



Tutoriel création de site web Laravel

Antoine Mouchamps - 15 novembre 2023



N-HiTec

Allée de la Découverte 10, 4000 Liège, Belgium

nhitec.com | info@nhitec.com

Table des matières

Table des matières	2
I Introduction	4
1 Laravel, c'est quoi ?	4
1.1 Prélude	4
1.2 Fonctionnement et philosophie	4
II Premier site web	6
1 Setup initial	6
2 Premières routes & views	6
2.1 welcome!	6
2.2 Controller	7
3 Bootstrap et CSS	10
3.1 Qu'est-ce que le CSS ?	10
3.2 A quoi sert Bootstrap ?	10
3.3 Vite	10
3.4 Installation	11
3.5 Utilisation	12
3.6 Navbar	14
4 Base de donnée	16
4.1 PHPMyAdmin	16
4.2 Models	16
4.3 Migrations : théorie	16
4.4 Migrations : exécution	17
4.5 PostsController	18
4.5.1 index	18
4.5.2 show	19
4.5.3 create	20
4.5.4 store	22
4.5.5 edit	23
4.5.6 update	23
4.5.7 delete	24
5 Messages	25
6 Authentification	27
6.1 Fortify	27
6.1.1 Installation	27
6.1.2 Configuration	27
6.1.3 Views	28
6.2 Middlewares	30
7 Conclusion	31
7.1 Postambule	31

7.2	Et ensuite ?	31
-----	------------------------	----



I. Introduction

Nous revoilà ! Si vous lisez ceci, j'espère que vous avez bien effectué la formation précédente sur l'installation de Docker et Laravel Sail, parce que tout ce qui suit va en découler !

1 Laravel, c'est quoi ?

1.1 Prélude

Laravel est ce qu'on appelle un *framework*. C'est à dire un ensemble d'outil fournissant une architecture de base sur laquelle n'importe quel site web peut être bâti. A la fin de ce "petit" tutoriel, vous serez je l'espère capable d'utiliser les fonctionnalités principales de Laravel, ainsi que les langages utilisés par ce framework et par la création de site web en général : PHP, HTML et en allant un petit peu plus loin, CSS, JQuery et Javascript. Cela paraît beaucoup d'un coup, mais en y allant méthodiquement et pas à pas, ça devrait bien se passer !

Bon alors, et ce framework alors ? Comment fonctionne-t'il ?

1.2 Fonctionnement & philosophie

Laravel utilise une architecture dite "MVC" (Modèle, Vue, Contrôleur) qui est décrite par la figure FIGURE 1. Elle se base donc sur 4 concepts :

1. **Routing**: Le routing est l'étape consistant à lier une URL, une route, à une action spécifique, qui sera effectuée par une méthode (dans le sens *Object-oriented-programming* du terme) contenue dans un controller.
2. **Controller**: Les controllers sont donc appelés par les routes, ce sont eux qui vont s'occuper de manipuler les données, effectuer x-y-z tâches, et enfin d'envoyer une certaine view à l'utilisateur.
3. **View**: Le concept de view est plutôt simple : Avec Laravel, chaque view correspond grosso-modo à une page que l'utilisateur voit affichée sur son écran.
4. **Model**: Enfin, les données stockées dans la base de donnée ne sont pas traitées telles quelles. Laravel nous facilite la vie en associant chaque type de donnée à un model, qui sera plus simple à utiliser par les controllers et comportera des fonctionnalités très utiles.



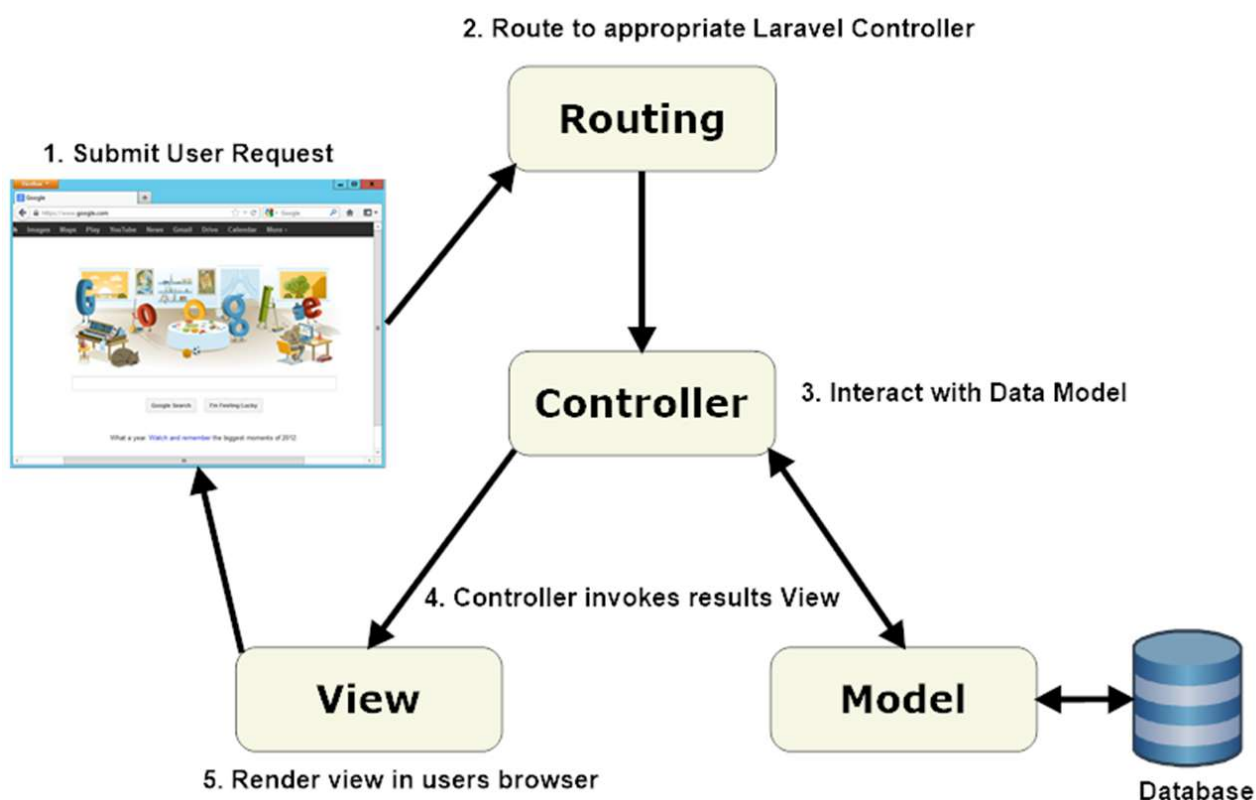
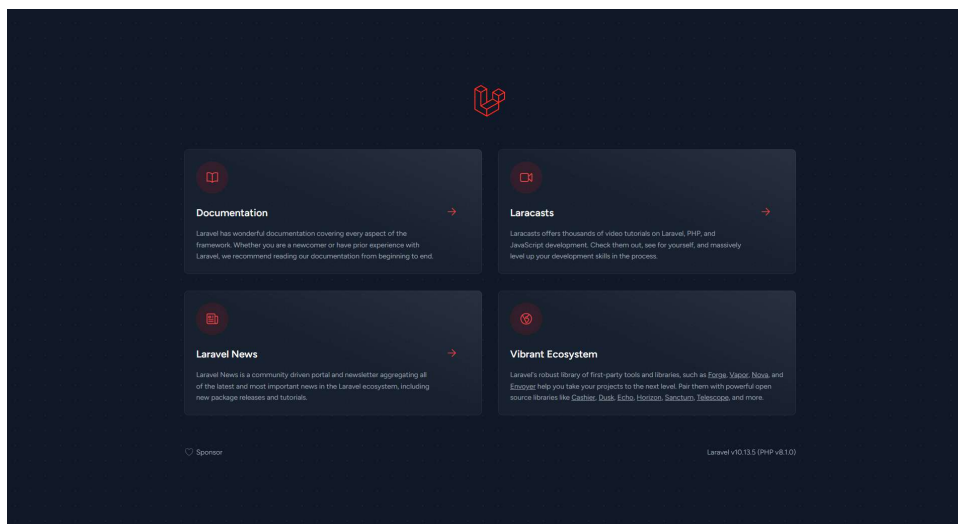


FIGURE 1

II. Premier site web

1 Setup initial

Toute cette partie est couverte par la formation précédente. Normalement, à la suite de tutoriel, vous devriez avoir obtenu le site suivant en vous rendant sur <http://lurlquevousavezchoisie>¹. Nous allons partir de ce site là. Pour ce tutoriel, mon projet sera appelé tutorialstepbystep



donc son URL sera <http://tutorialstepbystep/>.

2 Premières routes & views


2.1 welcome !

Les routes se trouvent dans le fichier `routes/web.php`.


Dans ce fichier se trouve cette route par défaut. Le premier argument de `get()` est l'adresse (absolue) qui sera visée par la route. En l'occurrence, la fonction en deuxième argument sera exécutée lorsque l'URL est '/', donc <http://tutorialstepbystep/>.

```
16 Route::get('/', function () {  
17     return view('welcome');  
18 });
```

Remarquons que la fonction exécutée retourne la view `welcome`, c'est pourquoi nous voyons la page d'accueil de laravel en nous rendant à cette adresse.

