# Introduction

Pages statiques vs pages dynamiques :

- Page web statique : écrite directement en HTML et retourne le même contenu aux clients
- Page web dynamique : écrite avec un mélange de HTML et un langage permettant de générer un contenu dynamique

Exemple de langage de programmation web : PHP, Java Entreprise Edition, ASP, Perl, etc.

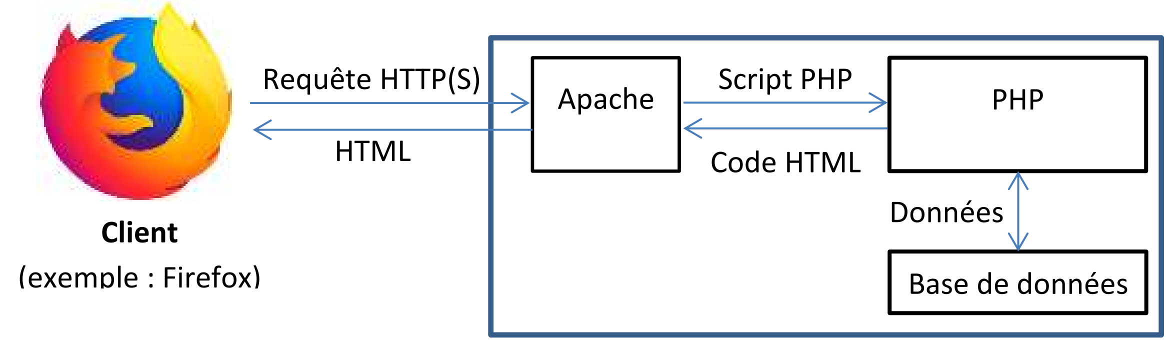
**Remarque** : Javascript est un langage qui s'exécute côté client

**Pages dynamiques et PHP** : Lors du chargement d'une page PHP, c'est le serveur qui va lire, interpréter et exécuter le code. Puis il renvoie le résultat, généralement sous la forme de code HTML au navigateur. Ainsi le navigateur et l'utilisateur ne voient jamais le véritable code PHP exécuté.

PHP est un langage de programmation permettant de créer des scripts PHP. Un script est exécuté côté serveur web pour générer un contenu HTML dynamique.

Exemple de serveurs web :

- WAMP : Windows Apache MySQL PHP
- LAMP : Linux Apache MySQL PHP




Framework : Permet de faciliter le développement des applications web

Exemples : Laravel, Symphony, Zend Framework, etc.

# Les pages WEB

La création d'une page web statique nécessite les langages suivants :

- **HTML** : Hypertext Markup Language (HTML) correspond au code que vous utiliserez pour structurer les différents contenus en ligne. Par exemple, le contenu sera-t-il un ensemble de paragraphes, ou une liste à puces ? Y aura-t-il des images insérées ? Aurai-je une table de données ? Exemple de balises HTML : https://www.w3schools.com/html/default.asp
- **CSS** : Le code Cascading Stylesheets (CSS) permet de mettre en forme votre site Web et améliorer son apparence. Par exemple, voulez-vous que le texte soit en noir ou en rouge ? Où le contenu doit-il être placé sur l'écran ? Quelles devront être les images de fond et les couleurs utilisées pour décorer votre site web ? Exemple de balises CSS : https://www.w3schools.com/css/default.asp
- **Javascript** : c'est le langage de programmation à utiliser pour ajouter des fonctionnalités interactives dans vos sites Web, par exemple des jeux, les événements déclenchés lorsqu'un bouton est pressé ou lorsque des données sont entrées dans un formulaire, des effets de style dynamiques, des animations, etc. Un code Javascript s'exécute côté client. Exemple de codes Javascript : https://www.w3schools.com/js/default.asp

Exemple de librairies (Framework) HTML/CSS/JS :

- **Bootstrap : https://getbootstrap.com/**
- Semantic UI : https://semantic-ui.com/
- Foundation : https://get.foundation/
- UIkit : https://getuikit.com/
- Skeleton : http://getskeleton.com/

## 1) HTML

HTML est un langage de balises qui définit la structure de votre contenu. HTML se compose d'une série d'éléments, utilisés pour entourer les diverses parties du contenu pour les faire apparaître d'une certaine façon.

Syntaxe d'un élément HTML :

```
<p class="class-css">This is a paragraph with one line.</p>
```

Les composants de l'élément précédent sont :

- La balise ouvrante : <p>
- La balise fermante : </p>
- Le contenu : « This is a paragraph with one line. »
- Les attributs : « class="class-css" »

Exemple d’une page HTML :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>

    <h1>This is a Heading</h1>
    <p>This is a paragraph.</p>

  </body>
</html>
```

## 2) CSS

Le code CSS peut être placé dans les 3 emplacements suivants :

- Dans un fichier indépendant
- Dans un fichier HTML entouré dans l’élément « **style** »
- Dans l’attribut « style » d’un élément HTML

**Exemple 1 : code CSS dans un fichier indépendant**

Fichier mystyle.css

```
h1 {
  color: blue;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
```

Fichier index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
    <link rel="stylesheet" href="assets/css/mystyle.css">
  </head>
  <body>
    <h1>Le code CSS</h1>
    <p>Comment utiliser les fichiers CSS dans une page HTML</p>
  </body>
</html>
```

**Exemple 2 : code CSS dans un fichier HTML entouré dans l'élément « style »**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
    <style>
h1 {
  color: blue;
  text-align: center;
}
p {
  font-family: verdana;
  font-size: 20px;
}
    </style>
  </head>
  <body>
    <h1>Le code CSS</h1>
    <p>Comment utiliser le code CSS dans une page HTML</p>
  </body>
</html>
```

**Exemple 3 : code CSS dans un l'attribut « style » d'un élément HTML**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
  </head>
  <body>
    <h1 style="color: blue; text-align: center;">Le code CSS</h1>
    <p>Comment utiliser le code CSS dans l'attribut d'un élément HTML</p>
  </body>
</html>
```

## 3) Javascript

Le code Javascript peut être placé dans les 3 emplacements suivants :

- Dans un fichier indépendant
- Dans un fichier HTML entouré dans l'élément « **script** »
- Dans un attribut d'un élément HTML

**Exemple 1 : code Javascript dans un fichier indépendant**

Fichier myscript.js

```
function myFunction() {
  document.getElementById("demo").innerHTML = Date();
}
```

Fichier index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
    <script src="assets/js/myscript.js"></script>
  </head>
  <body>
    <h1>Le code JavaScript </h1>
    <button type="button" onclick="myFunction()">Click to show Time.</button>
    <p id="demo"></p>
  </body>
</html>
```

**Exemple 2 : code Javascript dans un fichier HTML entouré dans l'élément « script »**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
    <script>
function myFunction() {
  document.getElementById("demo").innerHTML = Date();
}
    </script>
  </head>
  <body>
    <h1>Le code JavaScript </h1>
    <button type="button" onclick="myFunction()">Click to show Time.</button>
    <p id="demo"></p>
  </body>
</html>
```

**Exemple 3 : code CSS dans un attribut d'un élément HTML**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>
  </head>
  <body>
    <h1>Le code JavaScript </h1>
    <button type="button" onclick="document.getElementById("demo").innerHTML=Date();">
        Click to show Time.
    </button>
    <p id="demo"></p>
  </body>
</html>
```

## 4) Bootstrap

Bootstrap permet de créer des sites web rapides et réactifs (responsive) qui s'adaptent avec la taille de l'écran du client.

Bootstrap est un framework open source pour le développement web front-end. Il fournit une librairie CSS et JavaScript riche et puissante.

Quelques versions de Bootstrap :

- Bootstrap 5.1 (dernière version) : https://getbootstrap.com/docs/5.1/
- Bootstrap 4 :
  - https://getbootstrap.com/docs/4.6/
  - https://www.w3schools.com/bootstrap4/default.asp
- Bootstrap 3 :
  - https://getbootstrap.com/docs/3.4/
  - https://www.w3schools.com/bootstrap/default.asp

Il y a deux méthodes pour inclure Bootstrap (CSS et Javascript) dans une page HTML :

- Télécharger Bootstrap et inclure les fichiers téléchargés
- Inclure les fichiers Bootstrap à partir des liens externes

Exemple d'inclusion de Bootstrap 5 à partir de liens externes :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Développement WEB</title>

    <!-- Bootstrap core CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/
bootstrap.min.css">

    <!-- Bootstrap javascript -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/
bootstrap.bundle.min.js"></script>

  </head>
  <body>
    <h1>Inclure les fichiers Bootstrap à partir des liens externes</h1>
  </body>
</html>
```

# Le langage PHP

Une page php porte l'extension « .php ». Une page PHP peut être entièrement programmée en PHP ou mélangée avec du code html. PHP est un langage qui apparaît à n'importe quel endroit de la page HTML. Pour ça on le place dans des balises particulières : **<?php** et **?>**. On écrit donc une page HTML dans laquelle on intègre du code PHP.

Fichier PHP :

```
<html>
<head>
<title>Titre</title>
</head>
<body>
<?php
echo "Hello World !";
?>
</body>
</html>
```

Le code HTML généré sera le suivant :

```
<html>
<head>
<title>Titre</title>
</head>
<body>
Hello World !
</body>
</html>
```

L'utilisation de balises pour l'intégration de code dans une page web est très souple et permet de mélanger facilement du code PHP avec du code HTML :

```
<?php
if ( $expression ) {
?>
<strong>Ceci est vrai.</strong>
<?php
} else {
?>
<strong>Ceci est faux.</strong>
<?php
}
?>
```

## Déclarer une variable

Une variable commence par un dollar « **$** » suivi d'un nom de variable. Les variables ne sont pas typées au moment de leur création. Attention **PHP est sensible à la casse** ($var et $Var sont deux variables différentes). Les règles à respecter :

- Une variable peut commencer par une lettre
- Une variable peut commencer par un tiret bas (underscore) « _ »
- Une variable ne doit pas commencer par un chiffre.

```
// Déclaration et règles
$var=1; // $var est à 1
$Var=2; // $Var est à 2
$_toto='Salut'; // Ok
$3petits=5; // Invalide : commence par un chiffre
```

Le type d'une variable peut changer à tout moment :

```
// Déclaration et transtypage
$var='2'; // Une chaîne 2
$var+=1; // $var est maintenant un entier 3
$var=$var+0.3; // $var est maintenant un réel de type double 3.3
$var=5 + "3 mots"; // $var est un entier qui vaut 8
```

## Les fonctions

On peut passer 0 ou plusieurs arguments de différents types à une fonction. Une fonction peut ne pas retourner un résultat, ou retourner n'importe quel type de résultat (variable simple, tableau, objet, etc.) avec le mot clé « **return** ». Une fonction a la syntaxe suivante :

```
function fonction_php ($arg1, $arg2, ..., $argn) {
	// instructions

	return $valeur;
}
```

Exemple :

```
function somme ($arg1, $arg2) {
	$valeur = $arg1 + $arg2;

	return $valeur;
}

$s = somme(4, 2);
echo $s; // affiche 6
```

Par défaut, les variables sont passées par copie (la modification de la valeur d'un argument à l'intérieur de la fonction ne change pas la valeur de la variable passée en argument). Exemple :

```
function fonction1 ($var) {
	$var += 2;
}

$age = 20;
fonction1($age);
echo $age; // affiche 20 et non pas 22
```

Il est possible de passer des arguments par référence. On peut donc changer leurs valeurs à l'intérieur de la fonction. Pour passer des arguments par référence, il faut ajouter « & » devant l'argument.
Exemple :

```
function fonction1 (&$var) {
	$var += 2;
}

$age = 20;
fonction1($age);
echo $age; // affiche 22
```

## Portée des variables

La portée d'une variable déclarée dans un script et hors d'une fonction est globale mais par défaut sa portée est limitée au script (fichier) courant, ainsi qu'aux scripts éventuellement inclus (**include**, **require**). Elle n'est pas accessible dans les fonctions ou d'autres scripts non-inclus.

Exemple :

```
$var = 5;
function ma_fonction () {
	echo $var; // erreur : Undefined variable
}

ma_fonction();
```

Pour accéder à une variable globale dans une fonction, il faut utiliser le mot-clé « **global** ».

```
$var = 5;
function ma_fonction2() {
	global $var; // on récupère la variable globale
	$var *= 2;
}

ma_fonction2();
echo $var; // affiche 10
```

Une variable définit à l'intérieur d'une fonction est une variable locale, et ne peut pas être utilisée en dehors de la fonction.

