Vues, Blade et Tailwind CSS

https://github.com/heig-vd-devprodmed-course/heig-vd-devprodmed-course

Visualiser le contenu complet sur GitHub à cette adresse.

L. Delafontaine, avec l'aide de GitHub Copilot.

Plus de détails sur GitHub

Cette présentation est un résumé du contenu complet disponible sur GitHub.

Pour plus de détails, consulter le contenu complet sur GitHub ou en cliquant sur l'en-tête de ce document.

Objectifs (1)

- Décrire la partie "vue" du patron de conception MVC.
- Décrire le concept de moteur de template et son intérêt.
- Utiliser Blade pour créer des vues dans une application Laravel.
- Utiliser les directives de Blade pour structurer les vues et afficher des données.

Objectifs (2)

- Utiliser Blade pour créer des layouts réutilisables.
- Utiliser les slots (par défaut et nommés) pour passer du contenu aux composants.
- Utiliser Blade pour créer des composants réutilisables.
- Utiliser les layouts et les composants Blade pour structurer une application Laravel.

Objectifs (3)

- Mettre en place l'internationalisation (i18n) dans une application Laravel.
- Utiliser les fichiers de traduction pour supporter plusieurs langues.
- Décrire la différence entre du CSS "classique" et un framework CSS utilitaire comme Tailwind CSS.
- Utiliser Tailwind CSS pour styliser les interfaces utilisateur.

Introduction aux vues dans le patron MVC

Le patron MVC (Model-View-Controller) sépare une application en trois composants principaux :

- **Model** : données et logique métier (séance précédente).
- **View** : présentation des données à l'utilisateur.
- **Controller** : logique de contrôle (prochaine séance).

Dans cette séance, nous allons étudier les **vues**.

Les moteurs de templates (1)

Un moteur de template permet de générer du HTML dynamique en combinant des données avec des templates prédéfinis.

Sans moteur de template, il faudrait mélanger du code PHP directement dans le HTML.

Problèmes : syntaxe verbeuse, échappement manuel, difficulté de maintenance.

Les moteurs de templates (2)

```php
<!DOCTYPE html>
<html>
<head>
    <title><?php echo $title; ?></title>
</head>
<body>
    <h1><?php echo $title; ?></h1>
    <ul>
    <?php foreach ($posts as $post): ?>
        <li><?php echo htmlspecialchars($post->title); ?></li>
    <?php endforeach; ?>
    </ul>
</body>
</html>
```

Blade : le moteur de templates de Laravel

Blade est le moteur de template inclus avec Laravel.

- Fichiers avec l'extension `.blade.php`.
- Permet d'utiliser du PHP pur si nécessaire.
- Directives simples pour les opérations courantes.
- Une syntaxe plus légère et plus lisible que le PHP natif.
- Voici un exemple de vue Blade issu de l'exemple précédent :

```
<!DOCTYPE html>
<html>
<head>
    <title>{{ $title }}</title>
</head>
<body>
    <h1>{{ $title }}</h1>
    @if (isset($user))
        <p>Bienvenue, {{ $user->name }} !</p>
    @else
        <p>Veuillez vous connecter.</p>
    @endif

    <ul>
    @foreach ($items as $item)
        <li>{{ $item->name }}</li>
    @endforeach
    </ul>
</body>
</html>
```

Lien avec les routes

Les vues sont rendues depuis les routes (ou contrôleurs) :

```php
// routes/web.php
Route::get('/about', function () {
    return view('about');
});
```

La fonction `view()` indique à Laravel de rendre la vue correspondante (`resources/views/about.blade.php`).

Des données peuvent être passées à la méthode `view()` .

Passage et affichage de données (1)

Les données sont passées aux vues via un tableau associatif :

```php
Route::get('/profile', function () {
    $user = User::find(1);

    return view('profile', [
        'user' => $user,
        'title' => 'Profil utilisateur'
    ]);
});
```

Passage et affichage de données (2)

Affichage dans la vue :

```
<h1>{{ $title }}</h1>

<div>
    <p>Nom : {{ $user->name }}</p>
    <p>Email : {{ $user->email }}</p>
</div>
```

La syntaxe `{{ }}` échappe automatiquement les données (protection XSS).

Syntaxe de base et directives (1)

Blade propose des directives pour les opérations courantes :

- Affichage de données.
- Commentaires.
- Appel de fonctions.
- Structures de contrôle (conditions, boucles, etc.).
- Et bien plus encore (inclusions, composants, etc.).