Allons voir le contenu de cette view. Les views en Laravel  ne sont pas écrites en fichier `.html` classiques, mais sous la forme de fichiers `fichier.blade.php`, qui permettent d'ajouter des fonc-

1. PS : n'oubliez pas de lancer `sail up -d` et `sail npm run dev` si cela n'est pas déjà fait ! Ne vous inquiétez pas, l'explication pour la deuxième commande va bientôt arriver. . .




tionnalités en plus² à l'HTML  classique.

Il y a bien beaucoup de chose dedans mais pas de panique : supprimons tout.

Plus précisément, ne gardons que ceci :

```
resources > views > welcome.blade.php > html
1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3  <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1">
6
7      <title>Laravel</title>
8  </head>
9  <body>
10
11 </body>
12 </html>
13
```

On y voit déjà plus clair. Dans ce qu'il reste, il n'y a que deux choses principales à retenir pour le moment :

1. le tag <head> est l'endroit où les styles (CSS ) et scripts (JQuery  & Javascript ) sont importés, ainsi que 2-3 autres choses.
2. le tag <body> est le tag qui contiendra tout ce qui sera affiché par le navigateur. Donc pour le moment, en allant sur votre site, vous verrez une page vide.

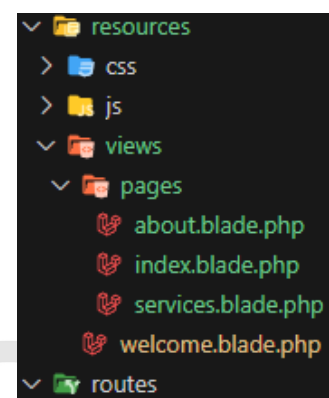
2.2 Controller

Bon, il est temps de remplir tout ça.

Commençons par créer un fichier pages dans resources/views/, et ajoutez trois autres fichiers comme indiqué ci-contre.

Ensuite, il va falloir créer un controller. Pour cela, tapez `sail artisan make:controller PagesController`³. Les controllers se trouvent dans `app\Http\controllers\`. Dans `PagesController`, créez trois fonctions comme à la FIGURE 2.

Vous l'aurez compris, `return view()` permet d'afficher la view donnée en argument : en l'occurrence, les 3 views `index`, `services` et `about`. Notez que `pages.` indique que ces 3 views se trouvent dans le dossier `pages`.



2. *HINT* : toutes les commandes commençant par un @ existent grâce au format blade, ainsi que la commande `{{ }}`. Plus d'informations [ici](#).

3. `sail artisan make:` est une commande très utile pour créer énormément de fichiers que nous verrons plus tard. Comme nous l'utilisons au travers de Laravel Sail , il faut utiliser `sail artisan` à la place

```

app > Http > Controllers > PagesController.php > ...
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class PagesController extends Controller
8  {
9      public function index() {
10         return view('pages.index');
11     }
12
13     public function services() {
14         return view('pages.services');
15     }
16
17     public function about() {
18         return view('pages.about');
19     }
20 }
21

```

FIGURE 2 – PagesController

Si vous vous rappelez bien de ce qu'on a vu plus tôt, chaque view HTML^{HTML} doit contenir un tag <head>. Celui-ci sera le même pour chaque page donc il serait judicieux⁴ de créer une sorte de template dans lequel on mettrait le <head> et qui sera ensuite utilisé pour les 26854 pages que comptera bientôt notre site !

COMME PAR HASARD les fichiers .blade.php nous permettent de faire cela : commencez par créer un dossier layouts dans les views, et créez un fichier dedans appelé app.blade.php. Ensuite, remplissez-le avec le contenu de welcome.blade.php (que vous pouvez désormais supprimer), puis ajoutez la commande @yield('content') dans le <body>. Enfin, il ne reste plus qu'à utiliser ce layout pour remplir vos 3 nouvelles views.

```

resources > views > pages > about.blade.php > p
1  @extends('layouts.app')
2
3  @section('content')
4      <h1> A propos </h1>
5      <p> Informations concernant N-HiTec </p>
6  @endsection

resources > views > pages > services.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4      <h1> Services </h1>
5      <p> Ceci est la page des services </p>
6  @endsection

resources > views > pages > index.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4      <h1> Accueil </h1>
5      <p> Ceci est le squelette de base pour le tutoriel d'introduction à la création de site web en utilisant Laravel v10 ! </p>
6  @endsection

```


FIGURE 3 – Contenu des views about (à gauche), services (à droite) et index (en bas)

Petit tuto HTML^{HTML} rapide : le tag <p> renferme un paragraphe, et le tag <h1> contient lui un h1 titre. De même, <h2> désignera un h2 sous-titre, <h3> un h3 sous-sous-titre, <h4> un h4 sous-sous-sous-titre, ...

Que se passe-t'il exactement ? Chacune des views va prendre le contenu du layout app (via @extends()), et remplir sa section content par ce qu'il y a entre @section et @endsection. Simple et efficace ! Nous rajouterons d'autres choses dans ce layout par la suite.

4. **D.R.Y** : *Don't Repeat Yourself* !

Enfin, pour pouvoir admirer le fruit de votre dur labeur, il faut créer les routes qui permettront d'afficher ces pages. Pour cela, rendez-vous dans `web.php` :

Afin d'utiliser notre nouveau controller, il faut le déclarer pour que Laravel  sache qu'il existe. C'est à ça que sert "use gngngn", (ce qui est suït est plutôt évident). Ensuite, décortiquons ce qui se passe : Comme précédemment, le 1er argument de `get()` donne l'adresse. Par exemple, la 2eme route est appelée à l'URL `http://tutorialstepbystep/services`. Le deuxième argument donne dans une array le controller ainsi que sa méthode à exécuter. Pour la deuxième route, aller à l'URL mentionnée va donc exécuter la fonction `services()` que nous avons créée il y a 5 (ou 40) minutes. Celle ci nous retourne la view correspondante, donc en allant sur cet URL nous voyons dans un coin de l'écran :

```
routes > web.php
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4
5  use App\Http\Controllers\PagesController;
6
7  /*
8   |-----
9   | Web Routes
10  |-----
11  |
12  | Here is where you can register web routes for your application. These
13  | routes are loaded by the RouteServiceProvider and all of them will
14  | be assigned to the "web" middleware group. Make something great!
15  |
16  */
17
18  Route::get('/', [PagesController::class, 'index'])->name('home');
19  Route::get('/services', [PagesController::class, 'services'])->name('services');
20  Route::get('/about', [PagesController::class, 'about'])->name('about');
```





FIGURE 4 – page internet moche



C'est laid, pas vrai ? Nous allons améliorer cela à la prochaine section.





3 Bootstrap & CSS



3.1 Qu'est-ce que le CSS ?

“De la même façon que HTML, CSS⁵ n'est pas vraiment un langage de programmation. C'est un langage de feuille de style, c'est-à-dire qu'il permet d'appliquer des styles sur différents éléments sélectionnés dans un document HTML”. (developer.mozilla.org). Cela signifie que le CSS  est le langage utilisé pour décrire comment chaque élément HTML  doit être affiché. Cela va de la taille et couleur du texte à la création de navbar, buttons, tables, etc... en passant par diverses animations simples ou plus complexes.

Le langage suit la philosophie suivante : à chaque sélecteur, on associe des propriétés. Les sélecteurs peuvent être des tags HTML  eux-mêmes, des class, id, ou d'autres choses. La bonne pratique est de styliser un maximum de composants en créant une multitude de class ayant chacune une tâche spécifique (taille, couleur, etc) afin d'obtenir une structure générale et modulaire, et ensuite d'assigner autant de class que l'on veut aux tags HTML  que l'on souhaite modifier. Plus de détails dans la section dédiée (WIP).

3.2 A quoi sert Bootstrap ?

Autant dire que la philosophie décrite conduit très rapidement à des fichiers énormes (milliers de lignes), illisibles, de successions de sélecteurs→propriétés, ce qui conduit à une maintenance plus que laborieuse, alors que là n'est souvent pas la partie sur laquelle les développeurs veulent passer du temps (sauf si c'est leur métier, évidemment⁶). C'est pour cela que de nombreux *frameworks front-end* existent afin d'amener de nombreuses class et plugins Javascript  prédéfinis. En l'occurrence, nous allons utiliser Bootstrap , qui est un *framework* utilisé pour construire des sites de manière *responsive*⁷ rapidement et facilement.

Pour plus de renseignement et pour découvrir les fonctionnalités de Bootstrap , rendez-vous sur [Doc Bootstrap !\[\]\(d47ad152ec3d86a04ad64c8049e1f17f_img.jpg\)](#). Dans la Section (WIP) et dans votre futur, vous utiliserez énormément de class Bootstrap . Il est donc important de se familiariser avec rapidement, par exemple en lisant la documentation des class que vous ne connaissez pas, même si c'est fort déroutant au début afin que cela roule tout seul à moyen terme.