## Types de variables

### Booléens

Un booléen peut prendre deux valeurs **TRUE** ou **FALSE**. Les deux constantes **TRUE** et **FALSE** peuvent être utilisées sans aucune distinction de casse (pas de différences entre les majuscules et les minuscules).

```
$var=FALSE; // FALSE, False, false, ...
$var2=True; // TRUE, True, true, ...
```

### Chaînes de caractères

Une chaîne de caractères peut être entourée par des guillemets simples '...' ou par des guillemets doubles "...". Parmi les séquences qu'on peut utiliser avec les guillemets doubles on trouve :

| \n | Nouvelle ligne |
|---|---|
| \t | Tabulation |
| \\ | Antislash |
| \" | Guillemets doubles |

Avec les guillemets doubles, n'importe quelle variable peut être affichée dans une chaîne. Exemple :

```
$age = 20;
//affiche l'age de cet utilisateur est 20 ans
echo "l'age de cet utilisateur est $age ans";
```

On peut concaténer deux chaînes avec l'opérateur point « . ». On peut ajouter du texte à une chaîne avec l'opérateur point égal « .= ». Exemple :

```
$nom = "Ali";
$str = "Le nom ";
$str .= "de cet utilisateur est ";
$str2 = $str . $nom;
echo $str2; //Le nom de cet utilisateur est Ali
```

### Les tableaux

Un tableau PHP est une association ordonnée qui fait correspondre des valeurs à des clés. Les tableaux sont très souples et peuvent contenir différents types de valeurs simultanément. Une valeur d'un tableau peut être elle-même un tableau.

Un tableau est créé avec la fonction **array()** qui prend comme arguments des paires « key => value » séparées par des virgules. La clé peut être soit un entier soit du texte. Si la clé est absente alors c'est la dernière clé entière plus 1 qui est choisie. Si c'est la première, c'est 0 zéro.

On accède aux éléments d'un tableau à l'aide des crochets « [ et ] ». On place entre ces crochets la clé entière ou la chaîne.

```php
$tab=array("a"=>12, "nom"=>"ali", "client", 17, 4=>5, "validation");
//$tab["a"] => 12
//$tab["nom"] => ali
//$tab[0] => client
//$tab[1] => 17
//$tab[4] => 5
//$tab[5] => validation
```

L'utilisation de la fonction **array()** n'est pas obligatoire et on peut déclarer un tableau à la volée.

Exemple :

```php
$tab2[1]=2;
$tab2[]=6; // equivaut $tab2[2]=6
$tab2["test"]='Ma chaîne';
//$tab2[1] => 2
//$tab2[2] => 6
//$tab2['test'] => Ma chaîne
```

On peut utiliser la fonction **foreach()** pour afficher et/ou modifier le contenu d'un tableau. Exemple :

```php
$tab = array(1 => 'un', 2 => 'deux', 3 => 'trois');
foreach($tab as $valeur) {
     echo "$valeur <br>"; // affiche un deux trois
}
foreach($tab as $cle => $valeur) {
     echo "$cle => $valeur <br>"; // 1 => un, 2 => deux, 3 => trois
}
```

## Inclusion des fichiers

L'inclusion d'un fichier (php ou non) à partir d'un autre fichier php est possible en utilisant les instructions suivantes :

- **include** : inclut et exécute le fichier spécifié en argument. Au cas d'échec, elle génère une alerte (E_WARNING) et termine l'exécution du programme.
- **require** : inclut et exécute le fichier spécifié en argument. Au cas d'échec, elle génère une erreur (E_COMPILE_ERROR) et arrête le programme.
- **include_once** : similaire à « **include** » mais inclut le fichier une seule fois.
- **require_once** : similaire à « **require** » mais inclut le fichier une seule fois.

L'inclusion des fichiers est très utile pour

- utiliser le même template avec plusieurs pages sans répéter le code HTML dans chaque fichier
- utiliser les fonctions et les classes définies dans d'autres fichiers

Exemple :

```
<?php
include "head.php";
?>

<p>Contenu de page</p>

<?php
include "foot.php";
?>
```

Corriger l'erreur dans l'exemple suivant :

<table>
<tr>
<td>Fichier head.php :

<pre>
<html>
<head>
<title>
<?php
echo $title;
?>
</title>
</head>
<body>
</pre>

Fichier foot.php :

<pre>
</body>
</html>
</pre>
</td>
<td>Fichier accueil.php :

<pre>
<?php
include "header.php";
?>

<p>Contenu de page</p>

<?php
include "footer.php";
?>
</pre>
</td>
</tr>
</table>

## La programmation objet

Une classe est un ensemble de variables et de fonctions (méthodes). Cet ensemble forme les membres ou les propriétés de l'objet. Une classe est définie en utilisant le mot-clé « **class** » :

```
class Personne {
	//déclaration des variables et des fonctions
}
```

La visibilité d'une variable ou d'une fonction indique à partir d'où on peut y accéder. Il y a 3 types de visibilité : **public**, **protected** et **private**.

| Accès | public | protected | private |
| --- | --- | --- | --- |
| Dans la classe elle-même | oui | oui | oui |
| Depuis une classe dérivée | oui | oui | × |
| Depuis l'extérieur | oui | × | × |

Une variable définie dans la classe est une variable globale, et doit commencer par un mot-clé de visibilité (**public**, **protected** et **private**). Elle peut commencer aussi par le mot-clé « **var** » et dans ce cas elle est considérée comme publique.

Il est possible d'initialiser les variables au moment de la déclaration. Pour accéder à une variable globale à l'intérieur d'une classe, on utilise le mot-clé « this » et « -> ». Dans ce cas, on place « $ » devant « this » et non pas devant la variable.

```
class Etudiant {
	private $nom;
	private $prenom;
	private $formation = 'ISI1';

	public function setNomPrenom($nom, $prenom) {
		$this->nom = $nom;
		$this->prenom = $prenom;
	}
	public function setFormation($formation) {
		$this->formation = $formation;
	}
	public function getInfo() {
		$info = "nom = " . $this->nom .
			", prenom = " . $this->prenom .
			", formation = " . $this->formation;
		return $info;
	}
}
```

**Création d'un objet**

Pour créer un objet on utilise le mot-clé « **new** ».

```
$et = new Etudiant;
$et->setNomPrenom("Abid", "Ahmed");
echo $et->getInfo(); //nom = Abid, prenom = Ahmed, formation = ISI1
```

**Constructeur**

Le constructeur en PHP est une fonction qui porte le même nom que la classe. Si aucun constructeur n'est définit, c'est le constructeur par défaut qui est appelé.

```
class Personne {
	private $nom;

	public function Personne($nom) {
		$this->nom = $nom;
	}
}

$p = new Personne("ali");
```

La fonction « __construct() » permet aussi de définir le constructeur de la classe (voir : https://www.w3schools.com/php/php_oop_constructor.asp ). Cette fonction commence par deux underscores.

**Namespace**

L'espace de nommage permet d'organiser le code source et de créer des classes dans des espaces de nommage bien définies.

La déclaration d'un espace de nommage doit être la première instruction du fichier php. Exemple :

```
namespace controller;
```

Utilisation de l'espace de nommage :

Fichier home.php :

```
<html>
<head>
<title>
<?php
include 'User.php';
use project1\User;
$user = new User;
$title = $user->getTitle();
echo $title;
?>
</title>
</head>
<body>
<p>Utilisation des Namespaces</p>
</body>
</html>
```

```
<?php
namespace project1;

class User {
	private $title = 'ISI1';

	public function getTitle() {
		return $ title;
	}
}

?>
```

# Framework Laravel

## Introduction

Un framework est un ensemble de composants logiciel (classes, fonctions, etc.) prêt à l'emploi. L'utilité d'un framework est d'éviter de passer du temps à développer ce qui a déjà été fait. Par exemple, pour vérifier si les données d'un formulaire sont saisies correctement, on utilise un composant de validation existant. Egalement, pour faire le routage dans une application web, on prend un composant de routage prêt et on l'utilise.

Laravel est un Framework PHP « open source » qui a fait son apparition en 2011 et qui évolue rapidement. Il fait le regroupement de plusieurs framework existants et définit aussi des nouveaux composants. Par exemple, Laravel utilise plusieurs fonctionnalités de Symphony, SwiftMailer et d'autres librairies.

Les différentes versions de Laravel sont (source : en.wikipedia.org/wiki/Laravel) :

| Version | Date d'apparition | Version minimale nécessaire de PHP |
|---|---|---|
| 1.0 | Juin 2011 | |
| 2.0 | Septembre 2011 | |
| 3.0 | Février 2012 | |
| 3.1 | Mars 2012 | |
| 3.2 | Mai 2012 | |
| 4.0 | Mai 2013 | ≥ 5.3.0 |
| 4.1 | Décembre 2013 | ≥ 5.3.0 |
| 4.2 | Juin 2014 | ≥ 5.4.0 |
| 5.0 | Février 2015 | ≥ 5.4.0 |
| 5.1 | Juin 2015 | ≥ 5.5.9 |
| 5.2 | Décembre 2015 | ≥ 5.5.9 |
| 5.3 | Aout 2016 | ≥ 5.6.4 |
| **5.4** | **Janvier 2017** | **≥ 5.6.4** |
| 5.5 | Aout 2017 | |
| 5.6 | Février 2018 | ≥ 7.1.3 |
| 5.7 | Septembre 2018 | ≥ 7.1.3 |
| 5.8 | Février 2019 | ≥ 7.1.3 |
| 6 | 3 Septembre 2019 | 7.2.0 – 8.0 |
| 7 | 3 Mars 2020 | 7.2.5 – 8.0 |
| 8 | 8 Septembre 2020 | |
| 9 | 8 Février 2022 | 8.0 – 8.1 |
| 10 | 7 Février 2023 | 8.0 – 8.1 |

## Installation

Il y a plusieurs versions de Laravel, et chaque version a ses propres besoins. Par exemple, la version 5.4 nécessite au minimum la version PHP 5.6.4. Pour trouver les différents besoins d'une version en particulier de Laravel, il faut consulter le site web de Laravel accessible sur l'adresse suivante :

**laravel.com**

Ensuite, aller sur « Documentation » et sélectionner la version de Laravel (exemple version « 5.5 »)

**Question :** trouver la version PHP minimale nécessaire pour Laravel 5.5 et pour Laravel 9.x

- ⇨ **Il faut aller à « Getting Started > Installation » dans le cas de Laravel 5.5**
- ⇨ **Il faut aller à « Prologue > Release Notes » dans le cas de Laravel 8.x**

Il est possible d'utiliser Laravel avec plusieurs serveurs (WAMP, XAMPP, etc.). Laravel utilise des composants provenant d'autres frameworks. Ces composants ne sont pas incorporés directement dans Laravel et doivent être téléchargés. La méthode classique consiste à télécharger Laravel et tous les frameworks dont il a besoin (tout en vérifiant la compatibilité des versions). Cette méthode n'est pas pratique !

- ⇨ Solution : on utilise l'outil « **Composer** »

« **Composer** » est un gestionnaire de dépendances qui permet de télécharger Laravel et toutes ses dépendances. Il permet aussi de télécharger d'autres projets de tous types. Il est possible de télécharger cet outil à partir de l'url suivante : **getcomposer.org**

Lors de l'installation de « **Composer** », il faut lui indiquer le chemin vers « **php.exe** »

#### Création d'une nouvelle application Laravel

Pour créer une nouvelle application Laravel nommée « **forum** » dans le dossier « **www** » de WampServer (ou dans le dossier « **htdocs** » de XAMPP), en utilisant Composer, il faut :

1) Lancer un terminal
2) Se déplacer dans le dossier « **www** » de WampServer (ou dans le dossier « **htdocs** » de XAMPP)
3) Taper la commande suivante :

```
composer create-project laravel/laravel forum-app
```

**Remarque** : La commande précédente installe la dernière version de Laravel qui soit compatible avec la version installée de PHP, dans le dossier « `www/forum` » (ou « `htdocs/forum` »).

Pour installer une version en particulier de Laravel en utilisant Composer, on ajoute la version en paramètre. L'exemple suivant permet d'installer Laravel 8.x dans le dossier « blog » :

```
composer create-project laravel/laravel forum-app "8.*"
```

**Exercice** : Créer une nouvelle application Laravel 9.x nommée « **laravel-app** » sous le dossier « **www** » de wamp (ou « **htdocs** » de xampp).

## Organisation d'une application Laravel

L'organisation par défaut d'une application peut changer selon la version Laravel. Cette organisation a le but de simplifier le développement d'une application web, mais il est possible de la modifier selon le besoin.

