



# Comment créer, lire, mettre à jour et supprimer (CRUD) avec Laravel

# Ship and manage your web projects faster

Deploy your projects on Google Cloud Platform's top tier infrastructure. You'll get 25+ data centers to choose from, 24/7/365 expert support, and advanced security with DDoS protection.

Try for free

■ Data centers (25+) ■ CDN locations (200)

[Laravel](#) est un [framework PHP](#) populaire qui permet de créer des applications web modernes et dynamiques dans le paysage actuel du développement web, en constante évolution. L'une de ses principales fonctionnalités est [Laravel Eloquent](#), un [mappeur objet-relationnel](#) (ORM) qui permet aux développeurs d'effectuer efficacement des opérations de création, de lecture, de mise à jour et de suppression (Create, Read, Update, Delete ou CRUD) sur une base de données.

Ce tutoriel montre comment effectuer ces opérations dans votre application Laravel en utilisant l'ORM Eloquent de Laravel et comment déployer votre application CRUD Laravel en utilisant [MyKinsta](#).

## Fonctionnalité CRUD dans Laravel

Les opérations CRUD sont l'épine dorsale de toute application basée sur une base de données. Elles vous permettent d'effectuer les opérations les plus élémentaires et les plus essentielles de la base de données, telles que la création de nouveaux enregistrements, la lecture des enregistrements existants, leur mise à jour et leur suppression. Ces opérations sont cruciales pour la fonctionnalité de toute [application Laravel](#) qui interagit avec une base

de données.

Eloquent fournit un moyen simple et intuitif d'interagir avec la base de données en réduisant la complexité de la gestion de la base de données afin que vous puissiez vous concentrer sur la construction de votre application. Ses méthodes et classes intégrées vous permettent d'accéder facilement aux enregistrements de la base de données.

## Pré-requis

Pour suivre ce tutoriel, assurez-vous d'avoir les éléments suivants :

- [XAMPP](#)
- [Composer](#)
- [Un compte MyKinsta](#)
- Un compte sur [GitHub](#), [GitLab](#), ou [Bitbucket](#) pour pousser votre code
- [Bootstrap version 5](#)

## Étapes

1. Installez Laravel et créez une nouvelle application
2. Créez une base de données
3. Créez une table
4. Créez un contrôleur
5. Configurez le modèle
6. Ajouter une route
7. Générer des fichiers Blade
8. Déployez et testez votre application CRUD

Pour vous guider tout au long du processus, consultez le [code complet](#) du tutoriel.

## Installer Laravel et créer une nouvelle application

Ouvrez le terminal dans lequel vous souhaitez créer votre application Laravel et suivez les étapes ci-dessous.

1. Pour installer Laravel, exécutez :

```
composer global require laravel/installer
```

2. Pour créer une nouvelle application Laravel, exécutez :

```
laravel new crudposts
```

## Créer une base de données

Pour créer une nouvelle base de données pour votre application :

1. Démarrez les serveurs Apache et MySQL dans le panneau de contrôle XAMPP et visitez `http://localhost/phpmyadmin` dans votre navigateur.
2. Cliquez sur **Nouveau** dans la colonne latérale de gauche. Vous devriez voir ce qui suit :

# Databases

Create database

Database name

utf8mb4\_general\_ci

Create

— Le formulaire de création de base de données.

3. Ajoutez un nom de base de données et cliquez sur **Créer**.
4. Modifiez le fichier **.env** de votre application à la racine de votre application Laravel. Il contient toutes les variables d'environnement utilisées par l'application. Localisez les variables préfixées par `DB_` et modifiez-les avec les informations d'identification de votre base de données :

```
DB_CONNECTION=  
DB_HOST=  
DB_PORT=  
DB_DATABASE=  
DB_USERNAME=  
DB_PASSWORD=
```

## Créer une table

Les lignes de données de votre application sont stockées dans des tables. Pour cette application, vous n'avez besoin que d'une seule table, créée à l'aide des [migrations Laravel](#).