3.3 Vite ? Comme la rapidité de cette formation ?

Les *frameworks* “règlent le problème” des 10000 lignes de code à écrire, mais pas celui de leur gestion et compilation. Rentre alors en scène Vite  :

“Vite is a modern frontend build tool that provides an extremely fast development environment and bundles your code for production⁸. When building applications with Laravel, you will typically

5. Cascading Style Sheets


6. on parle alors de développeurs *front-end*, responsables entre autres du rendu des sites.





7. *responsive* signifie que le site/composant adapte son rendu en fonction de la taille de l'écran, du format etc, ce qui est quand même très important.




8. *production* signifie le déploiement du site pour le public

use Vite to bundle your application's CSS and JavaScript files into production ready assets". (Doc [Laravel](#) .

3.4 Installation

Pour installer Bootstrap , il suffit d'exécuter les commandes suivantes :

1. `sail npm install bootstrap @popperjs/core`, qui permet d'installer Popper, une bibliothèque Javascript  utilisé par Bootstrap .
2. `sail npm install sass --save-dev`, qui permet d'utiliser le langage SASS , utilisé par Bootstrap .

Ensuite, dans le dossier `resources`, nous allons renommer le dossier `css` en `sass` et le fichier `app.css` à l'intérieur en un fichier `app.scss`. Il nous reste ensuite à ajouter un fichier `_variables.scss` dans le dossier `sass`, que nous utiliserons plus tard. Pour l'instant, nous allons juste le remplir avec le contenu de la FIGURE 6. Vous devriez donc obtenir un arrangement comme à la FIGURE 5. Comme nous le verrons à la FIGURE 8, le fichier `app.scss` est compilé par Vite  en un fichier CSS  dans le dossier `public` qui permettra de customiser notre site web. Il faut donc le remplir par ce qui est donné par la FIGURE 7, où la première ligne permet de changer la police d'écriture utilisée par défaut (avec celle ajoutée dans `_variables.scss`), la deuxième importe le fichier `variables.scss` que nous venons de créer (qui pour le moment est vide) et enfin la troisième importe toutes les fonctionnalités de Bootstrap . Nous verrons dans une future section (WIP) comment personnaliser ces importations pour n'importer que les fonctionnalités que l'on utilise, et donc gagner en performances.

Par ailleurs, lorsque nous créerons des codes SASS  custom, nous les utiliserons en les importants dans le fichier `app.scss`.

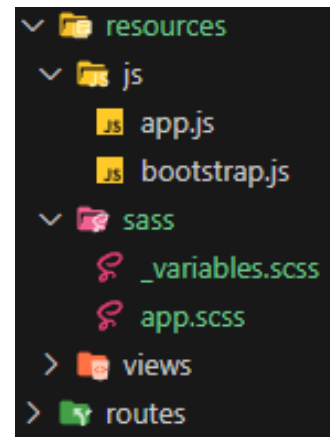




FIGURE 5


```
resources > sass > _variables.scss > ...
1 // Typography
2 $font-family-sans-serif: 'Nunito', sans-serif;
```

FIGURE 6

```
resources > sass > app.scss
1 // Fonts
2 @import url('https://fonts.bunny.net/css?family=Nunito');
3
4 // Variables
5 @import 'variables';
6
7 // Bootstrap
8 @import '~bootstrap/scss/bootstrap';
```

FIGURE 7

En ce qui concerne le code Javascript  utilisé par Bootstrap , il faut l'importer en ajoutant `import * as bootstrap from 'bootstrap'` dans le fichier `resources/js/app.js`.


Enfin, il faut modifier les paramètres de Vite  pour prendre en compte les changements que nous avons mis en place. Modifier donc la ligne 8 et ajoutez d'autres lignes dans le fichier vite.config.js situé dans le dossier racine du site.

```
vite.config.js > ...
1  import { defineConfig } from 'vite';
2  import laravel from 'laravel-vite-plugin';
3  import path from 'path';
4
5  export default defineConfig({
6    plugins: [
7      laravel({
8        input: ['resources/sass/app.scss', 'resources/js/app.js'],
9        refresh: true,
10      }),
11    ],
12    resolve: {
13      alias: {
14        '~bootstrap': path.resolve(__dirname, 'node_modules/bootstrap')
15      }
16    }
17  });
18
```



FIGURE 8

Remarquez que jusqu'ici, on ne voit toujours pas de changement quant on va sur notre site. La raison est que nous n'avons pas encore "dit" à nos views d'utiliser nos ajouts. Pour cela, rendez-vous dans notre layout (càd app.blade.php) et ajoutons la ligne suivante :

```
resources > views > layouts > app.blade.php > ...
1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3
4  <head>
5    <meta charset="utf-8">
6    <meta name="viewport" content="width=device-width, initial-scale=1">
7
8    <title>Laravel</title>
9
10   <!-- Scripts -->
11   @vite(['resources/sass/app.scss', 'resources/js/app.js'])
12
13 </head>
14 <body>
15   @yield('content')
16 </body>
17 </html>
18
```

Maintenant, tout est prêt pour commencer à utiliser Bootstrap .

3.5 Utilisation

Comme expliqué plus haut, Bootstrap  nous fournit une multitude de class que nous pouvons utiliser pour styliser nos tags HTML . Modifions donc nos views about.blade.php et index.blade.php comme aux FIGURES 9&10.

```
resources > views > pages > about.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4
5  <div class="container">
6    <h1> A propos </h1>
7    <p> Informations concernant N-HiTec </p>
8  </div>
9
10 @endsection
```

FIGURE 9 –
about.blade.php

```
resources > views > pages > index.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4
5  <div class="jumbotron text-center">
6    <h1> Accueil </h1>
7    <p> Ceci est le squelette de base pour le tutoriel d'introduction à la création de site web en utilisant Laravel v10 ! </p>
8    <a role="button" href="" class="btn btn-primary btn-lg">Se connecter</a>
9    <a role="button" href="" class="btn btn-success btn-lg">S'enregistrer</a>
10  </div>
11
12 @endsection
```

FIGURE 10 – index.blade.php

Par exemple, dans la FIGURE 10, class="text-center" assigne la classe text-center à l'élé-

ment, et en maintenant la touche **Ctl** enfoncée et en passant le curseur sur la classe, vous verrez une classe⁹ permettant de centrer un élément dans son conteneur.

Dans le petit cadre, nous voyons le code CSS¹⁰ correspondant à cette classe.

```
@section('content')
    .text-center{text-align:center!important}
<div class="jumbotron text-center">
```

Pour la customisation de la page des services, nous allons en profiter pour découvrir une nouvelle mécanique : passer des données aux pages. En effet, c'est quand même pratique de pouvoir afficher des informations dynamiquement ! Pour le moment, nous n'allons pas encore s'embêter avec la base de donnée, nous allons juste voir comment la mécanique de base fonctionne. Rappelez-vous de la Section 1.2, ce sont les controllers qui s'occupent de manipuler les données avant d'afficher une view. Dès lors, c'est dans la fonction `services()` de `PagesController.php` que nous allons ajouter des choses :

```
13 public function services() {
14     $titlefromcontroller = 'Voici les Services fournis par N-HiTec';
15     $services = ['Programmation web', 'introduction à la gestion d\'entreprise', 'des rencontres', 'du fun !'];
16
17     return view('pages.services')->with([
18         'title' => $titlefromcontroller,
19         'services' => $services
20     ]);
21 }
```

`$titlefromcontroller` et `$services` sont 2 variables, et nous les passons à la view par l'intermédiaire du `->with(...)`.

Ensuite, nous pouvons utiliser les variables `title` et `services` dans la view concernée. Dans la Section 2.1 nous avons pris connaissance des avantages du format `.blade.php`. Celui-ci nous apporte donc les commandes `{{}}`¹¹ et `@foreach()`¹².

```
resources > views > pages > services.blade.php > ...
1 @extends('layouts.app')
2
3 @section('content')
4
5 <div class="container">
6 <h1> Services </h1>
7 <p> {{ $title }} </p>
8 <ul class="list-group">
9     @foreach ($services as $service)
10         <li class="list-group-item">{{ $service }}</li>
11     @endforeach
12 </ul>
13 </div>
14
15 @endsection
```

9. Avec les bonnes extensions VS code en tt cas... voir les extensions dans la section (WIP)

10. voir Section 3.1

11. Cette commande agit comme un `printf()` en C. Elle permet d'afficher le contenu de la variable en argument.

12. Boucle `foreach` classique, pour itérer sur un tableau. Les tags `<a>` ainsi que `` et `` seront expliqués à la section suivante.

Ceci est un document interne. Ne pas diffuser.

Et voilà ! Maintenant, il suffit de taper `sail npm run dev`¹³ (si ce n'était pas déjà fait) pour admirer le résultat. C'est déjà vachement mieux, non ?