**Cas de Laravel 9.x** (voir : laravel.com/docs/9.x/structure)

- **Dossier « routes »** : contient toutes les routes vers les différentes pages de l'application. Ce dossier contient par défaut les fichiers « web.php », « api.php », « console.php » et « channels.php ». Pour une application simple, les différentes routes sont généralement définies dans le fichier **« web.php »**.
- **Dossier « resources »** : contient les 3 dossiers suivants :
  - **views** : contient les différentes vues (pages web)
  - **lang** : contient les fichiers de langues
  - **assets** : contient des fichiers js, etc.
- **Dossier « public »** : contient le fichier « **index.php** » qui est le point d'entrée pour toutes les requêtes venant à l'application. Ce dossier peut contenir des images, des fichiers js et css, etc.
  - Quelle est l'utilité du fichier « .htaccess » ?

**Questions :**

1) Quelle est la différence entre la structure de Laravel 5.5 et celle de Laravel 9.x ?
2) Est-ce que le contenu du dossier « **routes** » change dans la version 9.x ?
3) Est-ce que le contenu du dossier « **resources** » change dans la version 9.x ?

## Facades et Helpers

Voir : « **laravel.com/docs/9.x/facades** » et « **laravel.com/docs/9.x/helpers** »

Laravel propose plusieurs façades (alias) pour simplifier la programmation de l'application.

L'usage d'une façade est insensible à la casse, et les exemples suivants sont corrects et identique :

```
return view::make('accueil');
return View::make('accueil');
return VIEW::make('accueil');
```

Pour utiliser des nouvelles façades, il faut les déclarer dans « **config/app.php** » sous la clé « **aliases** ». Par exemple, après l'ajout du composant « html » on peut ajouter les deux façades suivantes :

```
    'aliases' => [

        ...
        'Form' => Collective\Html\FormFacade::class,
        'Html' => Collective\Html\HtmlFacade::class,
        ...

    ],
```

Laravel définit aussi les « **helpers** » qui permettent de simplifier davantage la syntaxe. Plusieurs « **helpers** » réalisent les mêmes fonctionnalités des façades mais avec une syntaxe simplifiée. Par exemple, les deux instructions suivantes sont identiques :

| Retourner une vue en utilisant une **façade** : | Retourner une vue en utilisant un **helper** : |
|---|---|
| `return view::make('accueil');` | `return view('accueil');` |

**Remarque** : tout comme les façades, les **helpers** sont aussi insensibles à la casse !

## Les routes

Voir « **laravel.com/docs/9.x/routing** »

Lorsque la requête arrive au fichier « **public/index.php** », l'application Laravel est créée et configurée. Ensuite, les fichiers du dossier « **routes** » (y compris « **web.php** ») sont chargés. Le fichier « **web.php** » définit les routes vers les différentes pages web.

A l'installation de Laravel, il y a une seule route dans Le fichier « **web.php** » qui correspond à l'url de base :

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

Si :

- la requête est reçu avec la méthode « get » avec ou sans paramètres (`Route::get`), et
- la requête est envoyée à l'url de base (/)

Alors : retourner la vue « **welcome** » qui se trouve sous « **resources/views** » dans un fichier nommé « **welcome.blade.php** » ou « **welcome.php** »

Une route en Laravel accepte un URI et une fonction permettant de produire une réponse. Les différentes routes sont définies dans le dossier « **routes** », et principalement dans le fichier « **web.php** » de ce dossier. Le routeur permet d'enregistrer des routes qui répondent aux différentes méthodes HTML (GET, POST, etc.). Les fonctions du routeur sont :

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

**Exemples d'URI :**

- `'/'` : URI racine de l'application (emplacement du fichier « **index.php** » de Laravel). Exemple : « `www.nom-site.com` », « `localhost` » et « `localhost/forum/public` ».
- `'/home'` ou `'home'` : chemin relatif à ajouter à l'URI racine. Exemple : « `www.nom-site.com/home` », « `localhost/home` », « `localhost/forum/public/home` ».
- `'/home/account'` ou `'home/account'`

**Exercice :**

1) Dans l'application « **laravel-app** », ajouter une route pour l'url « **route1** » accessible par la méthode GET dans fichier « **web.php** » permettant d'afficher le texte « **nouvelle route** ».
2) Créer la vue « **accueil.php** » sous « **resources/views** » avec le contenu suivant :

```
<!doctype html>
<html>
    <body>
        <h1>bonjour à tous</h1>
        <p>Il est <?php echo date("H:i:s") ?></p>
    </body>
</html>
```

3) Ajouter une route pour l'url « **route2** » accessible par la méthode GET dans fichier « **web.php** » permettant d'afficher la vue « **accueil.php** »
4) Ajouter une route pour l'url « **route3** » accessible par la méthode POST dans fichier « **web.php** » permettant d'afficher le texte « **troisième route** ». Est-il possible d'accéder à cette route ?

Pour enregistrer une route capable de répondre à un ensemble de méthodes HTML, on utilise la fonction « **match** ». On peut aussi utiliser la fonction « **any** » pour répondre à toutes les méthodes HTML. Exemple :

```
Route::match(['get', 'post'], 'add', function () {
    return "Ajouter"; //afficher "Ajouter"
});

Route::any('cart', function () {
    return "Panier"; //afficher "Ajouter"
});
```

**Protection CSRF :**

Tout formulaire HTML utilisant la méthode HTML « post » doit inclure un jeton CSRF. Sinon, la requête sera rejetée (pour plus d'information sur la protection CSRF, voir lien : « **laravel.com/docs/9.x/csrf** »). La fonction PHP « **`csrf_field()`** » permet de retourner une chaine de caractères qui représente un champ contenant le jeton CSRF. On peut ajouter ce champ dans un formulaire HTML.

Exemple de résultat de « **`csrf_field()`** » :

```
<input type="hidden" name="_token" value="eQLnTzhL3c9aX0PjNy27xY3Kcay0ufgxWUOgkhU4">
```

Exemple d'usage de « **`csrf_field()`** » :

```
<form method="POST" action="/add">
    <?php echo csrf_field() ?>
    ...
</form>
```

## Les vues

Voir lien : « **laravel.com/docs/9.x/views** »

Les vues contiennent le code HTML et sont sauvegardées dans le dossier « **resources/views** ». Pour retourner une vue, on peut utiliser le helper « **view** » ou bien les méthodes de la façade « **view** ».

Exemple « **resources/views/accueil.php** » :

```
<!doctype html>
<html>
    <body>
        <h1>bonjour <?php echo $name ?></h1>
        <p>Il est <?php echo $date ?></p>
    </body>
</html>
```

La vue « **accueil.php** » nécessite les deux variables « **$name** » et « **$date** » qui ne sont pas initialisées dans le fichier. Retourner cette vue avec le code suivant génère des erreurs (**Undefined variable: name/date**) :

```
Route::get('accueil', function () {
    return view('accueil');
});
```

Pour corriger ces erreurs, il faut passer les variables « **$name** » et « **$date** » dans un tableau indexé (les clés représentent les noms des variables) à la vue. Ce tableau est le 2eme paramètre du helper « **view** » (le premier paramètre est la vue).

Exemple 1 :

```
Route::get('accueil', function () {
    $n = "Ali";
    $d = date("H:i:s");
    $data = ['name' => $n, 'date' => $d];
    return view('accueil', $data);
});
```

Exemple 2 :

```
Route::get('accueil', function () {
   return view('accueil', ['name' => "Ali", 'date' => date("H:i:s")]);
});
```

En général, il est possible de passer un nombre indéfini de variables à une vue en utilisant un tableau indexé. Ensuite, la vue peut utiliser ces variables.

Les vues peuvent être organisées dans des dossiers à l'intérieur de « **resources/views** ». Pour accéder à ces vues avec le helper « **view** », il faut lui donner le chemin d'une vue en séparant les noms des dossiers et de la vue avec des points.

Exemple :

```
//retourner la vue resources/views/admin/profile.php
return view('admin.profile');
```

Pour vérifier l'existence d'une vue, on peut utiliser la méthode « **exists** » de la façade « **view** » comme suit :

```
if (View::exists('emails.customer')) {
    //
}
```

## Les contrôleurs

Voir le lien « **laravel.com/docs/9.x/controllers** »

Au lieu de définir toutes les instructions dans le fichier « **routes/web.php** » dans une fonction de rappel, il est possible d'utiliser des contrôleurs. Les contrôleurs sont stockés sous « **app/Http/Controllers** ». L'exemple suivant est un contrôleur nommé « **HomeController** » définit dans le fichier « **HomeController.php** » du dossier « **app/Http/Controllers** » :

```
<?php

namespace App\Http\Controllers;

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

class HomeController extends Controller
{
    public function showHome(Request $request)
    {
        $n = "User";
        $d = date("H:i:s");
        $data = ['name' => $n, 'date' => $d];
        return view('accueil', $data);
    }

    public function showInfo()
    {
        $info = "Laravel est un Framework PHP simple et puissant. ";
        return $info;
    }
}
```

Après la création de la classe « **HomeController** », on peut définir des routes vers les méthodes de ce contrôleur dans le fichier « **routes/web.php** » comme suit :

```
use App\Http\Controllers\HomeController;

Route::get('/ctrl/home', [HomeController::class, 'showHome']);
Route::get('/ctrl/info', [HomeController::class, 'showInfo']);
```

Au lieu d'importer le contrôleur avec l'instruction « **use** », il est possible de créer une route et lui fournir le chemin complet du contrôleur.

Exemple:

```
Route::get('/ctrl/home', [App\Http\Controllers\HomeController::class, 'showHome']);
Route::get('/ctrl/info', [App\Http\Controllers\HomeController::class, 'showInfo']);
```

**Injection de dépendance**

Le « service container » de Laravel est utilisé pour résoudre tous les contrôleurs. Il est possible d'indiquer tous les paramètres utilisés par les méthodes et les constructeurs des contrôleurs. Ces paramètres seront automatiquement injectés dans les méthodes et les constructeurs. Un usage typique de l'injection de dépendance consiste à injecter l'objet « **Illuminate\Http\Request** » dans les méthodes du contrôleur (voir la méthode showHome() de la classe HomeController).

**Exercice :**

1) Dans l'application « **laravel-app** », créer le contrôleur « **ContactController** » avec les deux méthodes suivantes :
    a. **showView** : permet de retourner la vue « **contact.php** » (disponible dans les templates)
    b. **addContact** : permet de lire les données du formulaire de contact à partir du tableau **$_POST** et de retourner la vue « **confirm.php** » suivante :

```
<!doctype html>
<html>
    <body>
        <h1>Nous avons bien reçu votre demande :</h1>
        <p>Nom : <?php echo $name ?></p>
        <p>Email : <?php echo $email ?></p>
        <p>Téléphone : <?php echo $phone ?></p>
        <p>Sujet : <?php echo $subject ?></p>
        <p>Message : <?php echo $message ?></p>
    </body>
</html>
```

2) Ajouter une route pour l'url « **contact** » accessible par la méthode GET dans fichier « **web.php** » permettant d'appeler la méthode « **showView** » du contrôleur.
3) Ajouter une route pour l'url « **contact** » accessible par la méthode POST dans fichier « **web.php** » permettant d'appeler la méthode « **addContact** » du contrôleur. Tester l'envoi d'un formulaire.
4) Ajouter la variable « **`Request $request`** » en paramètre de la méthode « **addContact** » et utiliser cette variable à la place de **$_POST** pour lire les données du formulaire. Exemple :

```
//lien : https://laravel.com/docs/9.x/requests
$name = $request->input('name');
```

## Blade

Voir lien : « **laravel.com/docs/9.x/blade** »

Blade est un moteur de template qui simplifier la syntaxe. Pour utiliser la syntaxe Blade dans une vue, il faut que la vue soit sauvegardée dans un fichier « **blade** » : le nom de ce fichier se termine par « **.blade.php** ». La syntaxe Blade est incompréhensible dans un fichier PHP classique qui n'est pas un fichier « **blade** ». Il est possible d'utiliser les balises PHP classiques dans un fichier « **blade** ».