1. Pour créer une table et générer un fichier de migration à l'aide de l'interface en ligne de commande de Laravel, Artisan, exécutez :

```
php artisan make:migration create_posts_table
```

La commande ci-dessus crée un nouveau fichier,

**yyyy\_mm\_dd\_hhmmss\_create\_posts\_table.php**, dans **database/migrations**.

2. Ouvrez le fichier **yyyy\_mm\_dd\_hhmmss\_create\_posts\_table.php** pour définir les colonnes que vous voulez dans votre table de base de données dans la fonction up :

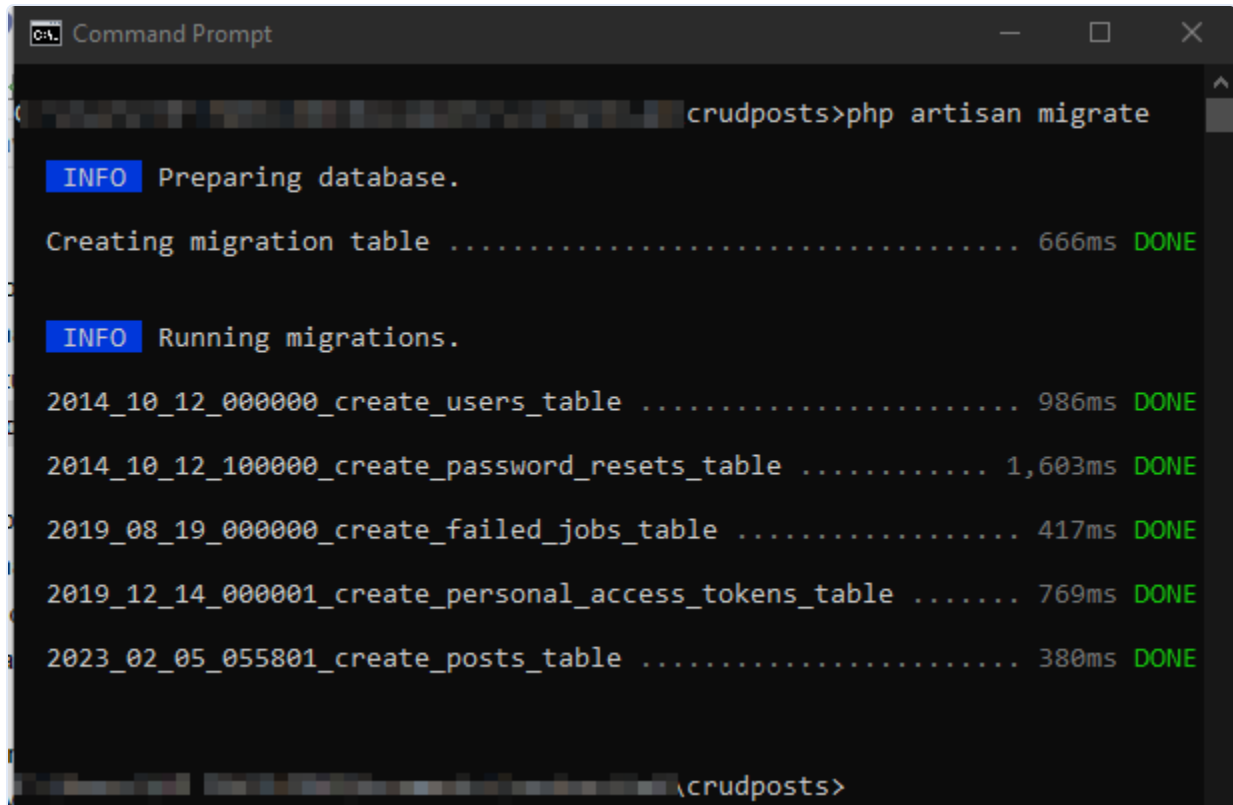
```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('body');
        $table->timestamps();
    });
}
```

Ce code définit le contenu de la table posts. Elle comporte quatre colonnes : `id`, `title`, `body`, et `timestamps`.

3. Exécutez les fichiers de migration dans le dossier **database/migrations** pour créer des tables dans la base de données :

```
php artisan migrate
```

Le résultat ressemble à ceci :



```
C:\> Command Prompt
crudposts>php artisan migrate

INFO  Preparing database.
Creating migration table ..... 666ms DONE

INFO  Running migrations.
2014_10_12_000000_create_users_table ..... 986ms DONE
2014_10_12_100000_create_password_resets_table ..... 1,603ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 417ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 769ms DONE
2023_02_05_055801_create_posts_table ..... 380ms DONE

crudposts>
```

— Exécution des migrations.