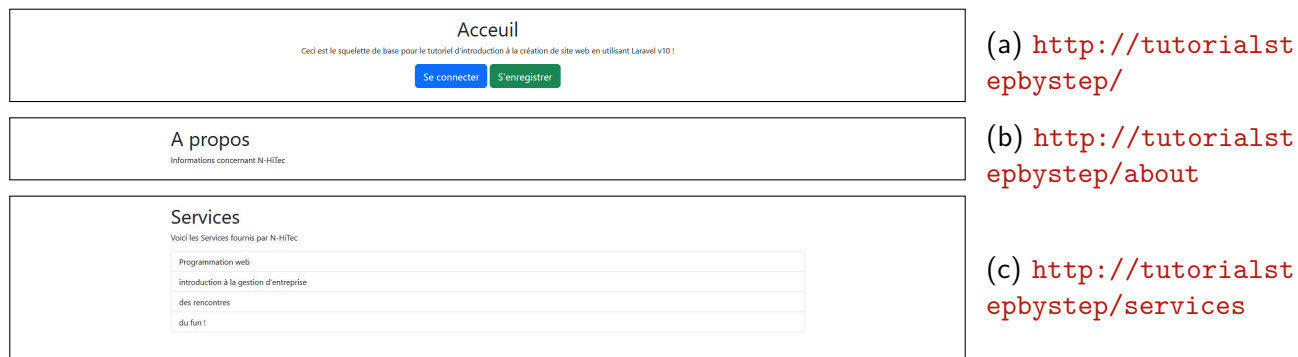


FIGURE 11 – 3 pages créées jusqu'à présent et stylisées avec Bootstrap ^B.

C'est bien beau, mais jusque ici le seul moyen de naviguer entre les pages est de rentrer leur URL, ce qui n'est ma foi pas fort pratique. Remédions à cela avant de passer à la suite.

3.6 Navbar

C'est un gros morceau qui utilise beaucoup des class de Bootstrap ^B, donc il va falloir s'accrocher. D'abord, créez un dossier `inc` dans `resources/views` et un fichier `navbar.blade.php` dans ce nouveau dossier. Remplissez ce fichier comme la FIGURE 12.

```
resources > views > inc > navbar.blade.php > ...
1 <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
2   <div class="container-fluid">
3     <a href="{{route('home')}}" class="navbar-brand">{{config('app.name', 'pasdetite')}}</a>
4     <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent">
5       <span class="navbar-toggler-icon"></span>
6     </button>
7     <div class="collapse navbar-collapse" id="navbarSupportedContent">
8       <ul class="navbar-nav me-auto mb-2 mb-lg-0">
9         <li class="nav-item">
10          <a href="{{route('services')}}" class="nav-link">Services</a>
11        </li>
12        <li class="nav-item">
13          <a href="{{route('about')}}" class="nav-link">A propos</a>
14        </li>
15      </ul>
16    </div>
17  </div>
18 </nav>
```

FIGURE 12

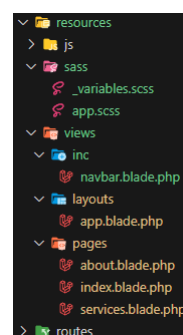


FIGURE 13

Ensuite, il faut ajouter notre *navbar* à notre *layout* afin qu'elle apparaisse sur toutes nos pages. Pour cela, rien de plus simple !

```
13 <body>
14   @include('inc.navbar')
15   <br>
16   @yield('content')
17 </body>
18 </html>
```

13. Cette commande permet de compiler le CSS^U et Javascript^U en créant un mini serveur localement. Cette commande utilisée lors du **DEVELOPPEMENT DU SITE** permet d'appliquer les modifications apportées à des fichiers rapidement sans devoir refresh la page. Pour compiler tout ça en production, il faut utiliser `npm run build`. Plus de `sail` car Laravel Sail ^U est un outil de développement, pas de déploiement !

Qu'est ce que c'est que tout ça ? Décomposons tout cela. Premièrement, nous découvrons ici trois nouveaux tags HTML 📄 :

1. `<a>` est un tag permettant la création d'un lien vers une autre URL, que l'on place dans l'attribut `href`. Au lieu de taper l'URL d'une route, Laravel 🐘 nous permet d'optimiser l'écriture en utilisant la commande `route('nomdelaroute')` afin d'obtenir l'URL en question. `{{ ... }}` permet ensuite de l' "afficher" dans le `href`.
2. `` est un tag signifiant la création d'une liste.
3. `` représente un élément d'une liste.

Pour le reste je vous invite à lire ce que font chaque `class` et de jeter un oeil sur [la doc Bootstrap](#) ^B sur les [navbars](#). Bien que ça soit indigeste lors d'une première lecture, ça l'est beaucoup moins que si nous devons analyser les `class` une par une...

Néanmoins, quelques notions clés :

- "*collapsing*" fait référence au fait de faire disparaître la navbar au profit d'une liste déroulable avec un bouton lorsque la largeur de l'écran devient plus petit qu'une valeur fixée (ici, 992px car on utilise le mot clé `lg`).
- le bouton spécial pour dérouler la navbar est créé par le tag `<button class="navbar-toggler">`, qui est invisible lorsque la largeur de l'écran est $> 992px$.
- rien à voir avec Bootstrap ^B, `config()` permet d'accéder à certaines valeurs, notamment celles du `.env`. En l'occurrence, `'app.name'` permet d'accéder à `APP_NAME` (et si cette valeur n'existe pas, le deuxième argument est affiché).

Dans la Section (WIP), nous verrons comment améliorer cette navbar. En attendant, voilà ce à quoi elle devrait ressembler :

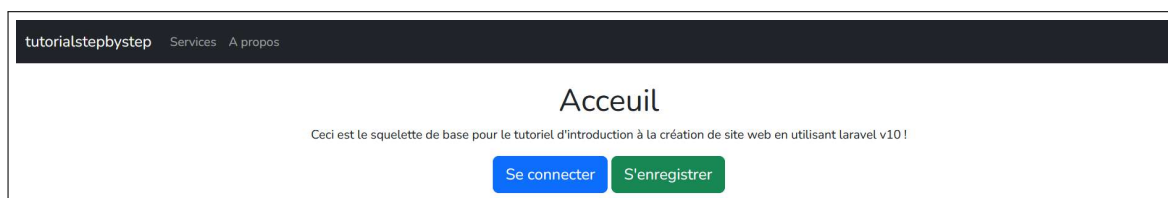


FIGURE 14 – navbar sur un écran de largeur $> 992px$

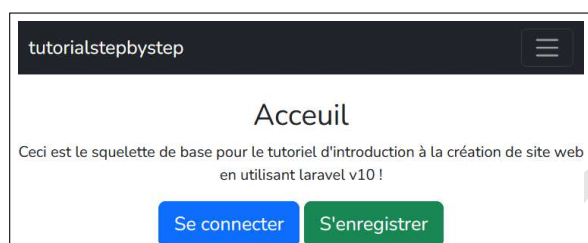


FIGURE 15 – navbar vue depuis un téléphone (largeur $< 992px$)

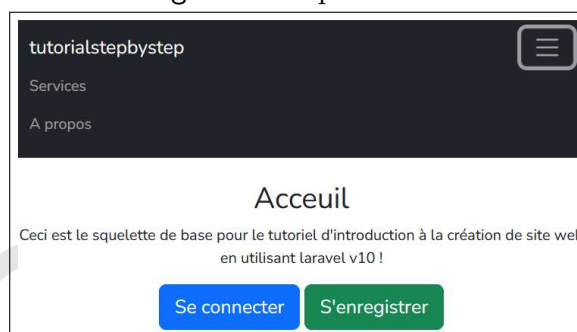


FIGURE 16 – navbar en ayant appuyé sur le bouton en haut à droite


4 Base de donnée

Nous allons créer une base de donnée en utilisant l'exemple de posts sur un blog.

4.1 PhpMyAdmin 🚢

PhpMyAdmin 🚢 est une interface permettant à des ignares comme nou... comme vous* de manipuler et de visualiser le contenu des bases de données facilement sans connaissances en MySQL. Normalement, la configuration a été effectuée lors de la formation précédente et vous devriez pouvoir y accéder en vous rendant sur <http://tutorialstepbystep:8080>.

4.2 Models

Comme dit dans la Section 1.2, le model est l'objet qui nous permettra d'interagir avec les *posts* stockés dans notre base de donnée¹⁴. Pour le créer, tapez `sail artisan make:model Post -m`, et observez la création d'un fichier `Post.php` dans `app/Models/`. Pour le moment, la class (au sens PHP  du terme) est vide, mais nous pouvons ajouter des méthodes spécifiques à ce model, ce qui, nous le verrons (WIP), est très pratique.

4.3 Migrations : comme les oiseaux ?

Une migration est un fichier qui permet de définir les tables de notre base de donnée. Une table est grosso modo un type de donnée que la base de donnée va stocker. Par exemple, la liste de tous les utilisateurs est une table, tout comme les *posts* que nous allons créer. Chaque table contient un certain nombre de *columns* qui elles sont les données stockées en temps que telles. En l'occurrence, notre table de *posts* doit contenir une *column* pour le titre d'un *post* et une pour son contenu en lui même¹⁵.

Pour créer notre migration, tapez rien du tout car la migration a été créée en même temps que notre model grâce au `-m` ! Elle se trouve dans `database/migration/xxxx_xx_xx_XXXXXX_create_posts_table.php`. Ensuite, remplissez là comme à la FIGURE 17¹⁶ :

Analysons tout a :

- `function up()` et `function down()` : La première est exécutée quand on souhaite créer les tables à l'intérieur tandis que la deuxième est exécutée lorsqu'on souhaite supprimer les tables de la base de donnée. Pour l'utilisation simple que nous faisons des migrations, pensez à `down` tout ce que vous `up`-per.
- `Schema::create('posts', ...` en gros, c'est la fonction utilisée pour créer la table 'posts', et les `$table->` qui suivent permettent de définir chaque *column* de la table créée.