**Remarque** : Les fichiers « **blade** » sont prioritaires par le helper/façade « **view** ». Par exemple, si les deux fichiers « **accueil.php** » et « **accueil.blade.php** » se trouvent dans le même dossier, l'instruction suivante permet de retourner le fichier « **blade** » (**accueil.blade.php**) :

```
return view('accueil');
```

Lorsqu'on veut insérer une donnée avec PHP on utilise cette syntaxe :

```
<?php echo $valeur; ?>
```

Avec Blade on a cette syntaxe :

```
{{ $valeur }}
```

Autre exemple (Protection CSRF) :

<table>
<tr><th>Syntaxe PHP</th><th>Syntaxe Blade</th></tr>
<tr><td><pre><form method="POST" action="/add">
    <?php echo csrf_field() ?>
    ...
</form></pre></td><td><pre><form method="POST" action="/add">
    {{ csrf_field() }}
    ...
</form></pre></td></tr>
</table>

**Les structures de contrôle avec Blade**

**foreach**

<table>
<tr><th>Syntaxe PHP</th><th>Syntaxe Blade</th></tr>
<tr><td><pre><?php
foreach($data as $element) {
    echo $element.'<br>';
}
?></pre></td><td><pre>@foreach ($data as $element)
    {{ $element }}<br>
@endforeach</pre></td></tr>
</table>

**for**

```
@for ($i =0; $i < count($data); $i++)
    {{ $data[$i] }}<br>
@endfor
```

**while**

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

**if**

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

## Validation

Voir le lien « **laravel.com/docs/9.x/validation** »

La méthode « **validate** » accepte la requête HTTP et un ensemble de règles de validation. Si la validation réussie, la suite du code s'exécute normalement. Sinon, Laravel redirige l'utilisateur à la page précédente (la page qui a généré la requête), et lui envoie une erreur.

**Exemple :**

```
public function ajouterVisiteur(Request $request) {
    $this->validate($request, [
        'prenom' => 'required|alpha|max:255',
        'nom' => 'required|alpha|max:255',
        'email' => 'required|string|email|max:191|unique:users',
        'tel' => 'string|nullable|min:8|max:30',
        'ville' => 'alpha_num|nullable|max:255',
        'code_postal' => 'alpha_num|nullable|max:10',
    ]);

    DB::table('visiteurs')->insert([
        'prenom' => $request->input('prenom'),
        'nom' => $request->input('nom'),
        'email' => $request->input('email'),
        'tel' => $request->input('tel'),
        'ville' => $request->input('ville'),
        'code_postal' => $request->input('code_postal')
     ]);
}
```

**Afficher les erreurs d'un formulaire** :

Voir section « **Displaying The Validation Errors** »

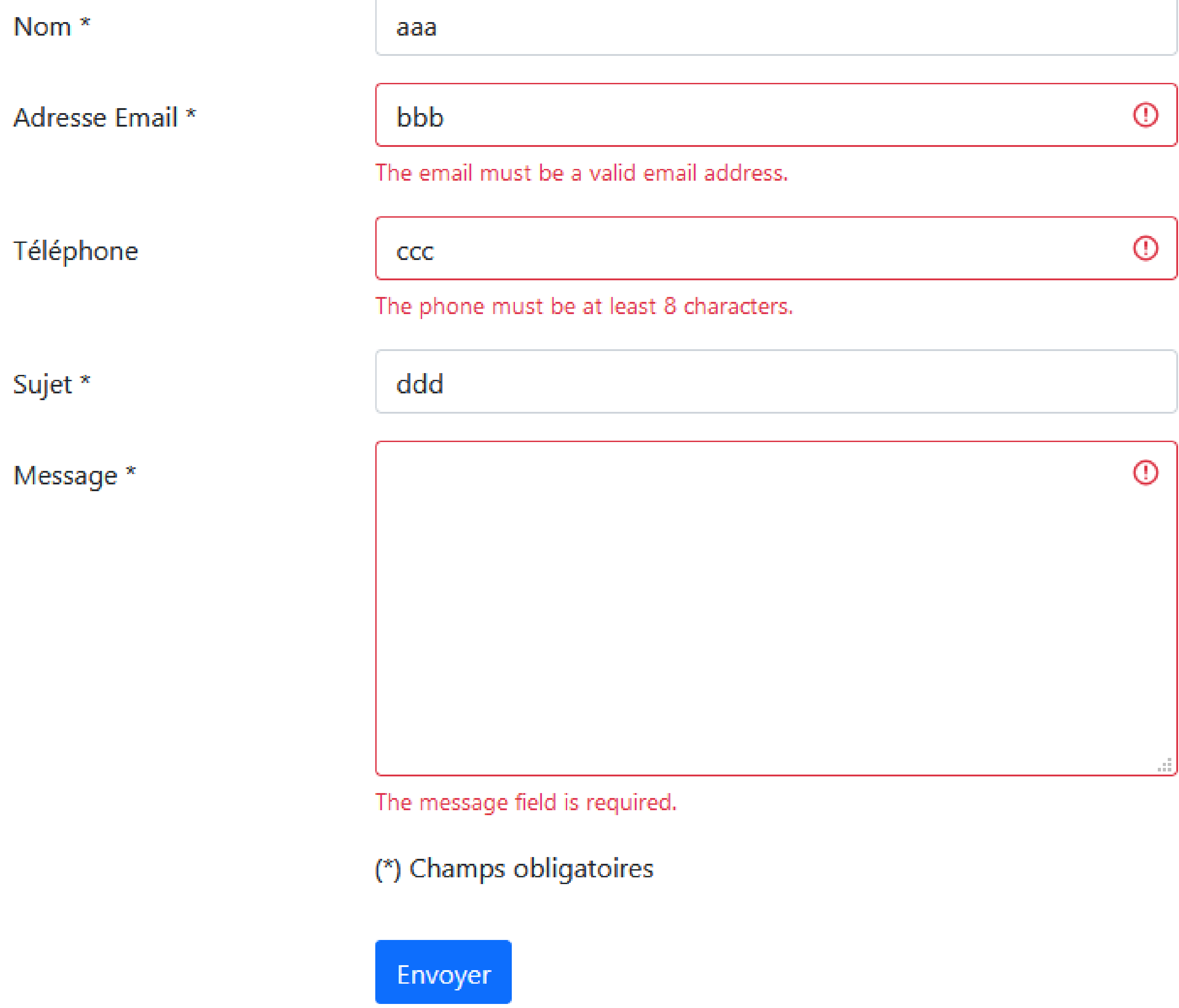
**Remplissage du formulaire suite à une erreur de validation** :

Lorsque la validation échoue, Laravel redirige l'utilisateur à la page du formulaire. Cette page a accès à la variable « **$errors** » qui permet de consulter les différents messages d'erreur. Il est possible de remplir les différents champs de ce formulaire avec les données invalidées en utilisant le helper « **old** ». Ce helper reçoit en paramètre le nom du champ et retourne la valeur saisie dans ce champ. Il est possible aussi de vérifier si un champ en particulier est invalide, et récupérer le message d'erreur correspondant.

**Exemple :**

```
...
<div class="row mb-3">
  <label for="name" class="col-sm-3 offset-sm-1 col-form-label">Nom *</label>
  <div class="col-sm-7">
    <input type="text" class="form-control @if ($errors->has('name')) is-invalid
                       @endif" id="name" name="name" value="{{ old('name') }}">
    @if ($errors->has('name'))
    <div class="invalid-feedback">{{ $errors->first('name') }}</div>
    @endif
  </div>
</div>

...

<div class="row mb-3">
  <label for="message" class="col-sm-3 offset-sm-1 col-form-label">Message *</label>
  <div class="col-sm-7">
    <textarea class="form-control @if ($errors->has('message')) is-invalid @endif"
              rows="7" id="message" name="message">{{ old('message') }}</textarea>
    @if ($errors->has('message'))
    <div class="invalid-feedback">{{ $errors->first('message') }}</div>
    @endif
  </div>
</div>
...
```

**Résultat au cas d'erreur :**




**Validation manuelle :**

Voir section « **Manually Creating Validators** »

## Base de données

Voir le lien « **laravel.com/docs/9.x/database** »

Laravel simplifie remarquablement la manipulation des bases de données, et supporte 4 SGBD. Ce framework permet de manipuler les données en utilisant :

- Des requêtes SQL brutes
- **Query Builder** (voir le lien « **laravel.com/docs/9.x/queries** ») : ce générateur de requêtes permet à une application Laravel de communiquer directement et facilement avec les tables de la base de données.

- **Eloquent ORM** : Une application Laravel peut communiquer directement avec les tables de la base de données. Cependant, il est très intéressant d'utiliser des modèles, c'est-à-dire une représentation objet de chacune des tables. Lorsqu'on utilise une telle couche, placée entre le code et la base de données, on dit qu'on fait du mapping « objet-relationnel », souvent référé comme ORM (Object-Relational Mapping). L'ORM livré avec Laravel s'appelle Eloquent. Les fonctionnalités qu'il vous offre sont époustouflantes.

Les SGBD actuellement supportés sont : MySQL, PostgreSQL, SQLite et SQL Server.

**Configuration**

La configuration des bases de données (**DB** : Database) se trouve dans le fichier « **config/database.php** » d'une application Laravel. Ce fichier permet de définir les différentes connexions avec les 4 SGBD supportés, et la connexion à utiliser par défaut. La clé « **default** » de ce fichier indique la connexion par défaut, et la clé « **connections** » contient la configuration des différentes connexions (y compris la connexion par défaut).

**Exemple de clés du fichier « config/database.php » :**

```
...
'default' => env('DB_CONNECTION', 'mysql'),

'connections' => [
    ...

    'mysql' => [
        'driver' => 'mysql',
        'url' => env('DATABASE_URL'),
        'host' => env('DB_HOST', '127.0.0.1'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'forge'),
        'username' => env('DB_USERNAME', 'forge'),
        'password' => env('DB_PASSWORD', ''),
        'unix_socket' => env('DB_SOCKET', ''),
        'charset' => 'utf8mb4',
        'collation' => 'utf8mb4_unicode_ci',
        'prefix' => '',
        'prefix_indexes' => true,
        'strict' => true,
        'engine' => null,
    ],

    ...
],
...
```

**Remarque** : le helper « **env** » permet de retourner la valeur d'une variable d'environnement lorsque cette variable est définie dans le fichier « **.env** » de l'application. Si la variable n'est pas définie dans ce fichier, c'est le deuxième paramètre du helper qui sera retourné.

**Exemple** : la fonction suivante retourne la valeur correspondante à la variable `'DB_CONNECTION'` si cette variable est définie dans le fichier « **.env** » (sinon, la fonction retourne la valeur « **mysql** ») :

```
env('DB_CONNECTION', 'mysql')
```

**Exemple de variables d'environnement du fichier « .env » :**

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=forum
DB_USERNAME=root
DB_PASSWORD=0000
```

**Requêtes SQL brutes**

La façade « **DB** » permet d'exécuter des requêtes SQL brutes. Elle offre les méthodes suivantes pour chaque type de requête SQL : **select**, **update**, **insert**, **delete**, et **statement**.

**Question** : Comment trouver toutes les méthodes de la façade « **DB** » et leur documentation ?

**Réponse** : Aller à la page « **laravel.com/docs/9.x/facades** » et trouver la classe correspondante à la façade « **DB (instance)** ». Aller sur la page de cette classe pour voir l'ensemble de ses méthodes (également les méthodes de la façade « **DB** »).

Requête « **select** » :

La méthode « **select** » de la façade « **DB** » permet d'exécuter une requête basique. Le premier argument de cette méthode est la requête SQL, et le deuxième argument est un tableau contenant les paramètres de la requête SQL.

La méthode « **select** » retourne un tableau contenant les résultats de la requête. Chaque résultat est un objet PHP de type « **StdClass** », permettant d'accéder aux valeurs du résultat.

**Exemple :**

```
$users = DB::select('select * from users where active=?', [1]);

foreach ($users as $user) {
    echo $user->name;
}
```

Requête « **insert** » :

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Abid']);
```

Requête « **update** » :

La méthode « **update** » retourne le nombre de lignes modifiées par la requête SQL.

**Exemple :**

```
$n = DB::update('update users set votes=100 where name=?', [Abid]);
echo "$n lignes modifiées";
```

Requête « **delete** » :

La méthode « **delete** » retourne le nombre de lignes supprimées.

**Exemple :**

```
$n = DB::delete('delete from users where id < ?', [10]);
echo "$n lignes supprimées";
```

Requête « **statement** » :

Certaines requêtes SQL ne retourne aucun résultat. Ces requêtes peuvent être exécutées avec la méthode « **statement** ».

