Curso de laravel versión 8

* **Rutas y Controladores**

/\*NOTA: laravel lee las rutas de arriba hacia abajo en el caso de que dos rutas sean parecidas como **'cursos/create'** y **'cursos/{curso}'** debemos tener en cuenta el orden en que las colocamos\*/

/\*para no ensuciar tanto nuestro código de rutas lo que hacemos en aplicar esa lógica

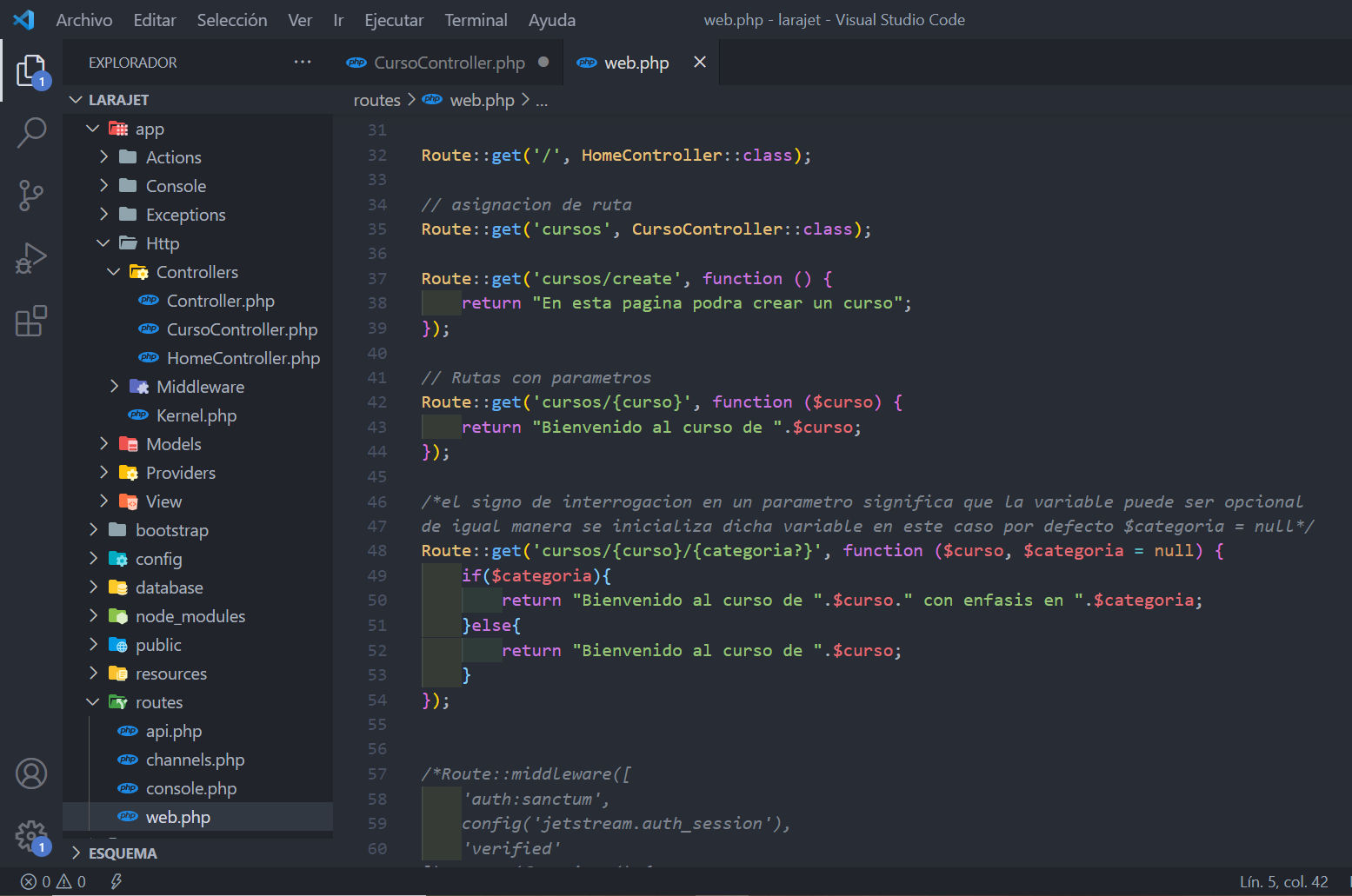
desde el controlador con una función de la cual haremos uso agregando a la ruta el controlador indicado en ves de la lógica directamente ejemplo:

**Route::get('/', HomeController::class);** \*/

Anteriormente desde la versión 7 de laravel para abajo

Se utilizaba la nomenclatura **Route::get('/', ‘HomeController’)**

Pero ya desde laravel 8 se realiza de la forma como se muestra en la imagen.



Para crear nuestros controladores vamos a la terminal de comandos entramos a la ruta de nuestro proyecto y escribimos el siguiente comando:

- **php artisan make:controller nombre\_del\_controlador**

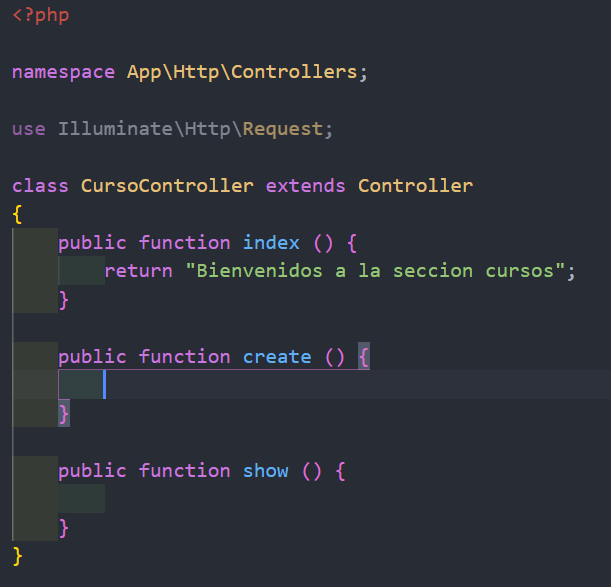
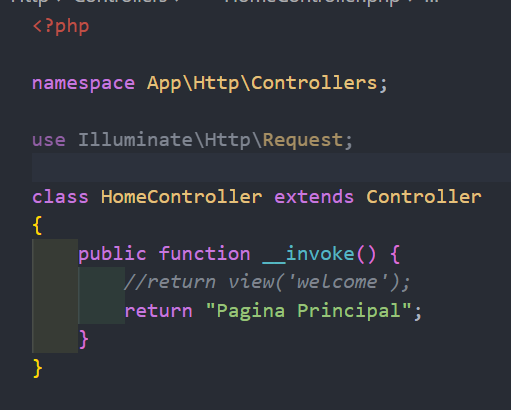
comando para crear controlador y por convención pegado al nombre se le agrega Controller

El método **\_\_invoke()** dentro del controlador se utiliza cuando queremos que este administre una sola ruta en caso contrario se crea los distintos métodos que se pueden llamar de la forma que queramos, pero por convención

al método encargado de mostrar la página principal se suele llamar **index ().**

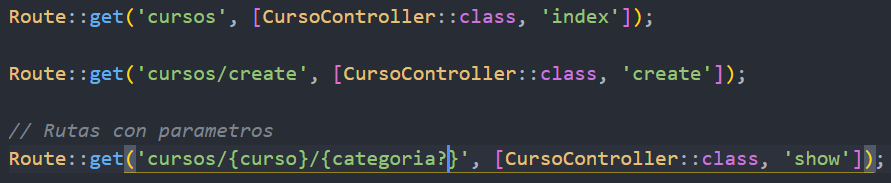
al método encargado de mostrar el formulario para crear se suele llamar **create ().**

al método encargado de mostrar un elemento en particular se suele llamar **show ().**

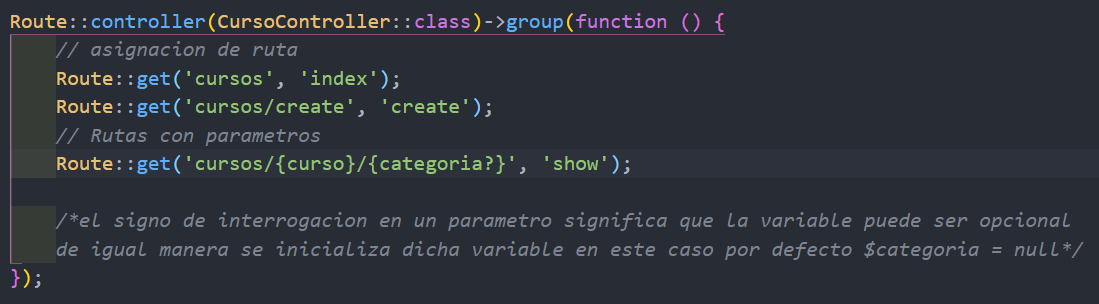


Para hacer uso de los métodos de un controlador por medio de la ruta se utiliza la siguiente

Nomenclatura ejemplo:



Para hacer el código mas ordenado otras de las novedades que trae específicamente laravel en su versión 8.81 para arriba es el poder agrupar las rutas que comparten un mismo controlador como observaremos en la siguiente imagen



Se referencia el controlador seguido de la función para agrupar:

**Route::controller(CursoController::class)->group(function () {})**

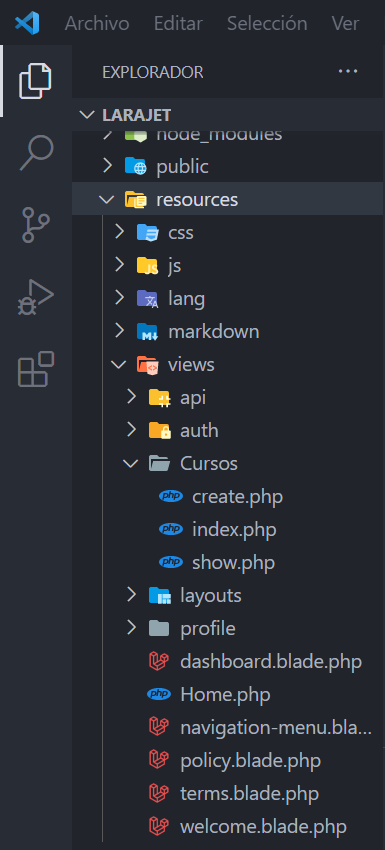
Al hacer esto ya no es necesario referenciar el controlador por parámetro en cada ruta, simplemente se colocaría la función que utilizara la ruta.

* **Vistas en laravel**

Laravel nos crea una estructura de carpetas en este caso para las vistas ingresamos a la carpeta **resources** esta nos despliega otras carpetas, pero nos dirigimos a la que dice **views** que es donde se encuentran alojadas las vistas de nuestro proyecto.

Dentro de views creamos nuestra vista principal seguido de esto para llevar todo mas organizado creamos subcarpetas para ir diferenciando las diferentes vistas que vayamos creando.

Al momento de crear las vistas de cada controlador laravel 8 nos recomienda por convención llamarlas como los métodos a los cuales están referenciados, no es obligatorio podemos llamar las vistas como queramos.





La idea es invocar cada vista desde el controlador con el método que le corresponda, esto lo hacemos de la siguiente forma





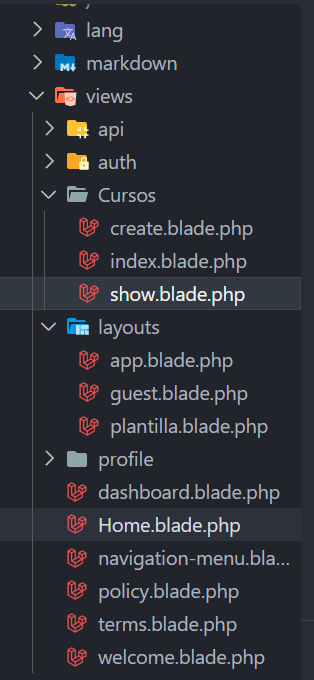
En la anterior imagen también podemos observar cómo se envían los parámetros a una vista desde el controlador por medio del método **view(),** los parámetros se envían dentro de un array referenciando la variable con cualquier palabra en el siguiente ejemplo se pone igual a la variable.

Ej:

**view(‘Cursos/show’,[‘curso’=>$curso,’categoria’=>$categoria]);**

Laravel maneja el sistema de plantillas Blade para la creación de las vistas y facilitar la creación de estas.