14. pour chaque nouvel objet, on aura un model correspondant

15. Tous ces termes sont compliqués à décrire avec des mots, mais sont en réalité très intuitifs quand on imagine les données affichées dans un grand tableau à double entrée.

16. lignes 16 et 17, de rien

- `id()` : l'id est ce qu'on appelle la Primary Key. Unique pour chaque row¹⁷, il permet d'identifier chaque élément de donnée. Il se trouve par défaut sur chaque table nouvellement créée.
- `string()` permet de créer une column de type string, de taille 255. La taille est modifiable en ajoutant un nombre <255 en second argument.
- `mediumtext()` permet de créer une column de 16.777.215 caractères (oula).
- `timestamps()` ajoute une date de création et de modification à la table. Ces deux columns sont également ajoutées par défaut.

Quelques remarques supplémentaires :

1. Par défaut, les columns doivent obligatoirement posséder une valeur.
2. On peut ajouter de nombreux paramètres aux columns pour modifier leur comportements (exemple : `->nullable()` pour leur permettre d'être vide).
3. Les types des column correspondent chacune à un type de valeur de MySQL, le "vrai" langage pour communiquer avec les base de donnée duquel Laravel 🐘 nous protège grâce aux models, migrations, tables que nous venons de voir.

Encore une fois, parcourir la doc officielle de Laravel 🐘 permet d'en apprendre beaucoup plus que ce que tutoriel ne pourra jamais vous apprendre !

```
database > migrations > 2023_07_22_194605_create_posts_table.php > ...
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  return new class extends Migration
8  {
9      /**
10       * Run the migrations.
11       */
12      public function up(): void
13      {
14          Schema::create('posts', function (Blueprint $table) {
15              $table->id();
16              $table->string('title');
17              $table->mediumText('body');
18              $table->timestamps();
19          });
20      }
21
22      /**
23       * Reverse the migrations.
24       */
25      public function down(): void
26      {
27          Schema::dropIfExists('posts');
28      }
29  };
30
```

FIGURE 17 – Exemple très simple de migration

4.4 Migrations : exécution

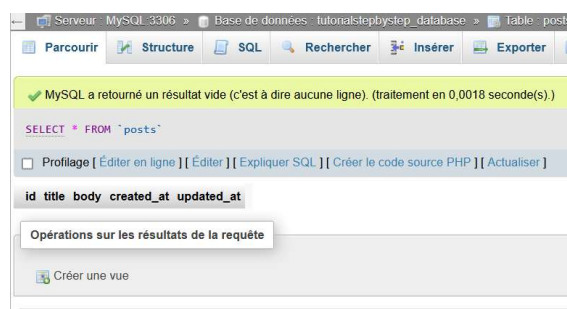
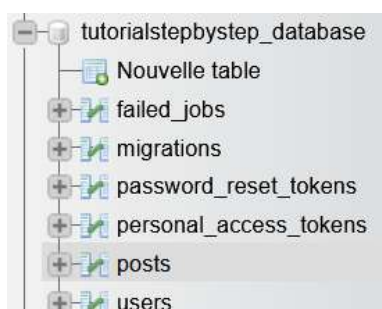
Bon, après tant de blabla, passons à l'action. Mais avant cela, Laravel 🐘 nous embête (pour une fois). Afin de n'avoir aucune erreur en exécutant la migration, il va falloir ajouter ces lignes dans `app/Providers/AppServiceProvider.php` :

```
4
5  use Illuminate\Support\ServiceProvider;
6  use Illuminate\Support\Facades\Schema;
7
8  class AppServiceProvider extends ServiceProvider
```

```
21      public function boot(): void
22      {
23          Schema::defaultStringLength(191);
24      }
25  }
```

Voilà ! maintenant tapez `sail artisan migrate:fresh --seed` (où `fresh` signifie que tout ce qui existait avant est supprimé et `--seed` permet d'exécuter les seeders vus dans la Section (WIP)) et, si tout va bien, vous verrez maintenant cela en allant dans PhpMyAdmin 🐙 :

17. une row est un élément de donnée dans une table



4.5 PostsController

Pour gérer ces posts, nous allons bien entendu avoir besoin d'un controller. Comme ce genre de donnée va avoir des manipulations basiques très communes (création, liste, affichage, modification, suppression,...), il existe un certain type de controller permettant de nous faire gagner du temps : le resource controller. Tapez donc

```
sail artisan make:controller PostsController --resource
```

 pour en créer un.

Ensuite, il faut évidemment définir des nouvelles routes. Une seule ligne toute simple nous permet de générer en réalité 7 routes différentes que nous verrons petit à petit. Ajoutez donc ces 2 lignes dans routes/web.php :

```
21 Route::get('/about', [PagesController::class, 'about'])->name('about');
22
23 Route::resource('posts', PostsController::class);
```

```
5 use App\Http\Controllers\PagesController;
6 use App\Http\Controllers\PostsController;
7
```

FIGURE 18

FIGURE 19

Notez que en donnant 'posts' en argument à resource(), celui-ci fait automatiquement lien avec le model Post¹⁸.

Petite parenthèse avant de continuer : un exemple de route générée par la commande de la FIGURE 18 est :


```
Route::get('/posts/{post}', [PostsController::class, 'show'])->name('posts.show')
```

le {...} dans l'URL est une sorte de paramètre dans l'URL, qui permet de passer des informations au controller. En effet, chaque paramètre dans l'URL sera passé (si on le souhaite) en argument à la fonction du controller appelée par cette route, dans le même ordre d'apparition que dans l'URL. Ce paramètre est extrêmement utile comme nous le verrons à la Section 4.5.2. Pour plus d'informations sur les routes générées par cette commande, voyez [ceci](#). Fin de la parenthèse.

Bon, maintenant, il faut remplir ce controller et créer les views qui vont avec. Commençons par les plus simples, index() et show().

4.5.1 index

Cette méthode est utilisée pour afficher la liste de tous les Posts créés.

18. Pour comprendre comment Laravel  fait pour être si intelligent, jetez un oeil à [ceci](#).

Avec cet exemple simple, on comprend vite à quel point il est simple d'interagir avec la base de donnée par l'intermédiaire des `models`. Nos posts sont stockés sous forme d'une array d'objets PHP `php`, contenant les champs que nous avons spécifiés dans la migration.

```
13 public function index()
14 {
15     $posts = Post::all();
16     return view('posts.index')->with([
17         'posts' => $posts
18     ]);
19 }
```

Ensuite, nous allons créer un nouveau dossier `posts` dans `resources/views` et ajouter un fichier dedans appelé `index.blade.php`. Il ne reste "plus qu'à" le remplir avec le contenu de la FIGURE 20.

```
resources > views > posts > index.blade.php > ...
1 @extends('layouts.app')
2
3 @section('content')
4
5 <div class="container">
6     <h1>Posts</h1>
7
8     @if(count($posts) > 0)
9         <ul class="list-group">
10             @foreach ($posts as $post)
11                 <li class="list-group-item">
12                     <h3><a href="{{route('posts.show', $post->id)}}">{{ $post->title }}</a></h3>
13                     <small>écrit le {{ $post->created_at }}</small>
14                 </li>
15             @endforeach
16         </ul>
17     @else
18         Aucun posts n'a été trouvé.
19     @endif
20 </div>
21
22 @endsection
```

Ici, tout est déjà connu mis à part le `@if`, mais c'est plutôt intuitif au vu de ce que je vous ai déjà appris. Ensuite, il y a le `->` permettant d'accéder aux différents champs des objets que sont nos `$post`.

FIGURE 20 – `index.blade.php`

Enfin, il ne nous reste plus qu'à ajouter un bouton à notre navbar afin d'accéder à cette page.

```
15 <li class="nav-item">
16     <a href="{{route('posts.index')}}" class="nav-link">Liste des posts</a>
17 </li>
18 </ul>
```

4.5.2 show

Même chose que pour la page précédente, il faut remplir la fonction `show()`.

Ici, nous utilisons ce que j'ai expliqué en dessous de la FIGURE 18. La fonction `show()` prend un argument, appelé `$id`. Pourquoi et d'où sort-il ? Son origine a été expliquée plus haut, il provient du paramètre `{post}` dans l'URL correspondant à la fonction¹⁹.

```
40 public function show(string $id)
41 {
42     $post = Post::find($id);
43
44     return view('posts.show')->with('post', $post);
45 }
```

Maintenant, pourquoi l'avoir appelé `$id` ? Simplement parce qu'il contient l'ID du post que nous souhaitons obtenir, puisque c'est la valeur que nous lui avons donné en écrivant `route(posts.show, $post->id)` dans la FIGURE 20²⁰.

PS : L'écriture un petit peu différente du `->with()` permet de simplifier l'écriture que nous avons utilisée jusqu'ici lorsque qu'il n'y a qu'un seul élément à ajouter.