La documentation officielle de Laravel offre une liste extrêmement complète des directives Blade disponibles.

Affichage de données

Pour afficher des données, Blade utilise la syntaxe `{{ }}` :

```
<h1>{{ $title }}</h1>
<p>Bienvenue, {{ $user->name }} !</p>
```

Cette syntaxe échappe automatiquement les données pour prévenir les attaques XSS (Cross-Site Scripting), comme étudié en ProgServ1 et ProgServ2.

C'est équivalent à utiliser `htmlspecialchars()` en PHP pur.

Commentaires

Les commentaires dans Blade sont écrits avec la syntaxe `{{-- --}}` et ne seront pas inclus dans le HTML généré :

```
{{-- Ceci est un commentaire Blade --}}
```

Appel de fonctions

Vous pouvez appeler des fonctions PHP et des méthodes d'objets directement dans les templates Blade :

```
<p>Membre depuis : {{ $user->created_at->format('d/m/Y') }}</p>
<p>Nombre d'articles : {{ count($user->posts) }}</p>
<p>Nom en majuscules : {{ strtoupper($user->name) }}</p>
```

Structures de contrôle (1)

Blade offre des directives élégantes pour les structures de contrôle courantes.

```
@if ($user->isAdmin())
    <p>Vous êtes administrateur.trice.</p>
@elseif ($user->isModerator())
    <p>Vous êtes modérateur.trice.</p>
@else
    <p>Vous êtes un.e utilisateur.trice standard.</p>
@endif
```

Structures de contrôle (2)

```
@foreach ($posts as $post)
    <article>
        <h2>{{ $post->title }}</h2>
        <p>{{ $post->content }}</p>
    </article>
@endforeach
```

Structures de contrôle (3)

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

@while ($condition)
    <p>En cours...</p>
@endwhile

@forelse ($posts as $post)
    <article>{{ $post->title }}</article>
@empty
    <p>Aucun post trouvé.</p>
@endforelse
```

Création de vues avec Artisan

Laravel fournit une commande pour créer rapidement des vues dans le répertoire `resources/views` ou un sous-répertoire.:

```
php artisan make:view profile
```

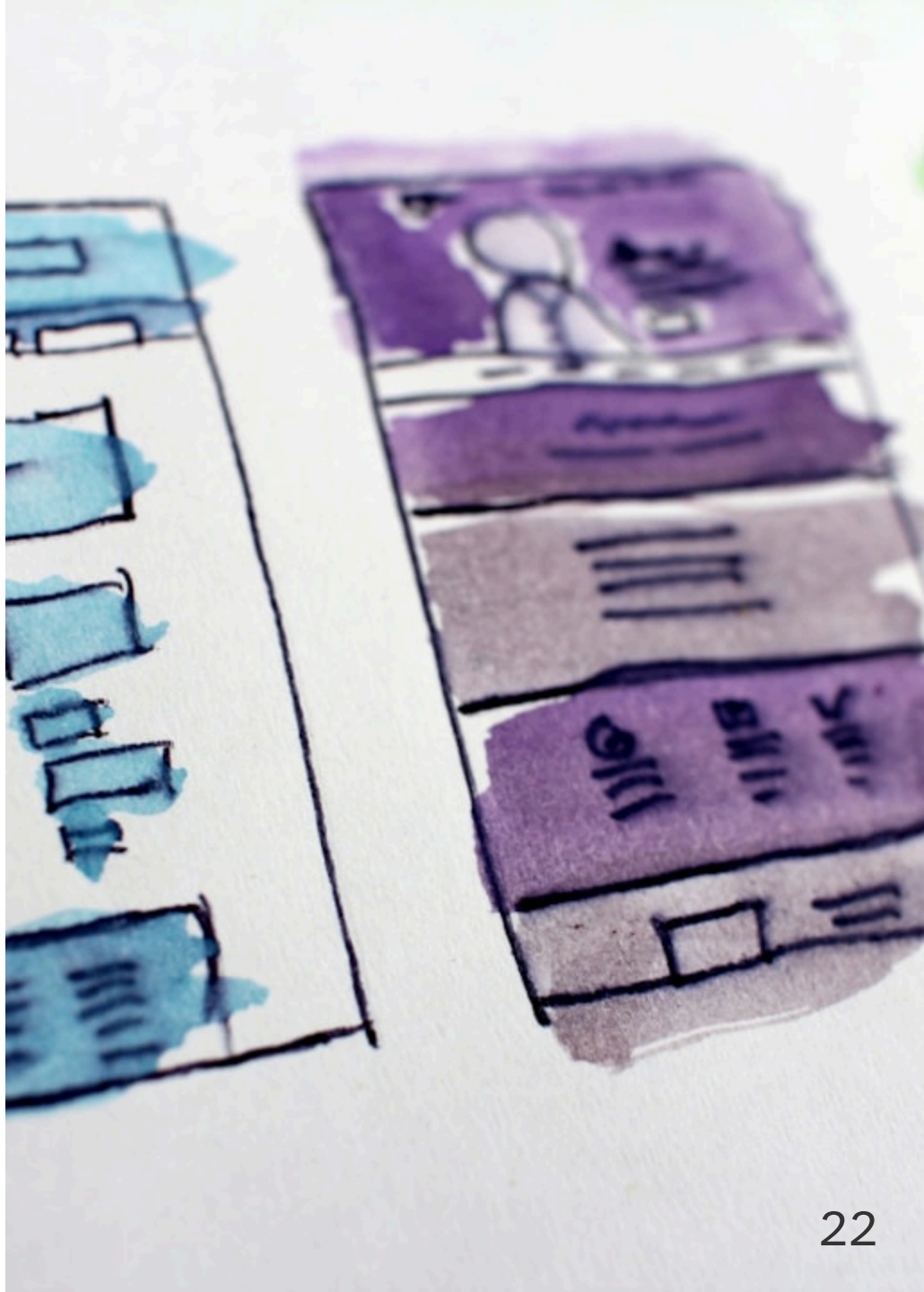
Crée `resources/views/profile.blade.php`.