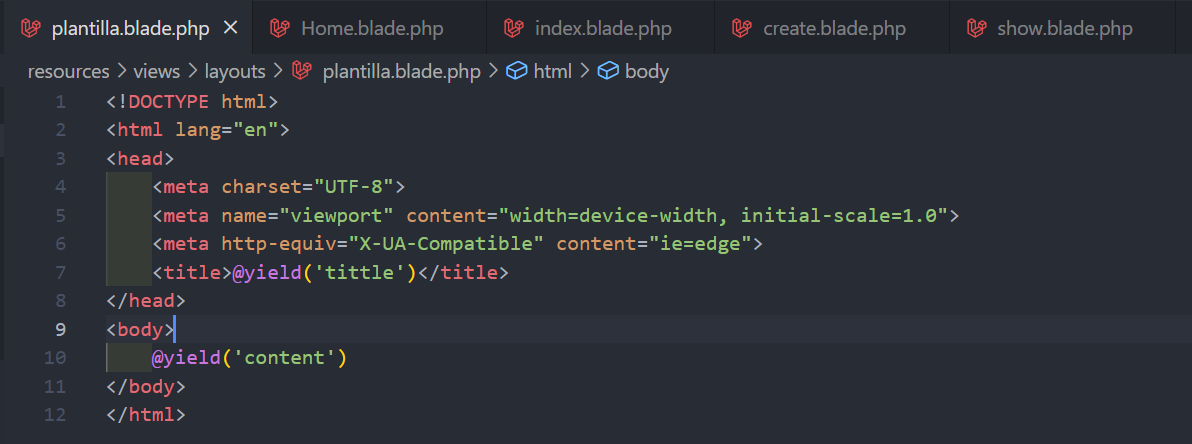
Una de ventajas de Blade al momento del hacer nuestras vistas es que podemos crear plantillas aparte las cuales podemos usar en distintas vistas que compartan esta en común, evitando así repetir código en nuestro HTML teniendo así un trabajo más limpio y organizado.





Como se observa en la imagen anterior las extensiones de nuestras vistas las cambiamos a la directiva **.blade.php** además de que se agrego la carpera **layout** que donde crearemos nuestras plantillas de forma organizada.

En la siguiente imagen observaremos el ejemplo de una plantilla creada la cual se utilizará en varias vistas:

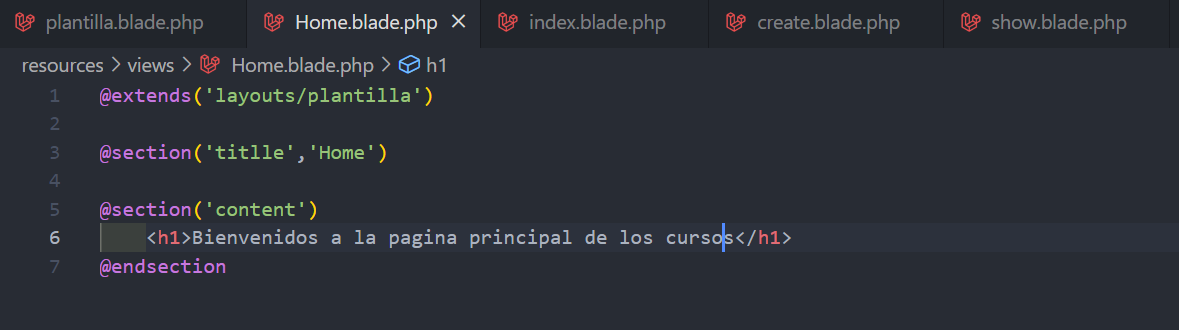




Como se observa en la imagen anterior se creo la plantilla y encontramos el uso de la función **@yield()** que nos proporciona Blade, con esta podemos darle dinamismo a la plantilla dependiendo en que vista se utilice haciendo el contenido dinámico.

Dentro de la función @yield() colocamos encerrado en comillas una variable de referencia la cual nos servirá para poder agregar distinto contenido en las distintas vistas como veremos en la siguiente imagen.



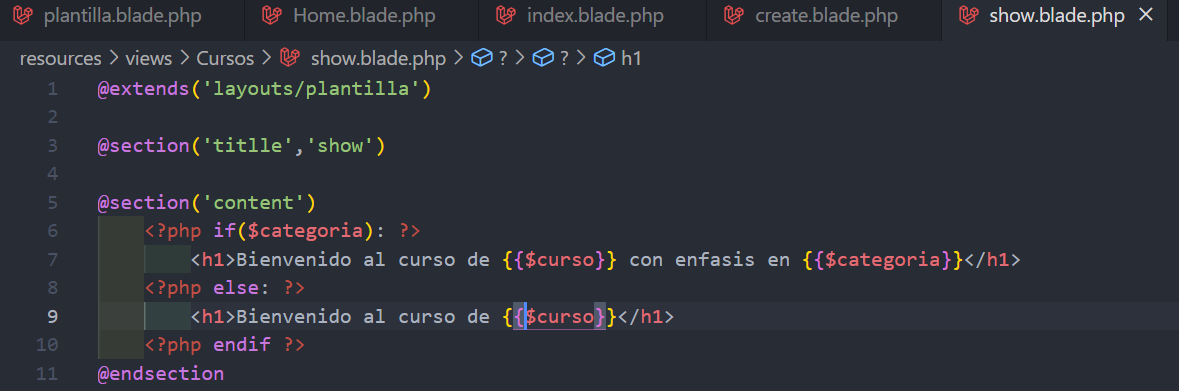




En la anterior imagen podemos ver como importamos nuestra plantilla a la vista Home haciendo uso de la función @extends() que recibe por parámetro la ruta de la plantilla a utilizar, luego de eso lo siguiente es hacer uso de la funcion **@section()** que recible por parámetro la variable de referencia que colocamos en nuestra plantilla y como segundo parámetro el valor que le vamos a dar como se ve en la imagen.

En el segundo **@section()** se observa que se hace un cierre con **@endsection,** esto se hacer para colocar contenido extenso o en cadena dentro de esa section como vemos se coloca la variable de referencia como parámetro y lego se agrega el HTML con la información que queremos mostrar.

Blade también nos brinda la siguiente directiva y el poder interactuar con las variables que vienen desde el controlador haciendo uso de las llaves dobles {{}} como por ejemplo.

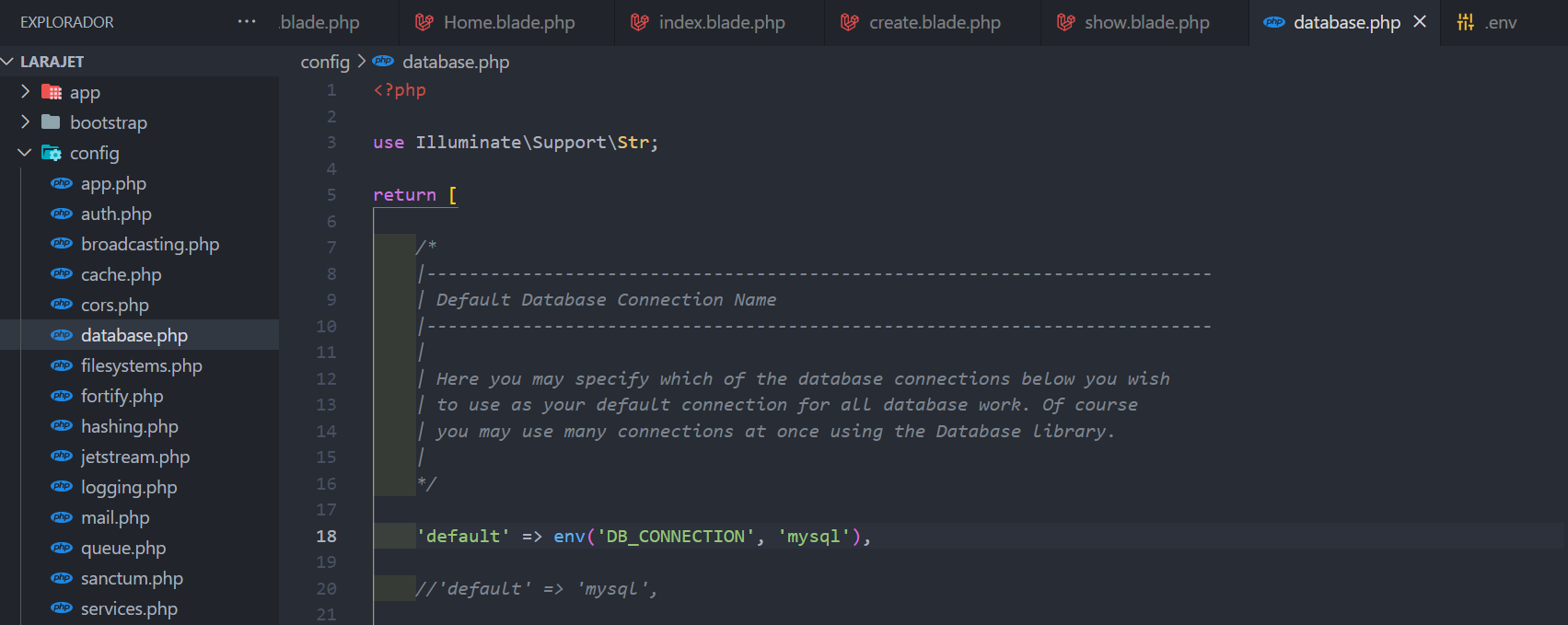




* **Manejo de Bases de datos con laravel**

Laravel maneja una carpeta llamada config donde encontraremos todo los relacionado a la configuración de nuestro proyecto.

Dentro de esta carpeta se encuentra el archivo llamado **database.php** donde podemos observar la configuración de nuestra base de datos y el tipo de base de datos que estemos utilizando por ejemplo si es MySQL o SQL Server, etc… cualquiera con las que laravel sea compatible. Se debe especificar el tipo de base de datos que vayamos a utilizar, esto lo hacemos de la siguiente forma como veremos en la siguiente imagen.





Dentro del mismo archivo database.php podemos encontrar las configuraciones del tipo de base de datos que hayamos puesto por default, en este caso como se observa en la imagen se utiliza **mysql** que por lo general ya viene por defecto en nuestro proyecto de laravel.

**'default' => env('DB\_CONNECTION', 'mysql'),**

En la siguiente imagen se apreciará la configuración.





En este caso como tenemos seleccionado mysql por default buscamos la configuración de esta como se ve en la imagen.

Adelante se muestra una imagen de las configuraciones de otros tipos de bases de datos.



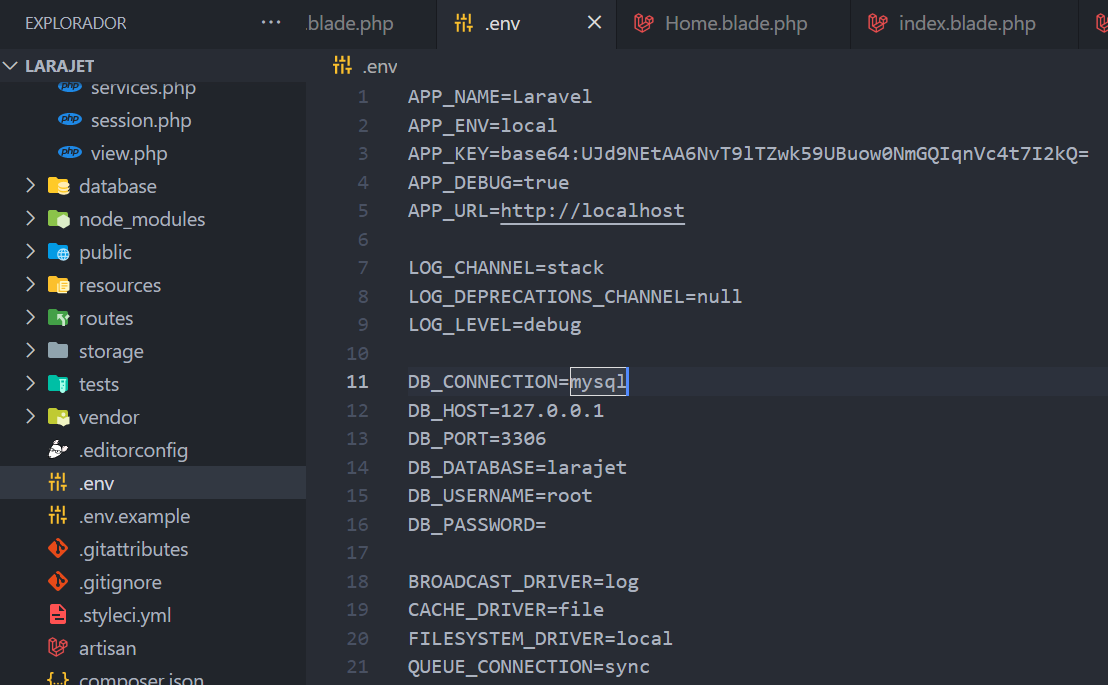
Base de datos [SQL Server](https://www.microsoft.com/es-es/sql-server/sql-server-2019)

Base de datos [PostgreSQL](https://www.postgresql.org/)



Ya habiendo ubicado la configuración de la base de datos que estemos utilizando podríamos configurar las credenciales y demás datos que necesitemos, pero al hacer esto dejaríamos nuestro proyecto con un grave problema de seguridad debido que al subir nuestro proyecto a cualquier repositorio como por ejemplo GitHub toda esta información podrá ser vista por cualquier persona que tenga acceso a ese repositorio y debemos recordar que estos son datos sensibles que no queremos que otras personas puedan verlo, entonces por esta razón laravel nos recomienda que al momento de colocar credenciales lo hagamos desde un archivo llamado **.env** que este nos ofrece y así realizar nuestra configuración de forma más segura.