19. Si on avait eu deux paramètres différents, par exemple `.../{post}/{comment}`, et qu'on avait gardé `show($id1)`, `$id1` aurait pris la valeur de `{post}`. Pour obtenir la valeur du deuxième paramètre on aurait du écrire `show($id1, $id2)` (et `$id2` aurait pris la valeur de `{comment}`).

20. Le nom du paramètre dans l'URL n'a donc aucun rapport avec le nom de la variable dans la fonction, si ce n'est que l'un donne sa valeur à l'autre

Enfin, la view correspondante peut être créée au nom et emplacement `resources/views/posts/show.blade.php`.

```
resources > views > posts > show.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4
5  <div class="container">
6      <h1>{{ $post->title }}</h1>
7
8      <div>
9          {{ $post->body }}
10     </div>
11     <hr>
12     <small> Ecrit le {{ $post->created_at }} </small>
13     <div>
14         <a href="{{ route('posts.index') }}" class="btn btn-default bg-warning">Retour</a>
15     </div>
16 </div>
17
18 @endsection
```

Pour le coup, R.A.S. en terme de nouveautés donc nous pouvons maintenant passer à la création de posts !

4.5.3 create

Cette partie est plus intéressante car elle va amener un nouveau concept, les *forms*. Jusque ici, nous n'avons jamais eu à rentrer nous mêmes des données et à les envoyer à notre site pour qu'il fasse des choses avec, c'est exactement à cela que servent les *forms*.

Tout d'abord, remplissons la méthode `create()` du controller et créons le fichier `resources/posts/create.blade.php` comme nous en avons maintenant l'habitude.

```
24 public function create()
25 {
26     return view('posts.create');
27 }
```

Maintenant, il faut remplir la view :

```
resources > views > posts > create.blade.php > ...
1  @extends('layouts.app')
2
3  @section('content')
4
5  <div class="container">
6      <h1>Créer un post</h1>
7      <form action="{{ route('posts.store') }}" method="POST">
8          @csrf
9          <div class="form-group">
10             <label for="titre">Titre</label>
11             <input type="text" name="titre" id="titre" class="form-control" placeholder="Titre" value="{{ old('titre') }}">
12          </div>
13          <div class="form-group">
14             <label for="body">Message</label>
15             <textarea name="message" id="body" class="form-control" placeholder="Message">{{ old('message') }}</textarea>
16          </div>
17          <br>
18          <button class="btn btn-primary">Créer</button>
19      </form>
20 </div>
21
22 @endsection
```

Ici il y'a pas mal de nouveautés. On voit en effet apparaître trois nouveaux tags HTML  :

1. `<form>` : Le form permet donc l'envoi d'information entrées par l'utilisateur. Pour cela, on doit lui fournir l'URL vers laquelle aller lorsque le form est envoyé, c'est ce que l'on met dans action. En l'occurrence, le resource controller nous a créé une route pour cela, appelée 'posts.store' et qui permettra de stocker les informations du form. Nous verrons cela en détail à la prochaine section. Ensuite, il faut préciser le type de requête effectuée, et lorsque l'on stocke quelque chose, c'est une requête POST²¹
2. `<label>` : Celui-ci est simplement utilisé pour donner un nom²² à un champ `<input>`. Il suffit de spécifier à quel champ le `<label>` fait référence en mettant le id de l'`<input>` dans l'attribut for du `<label>`.
3. `<textarea>` : Il fonctionne comme un `<input>` mais est spécialisé pour la gestion de paragraphes. L'attribut value est quant à lui substitué par l'espace entre les deux marqueurs `<textarea>` et `</textarea>`.
4. `<input>` : Enfin, ce tag correspond à un champ à remplir. Ce champ peut prendre plusieurs formes qui s'indiquent au moyen de l'attribut type. En l'occurrence nous utilisons un champ text pour une petite phrase/mots (ici, le titre de notre post)²³. Les `<input>` peuvent posséder de très nombreux attributs, ici nous utilisons les attributs :
 - name qui donne le nom de la variable accessible dans le controller, qui contiendra les données envoyées par cet `<input>`.
 - placeholder qui permet d'afficher un petit texte en fond dans le champ (pour par exemple donner un format de réponse)
 - value qui permet de donner une valeur prédéfinie au champ lorsque l'on affiche la page. Ici, `old()` est une fonction prenant en argument le nom d'un `<input>` et qui permet d'afficher la valeur du champ (si elle existait²⁴) lors de la requête précédente. C'est utile lorsque les données doivent être validées avant d'être stockées (ex : est-ce que tous les champs sont remplis?) et que l'utilisateur est renvoyé vers la page de saisie du form si la validation échoue.

La dernière commande inconnue est le `@csrf`. C'est une mesure de sécurité contre un certain type d'attaque. Plus d'information [ici](#).

Bon, ce fût beaucoup (trop) de bla-bla, mais on va (enfin) pouvoir passer à la suite ! Promis, les trois dernières méthodes seront bien plus simples. Mais avant cela, ne pas oublier d'ajouter un bouton sur notre navbar pour accéder à cette page :

21. Pour afficher une view, on utilisait des requêtes GET. [Pour en savoir plus](#)

22. un nom visible sur la page internet.

23. Il existe de nombreux autres types pour des fichiers, dates, boîtes à cocher, ... vous les trouverez [ici](#).

24. on peut donner en deuxième argument à la fonction un texte à afficher lorsque le champ était vide à la requête précédente.


```

7      <div class="collapse navbar-collapse" id="navbarSupportedContent">
8      <ul class="navbar-nav me-auto mb-lg-0">
9          <li class="nav-item">
10             <a href="{{route('services')}}" class="nav-link">Services</a>
11          </li>
12          <li class="nav-item">
13             <a href="{{route('about')}}" class="nav-link">A propos</a>
14          </li>
15          <li class="nav-item">
16             <a href="{{route('posts.index')}}" class="nav-link">Liste des posts</a>
17          </li>
18      </ul>
19      <hr class="d-lg-none text-white m-y-1">
20      <ul class="navbar-nav mb-2 mb-lg-0">
21          <li class="nav-item">
22             <a href="{{route('posts.create')}}" class="nav-link">Créer un post</a>
23          </li>
24      </ul>
25  </div>

```

4.5.4 store

On a vu à la section précédente que les données du form étaient envoyées à la route 'posts.store', or cette route redirige vers cette fonction ! C'est donc ici que nous allons stocker le post nouvellement créé.

Tout d'abord, nous allons vérifier si les 2 champs titre et message ont bien été remplis (s'ils doivent être remplis, ils sont requis \Rightarrow required). Pour cela, on utilise la fonction validate() de Laravel  ²⁵. Si la validation rate, l'utilisateur est renvoyé à la page précédente et si elle réussit, alors on continue.

L'étape suivante est de créer un nouveau post, via son model. Ensuite, on remplit ses champs avec les valeurs obtenues dans le form et on le sauvegarde dans la base de donnée. Enfin, il ne reste plus qu'à envoyer l'utilisateur quelque part (la liste des posts) et le tour est joué ! ²⁶ Plus qu'à rajouter les fonctionnalités de modification et de suppression et puis nous en aurons terminé.

```

32  public function store(Request $request)
33  {
34      $this->validate($request, [
35          'titre' => 'required',
36          'message' => 'required'
37      ]);
38
39      $post = new Post;
40      $post->title = $request->input('titre');
41      $post->body = $request->input('message');
42      $post->save();
43
44      return redirect(route('posts.index'))->with('success', 'Post créé avec succès !');
45  }

```

25. Plus d'informations ainsi que la liste des règles de validation [ici](#).

26. La variable 'success' sera utilisée dans la Section 5

4.5.5 edit

Pour l'édition, c'est plutôt simple. Tout d'abord remplissons la méthode du controller :

```
60 public function edit(string $id)
61 {
62     $post = Post::find($id);
63
64     return view('posts.edit')->with('post', $post);
65 }
```

Ensuite, créons la view dédiée :

```
resources > views > posts > edit.blade.php > ...
1 @extends('layouts.app')
2
3 @section('content')
4
5 <div class="container">
6 <h1>Modification du post</h1>
7 <form action="{{route('posts.update', $post->id)}}" method="POST">
8 @csrf
9 @method('PUT')
10
11 <div class="form-group">
12 <label for="titre">Titre</label>
13 <input type="text" name="titre" id="titre" class="form-control" placeholder="Titre" value="{{old('titre', $post->title)}}">
14 </div>
15 <div class="form-group">
16 <label for="body">Message</label>
17 <textarea name="message" id="body" class="form-control" placeholder="Message"> {{old('message', $post->body)}} </textarea>
18 </div>
19 <br>
20 <button class="btn btn-primary">Modifier</button>
21 </form>
22 </div>
23
24 @endsection
```

Remarquez qu'il n'y a presque aucune différence avec la view de création de post, et c'est bien normal : on peut l'utiliser quasi à l'identique à condition de mettre les valeurs du post dans les bons champs lorsque l'on accède à la page. Une différence est néanmoins à noter, la présence du `@method('PUT')`. Cette méthode permet de remplacer une donnée par une autre (notre post à modifier), mais elle n'est pas disponible directement dans les forms. Il faut donc effectuer une petite acrobatie pour l'utiliser.