4. Accédez à la base de données que vous avez créée précédemment pour confirmer que vous avez créé les tables :

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> failed_jobs	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<input type="checkbox"/> migrations	★ Browse Structure Search Insert Empty Drop	5	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> password_resets	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> personal_access_tokens	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	48.0 KiB	-
<input type="checkbox"/> posts	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> users	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<b>6 tables</b>	<b>Sum</b>	<b>5</b>	<b>InnoDB</b>	<b>utf8mb4_general_ci</b>	<b>160.0 KiB</b>	<b>0 B</b>

— Tables créées.

## Créer un contrôleur

Le contrôleur contient toutes les fonctions pour CRUD les posts de la base de données.

Générez un fichier de contrôleur dans votre application Laravel en utilisant Artisan :

```
php artisan make:controller PostController --api
```

L'exécution de cette commande crée un fichier **PostController.php** dans **app/Http/Controllers**, avec un code passe-partout et des déclarations de fonctions vides `index`, `store`, `show`, `update`, et `destroy`.

## Créer des fonctions

Ensuite, créez les fonctions qui stockent, indexent, mettent à jour, détruisent, créent, affichent et modifient les données.

Vous pouvez les ajouter au fichier [app/Http/Controller/PostController.php](#) illustré ci-dessous.



## La fonction store

La fonction `store` ajoute un message à la base de données.

Allez jusqu'à la fonction `store` et ajoutez le code suivant à l'intérieur des accolades vides :

```
$request->validate([
    'title' => 'required|max:255',
    'body' => 'required',
]);
Post::create($request->all());
return redirect()->route('posts.index')
    ->with('success','Post created successfully.');
```

Ce code prend un objet contenant le titre et le corps de l'article, valide les données, ajoute un nouvel article à la base de données si les données sont valides et redirige l'utilisateur vers la page d'accueil avec un message de réussite.

## La fonction index

La fonction `index` récupère tous les messages de la base de données et envoie les données au fichier [posts.index](#) à la page `posts.index`.

## La fonction update

La fonction `<update` contient le `id` de l'article à mettre à jour, le nouvel article `title` et le `body`. Après avoir validé les données, elle recherche l'article ayant le même `id`. Si elle le trouve, la fonction `update` met à jour le message dans la base de données avec les nouveaux `title` et `body`. Ensuite, elle redirige l'utilisateur vers la page d'accueil avec un message de réussite.

## La fonction `destroy`

La fonction `destroy` trouve un message avec le nom `id` et le supprime de la base de données, puis redirige l'utilisateur vers la page d'accueil avec un message de succès.

Les fonctions ci-dessus sont les fonctions utilisées pour CRUD les messages de la base de données. Cependant, vous devez définir d'autres fonctions dans le contrôleur pour rendre les pages nécessaires dans **`resources/views/posts/`**.

## La fonction `create`

La fonction `create` rend la page <resources/views/posts/create.blade.php> qui contient le formulaire d'ajout d'articles à la base de données.

## La fonction `show`

La fonction `show` trouve un article avec l'adresse `id` dans la base de données et affiche la page <resources/views/posts/show.blade.php> avec l'article.

## La fonction `edit`

La fonction `edit` trouve un article avec l'adresse `id` dans la base de données et affiche le fichier `resources/views/posts/show.blade.php` avec le message <resources/views/posts/edit.blade.php> avec les détails de l'article à l'intérieur d'un formulaire.

## Mise en place du modèle

Le modèle `Post` interagit avec la table **`posts`** de la base de données.

1. Pour créer un modèle avec Artisan, exécutez :

```
php artisan make:model Post
```

Ce code crée un fichier **Post.php** dans le dossier **App/Models**.

2. Créez un tableau `<fillable`. Ajoutez le code suivant dans la classe `Post` et sous la ligne `use HasFactory;`

```
protected $fillable = [  
    'title',  
    'body',  
];
```

Ce code crée un tableau `fillable` qui vous permet d'ajouter des éléments à la base de données depuis votre application Laravel.