En las imágenes anteriores se puede observar que en las configuraciones se hace uso de la función **env(‘DB\_DATABASE’, ‘forge’),** esta cuenta con dos valores o parámetros la cual hace referencia a que laravel va buscar el valor dentro del archivo .env del primer parámetro que hayamos colocado ejemplo DB\_DATABASE. De no encontrar el primer parámetro dentro del archivo .env la credencial toma por defecto el segundo parámetro.





La razón por la que es más seguro colocar las credenciales desde el archivo .env es por que laravel esta configurado para que en el caso de subir nuestro proyecto a algún repositorio se ignore el archivo .env y de esta forma todas mis credenciales están seguras.

* **Migraciones Introducción**

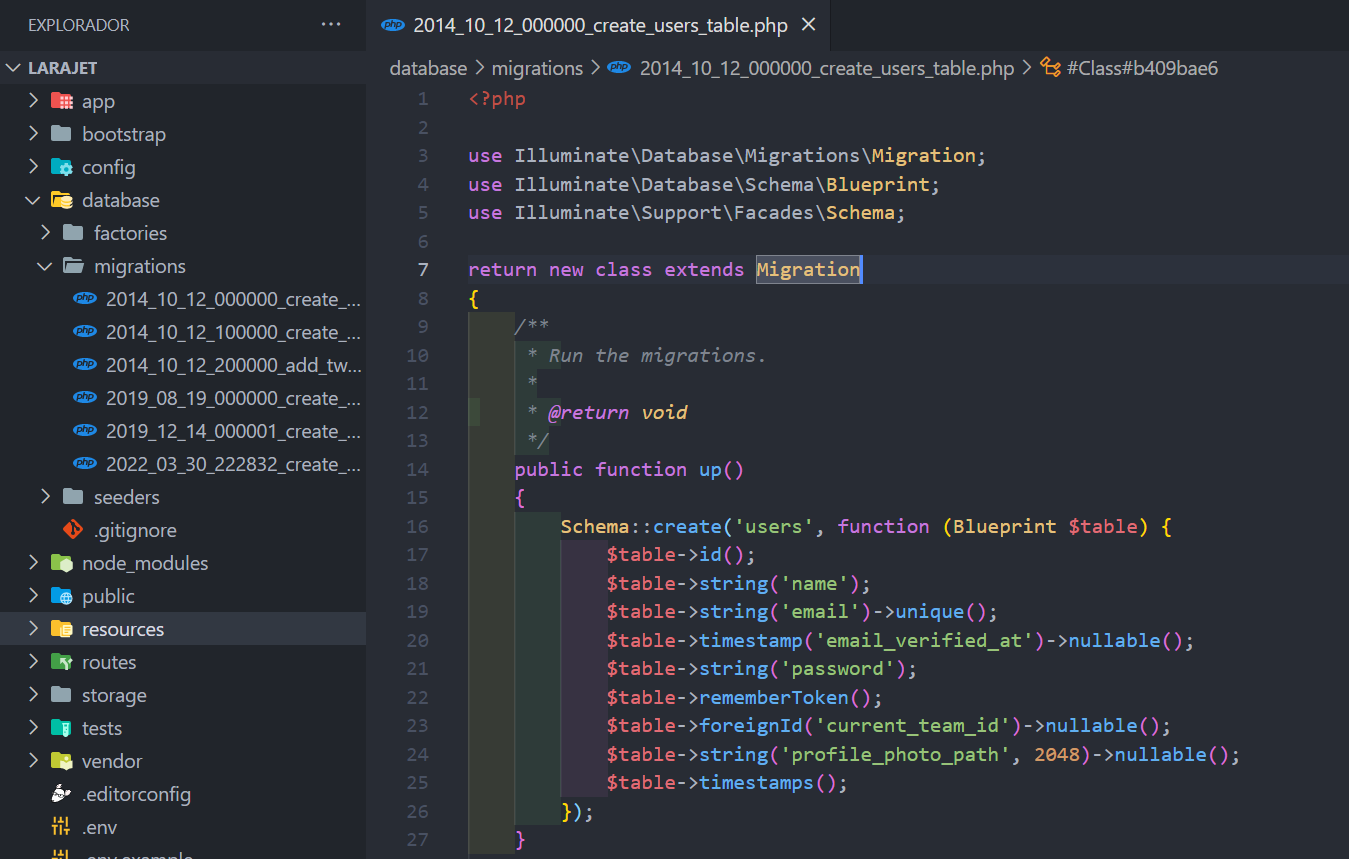
Laravel nos proporciona esta herramienta llamada migraciones la cual entre muchas ventajas nos permite crear las tablas de nuestra base de datos, estas se almacenan dentro de nuestro proyecto en la ruta database/migrations donde se pueden observar los archivos con el código de creación de cada tabla que nos genera laravel.

Al momento de subir nuestro proyecto a producción esto nos facilita la creación de la base de datos de forma automática evitando hacerlo de forma manual.

Las migraciones son como el control de versiones de su base de datos, lo que le permite a su equipo definir y compartir la definición del esquema de la base de datos de la aplicación. Si alguna vez tuvo que decirle a un compañero de equipo que agregue manualmente una columna a su esquema de base de datos local después de extraer los cambios del control de código fuente, se ha enfrentado al problema que resuelven las migraciones de base de datos.

Schema La fachada de Laravel proporciona soporte agnóstico de base de datos para crear y manipular tablas en todos los sistemas de base de datos compatibles con Laravel. Por lo general, las migraciones usarán esta fachada para crear y modificar tablas y columnas de bases de datos.

Cada modificación que queramos hacer a nuestra base de datos va quedar como un registro en las migraciones y en el caso de estar trabajando en equipo todas esos registros van a estar al alcance del equipo.



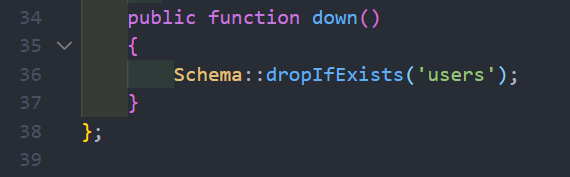


Como se observa en la imagen anterior se aprecia el uso de los métodos up() y down() el cual se mostrara en una imagen posteriormente.

Dentro del método **up()** vemos que se está ejecutando el método **create** de la clase **Shema,** el método create va recibir como parámetro una cadena la cual vendría siendo el nombre de la tabla que se va crear en el caso de la imagen se muestra ‘users’, como segundo parámetro de este método se pasa una función anónima la cual recibe un objeto llamado **$table** de tipo **Blueprint** – este objeto se utiliza para crear las columnas de nuestras tablas como se observa en la imagen.

Este objeto ejecuta distintos métodos los cuales podemos apreciar de forma mas clara en la documentación oficial de laravel, estos métodos se usan para definir el tipo de dato de cada columna además se le pueden agregar mas propiedades como por ejemplo la **unique()**, **nullable()** etc...

Con el método **up()** mencionado anteriormente se crea la tabla con todas sus columnas.



Con el método **down()** como se aprecia en la imagen se hace uso de otro método llamado **dropIfExists()** de la clase **Schema,** ese método lo que hace es eliminar la tabla que le indiquemos por parametro, en el caso de la imagen la tabla **‘users’.**

Para realizar una migración hacemos uso del siguiente comando:

* **php artisan migrate**

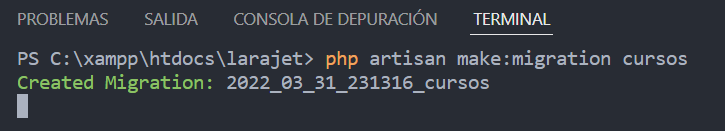
Este comando lo que hará es recorrer una a una las migraciones que hemos creado y va a ejecutar el método up().

Nota: antes de esto es necesario haber configurado ya el archivo .env con la base de datos.

* **Como crear migraciones en laravel**

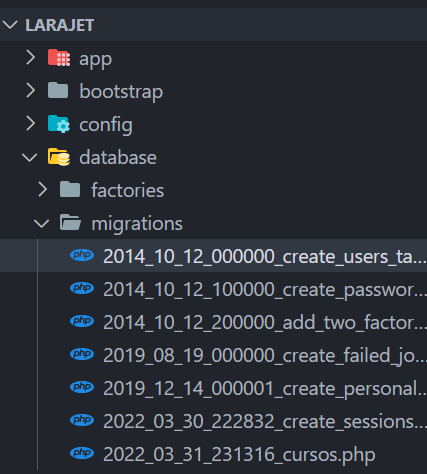
Para crear una migración nos dirigimos a nuestra terminal accedemos a nuestro proyecto y ejecutamos el siguiente comando:

* **php artisan make:migration nombre\_de\_la\_migracion**

****



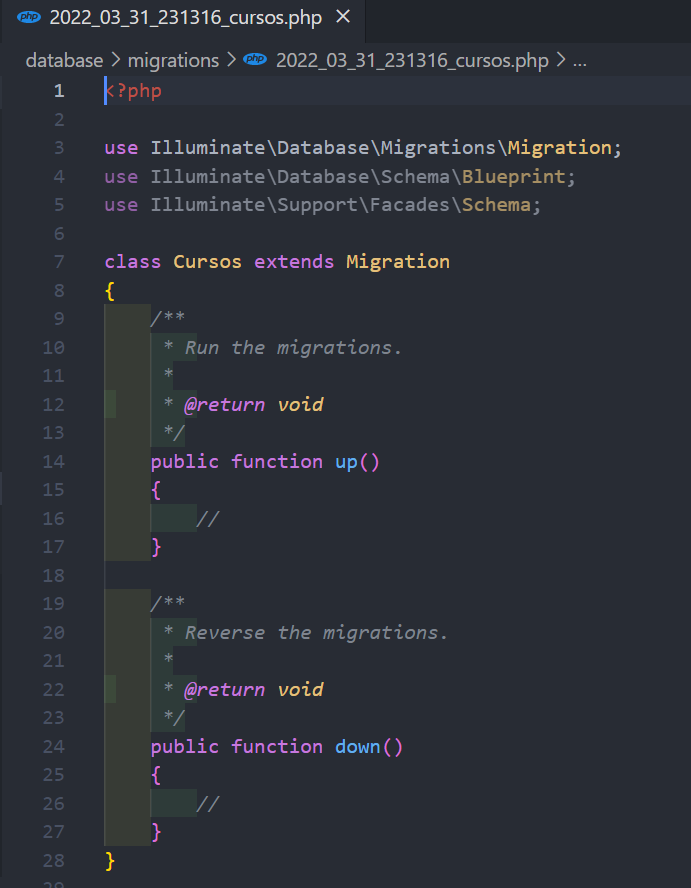
Automáticamente al ejecutar el comando la migración se creará en la ruta ya mencionada anteriormente las carpetas **database/migrations,** allí podremos ver el archivo de la migración que creamos como se apreciará en la siguiente imagen.





En este caso se creó la migración cursos.

En la siguiente imagen veremos lo que hay dentro de la migración que acabamos de crear.



Como se aprecia en la imagen vemos que creo la clase Cursos de acuerdo al nombre que le pusimos a la migración y dentro de esta se crearon los métodos **up()** y **down()** explicados anteriormente, dentro de cada función procedemos a utilizar los métodos respectivos de la clase **Schema** en up para crear nuestra tabla con cada una de sus columnas y en down para eliminarla en caso de ser necesario.

Luego de haber creado todas las migraciones de nuestra base de datos lo que se hace es utilizar el comando **php artisan migrate** el cual se explicó anteriormente para ejecutarlas y visualizarlas en nuestro gestor de base de datos que estemos utilizando.

Para revertir los cambios de la última migración que hayamos realizado laravel nos brinda el siguiente comando:

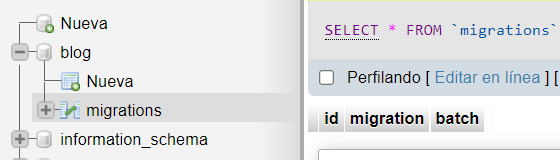
* **php artisan migrate:rollback**

Este comando lo que hacer el revertir los cambios en las migraciones por lotes, para entender un poco mejor que son los lotes se puede decir brevemente que cada vez que realizamos una migración aquellas migraciones nuevas que se vayan agregando pertenecerán a un lote distinto, por ejemplo:

Al realizar una primera migración en nuestro proyecto estas pertenecerán al lote numero 1, si se realiza otra migración en el mismo proyecto aquellas nuevas que se agreguen pertenecerán al lote numero 2 y así sucesivamente.