**Exemple :**

```
DB::statement('drop table users');
```

**Query Builder**

Voir le lien « **laravel.com/docs/9.x/queries** »

Pour commencer une requête, on utilise la méthode « **table** » de « **DB** ». Cette méthode reçoit le nom de la table et retourne un « **Query Builder** » (objet « **Builder** »). On peut ensuite utiliser les méthodes de ce Builder pour ajouter des nouvelles contraintes. Finalement on appelle la méthode « **get** » du Builder pour obtenir les résultats.

**Remarque** : Pour trouver toutes les méthodes de la classe « **Builder** », il faut d'abord chercher la classe associée à la façade « **DB** (instance) ». Ensuite, il faut trouver le type de retour de la méthode « **table** » (classe « **Builder** »). Finalement, aller à la page de la classe « **Builder** » et consulter l'ensemble de ses méthodes.

**Lire toutes les lignes d'une table :**

<table>
<tr><th>A partir du contrôleur</th><th>A partir de la vue</th></tr>
<tr><td><pre>$users = DB::table('users')->get();
return view('index', ['users' => $users]);</pre></td><td><pre>foreach ($users as $user) {
    echo $user->name;
}</pre></td></tr>
</table>

**Lire une seule ligne d'une table :**

```
$user = DB::table('users')->where('name', 'Abid')->first();
echo $user->id;
```

**Sélectionner des colonnes :**

La méthode « **select** » permet de lire des colonnes en particulier et évite de lire toutes les colonnes d'une table.

**Exemple :**

```
$users = DB::table('users')->select('name', 'email')->get();
```

**Remarque** : Il y a plusieurs autres exemples d'utilisation des différentes méthodes du « **Query Builder** » sur le site de Laravel.

## Pagination

Voir le lien « **laravel.com/docs/9.x/pagination** »

Laravel simplifie la pagination. Au lieu d'utiliser la méthode « **get** » du « **Query Builder** », il faut utiliser la méthode « **paginate** » en lui indiquant le nombre d'éléments à afficher.

**Exemple :**

```
$contacts = DB::table('contacts')->select('nom', 'prenom', 'sujet')->paginate(5);
```

On peut ensuite appeler la méthode « **links** » de l'objet retourné à partir de la vue afin d'afficher les liens vers les autres pages.

**Exemple :**

```
<table border="1" style="width:50%;">
    <tr>
        <th>Nom</th>
        <th>Prenom</th>
        <th>Sujet</th>
    </tr>
    @foreach ($contacts as $contact)
    <tr>
        <td>{{ $contact->nom }}</td>
        <td>{{ $contact->prenom }}</td>
        <td>{{ $contact->sujet }}</td>
    </tr>
    @endforeach
</table>

{{ $contacts->links() }}
```

**Résultat :**

| Nom | Prenom | Sujet |
|---|---|---|
| n1 | p1 | s1 |
| n2 | p2 | s2 |
| n3 | p3 | s3 |
| n4 | p4 | s4 |
| n5 | p5 | s5 |

## Migrations

Voir le lien « **laravel.com/docs/9.x/migrations** »

Les « **Migrations** » permettent de partager, modifier et supprimer facilement les tables d'une base de données. Un « **Migration** » est un fichier PHP utilisant la façade « **Schema** ». L'ensemble des fichiers « **Migrations** » se trouvent dans le dossier « **database\migrations** ».

Créer un « **Migration** » vide nommé « **create_infos_table** » :

```
C:\xampp\htdocs\forum>php artisan make:migration create_infos_table
Created Migration: 2022_03_10_070158_create_infos_table
```

Laravel utilisera le nom du « **Migration** » pour deviner le nom de la table.

**Résultat :**

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('infos', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('infos');
    }
};
```

Exécuter les fichiers « **Migrations** » : Pour créer les tables des « **Migrations** » en attente (qui ne sont pas encore crées) :

```
C:\xampp\htdocs\forum>php artisan migrate
```

Supprimer toutes les tables des « **Migrations** » :

```
C:\xampp\htdocs\forum>php artisan migrate:reset
```

**Remarque :** La suppression d'un migration consiste à supprimer le fichier du dossier « **database\migrations** ».

**Exercice :**

1) Citer les noms des fichiers « **Migration** » disponibles par défaut dans l'application Laravel
2) Créer un « **Migration** » vide nommé « **create_infos_table** » et visualiser son contenu.
3) Essayer de créer à nouveau le « **Migration** » nommé « **create_infos_table** ». Quel est le message qui s'affiche ?
4) Créer les tables des « **Migrations** ». Quel est le message qui s'affiche ?
5) Supprimer toutes les tables des « **Migrations** » avec la commande :
   **`php artisan migrate:reset`**
6) Supprimer manuellement le dernier « **Migration** » nommé « **create_infos_table** » et créer les tables à nouveau. Quel est le message qui s'affiche ?
7) Visualiser les tables avec « **phpMyAdmin** ».

## Commandes Utiles :

**Remarque** : Les commandes suivantes doivent être exécutées à partir du dossier racine de l'application.

Vérifier la version d'une application laravel :

```
php artisan --version
```

Lister les différentes commandes artisan :

```
php artisan list
```

Vider tout type de cache de l'application (cache des vues, cache de l'application, cache de route, cache de configuration et cache des services) :

```
php artisan optimize:clear
```

Vider le cache de l'application (vider le dossier storage/framework/cache/data) :

```
php artisan cache:clear
```

Vider le cache des vues (vider le dossier storage/framework/views) :

```
php artisan view:clear
```

Vider le cache des routes :

```
php artisan route:clear
```

Vider le cache de configuration :

```
php artisan config:clear
```

## Authentification

Voir les liens « **laravel.com/docs/9.x/authentication** » et « **https://laravel.com/docs/6.x/frontend** »

Laravel simplifie l'implémentation de l'authentification dans une application. Le fichier de configuration de l'authentification est « **config/auth.php** ». Ce fichier permet par exemple de configurer le temps d'expiration d'une session (60 minutes par défaut).

Laravel contient par défaut deux fichiers Migration permettant de construire les tables nécessaires à l'authentification :

- **create_users_table** : permet de construire la table « **users** »
- **create_password_resets_table** : permet de construire la table « **password_resets** » et gère les jetons de réinitialisation des mots de passe

L'authentification utilise l'Eloquent ORM pour manipuler la table « **users** » moyennant la classe Eloquent « **App\Models\User** ». Cette classe est disponible par défaut dans une application.

Pour implémenter l'authentification dans une application, il faut exécuter les commandes suivantes ( **https://laravel.com/docs/6.x/frontend** ) :

```
C:\xampp\htdocs\forum> php artisan migrate
C:\xampp\htdocs\forum> composer require laravel/ui
C:\xampp\htdocs\forum> php artisan ui bootstrap --auth
```

La commande « **`php artisan migrate`** » permet de construire les tables de la base de données à partir des fichiers Migration.

**Remarque** : si un message d'erreur indique que la taille de la colonne « email » est insupportable, il faut réduire la taille de cette colonne (par défaut 255 caractères) dans les deux fichiers Migrations comme suit :

```
$table->string('email', 190)->unique();
```

La commande « **`composer require laravel/ui`** » permet de télécharger et d'installer le paquetage « **`laravel/ui`** ».

La commande « **`php artisan ui bootstrap --auth`** » permet de générer les vues, les contrôleurs et les routes nécessaires pour implémenter l'authentification. Le paramètre « **`bootstrap`** » de la commande permet de générer des vues utilisant le framework « **bootstrap** » (la version de utilisé de bootstrap est indiquée dans le fichier « package.json »).

**Quelle est la version de bootstrap utilisée par les vues d'authentification ?**

**Ajouter Bootstrap à l'application :**

Suite à l'exécution de la dernière commande, on obtient le message suivant :

```
C:\xampp\htdocs\forum> php artisan ui bootstrap --auth
Bootstrap scaffolding installed successfully.
Please run "npm install && npm run dev" to compile your fresh scaffolding
Authentication scaffolding generated successfully.
```

2 méthodes pour ajouter Bootstrap à l'application :

- Téléchargement et compilation du code source de Bootstrap en utilisant les commandes :

```
C:\xampp\htdocs\forum> npm install
C:\xampp\htdocs\forum> npm run dev
C:\xampp\htdocs\forum> npm run dev
```

  Ces commandes génèrent les deux fichiers « **`public\css\app.css`** » et « **`public\js\app.js`** »

  **Remarque** : la commande « **npm** » nécessite l'installation de « **node.js** »

- Utilisation de Bootstrap à partir d'un réseau CDN : il faut ajouter les lignes suivantes dans la balise « **head** » du fichier « **`resources\views\layouts\app.blade.php`** » :

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/boots
trap.min.css">

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstr
ap.bundle.min.js"></script>
```

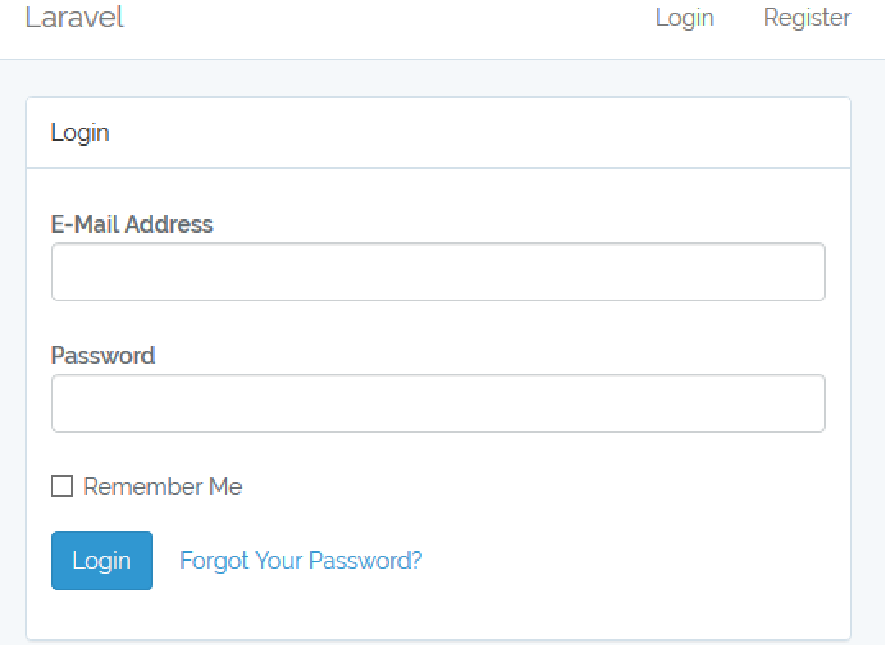
L’implémentation de l’authentification permet de créer les contrôleurs, les vues et les routes suivants :

**Les contrôleurs** (voir le dossier « **app\Http\Controllers** ») :

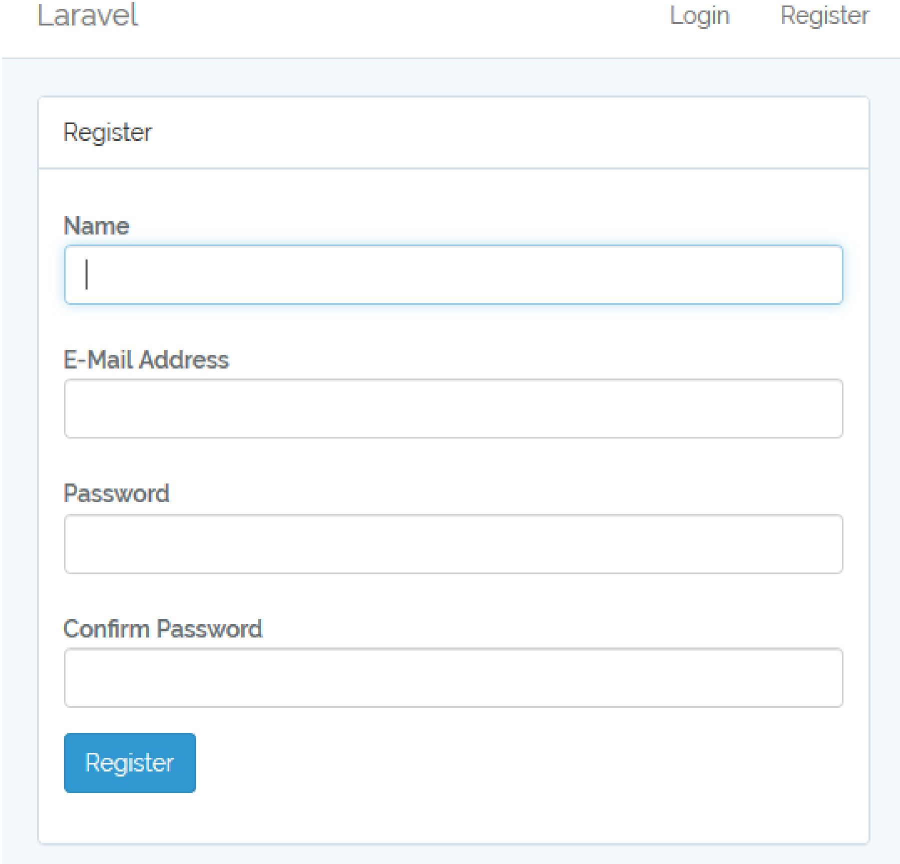
- **HomeController** : affiche la page d’accueil des utilisateurs authentifiés (**home.blade.php**)
- **Auth\RegisterController** : gère l’enregistrement des nouveaux utilisateurs
- **Auth\LoginController** : gère l’authentication
- **Auth\ForgotPasswordController** : génère et envoi par Email les liens de réinitialisation des mots de passe
- **Auth\ResetPasswordController** : contient la logique de réinitialisation des mots de passe
- **Auth\VerificationController** : gère la vérification des emails des nouveaux inscrits