3. Connectez le modèle `Post` au fichier **PostController.php**. Ouvrez **PostController.php** et ajoutez la ligne ci-dessous sous `use Illuminate\Http\Request;`. Elle se présente comme suit :

```
use Illuminate\Http\Request;  
use App\Models\Post;
```

Le fichier **PostController.php** devrait maintenant ressembler à ceci :

```

<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Models\Post;
class PostController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $posts = Post::all();
        return view('posts.index', compact('posts'));
    }
    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $request->validate([
            'title' => 'required|max:255',
            'body' => 'required',
        ]);
        Post::create($request->all());
        return redirect()->route('posts.index')
            ->with('success', 'Post created successfully.');
```

```

    * @return \Illuminate\Http\Response
    */
public function update(Request $request, $id)
{
    $request->validate([
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
    $post = Post::find($id);
    $post->update($request->all());
    return redirect()->route('posts.index')
        ->with('success', 'Post updated successfully.');
```

```

}
/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    $post = Post::find($id);
    $post->delete();
    return redirect()->route('posts.index')
        ->with('success', 'Post deleted successfully');
```

```

}
// routes functions
/**
 * Show the form for creating a new post.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    return view('posts.create');
```

```

}
/**
 * Display the specified resource.
 *
```

```

    * @param int $id
    * @return \Illuminate\Http\Response
    */
    public function show($id)
    {
        $post = Post::find($id);
        return view('posts.show', compact('post'));
    }
    /**
     * Show the form for editing the specified post.
     *
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        $post = Post::find($id);
        return view('posts.edit', compact('post'));
    }
}

```

## Ajouter des routes

Après avoir créé les fonctions du contrôleur et le modèle `Post`, vous devez ajouter des routes pour les fonctions de votre contrôleur.

1. Ouvrez **routes/web.php** et supprimez la route « boilerplate » générée par l'application. Remplacez-la par le code ci-dessous pour connecter les fonctions du contrôleur à leurs routes respectives :

```

// returns the home page with all posts
Route::get('/', PostController::class . '@index')->name('posts.index');

```

```
// returns the form for adding a post
Route::get('/posts/create', PostController::class . '@create')->name('pos
// adds a post to the database
Route::post('/posts', PostController::class . '@store')->name('posts.store
// returns a page that shows a full post
Route::get('/posts/{post}', PostController::class . '@show')->name('posts.
// returns the form for editing a post
Route::get('/posts/{post}/edit', PostController::class . '@edit')->name('p
// updates a post
Route::put('/posts/{post}', PostController::class . '@update')->name('post
// deletes a post
Route::delete('/posts/{post}', PostController::class . '@destroy')->name('p
```

2. Pour connecter les routes, ouvrez **app/Http/Controllers/PostController.php** et ajoutez la ligne ci-dessous sous la ligne `use Illuminate\Support\Facades\Route;`

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\PostController;
```

Le fichier **routes/web.php** devrait maintenant ressembler à ceci :

```
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\PostController;
// returns the home page with all posts
Route::get('/', PostController::class . '@index')->name('posts.index');
// returns the form for adding a post
Route::get('/posts/create', PostController::class . '@create')->name('pos
```

```
// adds a post to the database
Route::post('/posts', PostController::class . '@store')->name('posts.store');
// returns a page that shows a full post
Route::get('/posts/{post}', PostController::class . '@show')->name('posts.show');
// returns the form for editing a post
Route::get('/posts/{post}/edit', PostController::class . '@edit')->name('posts.edit');
// updates a post
Route::put('/posts/{post}', PostController::class . '@update')->name('posts.update');
// deletes a post
Route::delete('/posts/{post}', PostController::class . '@destroy')->name('posts.destroy');
```

## Générer les fichiers Blade

Maintenant que vous avez les routes, vous pouvez créer les fichiers [Laravel Blade](#). Avant d'utiliser Artisan pour générer les fichiers Blade, créez la commande `make:view`, avec laquelle vous pouvez générer les fichiers **blade.php**.

1. Exécutez le code suivant dans le CLI pour créer un fichier de commande **MakeViewCommand** dans le dossier **app/Console/Commands** :

```
php artisan make:command MakeViewCommand
```

2. Créez une commande pour générer des fichiers **.blade.php** à partir de l'interface de programmation en remplaçant le code du fichier **MakeViewCommand** par le suivant :