Entonces dicho esto ya podemos entender como funciona el anterior comando, que tiene como función revertir los cambios desde el ultimo lote es decir si tenemos 3 lotes al momento de revertir lo que hace es que elimina los cambios del ultimo lote regresando como estaban las migraciones antes de realizar la última regresándonos en este caso a las migraciones con el lote numero 2 como ultimo lote.

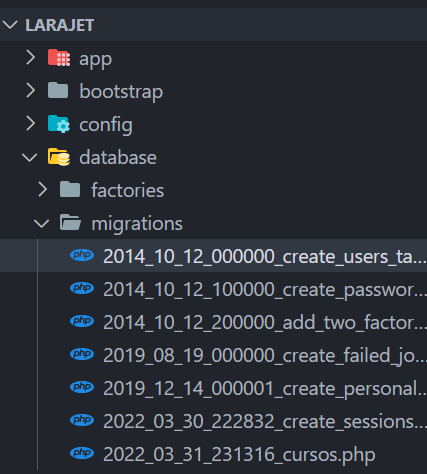
Debemos tener en cuenta que, en caso de querer eliminar un archivo de migración desde la carpeta en nuestro proyecto, primero tenemos que verificar que el lote donde se encuentra esa migración y la migración en si ya no exista en nuestra base de datos o esto podría ocasionarnos errores.





En este caso vemos que con el comando **php artisan migrate:rollback** al ejecutarlo como solo había realizado una migración solo tenia un lote, esto lo que hizo fue revertir los cambios a antes de haber realizado la migración ósea cuando no tenia nada en este caso, por eso podemos apreciar en la imagen donde esta subrayado no tenemos nada, ninguna tabla de nuestra base de datos blog.

Habiendo hecho esto ya podemos eliminar el archivo desde nuestro proyecto si así deseamos.



Archivos de migraciones, como ya revertimos con rollback y no existen en nuestra base de datos podemos eliminarlas directamente y luego las que quedan volver a ejecutarlas con el comando **php artisan migrate** y así nuevamente se crearan en nuestra base de datos sus respectivas tablas.



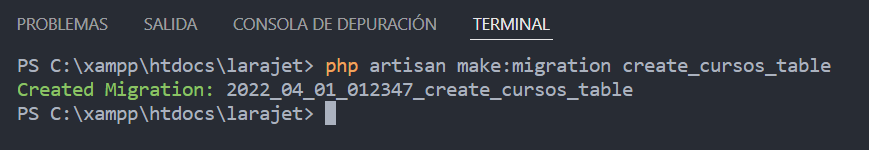
Una de las novedades de la versión 8 de laravel es que siguiendo la siguiente convención al momento de crear una migración laravel nos ayuda agregando automáticamente el método de creación y eliminación de tabla con el nombre de la migración, cosa que no sucedía haciendolo de la manera explicada anteriormente.

La convención que hay que seguir es la siguiente. Se escribe el comando de la siguiente manera:

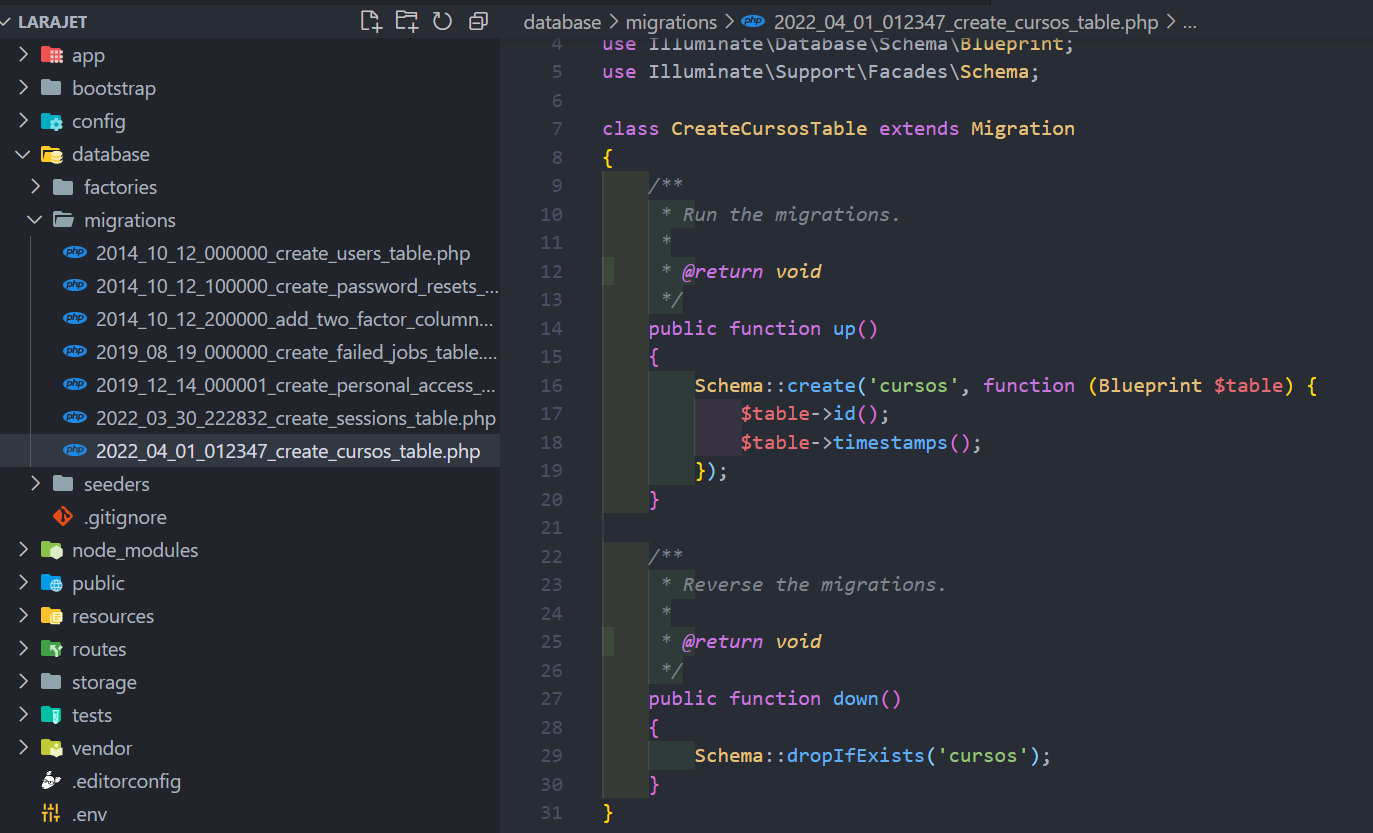
* **php artisan make:migration create\_cursos\_table**

Lo que hicimos fue agregarle al nombre de nuestra migración las palabras create y table seguidas de un guion bajo, recordemos que antes simplemente la habíamos creado colocando solo el nombre de la migración “php artisan make:migration cursos”

Observemos en las siguientes imágenes.









En las imágenes se da como ejemplo poniéndole el nombre cursos a la migración cabe resaltar que puede ser cualquier nombre pero agregándole lo anteriormente dicho y vemos como esto nos facilita el estar reescribiendo el código base donde solamente en lo que es el método up() nos queda crear las columnas de la tabla.

Por último, se ejecutan las migraciones con php artisan migrate para que se creen las tablas en nuestra base de datos.





En la imagen anterior se aprecia con mas claridad las migraciones ejecutadas y las tablas que se crearon junto al lote al que pertenecen en la columna batch.

* **Como agregar o quitar columnas a una tabla por medio de las migraciones**

Para agregar o quitar columnas por medio de las migraciones, laravel nos brinda diferentes maneras.

Una de estas maneras es utilizando el comando

* **php artisan migrate:fresh**

Cabe resaltar que antes de ejecutar este comando primero debemos ir a la migración y modificarla agregándole o quitándole columnas.

La desventaja de este comando es que el mismo es destructivo, es decir que al momento de ejecutarlo este primero elimina todas las tablas con el método dropIfExist() y luego vuelve a crearlas con el método up() agregando los nuevos cambios, esto significa que la información que había guardada en las tablas se perderá por eso laravel no recomienda usarlo cuando el proyecto esté en producción.

Otro comando que podemos utilizar para realizar este proceso es el siguiente:

* **php artisan migrate:refresh**

El cual hace prácticamente lo mismo, a diferencia que este refresh en vez de ejecutar el método drop utiliza el Rolling back ósea el método down() y al final al igual que el comando anterior ejecuta el método up().

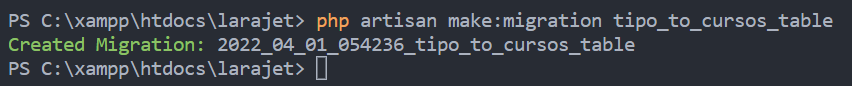
Sin embargo, existe una forma no destructiva de realizar estas modificaciones en una tabla siendo este forma la más viable. Los pasos son los siguientes

1. Lo primero que vamos hacer es crear una migración con la siguiente convención:

- php artisan make:migration NombreColumna\_to\_nombreDeLaTabla\_table

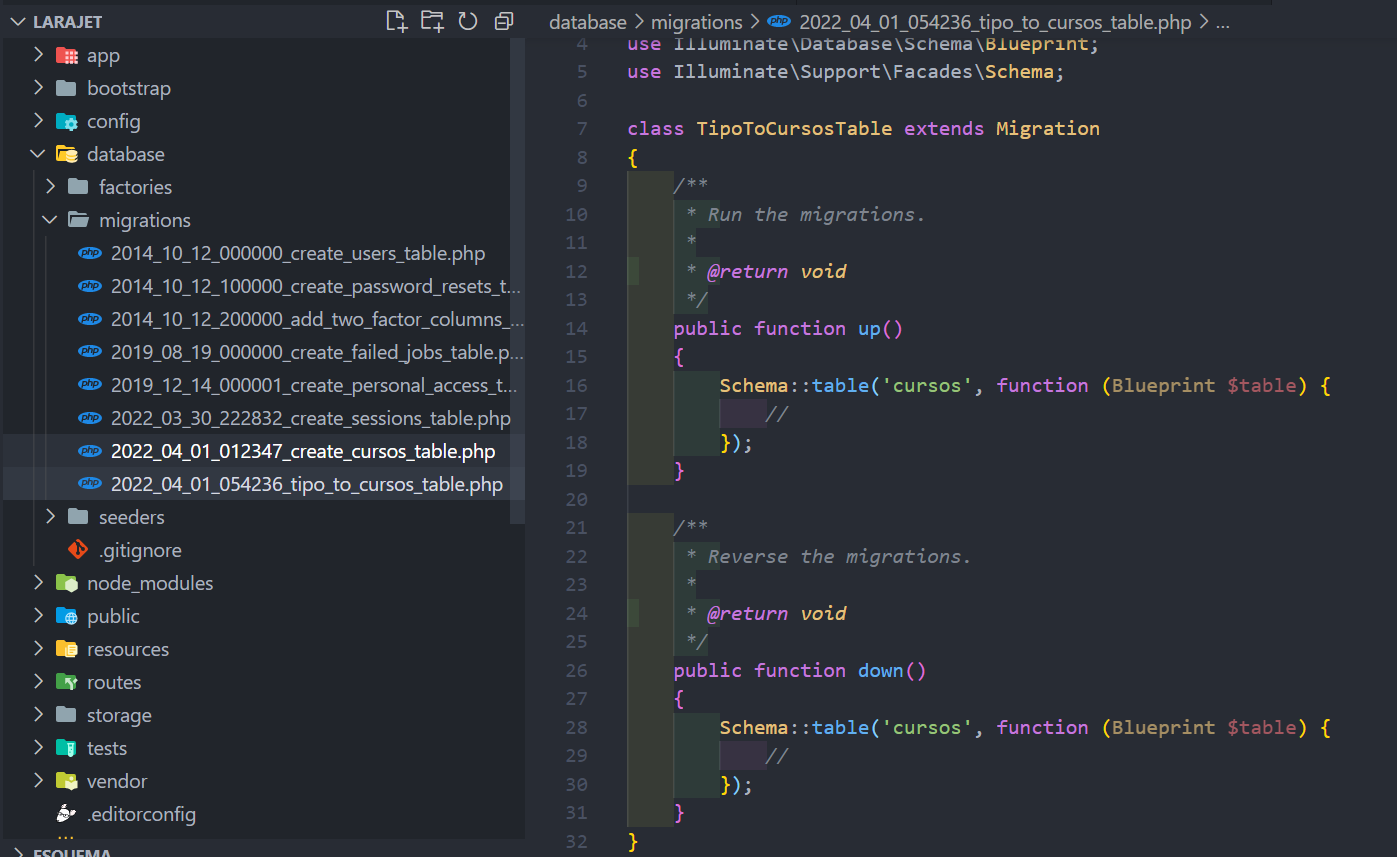
La convención es agregarle al comando el **\_to\_nombreDeLaTabla\_table** donde hacemos referencia a la tabla que se va modificar.