**Les vues :**

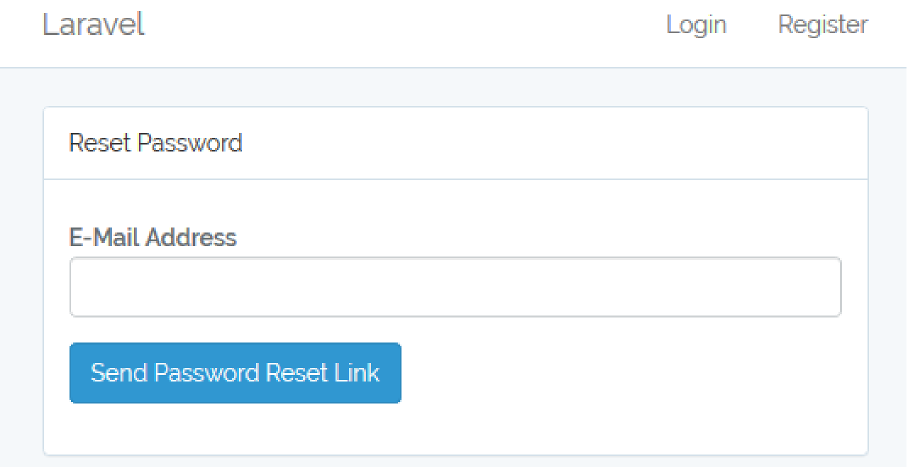
- resources\views**auth\login.blade.php** : formulaire d’authentification



- resources\views**auth\register.blade.php** : formulaire d'enregistrement d'un nouvel utilisateur




- resources\views**auth\passwords\email.blade.php** : vue permettant de demander la réinitialisation du mot de passe



- resources\views**auth\passwords\reset.blade.php** : permet de réinitialiser le mot de passe

Laravel Login Register

Reset Password

E-Mail Address

Password

Confirm Password

Reset Password

- resources\views**home.blade.php** : espace d’accueil réservé pour les utilisateurs authentifiés

Laravel Ali Dridi

Dashboard

You are logged in!

- resources\views**layouts\app.blade.php** : contient un layout de base

Laravel Login Register

Utilisateur **non**-authentifié

Laravel Ali Dridi

Utilisateur authentifié

**Les routes :**

```
Auth::routes();
Route::get('/home',
[App\Http\Controllers\HomeController::class, 'index'])->name('home');
```

Pour afficher toutes les routes d'une application, il est possible d'utiliser la commande suivante :

**`php artisan route:list`**

**Exemple :**

```
GET|HEAD   / .................................................................................................
POST       _ignition/execute-solution .. ignition.executeSolution › Spatie\LaravelIgnition › ExecuteSolutionController
GET|HEAD   _ignition/health-check ............. ignition.healthCheck › Spatie\LaravelIgnition › HealthCheckController
POST       _ignition/update-config .......... ignition.updateConfig › Spatie\LaravelIgnition › UpdateConfigController
GET|HEAD   api/user ..........................................................................................
GET|HEAD   home ........................................................................ home › HomeController@index
GET|HEAD   login ........................................................ login › Auth\LoginController@showLoginForm
POST       login ....................................................................... Auth\LoginController@login
POST       logout .......................................................... logout › Auth\LoginController@logout
GET|HEAD   password/confirm ........................ password.confirm › Auth\ConfirmPasswordController@showConfirmForm
POST       password/confirm ................................................... Auth\ConfirmPasswordController@confirm
POST       password/email ......................... password.email › Auth\ForgotPasswordController@sendResetLinkEmail
GET|HEAD   password/reset ...................... password.request › Auth\ForgotPasswordController@showLinkRequestForm
POST       password/reset ....................................... password.update › Auth\ResetPasswordController@reset
GET|HEAD   password/reset/{token} ........................ password.reset › Auth\ResetPasswordController@showResetForm
GET|HEAD   register ......................................... register › Auth\RegisterController@showRegistrationForm
POST       register ................................................................ Auth\RegisterController@register
GET|HEAD   sanctum/csrf-cookie ......................................... Laravel\Sanctum › CsrfCookieController@show
```

**Redirection après authentification**

Lorsqu'un utilisateur est authentifié correctement, il sera redirigé par défaut à l'URI « **/home** ». Il est possible de changer cette URI en modifiant la variable « **`$redirectTo`** » dans les contrôleurs « **LoginController** », « **RegisterController** », et « **ResetPasswordController** » :

```
// rediriger à l'URI "/" après authentification
protected $redirectTo = '/';
```

Si le chemin de redirection nécessite une logique personnalisable, il est possible de définir la méthode « **`redirectTo`** » au lieu de la variable « **`$redirectTo`** » (la méthode est plus prioritaire que la variable).

**Exemple :**

```
protected function redirectTo()
{
    return '/';
}
```

Définir la méthode « `redirectTo` » peut être utile dans plusieurs scénarios, tels que :

- Si l'utilisateur est un administrateur, il sera redirigé à la page d'accueil « Admin ». Sinon, il sera redirigé à la page d'accueil « User ».
- Si l'utilisateur a un nouveau message, il sera redirigé à la page des messages. Sinon, il sera redirigé à la page d'accueil.
- Etc.

**Modification des données d'enregistrement**

Pour modifier les champs d'enregistrement d'un utilisateur, il faut faire les modifications suivantes :

- Modifier les colonnes de la table « **users** » de la base de données (Modifier le fichier Migration correspondant et créer à nouveau la table).
- Modifier les champs du formulaire à partir de la vue « **auth\register.blade.php** ».
- Modifier la méthode « **validator** » de « **RegisterController** » : cette méthode contient les règles de validation des données du formulaire. Elle doit être modifiée selon les besoins.
- Modifier la méthode « **create** » de « **RegisterController** » : cette méthode permet d'ajouter le nouvel utilisateur dans la base de données en utilisant Eloquent ORM. Elle doit être modifiée selon les besoins.
- Modifier le fichier « **app\Models\User.php** » et renseigner le nom des colonnes modifiables dans la table « **$fillable** ».

**Accès à l'utilisateur authentifié**

Il est possible d'accéder à l'utilisateur authentifié avec la façade « **Auth** ».

Exemples :

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


//vérifier si l'utilisateur est authentifié
if (Auth::check()) {
    // l'utilisateur est authentifié


    // accéder à l'utilisateur
    $user = Auth::user();


     // lire les données de l'utilisateur
    $id = $user->id;
    $name = $user->name;
    $email = $user->email;
```

```
    //modifier et enregistrer les données
    $user->name = "Kamel";
    $user->save();


    //déconnexion
    Auth::logout();
}
```

**Protection des routes**

Laravel utilise les « **Middleware** » (composants interlogiciel) pour filtrer les requêtes HTTP entrantes (voir le lien : « **laravel.com/docs/9.x/middleware** »). Par exemple, Laravel inclut un middleware qui vérifie que l'utilisateur est authentifié. S'il n'est pas authentifié, le middleware redirigera l'utilisateur vers l'écran de connexion. Mais si l'utilisateur est authentifié, le middleware autorisera la requête à aller plus loin dans l'application.

Le middleware peut être utilisé pour autoriser uniquement les utilisateurs authentifiés à accéder à une route en particulier. Laravel contient par défaut le middleware « **auth** » définit dans la classe suivante :

**`Illuminate\Auth\Middleware\Authenticate`**

Ce « **Middleware** » est déjà enregistré dans le kernel (noyau) HTTP de l'application, et il suffit de l'attacher à une route pour la protéger :

```
Route::get('profile', function () {
    return view('user.profile');
})->middleware('auth');
```

Pour les contrôleurs, il faut appeler la méthode « **middleware** » depuis le constructeur du contrôleur au lieu de l'attacher directement dans la définition de la route :

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

TP4

## Middleware

Voir le lien « **laravel.com/docs/9.x/middleware** »

Un middleware permet de simplifier le filtrage des requêtes HTTP entrantes dans l'application. Il offre plusieurs fonctionnalités utiles, telles que :

- La vérification de l'authentification des utilisateurs
- La protection CSRF
- Journalisation (logging) des requêtes entrantes
- Etc.

Tous les middlewares se trouvent dans le dossier « **app/Http/Middleware** ». Pour créer un nouveau middleware, on peut utiliser la commande Artisan suivante :

```
php artisan make:middleware CheckAdmin
```

Cette commande permet de créer le fichier « **CheckAdmin.php** » (classe « **CheckAdmin** ») dans le dossier « **app/Http/Middleware** » avec le contenu suivant :

```
namespace App\Http\Middleware;

use Closure;

class CheckAdmin
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

Dans cet exemple, la méthode « **handle** » reçoit la requête et lui permet d'accéder à l'application. Si l'accès est autorisé uniquement pour un administrateur, on peut modifier la méthode « **handle** » comme suit :

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class CheckAdmin
{
    public function handle($request, Closure $next)
    {
            if (Auth::check()) {
                 if (Auth::user()->role == "admin") {
                      return $next($request);
                 }
            }

            return redirect('/');
    }
}
```

Après la création d'un middleware, il faut l'enregistrer dans le kernel (noyau) de l'application (fichier « **app/Http/Kernel.php** »). Un middleware peut être enregistré dans l'un des cas suivants :

1) Pour toutes les requêtes HTTP d'une application
2) Pour des routes en particulier
3) Dans des groupes

Si le middleware doit être exécuté pour chaque requête HTTP d'une application, il faut le lister dans le tableau « **$middleware** » :

```
protected $middleware = [
    \App\Http\Middleware\TrustProxies::class,
    \Illuminate\Http\Middleware\HandleCors::class,
    \App\Http\Middleware\PreventRequestsDuringMaintenance::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

Afin d'enregistrer un middleware pour des routes en particulier, il faut le lister dans le tableau indexé « **$routeMiddleware** » et lui attribuer une clé. Par exemple, on attribue la clé « **admin** » au middleware « **CheckAdmin** » et on le liste à la fin du tableau « **$routeMiddleware** » comme suit :

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'admin' => \App\Http\Middleware\CheckAdmin::class,
];
```

Après la création et l’enregistrement du middleware, il suffit de l’attacher à une route pour la protéger :

```
Route::get('admin/profile', function () {
    return view('admin.profile');
})->middleware('admin');
```

Il est possible d’attacher plusieurs middlewares à une seule route :

```
Route::get('admin/profile', function () {
    return view('admin.profile');
})->middleware('auth', 'admin');
```

Pour les contrôleurs, il faut appeler la méthode « **middleware** » (en lui donnant le nom du middleware) depuis le constructeur du contrôleur au lieu de l'attacher directement dans la définition de la route :

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

Pour protéger un contrôleur avec plusieurs middlewares, il faut appeler la méthode « **middleware** » pour chaque middleware depuis le constructeur du contrôleur :

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

## Héritage des vues

Voir le lien « **laravel.com/docs/9.x/blade** »

Blade simplifie l’héritage des templates. Lorsque plusieurs pages utilisent le même layout (exemple : entête et bas de page), il est possible de définir ce layout dans un fichier et l’hériter par les autres pages. Cet héritage est possible avec Blade en utilisant les instructions suivantes :

- **`@section`** : permet de définir une section avec un contenu
- **`@yield`** : permet de placer le contenu d’une section dans la page
- **`@extends`** : utilisé par une page pour indiquer le layout qui doit être hérité

**Exemple de layout** (fichier « **resources/views/layouts/app.blade.php** ») :

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            <p>This is the layout sidebar.</p>
        @show
        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

**Remarque** : lorsqu'une section est entourée par `@section` et `@show`, elle sera définie et affichée. Mais si elle est entourée par `@section` et `@endsection`, elle sera définie uniquement (il faut utiliser l'instruction `@yield` pour afficher la section).

**Exemple de vue héritant le layout** (fichier « **resources/views/admin/profile.blade.php** ») **:**

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

@section('title', 'Profile Admin')

@section('sidebar')
    @parent

    <p>This is appended to the layout sidebar.</p>
@endsection

@section('content')
    <p>This is the profile of the Administrator.</p>
@endsection
```

**Remarque** : l'usage de l'instruction `@parent` dans la section « **sidebar** » permet d'ajouter du contenu à la section au lieu de la redéfinir.