```

<?php
namespace App\Console\Commands;
use Illuminate\Console\Command;
use File;
class MakeViewCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'make:view {view}';
    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Create a new blade template.';
    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $view = $this->argument('view');
        $path = $this->viewPath($view);
        $this->createDir($path);
        if (File::exists($path))
        {
            $this->error("File {$path} already exists!");
            return;
        }
        File::put($path, $path);
        $this->info("File {$path} created.");
    }
    /**

```

```

    * Get the view full path.
    *
    * @param string $view
    *
    * @return string
    */
public function viewPath($view)
{
    $view = str_replace('.', '/', $view) . '.blade.php';
    $path = "resources/views/{ $view }";
    return $path;
}
/**
 * Create a view directory if it does not exist.
 *
 * @param $path
 */
public function createDir($path)
{
    $dir = dirname($path);
    if (!file_exists($dir))
    {
        mkdir($dir, 0777, true);
    }
}
}

```

## Créer une page d'accueil

Créez ensuite votre page d'accueil. La page d'accueil est le fichier **index.blade.php**, qui répertorie tous les articles.

1. Pour créer la page d'accueil, exécutez :

```
php artisan make:view posts.index
```

Cela crée un dossier **posts** dans le dossier **/resources/views** et un fichier **index.blade.php** en dessous. Le chemin résultant est **/resources/views/posts/index.blade.php**.

2. Ajoutez le code suivant dans le fichier **index.blade.php**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" integrity="sha384-GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1zL3/Jr59b6EGGoI1aFkw7O" rel="stylesheet">
  <title>Posts</title>
</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-warning">
    <div class="container-fluid">
      <a class="navbar-brand h1" href={{ route('posts.index') }}>CRUDPosts</a>
      <div class="justify-end">
        <div class="col">
          <a class="btn btn-sm btn-success" href={{ route('posts.create') }}>Create</a>
        </div>
      </div>
    </div>
  </nav>
  <div class="container mt-5">
    <div class="row">
      @foreach ($posts as $post)
```

```

<div class="col-sm">
  <div class="card">
    <div class="card-header">
      <h5 class="card-title">{{ $post->title }}</h5>
    </div>
    <div class="card-body">
      <p class="card-text">{{ $post->body }}</p>
    </div>
    <div class="card-footer">
      <div class="row">
        <div class="col-sm">
          <a href="{{ route('posts.edit', $post->id) }}"
            class="btn btn-primary btn-sm">Edit</a>
        </div>
        <div class="col-sm">
          <form action="{{ route('posts.destroy', $post->id) }}"
            @csrf
            @method('DELETE')
            <button type="submit" class="btn btn-danger btn-sm">
              Supprimer
            </button>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
@endforeach
</div>
</body>
</html>

```

Le code ci-dessus crée une page HTML simple qui utilise [Bootstrap](#) pour le style. Il établit une barre de navigation et un modèle de grille qui répertorie tous les articles de la base de données avec des détails et deux boutons d'action – modifier et supprimer – en utilisant l'aide `@foreach` Blade.

Le bouton **Modifier** permet à l'utilisateur d'accéder à la page **Modifier l'article**, où il peut modifier l'article. Le bouton **Supprimer** supprime l'article de la base de données en utilisant

```
{{ route('posts.destroy', $post->id) }} avec une méthode DELETE.
```

**Remarque** : le code de la barre de navigation pour tous les fichiers est le même que celui du fichier précédent.

3. Créez la page **create.blade.php**. Le fichier Blade appelé **create** ajoute des articles à la base de données. Utilisez la commande suivante pour générer le fichier :

```
php artisan make:view posts.create
```

Cela génère un fichier **create.blade.php** dans le dossier **/resources/views/posts**.

4. Ajoutez le code suivant au fichier **create.blade.php**:

```
// same as the previous file. Add the following after the nav tag and before the main content
<div class="container h-100 mt-5">
  <div class="row h-100 justify-content-center align-items-center">
    <div class="col-10 col-md-8 col-lg-6">
      <h3>Add a Post</h3>
      <form action="{{ route('posts.store') }}" method="post">
        @csrf
        <div class="form-group">
          <label for="title">Title</label>
          <input type="text" class="form-control" id="title" name="title">
        </div>
        <div class="form-group">
          <label for="body">Body</label>
          <textarea class="form-control" id="body" name="body" rows="3">
        </div>
        <br>
```

```

        <button type="submit" class="btn btn-primary">Create Post</button>
    </form>
</div>
</div>
</div>

```

Le code ci-dessus crée un formulaire avec les champs `title` et `body` et un bouton `submit` pour ajouter un message à la base de données via l'action `{{ route('posts.store') }}` avec une méthode `POST`.