Al ejecutar el comando se nos crea la siguiente migración





1. Como se puede apreciar en la imagen, vemos como esta vez la clase Schema en la funcion up() en vez de utilizar el método **create()** usa el método **table()** que recibe como parámetros la tabla que queremos modificar y una función anónima con un objeto de tipo Blueprint que utilizaremos para hacer la modificación creado la columna por ejemplo el campo tipo el cual agregamos de la siguiente manera:

**$table->string(‘tipo’)->nullable()->after(‘Nombre\_Columna’)**

El método nullable() es para hacer referencia a que ese campo puede ser null y el método after simplemente para ubicar la nueva columna después de la columna que queramos este método recibe como parametro el nombre de la columna.

Dentro del método down() lo que hacemos es agregar el método de eliminación de columna que seria de la siguiente manera siguiendo el ejemplo anterior:

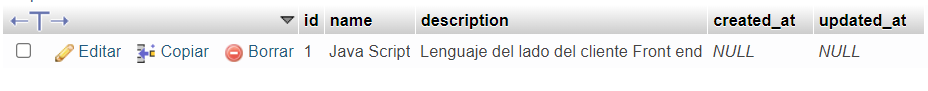
**$table->dropColum(‘tipo’)**

La funcion dropColum() recibe como parametro el nombre de la columna que vamos a eliminar.

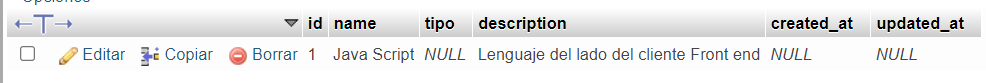
1. Ejecutamos el comando php artisan migrate para ejecutar la migración creada y se realicen los cambios.

Apreciemos lo que sucedió en las siguientes imágenes.

Antes,



Después,





Como se puede notar en el antes y el después vemos que se agrego la nueva columna con campo null y con la indicación que pusimos con el metodo after() que se creara después del campo name.

Pero lo mas importante es que la columna se agrego sin perder la información contenida en la tabla.

Nota: en la documentación oficial podremos apreciar muchas más cosas que se pueden hacer con esta forma de modificación.

* **Introducción a eloquent**

Laravel incluye Eloquent, un mapeador relacional de objetos (ORM) que hace que sea agradable interactuar con su base de datos. Al usar Eloquent, cada tabla de la base de datos tiene un "Modelo" correspondiente que se usa para interactuar con esa tabla. Además de recuperar registros de la tabla de la base de datos, los modelos Eloquent también le permiten insertar, actualizar y eliminar registros de la tabla.

Para poder utilizar eloquent necesitamos crear un modelo, para esto hacemos uso del siguiente comando:

* **php artisan make:model nombre**

Los modelos se encargarán de administrar una respectiva tabla, - pero como sabe el modelo que tabla va a administrar? Para esto debemos seguir una convención que nos brinda laravel. Al momento de crear nuestro modelo cabe recalcar que podemos ponerle el nombre que queramos, pero para facilitar las cosas la convención nos dice que al crearlos lo hagamos colocándole el nombre en singular pero antes debemos verificar que la tabla que queremos administrar con nuestro modelo tenga el mismo nombre, pero plural ej:

Tenemos la siguiente tabla llamada **users**

luego creamos el modelo -**php artisan make:model User**

Al hacer esto eloquent automáticamente va a entender que ese modelo tiene que administrar a la tabla users.

Algo que debemos tener en cuenta es que eloquent saca el singular y el plural del Ingles por tanto para que esta convención funcione bien y sin problemas es preferible trabajar utilizando el inglés al momento de programar.

**Tinker**

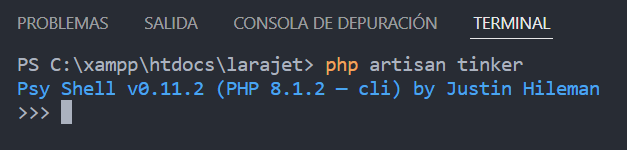
Es una tecnología que permite la interacción de toda nuestra aplicación laravel en la línea de comando, incluido el ORM de eloquent, trabajos, eventos y más, sin necesidad de tener una estructura de formularios creados ya que con Tinker lo manejamos todo por la terminal.

Para poder abrir Tinker debemos hacer uso del siguiente comando:

* php artisan tinker

Al usar este comando podemos notar como nuestra terminal cambia mostrando

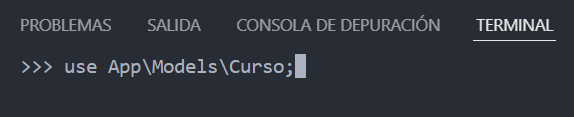
**>>>**

****

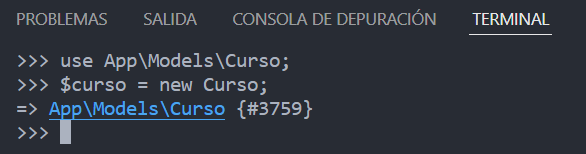
Al trabajar con eloquent este toma cada registro como un objeto, entonces si quisiéramos agregar un nuevo registro lo que hacemos es crear un objeto de la clase en este caso siguiendo el ejemplo la clase Curso.

Lo primero es entrar a Tinker y especificar que queremos hacer uso del modelo mencionado, esto lo hacemos de la siguiente forma:

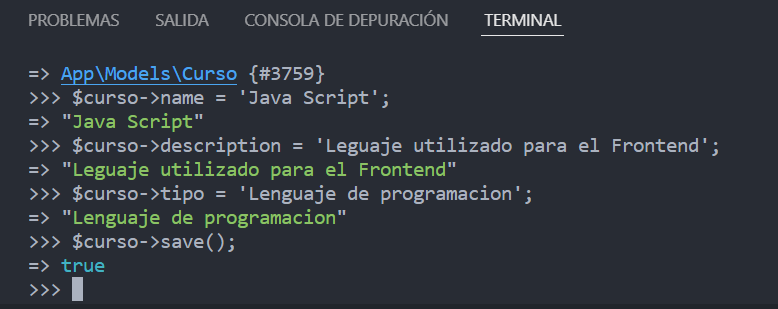
Escribimos use y la ruta ejemplo.



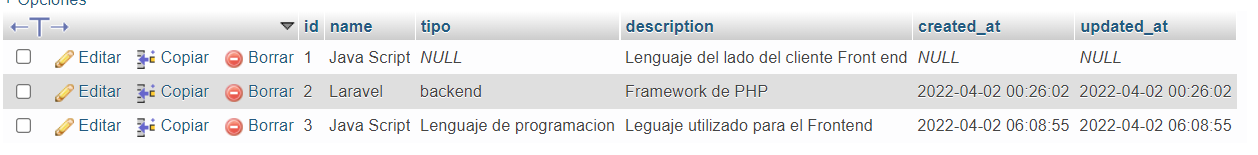
Como ya indiqué que voy hacer uso del modelo curso lo siguiente es crear una instancia de la clase, para esto declaramos la variable con la que haremos uso de nuestra instancia.



Ya con esto podemos asignarle valores a las propiedades de nuestro objeto.

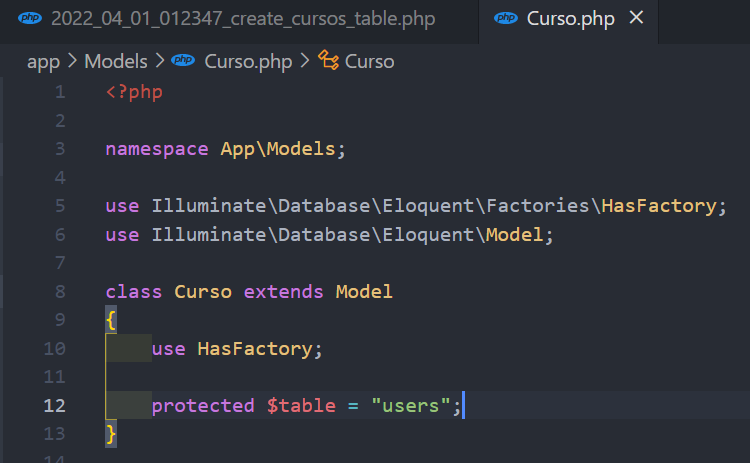


En la imagen se aprecia como se les asignaron valores a los atributos del objeto y luego con la función **save()** guardamos en nuestra base de datos.



Anteriormente habíamos hablado sobre la convención para que un modelo automáticamente administrara una tabla. pero en caso de que no queramos seguir la convención, como asignamos a nuestro modelo la tabla que queremos que administre? – esto los podemos hacer de la siguiente manera.

Lo primero es entrar a modelo y dentro de la clase escribimos lo siguiente



Ya con eso eloquent ignora la convención y al momento de entrar a tinker y hacer uso de nuestro modelo podremos manipular la tabla que le indicamos.

* **Seeders**

Laravel incluye la capacidad de generar su base de datos con datos usando clases seed. Todas las clases seed se almacenan en el directorio database/seeders. De forma predeterminada Laravel al crear nuestro proyecto nos crea una clase llamada DatabaseSeeder. Desde esta clase, puede usar el método **call()** para ejecutar otras clases seed, esto llevar control del orden de los seeders que vayamos creando.

Antes de seguir con los Seeders, en el caso de querer eliminar todas las tablas de nuestra base de datos podemos usar el comando

* **php artisan migrate:reset**

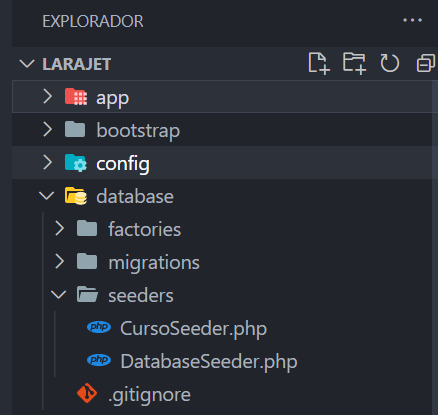
Siguiendo con los Seeders, la forma para crearlos es haciendo uso del comando ej:

* **php artisan make:seeder CursoSeeder**

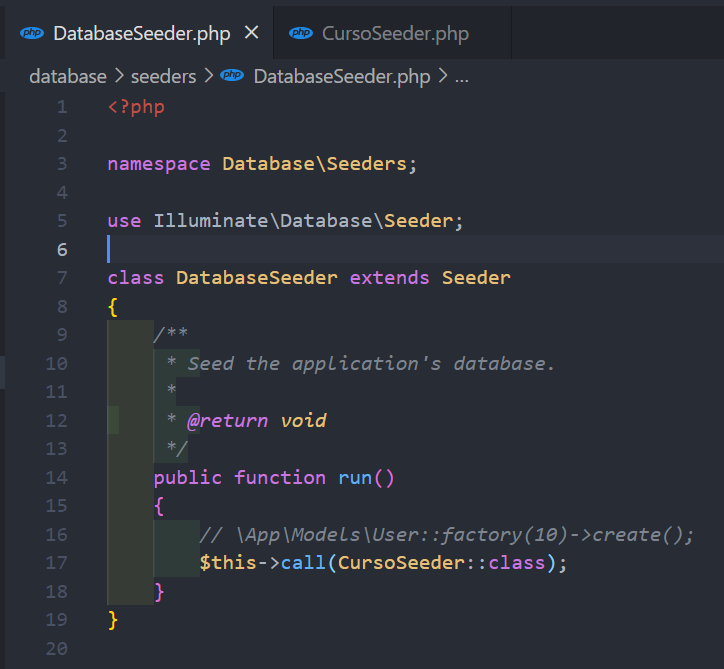
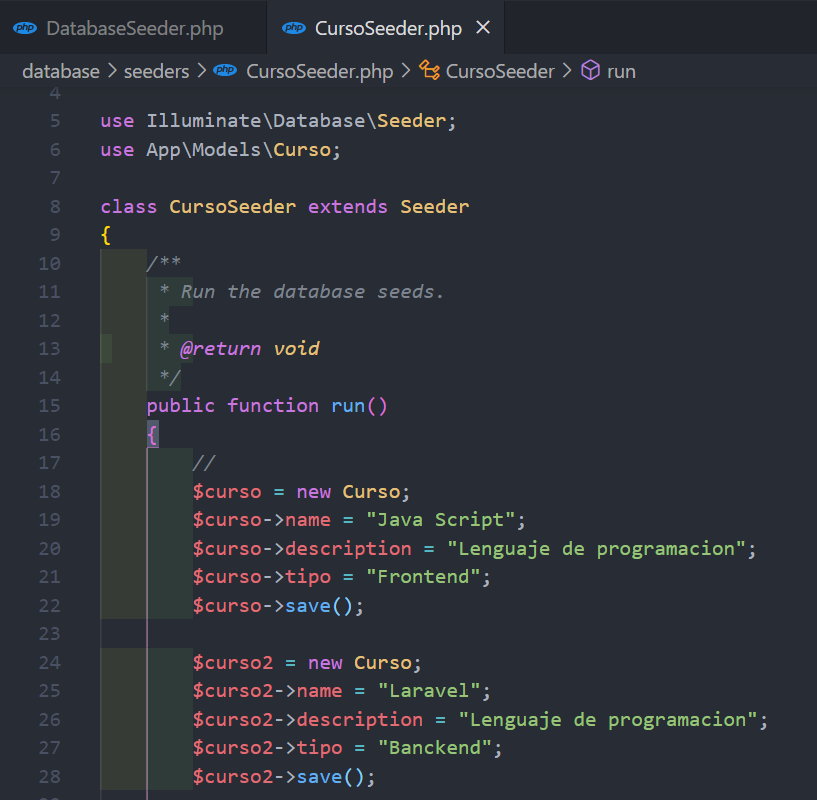
Cabe resaltar que podemos ponerle el nombre que deseemos a nuestros seeders, en el ejemplo anterior se le puso CursoSeeder como nombre al nuevo seeder.