4.5.6 update

Lorsque le form est envoyé, il faut bien sûr enregistrer le post dans notre base de donnée. La route `'posts.update'` nous envoie donc vers la méthode `update()` que vous pouvez remplir comme ci-contre. Notez que ici aussi, la démarche est la même que pour la création de post (et c'est logique), mis à part qu'on cherche le post à modifier au lieu d'en créer un nouveau.

```
70 public function update(Request $request, string $id)
71 {
72     $this->validate($request, [
73         'titre' => 'required',
74         'message' => 'required'
75     ]);
76
77     $post = Post::find($id);
78     $post->title = $request->input('titre');
79     $post->body = $request->input('message');
80     $post->save();
81
82     return redirect(route('posts.show', $id))->with('success', 'Post modifié avec succès !');
83 }
```


4.5.7 delete

Enfin, la suppression de post. Tout d'abord, ajoutons un bouton dans `posts/show.blade.php` qui permet d'accéder à notre page de modification d'un post, un autre qui permettra de le supprimer, et stylisons un peu ces trois boutons afin qu'ils soient plaisant à voir (FIGURE 21)

Ensuite, remplissons la méthode du controller. La mécanique est ici triviale, il suffit de sélectionner le post que l'on souhaite et ensuite de le supprimer.

```
88 public function destroy(string $id)
89 {
90     $post = Post::find($id);
91     $post->delete();
92
93     return redirect(route('posts.index'))->with('success', 'Post supprimé avec succès !');
94 }
```

```
12 <small> Ecrit le {{$post->created_at}} </small>
13 <div class="d-flex justify-content-center mt-4">
14     <div class="mx-2">
15         <a href="{{route('posts.edit', $post->id)}}" class="btn btn-default bg-info">Modifier</a>
16     </div>
17     <div class="mx-2">
18         <form action="{{route('posts.destroy', $post->id)}}" method="POST">
19             @csrf
20             @method('DELETE')
21             <button class="btn btn-danger">Supprimer</button>
22         </form>
23     </div>
24     <div class="mx-2">
25         <a href="{{route('posts.index')}}" class="btn btn-default bg-info">Retour</a>
26     </div>
27 </div>
28 </div>
29
30 @endsection
```

FIGURE 21

Et voilà ! Nous avons désormais un système (simple) de création et gestion de posts ! Evidemment, nous pourrions rajouter énormément de fonctionnalités (commentaires, auteurs, images, ...). Nous explorerons certaines de ces options dans de futures sections, pour cette introduction, c'est déjà plus que suffisant.

A titre d'indication, voici le résultat auquel vous devriez arriver si tout fonctionne correctement :



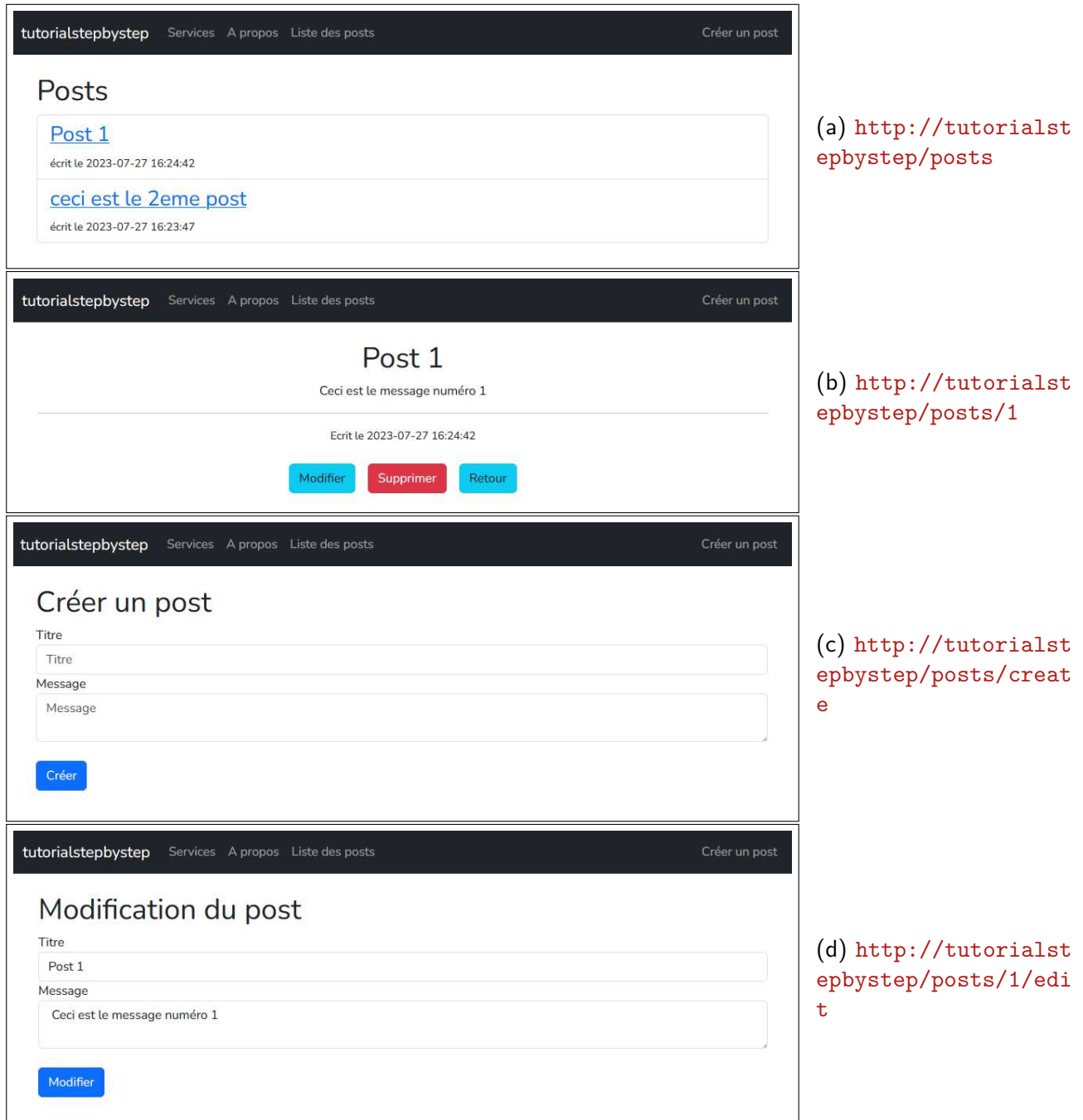


FIGURE 22 – Les quatre pages de gestions des posts.

5 Messages

Bon, c'est bien beau mais vous avez peut-être remarqué un gros inconvénient de notre système : lorsqu'un post est créé et qu'il manque un champ, rien ne nous dit clairement le problème ! De plus, il serait profitable d'avoir une confirmation lorsqu'un post est créé/modifié/supprimé. C'est exactement à cela que vont servir les variables 'success' que nous avons créées dans la section précédente,

sans les utiliser.

Créez donc un fichier `resources/views/inc/messages.blade.php` et remplissez le comme sur la figure de droite.

Ici, les deuxième et troisième `@if()` permettent de voir si la session (\Rightarrow la page actuelle, en gros) possède les variables 'success' et 'error', et de les afficher.

Ces variables sont celles que nous avons créées aux Section 4.5.4, 4.5.6, 4.5.7

Enfin, le premier `@if()` est utilisé afin d'afficher toutes les erreurs provenant d'un fail de validation après la soumission des `<form>`.

```
resources > views > inc > messages.blade.php > ...
1  @if(count($errors) > 0)
2      @foreach ($errors->all() as $error)
3          <div class="alert alert-danger">
4              {{$error}}
5          </div>
6      @endforeach
7  @endif
8
9  @if(session('success'))
10     <div class="alert alert-success">
11         {{session('success')}}
12     </div>
13 @endif
14
15 @if(session('error'))
16     <div class="alert alert-danger">
17         {{session('error')}}
18     </div>
19 @endif
```

Enfin, voici le résultat que vous devriez obtenir lorsque, par exemple, un post est créé (FIGURE 23A) ou lorsque le champ du message n'est pas rempli (FIGURE 23B) :

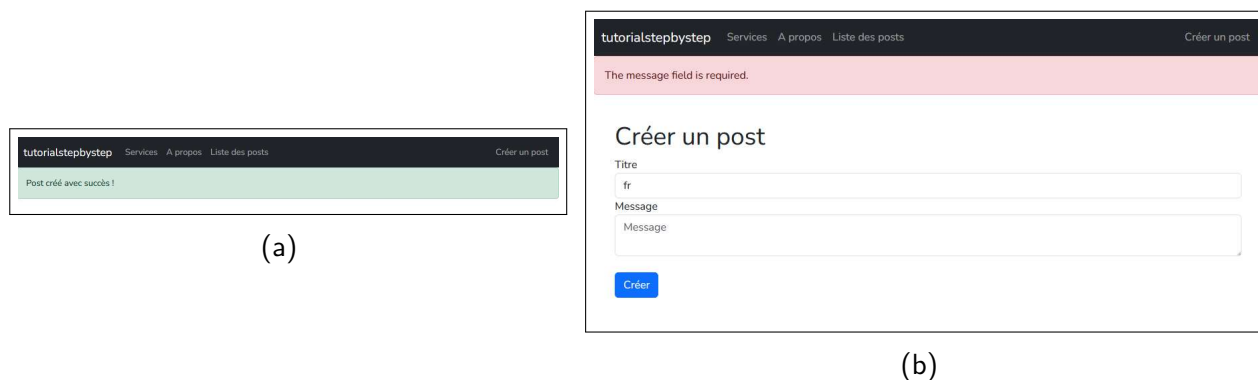


FIGURE 23


Vous pouvez avoir l'impression que nous en avons enfin fini avec cette formation, mais... il reste quelque chose qui devrait vous chiffrer : Les deux boutons login et register sur notre page d'accueil, ils ne servent toujours à rien.