5. Créez la page **Modifier l'article** pour modifier les articles dans la base de données. Utilisez la commande suivante pour générer le fichier :

```
php artisan make:view posts.edit
```

Cela crée un fichier **edit.blade.php** dans le dossier **/resources/views/posts**.

6. Ajoutez le code suivant au fichier **edit.blade.php**:

```

<div class="container h-100 mt-5">
    <div class="row h-100 justify-content-center align-items-center">
        <div class="col-10 col-md-8 col-lg-6">
            <h3>Update Post</h3>
            <form action="{{ route('posts.update', $post->id) }}" method="post"
                @csrf
                @method('PUT')
            >
                <div class="form-group">

```

```

        <label for="title">Title</label>
        <input type="text" class="form-control" id="title" name="title"
            value="{{ $post->title }}" required>
    </div>
    <div class="form-group">
        <label for="body">Body</label>
        <textarea class="form-control" id="body" name="body" rows="3" r
    </div>
    <button type="submit" class="btn mt-3 btn-primary">Update Post</b
</form>
</div>
</div>
</div>

```

Le code ci-dessus crée un formulaire avec des champs `title` et `body` et un bouton de soumission pour modifier un article avec le numéro spécifié `id` dans la base de données via l'action `{{ route('posts.update') }}` avec une méthode `PUT`.

7. Redémarrez ensuite votre serveur d'application à l'aide du code ci-dessous :

```
php artisan serve
```

Visitez `http://127.0.0.1:8000` sur votre navigateur pour voir votre nouveau blog. Cliquez sur le bouton **Ajouter un article** pour ajouter de nouveaux articles.

## Déployer et tester votre application CRUD

Préparez votre application pour le déploiement comme suit.

1. Facilitez le déploiement en déclarant le dossier public. Ajoutez un fichier **.htaccess** à la racine du dossier de l'application avec le code suivant :

```
<IfModule mod_rewrite.c >
  RewriteEngine On
  RewriteRule ^(.*)$ public/$1 [L]
</IfModule >
```

2. Forcez votre application à utiliser HTTPS en ajoutant le code suivant au-dessus de vos routes dans le fichier **routes/web.php**:

```
use Illuminate\Support\Facades\URL;
URL::forceScheme('https');
```

3. Poussez votre code vers un dépôt Git. Kinsta prend en charge les déploiements à partir de GitHub, GitLab ou Bitbucket.


## Configurer un projet sur MyKinsta

1. Créez un compte [MyKinsta](#) si vous n'en avez pas déjà un.
2. Connectez-vous à votre compte et cliquez sur le bouton **Ajouter un service** dans le tableau de bord pour créer une nouvelle application.
3. Si vous ne connaissez pas encore l'application, connectez-vous à votre compte GitHub, GitLab ou Bitbucket et donnez des autorisations spécifiques.




4. Remplissez le formulaire et ajoutez la valeur `APP_KEY`. Vous pouvez trouver la valeur correspondante dans votre fichier `.env`.
5. Sélectionnez vos ressources de construction et si vous voulez utiliser le chemin de construction de votre application ou construire votre application avec Dockerfile. Pour cette démonstration, laissez MyKinsta construire l'application sur la base de votre dépôt.
6. Spécifiez les différents processus que vous voulez exécuter pendant le déploiement. Vous pouvez laisser ce champ vide à ce stade.
7. Enfin, ajoutez votre moyen de paiement.

Après avoir confirmé votre moyen de paiement, MyKinsta déploie votre application et vous attribue une URL comme indiqué ci-dessous :

By	Deploy started	Deploy time
 multi-Adams	Mar 5, 2023, 5:01 PM	1m 41s
Data center location		
europe-west2		
Commit message		
fixes		


### Deployment progress



Build process completed

Mar 5, 2023, 5:01 PM


▼




Rollout process completed

Mar 5, 2023, 5:02 PM

▼



Deployment available at [crudposts-94z41.kinsta.app](https://crudposts-94z41.kinsta.app) 

Mar 5, 2023, 5:02 PM

[View runtime logs](#)

— Déploiement réussi.