Recomendable después de colocarle el nombre agregarle la palabra Seeder para hacer referencia a que es un seeder administrando una clase.

En la siguiente imagen podemos observar nuestros seeder tanto el que creamos en el ejemplo anterior como el que viene de forma predeterminada en nuestro proyecto que es esencial ya que desde este llamaremos las demas clases seeder que vayamos creando.



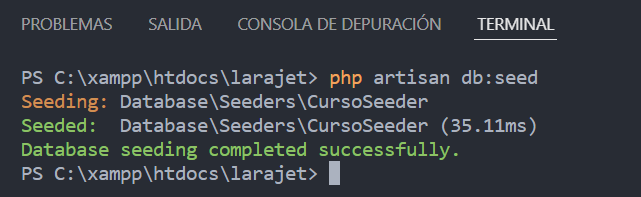
En la imagen de a continuación podremos observar mas de cerca nuestros Seeders

En la imagen de la izquierda tenemos nuestra clase DatabaseSeeder desde donde llamamos la clase CursoSeeder con el metodo **call()** que mencionamos anteriormente y en la imagen de la derecha tenemos la clase CursoSeeder creada en el ejemplo anterior y podemos observar como importamos nuestro modelo Curso para luego crear nuestros datos de prueba o por default para la tabla que el modelo importado administra.

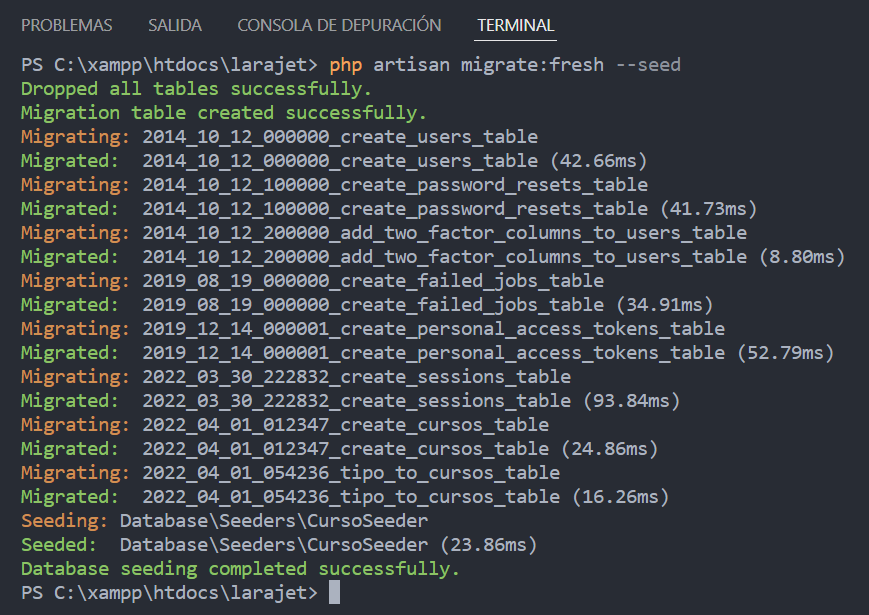
Ahora para ejecutar nuestros Seeders y que se creen los datos de prueba que codificamos se usa el siguiente comando

* **php artisan db:seed**

****

Y listo al ejecutarlo se verá reflejada la información en la base de datos. También podemos ejecutarlos al momento de una migración de la siguiente forma

* **php artisan migrate --seed**
* **php artisan migrate:fresh --seed**

****

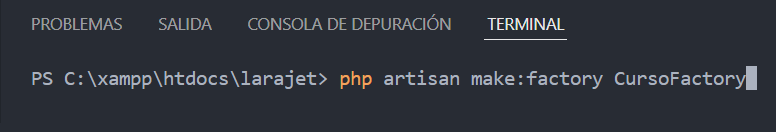
* **Factory´s**

Laravel proporciona una variedad de herramientas y aserciones útiles para facilitar la prueba de sus aplicaciones basadas en bases de datos. Además, los modelos factories y seeders de Laravel facilitan la creación de registros de base de datos de prueba utilizando los modelos y relaciones de Eloquent de su aplicación.

Los factories se van a encargar de llenar nuestra base de datos con datos de prueba, lo único que debemos hacer es indicarle con qué tipo de información queremos que se llene e indicar la cantidad de registros con los que queremos de que se llene nuestra base de datos. Mas adelante lo podremos apreciar mejor con ejemplos.

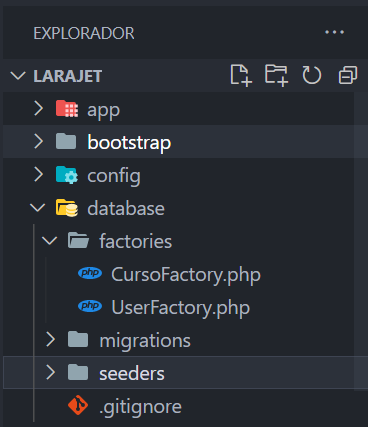
Para crear un factory hacemos uso del siguiente comando:

* **php artisan make:factory Nombre**

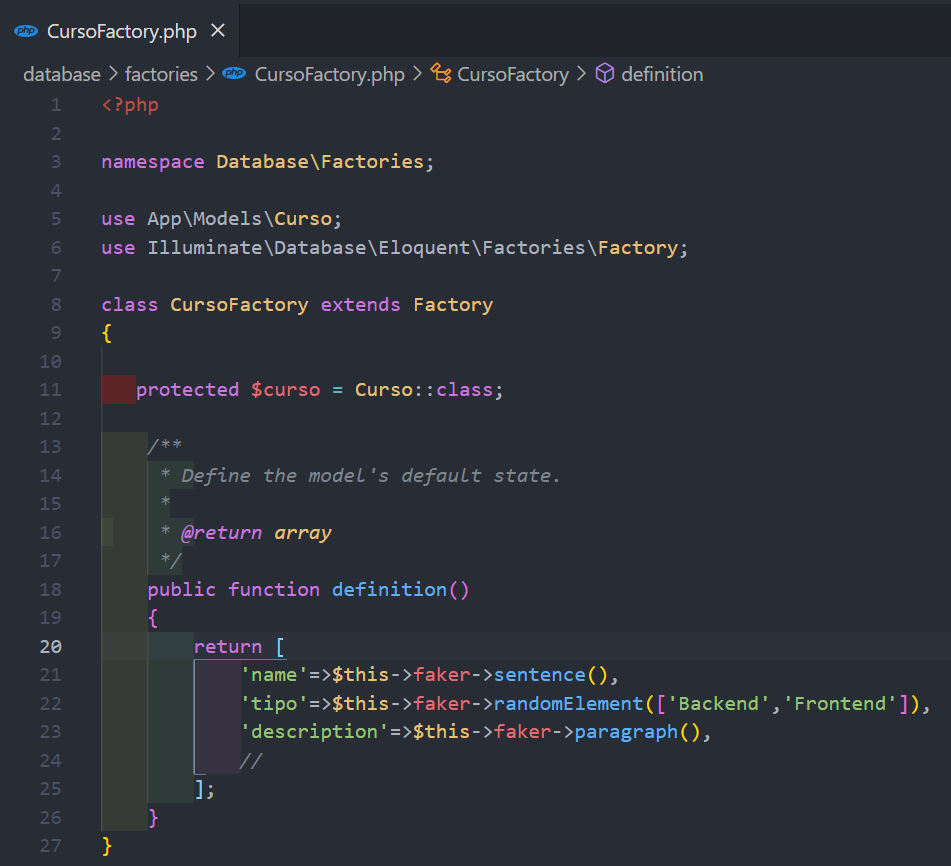


En la imagen podemos apreciar como ejecutamos el comando mostrado para crear un factory con el nombre CursoFactory, cabe recalcar que le podemos poner el nombre que queramos, pero por convención y llevar un código mas organizado es recomendable nombrarlo de esta manera, es decir, el nombre de la tabla a la que hará referencia seguido de la palabra factory para identificar que es un factory.

Estos factories se crean automáticamente en la ruta database/factories de nuestro proyecto



Veamos mas de cerca el código que se genera en nuestro factory

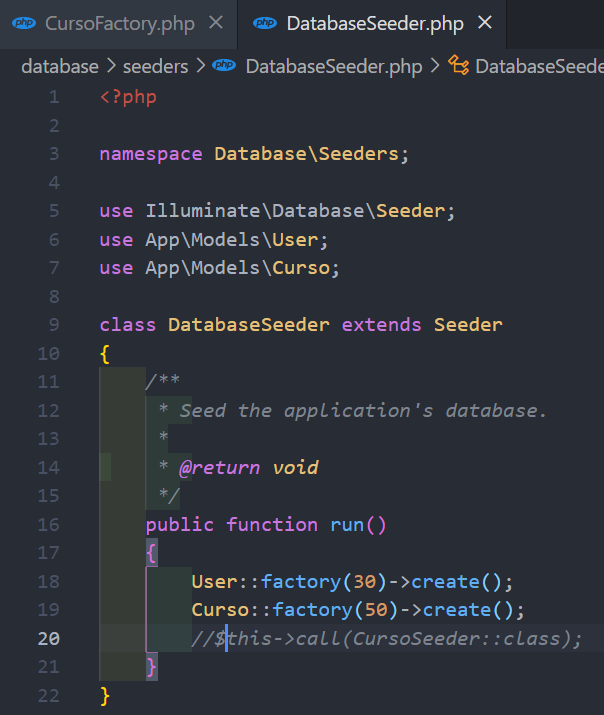


En la imagen anterior observamos el contenido de nuestro factory y si vemos mas de cerca lo que señalan la flechas en la imagen es código que he agregado luego de haber creado el factory, esto para primero indicarle al factory que va se administrado por el modelo Curso entonces lo que hice fue importar nuestro modelo, luego indicarle el modelo que se va a encargar y por ultimo dentro de la función **definition** lo que hice fue poner los atributos de la tabla indicándole el tipo de información que quiero que se genere haciendo uso de uno de los métodos de la clase **Factory** en este caso el método **faker** llamándolo con el this y sus funciones derivadas dependiendo del tipo de información que queramos generar ya sea una oración, párrafos y demás. Se puede hace uso de los métodos de la clase Factory por que si nos fijamos bien al crear nuestro factory este crea la clase y luego hereda la clase Factory automáticamente.

La clase **Factory** cuenta con muchos métodos que podemos ver mas a fondo en la documentación oficial de laravel, en el ejemplo anterior pudimos solo observar el método faker y junto con sentence(), paragraph() y randomElement().

Ahora veremos como hacer uso de estos factories como los llamamos y ejecutamos.

Recuerdan la clase DatabaseSeeder de la que hablamos con anterioridad y como hacíamos uso de nuestros Seeder´s por medio de esta, bueno con los factories sucede lo mismo para hacer uso de estos lo hacemos por medio de la clase DatabaseSeeder de la siguiente manera.



Como se observa en la imagen en la línea señalada 19 y 20 se están llamando los factories creados por medio de los modelos que los administrando en este caso User y Curso pero cabe recalcar que antes debemos importar los modelos.

En la función run lo que hacemos llamar el modelo que administra el factory seguido del método factory que recibe como parámetro la cantidad de registros que queramos generar seguido del método créate y listo.

Ej:

* **User::factory(100)->créate();**

Con el ejemplo anterior estamos diciendo que queremos generar 100 registros para la tabla user.