Le contenu HTML généré par la vue « **profile.blade.php** » est :

```
<html>
    <head>
        <title>App Name - Profile Admin</title>
    </head>
    <body>
        <p>This is the layout sidebar.</p>

        <p>This is appended to the layout sidebar.</p>

        <div class="container">
            <p>This is the profile of the Administrator.</p>
        </div>
    </body>
</html>
```

# Email

Voir le lien « **laravel.com/docs/9.x/mail** »

Le fichier de configuration du mail est « **config/mail.php** », et contient les paramètres permettant de modifier le serveur SMTP, le port, les identifiants de connexion, etc. La modification de ces paramètres peut se faire directement à partir de ce fichier ou à partir du fichier « **.env** » (il faut renseigner les variables d'environnement correspondants).

**Variables d'environnement du fichier « .env » pour configurer le mail :**

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=25
MAIL_USERNAME=tp.isi.fsm@gmail.com
MAIL_PASSWORD=abcd0000
MAIL_ENCRYPTION=tls
MAIL_FROM_NAME=MP-ISI1
MAIL_FROM_ADDRESS=tp.isi.fsm@gmail.com
MAIL_FROM_NAME="${APP_NAME}"
```

Les variables d'environnement précédents permettent à l'application Laravel d'utiliser l'adresse email « **`tp.isi.fsm@gmail.com`** » et le mot de passe correspondant (**`abcd0000`**) pour envoyer des emails. La variable « **`MAIL_FROM_NAME`** » renseigne le nom de l'expéditeur de l'email.

Après la configuration du fichier « **.env** » (ou aussi le fichier « **config/mail.php** »), l'application Laravel peut envoyer des emails en utilisant 1) l'envoi basique, ou 2) les objets **Mailables**. Dans les deux cas, il faut importer la façade « **Mail** » et utiliser ses méthodes.

~~**Remarque** : Pour utiliser le chiffrement des emails sans avoir un certificat reconnu, il faut autoriser les certificats auto-signé en ajoutant les lignes suivantes à la fin du fichier « **config/mail.php** » :~~

```
        'stream' => [
                'ssl' => [
                        'allow_self_signed' => true,
                        'verify_peer' => false,
                        'verify_peer_name' => false,
                ],
        ],
```

**1. Envoi basique**

Cette méthode est documentée par les anciennes versions de Laravel (voir la documentation de la version 5.0 sur le lien « **laravel.com/docs/5.0/mail** »)

La méthode « `Mail::send` » peut être utiliser pour envoyer un email.

**Exemple** :

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class EmailsController extends Controller
{
    public function sendEmail(Request $request)
    {
        Mail::send('emails.hello', [], function ($message) {
            $message->to('destinataire@gmail.com');
            $message->subject('Envoi email avec Laravel');
        });
        return "email envoyé\n<br>";
    }
}
```

La méthode « `Mail::send` » reçoit les arguments suivants :

- le nom de la vue qui contient le message à envoyer
- un tableau indexé contenant les données passées à la vue (la clé correspond au nom de la variable à utiliser pour récupérer la valeur)
- un « **Closure** » permettant de renseigner certains paramètres

**Exemple** :

```
$d = date("H:i:s");

Mail::send('emails.hello', ['date' => $d], function ($message) {
     $message->to('destinataire@gmail.com');
     $message->subject('Envoi email avec Laravel');
});
```

La vue est un fichier **php** (ou blade.php) permettant de générer une page **html**. La vue « **emails.hello** » correspond au fichier « **hello.php** » (ou **hello.blade.php**) qui se trouve dans le dossier « **resources\views\emails** ».

**Exemple de la vue « hello.blade.php » :**

```
<!doctype html>
<html>
    <body>
        <h1>Email bonjour</h1>
        <p>heure : {{ $date }}</p>
    </body>
</html>
```

La méthode « **`Mail::raw`** » permet d'envoyer un message sous forme d'une chaine de caractères (à la place d'une vue entière). Elle reçoit en premier argument le texte à envoyer, et en deuxième argument un « **Closure** ».

**Exemple** :

```
$d = date("H:i:s");
$texte = "Nous avons reçu votre demande à $d";

Mail::raw($texte, function($message) {
    $message->to('destinataire@gmail.com');
    $message->subject('Envoi email avec Laravel');
});
```

## 2. Envoi avec les objets Mailables

Les nouvelles versions de Laravel utilisent les objets « **Mailables** » (voir le lien « **laravel.com/docs/9.x/mail** »). Pour envoyer un email, on utilise la méthode « **`Mail::to`** » qui accepte l'adresse email du destinataire. Ensuite on utilise la méthode « **`send`** » et on lui donne un objet « **Mailable** ». Avec Laravel, toutes les classes « **Mailable** » se trouvent sous le dossier « **app/Mail** ».

**Remarque** : ne pas oublier d'importer la nouvelle classe « **Mailable** » dans le contrôleur !

### 2.1 Création d'une nouvelle classe mailable

Pour créer la classe « **EmailConfirmation** » de type « **Mailable** », on peut utiliser la commande suivante (à exécuter à partir du dossier de l'application Laravel) :

```
C:\xampp\htdocs\fsm\forum>php artisan make:mail EmailConfirmation
Mail created successfully.
```

La commande précédente permet de créer la classe « **EmailConfirmation** » dans le dossier « **app/Mail** » avec le code suivant :

```php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailConfirmation extends Mailable
{
    use Queueable, SerializesModels;

    public function __construct()
    {
        //
    }

    public function build()
    {
        return $this->view('view.name');
    }
}
```

L'envoi d'un email avec la classe « **Mailable** » se fait dans la méthode « **build** ». A partir de cette méthode, il est possible d'appeler les méthodes hérités telles que : **from**, **subject**, **view**, et **attach**.

**Exemple :**

```php
return $this->view('view.name');

return $this->from('example@example.com', 'Example')
          ->view('emails.hello');

return $this->from('example@example.com', 'Example')
          ->subject('Envoi email avec Laravel')
          ->view('emails.hello');
```

Dans la méthode « **build** », on peut utiliser la méthode héritée « **view** » pour envoyer le contenu d'une vue dans un email. La vue peut être un fichier php ou « blade.php ».

Toutes les variables publiques d'une classe « **Mailable** » sont accessibles à partir d'une vue.

**Exemple** :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailConfirmation extends Mailable
{
    use Queueable, SerializesModels;

    public $date;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        $this->date = date("H:i:s");

        return $this->subject('Envoi email avec Laravel')
                    ->view('emails.hello');
    }
}
```

Dans l'exemple précédent, la variable « **$date** » est publique dans la classe et sera accessible à partir de la vue « **emails.hello** » (fichier « **hello.php** » ou « **hello.balde.php** » du dossier « **resources\views\emails** »).

**Remarque** : Laravel supporte le template « **Markdown** » qui simplifie l'écriture des messages. Lorsque la vue « **emails.hello** » contient la syntaxe « **Markdown** », il faut utiliser la méthode « **markdown** » au lieu de « **view** » à partir de la méthode « **build** » pour envoyer la vue.

**2.2 Utilisation de la classe Mailable dans le contrôleur**

D'abord, il faut importer la nouvelle classe « **Mailable** » dans le contrôleur.

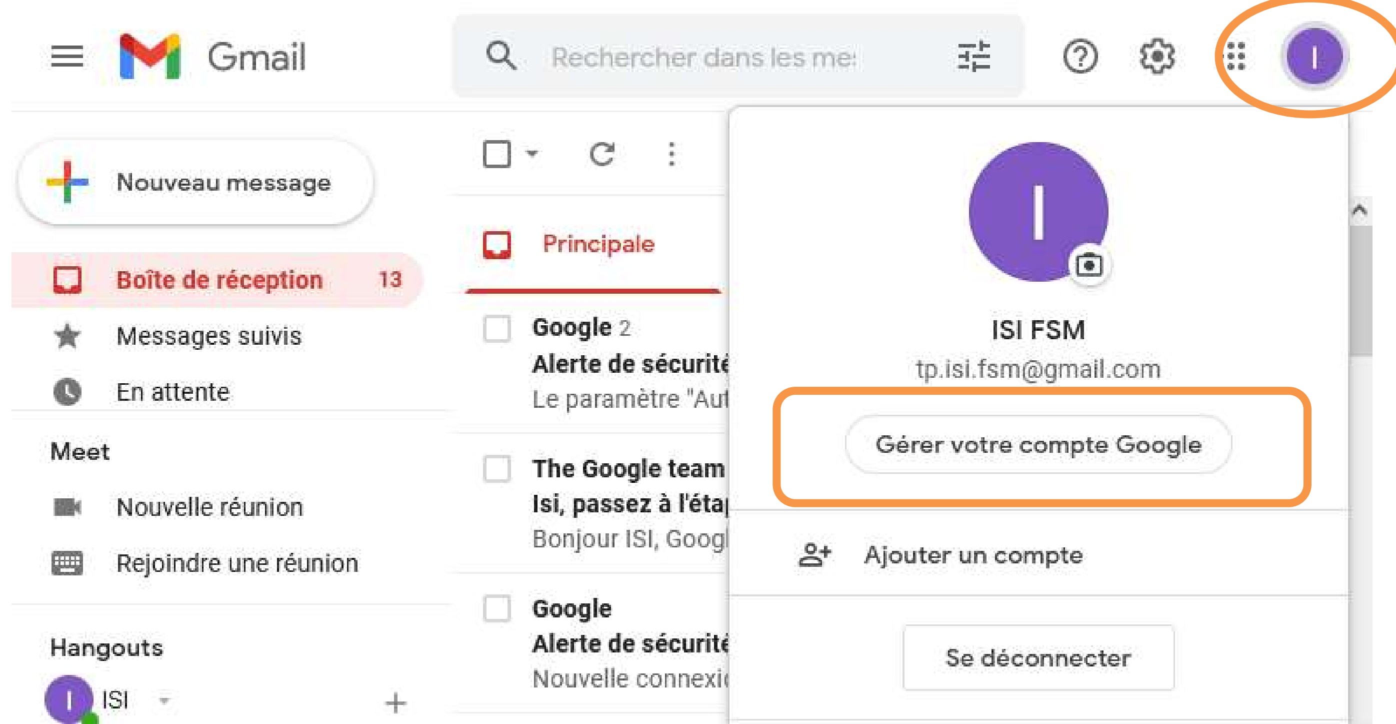
Ensuite, on utilise les méthodes « **`Mail::to`** » et « **`send`** ». La méthode « **`send`** » reçoit un objet de type « **Mailable** ».

**Exemple des méthodes « `Mail::to` » et « `send` » :**

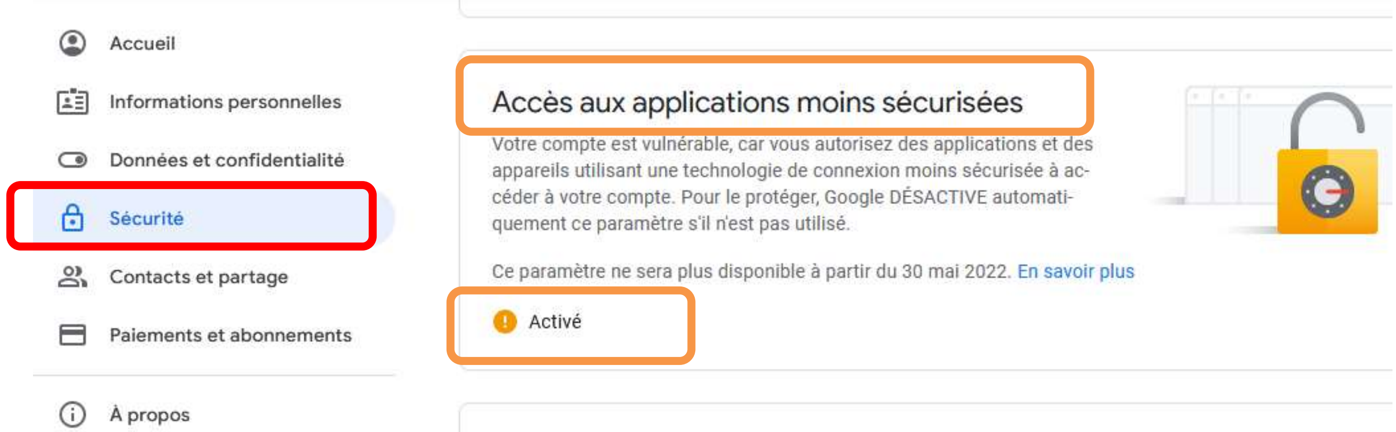
```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