Vous pouvez visiter le lien, mais vous obtenez une page 500 | `Server Error` parce que l'application a besoin d'une connexion valide à la base de données pour fonctionner. La section suivante résout ce problème.

## Créer une base de données via MyKinsta

1. Pour créer une base de données, allez sur votre tableau de bord MyKinsta et cliquez sur **Ajouter un service**.
2. Sélectionnez **Base de données** et remplissez le formulaire avec le nom et le type de votre base de données, votre nom d'utilisateur et votre mot de passe. Ajoutez un centre de données et une taille de base de données adaptée à votre application.
3. La page suivante affiche le récapitulatif des coûts et votre moyen de paiement. Cliquez sur **Créer une base de données** pour terminer la procédure.
4. Après avoir créé la base de données, MyKinsta vous redirige vers votre liste de services. Cliquez sur la base de données que vous venez de créer et descendez jusqu'à **Connexions externes**. Copiez les informations d'identification de la base de données.
5. Retournez à la page **Déploiement** de l'application et cliquez sur **Réglages**. Ensuite, faites défiler vers le bas jusqu'à **Variables d'environnement** et cliquez sur **Ajouter des variables d'environnement**. Ajoutez les informations d'identification de la base de données en tant que variables d'environnement dans l'ordre suivant :

```
DB_CONNECTION=mysql
DB_HOST=External hostname
DB_PORT=External port
DB_DATABASE=Database name
DB_USERNAME=Username
DB_PASSWORD=Password
```

La liste des variables d'environnement de l'application devrait maintenant ressembler à ceci :

<input type="checkbox"/>	Key ↑	Value	Available	
<input type="checkbox"/>	APP_KEY	.....	Build & Runtime	
<input type="checkbox"/>	APP_URL	.....	Build & Runtime	
<input type="checkbox"/>	DB_CONNECTION	.....	Build & Runtime	
<input type="checkbox"/>	DB_DATABASE	.....	Build & Runtime	
<input type="checkbox"/>	DB_HOST	.....	Build & Runtime	
<input type="checkbox"/>	DB_PASSWORD	.....	Build & Runtime	
<input type="checkbox"/>	DB_PORT	.....	Build & Runtime	
<input type="checkbox"/>	DB_USERNAME	.....	Build & Runtime	

— La liste des variables .env.

- Allez sur la page **Déploiements** de votre application et déployez manuellement votre application en cliquant sur **Déployer maintenant** pour appliquer ces changements. Jusqu'à présent, vous avez créé une base de données et l'avez connectée à votre application.

7. Enfin, pour créer les tables de votre base de données MyKinsta, connectez la base de données à votre application locale en mettant à jour votre fichier `.env` avec les mêmes informations d'identification que vous avez saisies dans votre application MyKinsta et exécutez la commande suivante :

```
php artisan migrate
```

Cette commande exécute tous les fichiers de migration. Elle crée toutes les tables définies dans votre application MyKinsta.

Maintenant, vous pouvez tester votre application avec l'URL assignée après le premier déploiement.

## Résumé

Laravel est un framework complet pour créer des applications web robustes et évolutives qui nécessitent des fonctionnalités CRUD. Grâce à sa syntaxe intuitive et à ses puissantes fonctionnalités, Laravel facilite l'intégration d'opérations CRUD dans votre application.

Cet article couvre les concepts fondamentaux des opérations CRUD et la manière de les mettre en œuvre en utilisant les fonctionnalités intégrées de Laravel. Il explique également :

- Comment créer une base de données dans MyKinsta et la connecter à votre application
- Comment utiliser les migrations de Laravel pour définir la table de la base de données, créer le fichier du contrôleur et ses fonctions
- Définir un modèle et le connecter au contrôleur. Le routage de Laravel génère des fichiers Blade pour créer les pages et formulaires correspondants et pour déployer et tester l'application en utilisant MyKinsta

Maintenant que vous avez vu à quel point il est facile d'effectuer des opérations CRUD dans Laravel, [consultez MyKinsta](#) pour le développement et l'[hébergement d'applications](#) web.