```
php artisan make:view users.profile
```

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

Layout Blade

- Un layout définit la structure de base d'une page (HTML, head, body, navigation, footer) réutilisable sur plusieurs pages.
- Évite la duplication de code et facilite la maintenance.
- Des parties du layout peuvent être dynamiques grâce à des *"slots"*.



Approche avec les composants

Laravel recommande d'utiliser des composants Blade pour créer des layouts :

```
php artisan make:component DefaultLayout
```

Crée deux fichiers :

- `app/View/Components/DefaultLayout.php` : classe du composant.
- `resources/views/components/default-layout.blade.php` : vue du composant.

Structure d'un layout

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Mon Application</title>
</head>
```

```
<body>
    <header>
        <nav>
            <a href="{{ url('/') }}">Accueil</a>
            <a href="{{ url('/about') }}">À propos</a>
        </nav>
    </header>

    <main>
        {{ $slot }}
    </main>

    <footer>
        <p>&copy; {{ date('Y') }} Mon Application</p>
    </footer>
</body>
</html>
```

Utilisation du layout

Pour utiliser le layout dans une vue :

```
<x-default-layout>
    <h1>Page d'accueil</h1>
    <p>Bienvenue sur notre site !</p>
</x-default-layout>
```

Le contenu entre les balises est automatiquement inséré à l'emplacement de `{{ $slot }}`.

Slots par défaut et slots nommés (1)

- **Slot par défaut** : `{{ $slot }}` reçoit tout le contenu entre les balises.
- **Slots nommés** : pour passer plusieurs sections de contenu à un composant en utilisant la syntaxe `<x-slot:nom>`.
- L'exemple suivant illustre les deux types de slots dans un composant.

Slots par défaut et slots nommés (2)

```
<article>
    <header>
        {{ $header }}
    </header>
    <div>
        {{ $slot }}
    </div>
    @isset($footer)
        <footer>
            {{ $footer }}
        </footer>
    @endisset
</article>
```

```
<x-card>
    <x-slot:header>
        <h2>Titre</h2>
    </x-slot:header>

    <p>Contenu principal.</p>

    <x-slot:footer>
        <button>Acheter</button>
    </x-slot:footer>
</x-card>
```

Composants Blade

- Les composants Blade sont des éléments d'interface réutilisables.
- Évitent la duplication de code et facilitent la maintenance.
- Ils s'apparentent à des fonctions ou classes qui génèrent du HTML à partir de données et de templates.
- Les composants peuvent être utilisés pour créer des éléments d'interface complexes (cartes, boutons, formulaires, etc.) de manière modulaire.

Création d'un composant

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

Crée deux fichiers :

- `app/View/Components/Alert.php` : classe du composant.
- `resources/views/components/alert.blade.php` : vue du composant.