use App\Mail\EmailConfirmation;

class EmailsController extends Controller
{
  public function sendEmail(Request $request)
  {
    Mail::to('destinataire@gmail.com')->send(new EmailConfirmation());
    return "email envoyé\n<br>";
  }
}
```

**Remarque** : Pour utiliser une adresse GMAIL pour l'envoi des emails, il faut autoriser l'accès aux applications non sécurisées en suivant les étapes suivantes :

Click sur le cercle en haut à droite et ensuite click sur « Gérer votre compte Google » :



Click sur « Sécurité » et ensuite sur la section « Accès aux applications moins sécurisées »




Ensuite il faut activé ce paramètre

## Cookies

Un serveur web peut stocker certaines informations dans le navigateur du client pour les utiliser ultérieurement. Ces informations sont conservées dans des cookies.

On peut utiliser les cookies pour mémoriser certaines préférences de l'utilisateur (couleurs, paramètres, identifiant d'une session, etc.).

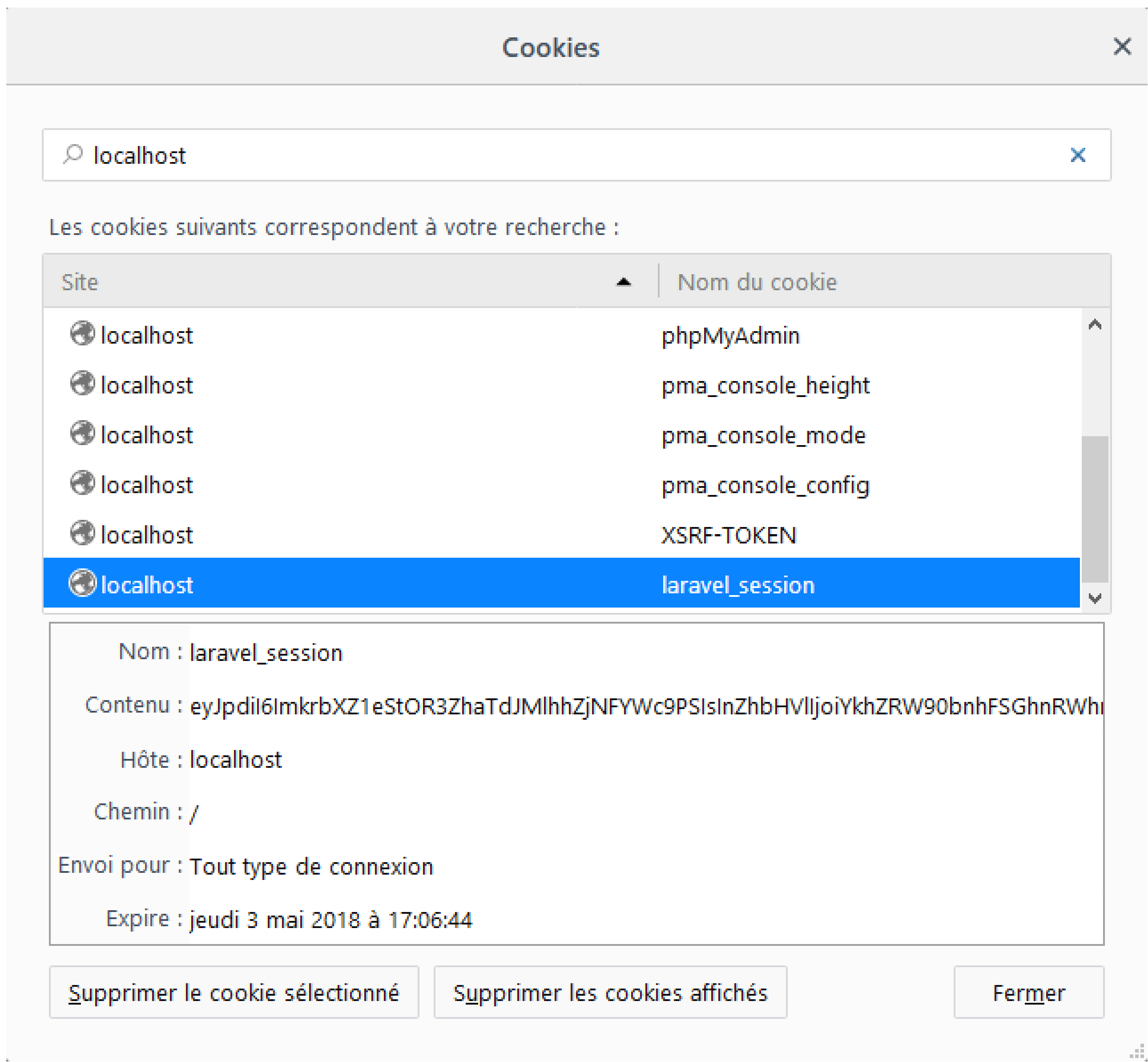
Un cookie est envoyé par le serveur sous la forme d'une chaine de caractères « **nom=valeur** » et en utilisant l'entête HTTP « **Set-Cookie** », exemple : « **`Set-Cookie: sessionid=38afes7a8`** ».

En plus de la paire « nom/valeur », un cookie peut contenir d'autres attributs : date d'expiration, chemin, nom de domaine et le type de connexion (chiffrée ou non). Exemple :

```
Set-Cookie: sessionid=38afes7a8; Expires=Wed, 21 Oct 2015 07:28:00 GMT
```

Un navigateur enregistre les cookies reçus et les envois dans toutes les futures requêtes au serveur.

Pour consulter les cookies de Firefox, aller à « **Outils > Options > Vie privée et sécurité** ». Ensuite cliquer sur le lien « **supprimer des cookies spécifiques** ». Dans le champ « **Rechercher** » de la fenêtre qui s'ouvre, taper « **localhost** » pour visualiser les cookies du domaine « **localhost** ». Appuyer sur ces cookies pour visualiser leurs informations.




Un serveur peut modifier un cookie en envoyant le même nom et une nouvelle valeur :
« **`Set-Cookie: nom=nouvelle_valeur`** ».

Un serveur peut supprimer un cookie en modifiant la date d'expiration en une date du passé.

**Les cookies avec Laravel**

Voir les liens « **laravel.com/docs/9.x/requests** » et « **laravel.com/docs/9.x/responses** »

Par défaut, tous les cookies créés par Laravel sont cryptés et signés avec un code d'authentification. Ça signifie que :

- les cookies seront considérés invalides s'ils sont modifiés par l'utilisateur
- la valeur correspondante à un cookie est illisible à partir du navigateur

**Envoi d'un cookie** : il est possible au serveur d'envoyer un cookie dans une réponse en utilisant le helper « **`response()`** » et la méthode « **`cookie()`** ».

**Exemple :**

```
//envoyer un cookie
$minutes = 60;
$content = view('home', ["txt" => $txt]);
return response($content)->cookie('nom', 'valeur', $minutes);
```

```
//envoyer plusieurs cookies
$minutes = 60;
$content = view('home', ["txt" => $txt]);
return response($content)
        ->cookie('nom', 'valeur', $minutes)
        ->cookie('car', 'voiture', $minutes)
        ->cookie('computer', 'ordinateur', $minutes);
```

La méthode « **`cookie()`** » accepte d'autres arguments qui représentent les autres attributs d'un cookie. Exemple :

```
->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

**Modification** : la modification d'un cookie consiste à modifier la valeur et/ou les attributs (date de validité, nom de domaine, etc.) d'un cookie existant. Elle est possible en utilisant le helper « **`response()`** » et la méthode « **`cookie()`** ».

**Suppression** : la suppression d'un cookie consiste à modifier sa date d'expiration en une date du passé. Ça est possible en utilisant le helper « **`response()`** » et la méthode « **`cookie()`** ». Exemple :

```
->cookie('nom', 'valeur', -600);
```

**Lecture** : contrairement à l'envoi des cookies (qui se fait avec le helper « **`response()`** »), la lecture des cookies est possible en utilisant les méthodes de l'objet « **`Request`** ».

**Exemple :**

```
$txt = 'aucun cookie "nom" : ';
if ($request->hasCookie("nom")) {
     $txt = 'cookie "nom" existe : ';
}

//lire un seul cookie
$nom = $request->cookie("nom");

//lire tous les cookies sous forme d'un tableau indexé : nom=>valeur
$cookies = $request->cookie();
```

# Sessions

Voir le lien « **laravel.com/docs/9.x/session** »

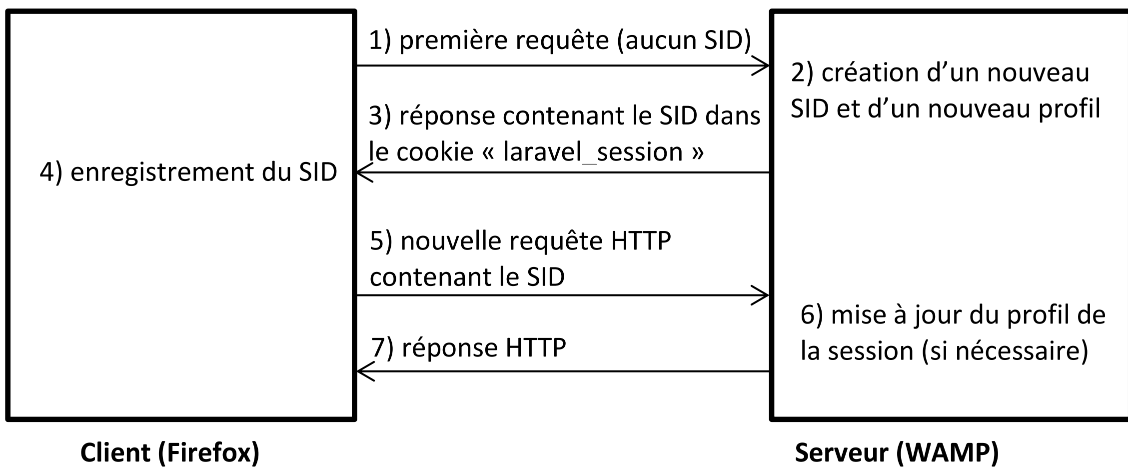
La session permet au serveur de mémoriser des informations relatives à l'utilisateur du site web (profil de l'utilisateur). Le serveur attribut à chaque session un identifiant unique (**SID** : Session Identifier). Cet identifiant est échangé entre le client et le serveur tout au long de la session.

Laravel permet de mémoriser les profils des utilisateurs de différentes façons :

- dans des fichiers du dossier « **storage/framework/sessions** »
- dans une base de données
- dans memcached/redis

L'identifiant d'une session est généralement échangé dans un cookie nommé « **laravel_session** ».

Le fichier de configuration de session est « **config/session.php** ».




Laravel permet de manipuler les sessions avec deux méthodes différentes :

- avec le helper « session »
- à travers l'objet « Request »

Accès à une seule variable de session :

```
public function afficher(Request $request)
{
	$valeur = 'Aucune valeur';
	if ($request->session()->has('cle')) {
		$valeur = $request->session()->get('cle');
	}

	return $valeur;
}
```

Il est possible de donner une valeur par défaut comme 2eme argument de la méthode « **get()** ». Cette valeur sera retournée si la clé n'existe pas dans la session :

```
$valeur = $request->session()->get('cle', 'Aucune valeur');
```

Accès à toutes les données d'une session :

```
$data = $request->session()->all();
```

Ajouter/modifier une variable de session : si la clé n'existe pas elle sera ajoutée, et si elle existe elle sera modifiée.

```
$request->session()->put('cle', 'valeur');
$request->session()->put("sujet", "Demande d'information");
```

Supprimer une seule variable de session :

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

Supprimer toutes les variables de session :

```
$request->session()->flush();
```