Ahora para ejecutarlos, lo hacemos de la misma forma como hacíamos con los Seeder´s. Con los comandos:

* **php artisan db:seed**
* **php artisan migrate --seed**
* **php artisan migrate:fresh --seed**

Cualquiera de esos 3 dependiendo de cual necesitemos en el momento.

* **Generador de consultas de eloquent**

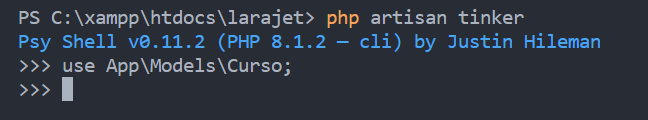
Como mencionamos anteriormente el ORM de eloquent nos brinda herramientas para interactuar con nuestra base de datos de forma más dinámica por medio de los modelos en nuestro proyecto.

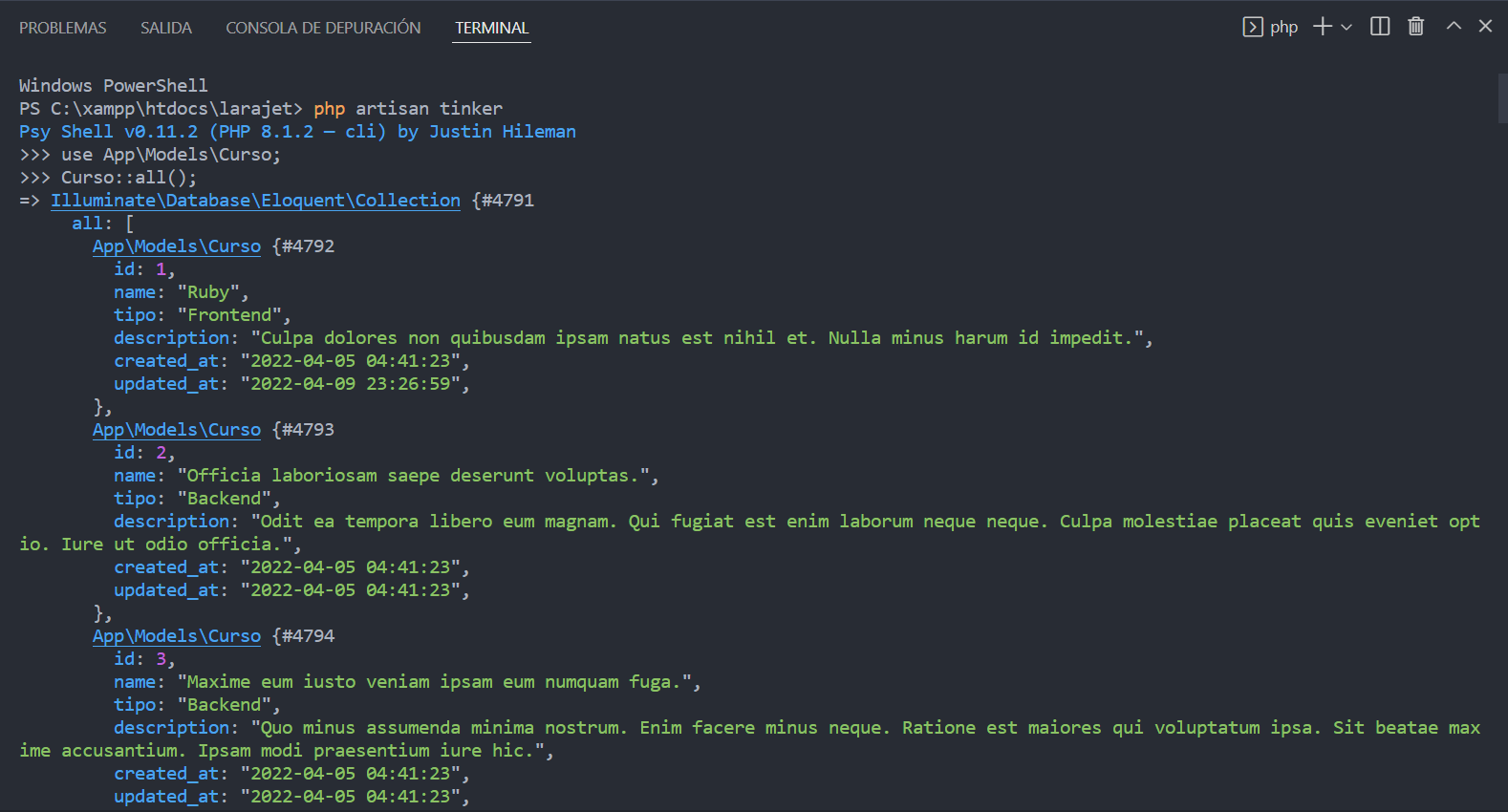
Podemos generar consultas con los distintos métodos que nos brinda, en esta documentación solo veremos algunas pocas de las muchas que existen, pero para ver todas solo necesitas entrar a la documentación oficial de Laravel.

Primero veremos cómo hacemos uso de esta tecnología usando **Tinker** del que se hablo con anterioridad**.**

Para empezar, veremos como mostrar la información de una de las tablas de mi base de datos:

* Lo primero es entrar a tinker.
* Luego indicamos el modelo que administra la tabla con la que vamos a interactuar, esto lo hacemos escribiendo la instrucción **use App\Models\Curso** Siguiendo los ejemplos anteriores haremos uso del modelo Curso**.**
* Y luego procedemos ah realizar la consulta, en este caso para mostrar todos los datos seria de la siguiente manera **Curso::all();** y listo se nos mostrara desde la consola automáticamente toda la información que hay en nuestra tabla cursos.

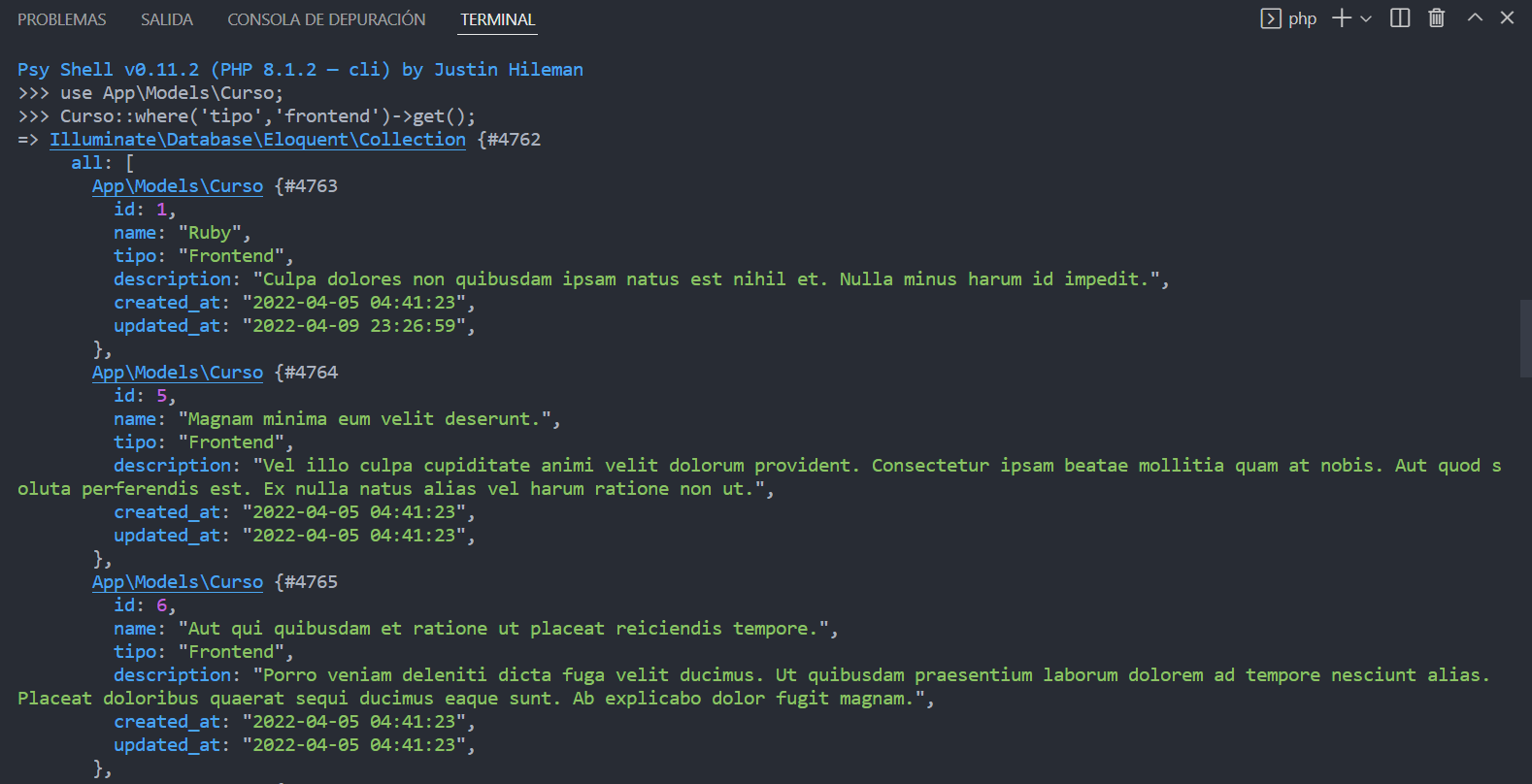




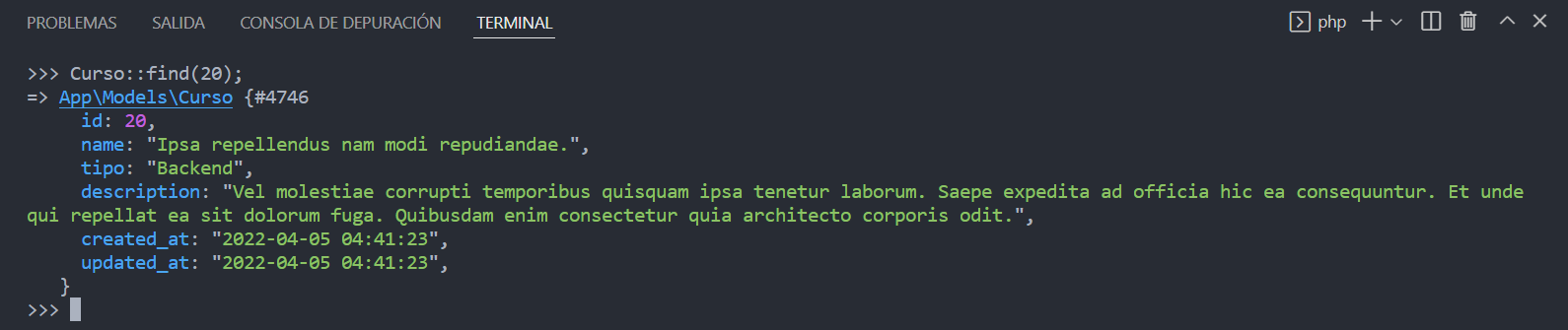
En las imágenes anteriores se puede observar de forma mas visual los pasos anteriores para mostrar la información de la tabla cursos desde tinker.

Mas adelante iré dejando ejemplos de distintas consultas y como mencione anteriormente para sabe run poco mas acerca de esta tecnología échale un vistazo a la documentación oficial de laravel.

En la siguiente imagen veremos una consulta a la tabla curso donde solo queremos mostrar aquellos cursos de tipo Frontend y además les dejo señalada la consulta realizada para lograrlo. En este caso la función **where()** que recibe como parámetro el campo que queremos filtrar en este caso ‘tipo’ y como segundo parámetro el valor en especifico del campo y al final la instrucción **get()** para obtener la información.



Para la imagen de a continuación se mostrará una consulta para obtener un dato en específico. Haciendo uso de la tabla cursos ahora solo quiero leer un curso específicamente para esto hacemos uso de la función find() que recibe como parámetro el id del curso.