Vue du composant

Exemple de composant d'alerte (`resources/views/components/alert.blade.php`) :

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

Définit la structure HTML avec une classe dynamique basée sur `$type` .

Classe du composant

```php
<?php
// Imports et namespace omis pour la clarté...

class Alert extends Component
{
    public function __construct(
        public string $type = 'info',
    ) {}

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

Utilisation du composant et passage de propriétés

```
<x-alert type="success">
    Votre profil a été mis à jour avec succès !
</x-alert>

<x-alert type="error">
    Une erreur s'est produite lors de l'enregistrement.
</x-alert>

<x-alert>
    Information générale (type par défaut : info).
</x-alert>
```

Passer des variables à des composants

Pour passer des variables PHP (et non des chaînes) :

```
@foreach ($posts as $post)
    <x-post-card :post="$post" :show-author="true" />
@endforeach
```

- Préfixe `:` devant l'attribut indique une variable PHP.
- Sans le préfixe, Laravel traite la valeur comme une chaîne littérale (= une chaîne de caractères).

Internationalisation (i18n)

L'internationalisation (i18n) est le processus de conception d'une application pour qu'elle puisse être facilement adaptée à différentes langues et régions.

18 lettres entre le `i` et le `n` de *"internationalization"*.

Pourquoi l'internationalisation est importante

Même pour une seule langue initialement :

- **Évolutivité** : facilite l'ajout de nouvelles langues.
- **Maintenance** : centralise tous les textes.
- **Professionnalisme** : sépare le code de la présentation.
- **Réutilisabilité** : permet de changer les textes sans toucher au code.

Vocabulaire

Lorsqu'on parle d'internationalisation, il est important de connaître les termes suivants :

- **Locale** : code langue + région (ex : `fr`, `fr_CH`, `en_US`).
- **Clé de traduction** : identifiant unique pour un texte traduisible.
- **Fichier de traduction** : fichier contenant les traductions pour une locale.
- **Fallback locale** : langue de secours si traduction manquante.

Configuration de la locale

Configuration dans le fichier `.env` :

- `APP_LOCALE` : langue par défaut de l'application.
- `APP_FALLBACK_LOCALE` : langue de secours si une traduction est manquante.
- `APP_FAKER_LOCALE` : locale utilisée par Faker (une librairie pour générer des données factices).

Ne pas oublier de mettre à jour le fichier `.env.example` pour documenter ces variables !

Fichiers de traduction (1)

Organisés par langue dans le dossier `lang/` :

```
lang/
├─── en/
│    ├─── auth.php
│    └─── ...
└─── fr/
     ├─── auth.php
     ├─── pagination.php
     ├─── passwords.php
     ├─── ui.php
     └─── validation.php
```

Fichiers de traduction (2)

Exemple `lang/fr/ui.php` :

```php
<?php
return [
    'home' => [
        'title' => 'Accueil',
        'welcome' => 'Bienvenue sur :app_name',
    ],
    'profile' => [
        'title' => 'Profil',
        'edit' => 'Modifier le profil',
    ],
];
```

Utilisation des traductions dans les vues

Fonction `__()` pour récupérer une traduction :

```
<h1>{{ __('ui.home.title') }}</h1>
<p>{{ __('ui.home.welcome', ['app_name' => "Mon Application"]) }}</p>
```

Notation point : `fichier.clé.sous-clé`.

Paramètres passés dans un tableau et référencés avec `:nom`.

Traductions au pluriel

Laravel gère automatiquement les formes plurielles :

```
<p>{{ trans_choice('ui.posts.likes_count', $likesCount) }}</p>
```

Dans le fichier de traduction :