Dans la Section 6, nous allons donc ajouter un système d'authentification très simple à notre site.

6 Authentification

6.1 Fortify

Pour cela, nous allons utiliser le package Fortify de Laravel .

"Laravel Fortify is a frontend agnostic authentication backend implementation for Laravel. Fortify registers the routes and controllers needed to implement all of Laravel's authentication features, including login, registration, password reset, email verification, and more". (Doc Laravel )

Cela veut donc dire qu'après configuration, nous n'aurons "plus qu'à" créer les views, et nous aurons finis! Vous l'aurez deviné, la difficulté viendra donc de cette configuration.


6.1.1 Installation

L'installation est plutôt rapide, ces deux commandes suffisent :

1. `sail composer require laravel/fortify`
2. `sail artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProvider"`

Si vous êtes observateurs, vous aurez remarqué que cette dernière commande aura créé une nouvelle migration, qu'il faut donc déployer. Exécutez donc `sail artisan migrate`.

6.1.2 Configuration

Premièrement, ajoutez cette ligne dans `config/app.php`, dans la liste des providers utilisés par Laravel .

```
183 Illuminate\View\ViewServiceProvider::class,
184 App\Providers\FortifyServiceProvider::class,
```

Ensuite, dans `config/fortify.php`, ajoutez `/* */` pour commenter les éléments comme ci-contre. L'array `features` permet de dire à Fortify quelles fonctionnalités vous souhaitez activer. En l'occurrence, nous sommes intéressés par `registration()` et `resetPasswords()`. Voici la liste des fonctionnalités :

```
134 'features' => [
135     Features::registration(),
136     Features::resetPasswords(), /*
137     // Features::emailVerification(),
138     Features::updateProfileInformation(),
139     Features::updatePasswords(),
140     Features::twoFactorAuthentication([
141         'confirm' => true,
142         'confirmPassword' => true,
143         // 'window' => 0,
144     ]), /*
145 ],
```

- `registration()` permet l'authentification de base (register/login).
- `resetPasswords()` permet de réinitialiser son mot de passe.
- `emailVerification()` permet de vérifier les adresses emails des utilisateurs.
- `updateProfileInformation()` permet de modifier son profil.
- `updatePassword()` permet de modifier son mot de passe depuis son profil.
- `twoFactorAuthentication` permet d'activer la double authentification.

Ensuite, il faut dire à Fortify quelles views correspondent à quelles routes. Pour cela, ajoutez ceci dans `app/Providers/FortifyServiceProvider.php`, plus précisément dans la méthode `boot()` :

```

46 Fortify::loginView(function () {
47     return view('auth.login');
48 });
49
50 Fortify::registerView(function () {
51     return view('auth.register');
52 });
53
54 Fortify::requestPasswordResetLinkView(function () {
55     return view('auth.password.forgot');
56 });
57
58 Fortify::resetPasswordView(function () {
59     return view('auth.password.reset');
60 });
61 }

```

6.1.3 Views

Ensuite, il faut évidemment créer les views pour les fonctionnalités que nous gardons. Comme elles sont (très) longues et proviennent d'un package qui n'est plus officiellement supporté (Laravel UI), vous pouvez les trouver ici : <https://github.com/nhitec/Laravel-tutorial-auth-blades>. Placez le dossier auth dans resources/views, et le tour est joué !²⁷

Afin d'accéder à ces nouvelles pages, il faut que l'on modifie nos boutons dans index.blade.php. Nous allons ajouter les routes aux deux boutons existants, et en rajouter un troisième pour se déconnecter :

```

5 <div class="jumbotron text-center">
6 <h1> Accueil </h1>
7 <p> Ceci est le squelette de base pour le tutoriel d'introduction à la création de site web en utilisant laravel v10 ! </p>
8
9 <div class="d-inline-flex">
10 <div class="mx-1">
11 <a href="{{ route('login') }}" class="btn btn-lg btn-primary">Se connecter</a>
12 </div>
13 <div class="mx-1">
14 <a href="{{ route('register') }}" class="btn btn-lg btn-success">S'enregistrer</a>
15 </div>
16 <div class="mx-1">
17 <form action="{{ route('logout') }}" method="POST">
18 @csrf
19 <button href="{{ route('logout') }}" class="btn btn-lg btn-danger">Logout</button>
20 </form>
21 </div>
22 </div>
23
24 @endsection

```

27. Remarquez que nous n'avons eu besoin ni de controllers ni de routes. C'est parce qu'ils sont déjà créés pour nous. Pour votre curiosité, vous pouvez trouver les routes créées par Fortify dans le fichier vendor/laravel/fortify/routes/routes.php.

Ceci est un document interne. Ne pas diffuser.

Voici les pages que vous devriez obtenir :

tutorialstepbystep
Services
A propos
Liste des posts
Créer un post

Register

Name

Email Address

Password

Confirm Password

Register

tutorialstepbystep
Services
A propos
Liste des posts
Créer un post

Login

Email Address

Password

☐ Remember Me

Login
[Forgot Your Password?](#)

tutorialstepbystep
Services
A propos
Liste des posts
Créer un post

Reset Password

Email Address

Send Password Reset Link

(a) <http://tutorialstepbystep/register>

(b) <http://tutorialstepbystep/login>

(c) <http://tutorialstepbystep/forgot-password>

FIGURE 24 – Les quatre pages de gestions des posts.



Remarquez que le bouton de la FIGURE 24C ne fonctionne pas, nous réglerons ce problème dans une future Section (WIP).

6.2 Middlewares

Nous arrivons à la fin ! La dernière chose à faire, c'est protéger quelques routes. En effet, il est préférable que n'importe qui ne puisse pas accéder aux posts sans s'être préalablement authentifié. Il faut donc trouver un moyen de vérifier si un utilisateur est connecté avant de lui donner accès à la gestion/création de posts. C'est exactement à cela que servent les middlewares.

De manière générale, les middlewares sont des fonctions qui seront exécutées "entre deux requêtes", pour par exemple interdire d'accès une certaine route et rediriger l'utilisateur selon certaines conditions.

Les middlewares se trouvent dans le dossier `app/Http/Middleware` et sont configurés par le fichier `app/Http/Kernel.php`. Les middlewares que nous pouvons assigner manuellement aux routes/controllers sont ceux listés dans `$middlewareAliases`.

En ce qui nous concerne, le middleware 'auth' est celui qu'il nous faut. Nous allons donc l'ajouter à aux routes responsables des posts dans `routes/web.php` comme cela :

```
23 Route::resource('posts', PostsController::class)->middleware('auth');
```

Et voilà ! Désormais, lorsque vous essayez de voir ou de créer un post sans être connecté, vous devriez vous faire rediriger vers la page de connection.



7 Conclusion

7.1 Postambule

Ce tutoriel touche (enfin) à sa fin ! Ce fût long et surement douloureux pour vous, mais vous y êtes arrivés, ce qui mérite déjà des félicitations en soi. Donc bravo !

Si vous avez toujours l'impression d'être un peu perdu, c'est tout à fait normal. Nous avons vu beaucoup de choses différentes, plusieurs langages différents, et des mécaniques qui s'entrecroisent. Ce n'est pas en suivant un tutoriel que vous deviendrez les maîtres de la programmation, et ce n'est pas l'objectif de ce tutoriel. Son objectif est de vous familiariser avec différentes notions afin de vous donner les clés nécessaires pour continuer de vous perfectionner et de comprendre les choses par vous mêmes. J'espère avoir accompli cet objectif !

7.2 Et ensuite ?

Il y a plusieurs suites possibles pour vous améliorer.

Premièrement, vous pouvez lire les documentations données au fil de tutoriel afin d'assimiler les notions et d'aller plus loin. Vous pouvez vous mêmes essayer d'ajouter des fonctionnalités à ce mini-site (ou créer le vôtre de A à Z) pour tester vos nouvelles compétences.

Ou alors, si vous continuez l'aventure N-HiTec avec nous, vous aurez accès à encore plus de ressources pour continuer de vous former et d'apprendre des choses avec nous, et si vous le souhaitez, vous pourrez même ajouter votre pierre à l'édifice en 1) contribuant à cette formation, and 2) travaillant sur les nombreux projets que nous avons à proposer !

N-HiTec vous attend !