```
'likes_count' => "{0} Aucun like|{1} :count like|[2,*] :count likes",
```

Laravel choisit automatiquement la forme appropriée selon le nombre à l'aide des séparations `|` et des conditions associées.

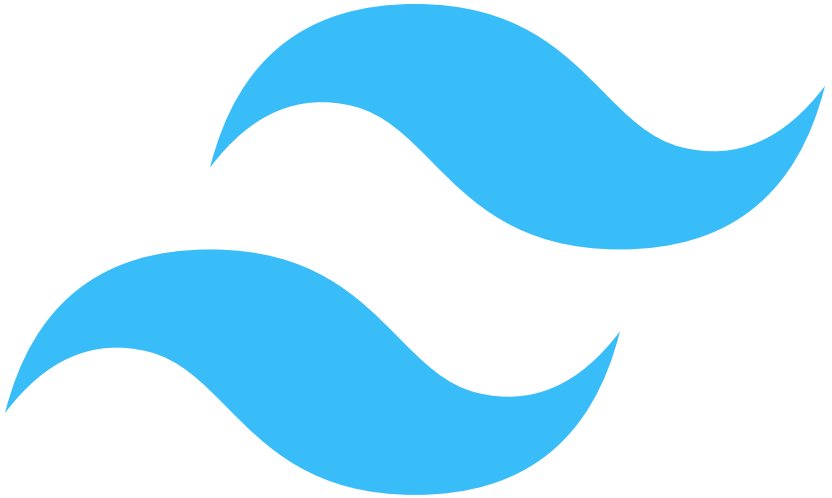
Librairie

`laravel-lang/lang`

- Fournit des traductions de base prêtes à l'emploi pour Laravel.
- Utilisé dans par certaines fonctionnalités de Laravel.
- Sera utilisé dans de futures séances pour les messages d'erreur de validation, etc.

Tailwind CSS

- Framework CSS utilitaire qui permet de construire des interfaces modernes rapidement.
- Utilise des classes CSS prédéfinies directement dans le HTML.
- Permet de styliser les éléments sans écrire de CSS personnalisé.



Approche CSS classique

```html
<div class="card">
        <h2 class="card-title">Titre</h2>
        <p class="card-content">Contenu de la carte.</p>
</div>
```

```css
.card {
        background-color: white;
        border-radius: 8px;
        padding: 20px;
        box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

/* ...autres styles pour .card-title, .card-content, etc. */
```

Approche avec Tailwind CSS (1)

Classes utilitaires directement dans le HTML :

```
<div class="bg-white rounded-lg p-5 shadow-md">
        <h2 class="text-2xl font-bold mb-3">Titre</h2>
        <p class="text-gray-600 leading-relaxed">Contenu de la carte.</p>
</div>
```

Approche avec Tailwind CSS (2)

Avantages

- Pas besoin d'inventer des noms de classes.
- Tout au même endroit.
- Système de design unifié.
- CSS minimal généré automatiquement.

Inconvénients

- HTML plus verbeux.
- Courbe d'apprentissage pour les conventions de classes.
- Peut être difficile à lire pour les non-initiés.
- Dépendance à un framework spécifique.

Intégration avec Laravel et Vite

Vite est intégré dans Laravel pour gérer les dépendances et les processus de build lié au CSS, JavaScript, etc., dont Tailwind CSS.

```
<!DOCTYPE html>
<html>
<head>
    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
<body>
    <!-- Votre contenu -->
</body>
</html>
```

Aller plus loin avec Tailwind CSS

- Tailwind CSS est un framework très puissant avec de nombreuses fonctionnalités avancées (variants, plugins, etc.).
- Le support de cours vous donne d'autres ressources pour aller plus loin.
- **Ce cours ne mettra pas l'accent sur Tailwind CSS, l'important est Laravel.**
- Tailwind CSS est utilisé pour fournir une base d'interface élégante et cohérente, mais l'accent du cours est sur Laravel et la programmation côté serveur.

Conclusion

- Les vues affichent les données aux utilisateur.trices et sont une partie essentielle du patron MVC.
- Blade est un moteur de template puissant et facile à utiliser pour créer des vues dans Laravel.
- Les layouts et les composants Blade permettent de structurer et de réutiliser les interfaces.
- L'i18n est cruciale pour supporter plusieurs langues et régions.
- Tailwind CSS est un framework utilitaire qui facilite la création d'interfaces modernes.

Questions

Est-ce que vous avez des questions ?

À vous de jouer !

- (Re)lire le contenu de cours.
- Faire les exercices.
- Faire le mini-projet.
- Poser des questions si nécessaire.

➡️ Visualiser le contenu complet sur GitHub.

N'hésitez pas à vous entraidez si vous avez des difficultés !

Sources

- Illustration principale par Richard Jacobs sur Unsplash
- Illustration par Aline de Nadai sur Unsplash
- Illustration par Hal Gatewood sur Unsplash
- Illustration de la gestion multilingue (i18n) générée avec ChatGPT à partir de la saisie suivante :
  "Make a realistic scene of a cookie as the Earth with all continents as the chocolate chips."
- Illustration par Nikita Kachanovsky sur Unsplash