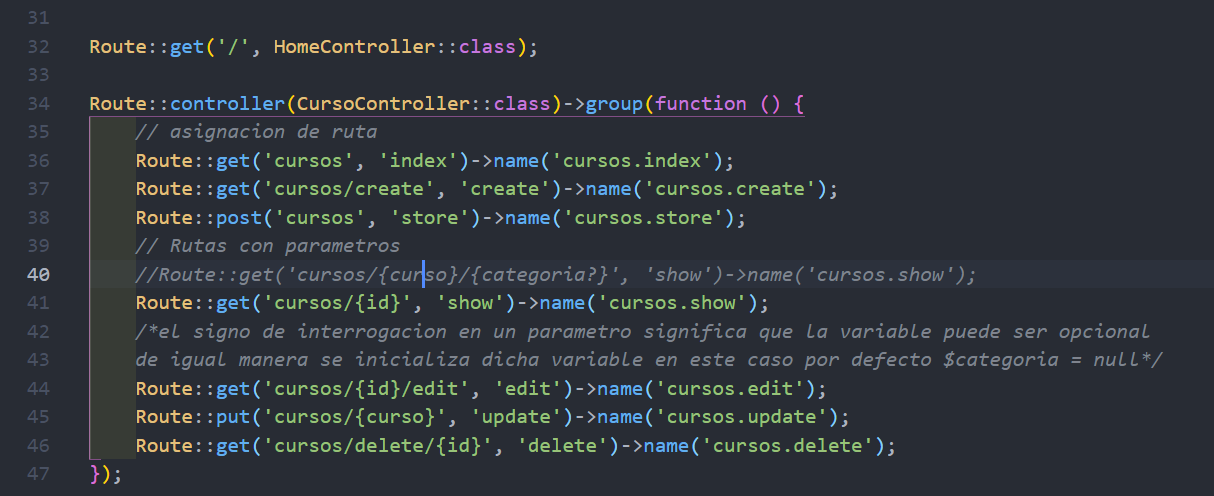
Lo anterior fueron solo algunos ejemplos de las múltiples funciones que laravel nos ofrece para interactuar con nuestra base de datos.

* **¿Como hacer un CRUD con lo aprendido hasta ahora?**

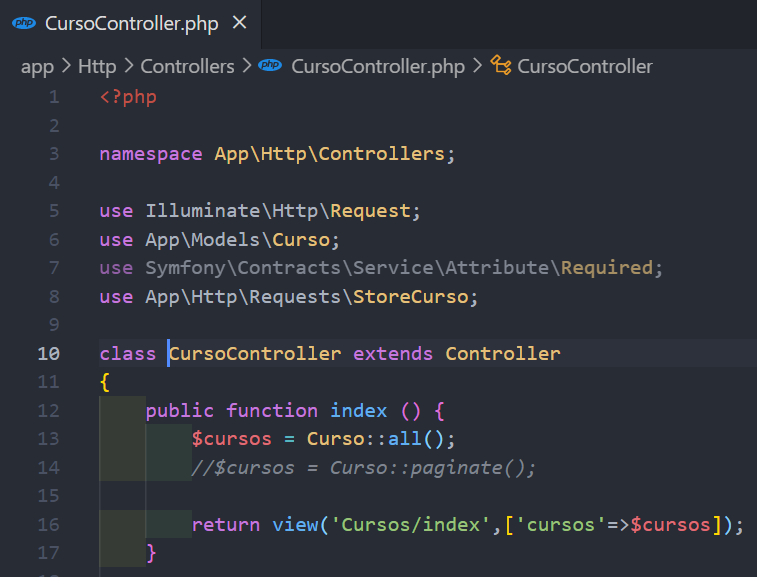
En los ejemplos anteriores utilizamos tinker para trabajar con eloquent desde la terminal. Ahora trabajaremos haciendo uso de las rutas, vistas, controladores y modelos que administran las tablas de nuestra base de datos.

* **Listar**

Empezaremos por mostrar todos los cursos almacenados en nuestra página desde la vista index, para ello miramos que controlador administra esa ruta y luego dentro del controlador nos dirigimos al método index() donde llamamos la respectiva vista que va mostrar la información

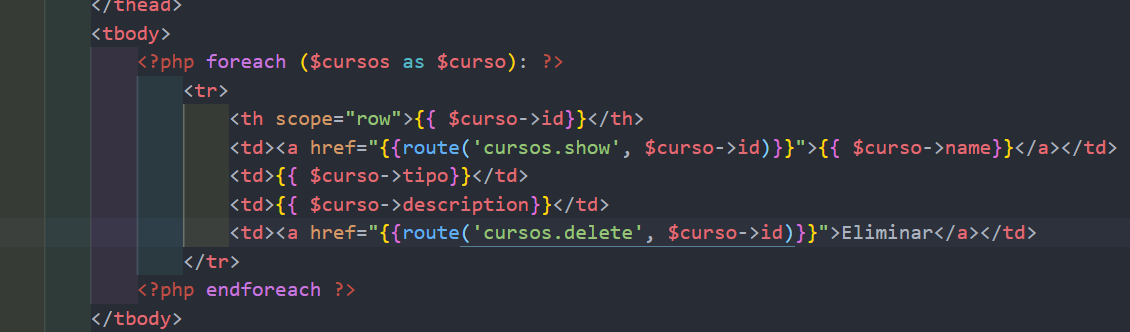


Como se observa en la imagen la ruta que utiliza el método index se encuentra dentro del grupo de rutas que son administradas por el controlador CursoController, entonces ya habiendo identificado el controlador nos dirigimos a el.



Dentro del controlador lo primero que hacemos es importar el modelo que administra la tabla cursos en este caso es el modelo Curso, luego nos dirigimos al método index y dentro realizamos nuestra consulta usando los métodos que nos ofrece eloquent parecido a como hacíamos en tinker solo que ahora lo hacemos desde el controlador capturamos lo que trae la consulta en una variable y por último retornamos a la vista con la variable como se observa en la imagen.

Luego en nuestra vista esa información que recibimos la mostramos con un ciclo for como se mostrara a continuación.



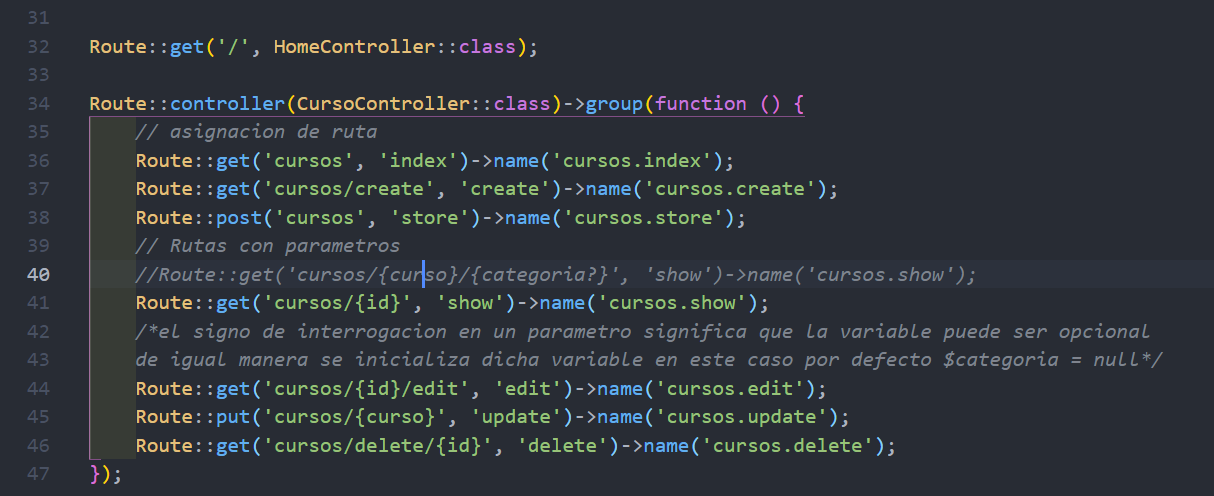
En mi caso cree una tabla en HTML para mostrar la información de forma más organizada.

Con lo anterior ya logramos mostrar información en nuestra página web, ahora lo que haremos mostrar un curso en específico en vez de mostrar todos, haremos que al seleccionar un curso se no muestre aparte la información de este.

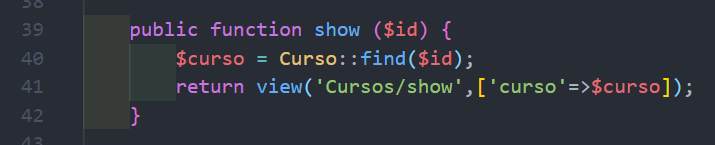
* **Leer**

Ah diferencia del mostrar en este caso lo que haremos será mostrar un registro en especifico con toda la información de este.

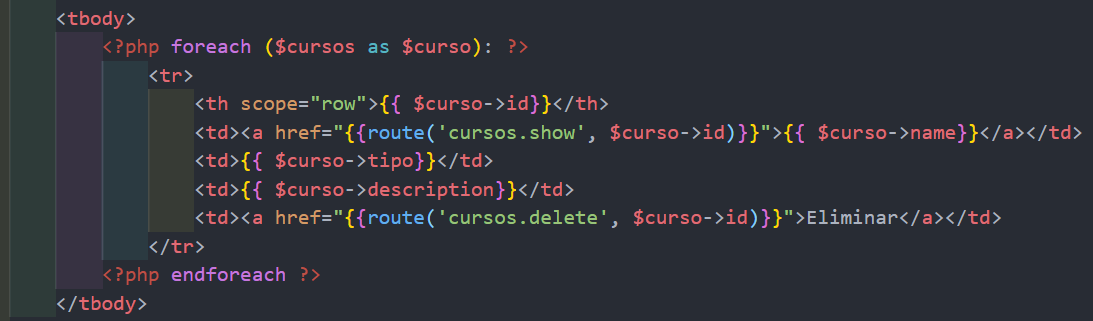
Lo primero es identificar la ruta que administra para este caso el método show que es el encargado de filtrar un registro.



Habiendo identificado la ruta con el método nos dirigimos al respectivo controlador, luego ubicamos el método show y ya dentro de este hacemos la consulta para traer la información.

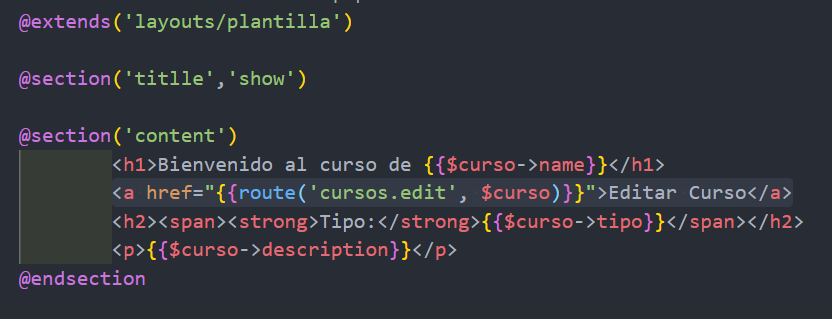


Si te preguntas como se captura el parámetro que recibe la función, la respuesta es sencilla en nuestro vista índex se hizo lo siguiente.



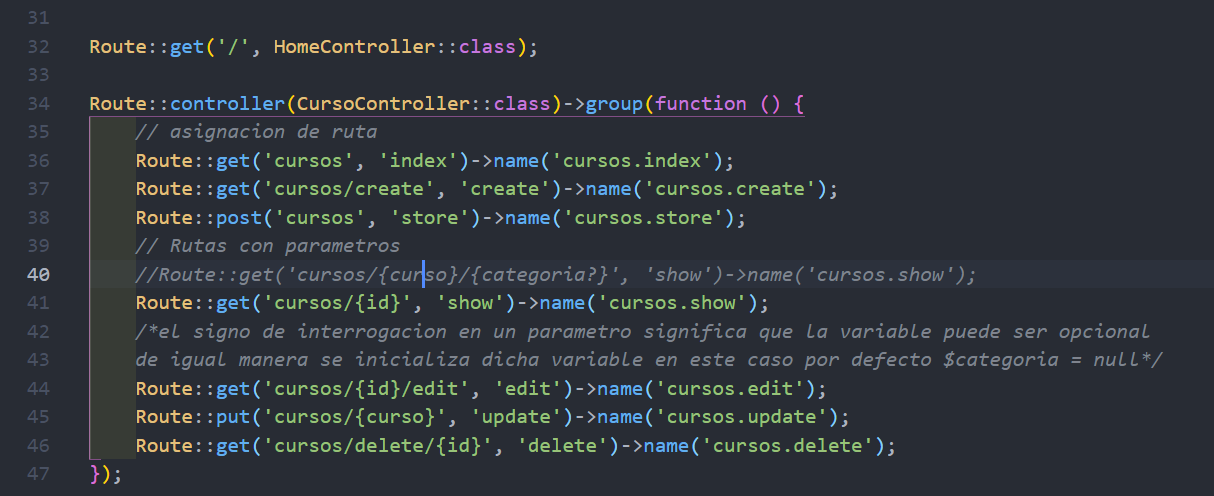
En la imagen anterior se puede apreciar como el campo nombre lo agregue dentro una etiqueta <a> con el fin de que al presionar en ese campo lo pueda redirigir a una vista donde pueda mostrar la información completa del curso seleccionado.

Para hacer que ese campo nos dirija a la vista show lo que hacemos es colocar en el atributo **href** la ruta de que muestra la vista show,Laravel nos proporciona la función **route** en donde ponemos la ruta o el name asignado a la ruta y luego el parámetro en este caso el id del curso que es el que recibimos en nuestro controlador como se puedo apreciar. Ya con esto podemos mostrar la información del curso en la vista show como se mostrará en la siguiente imagen.



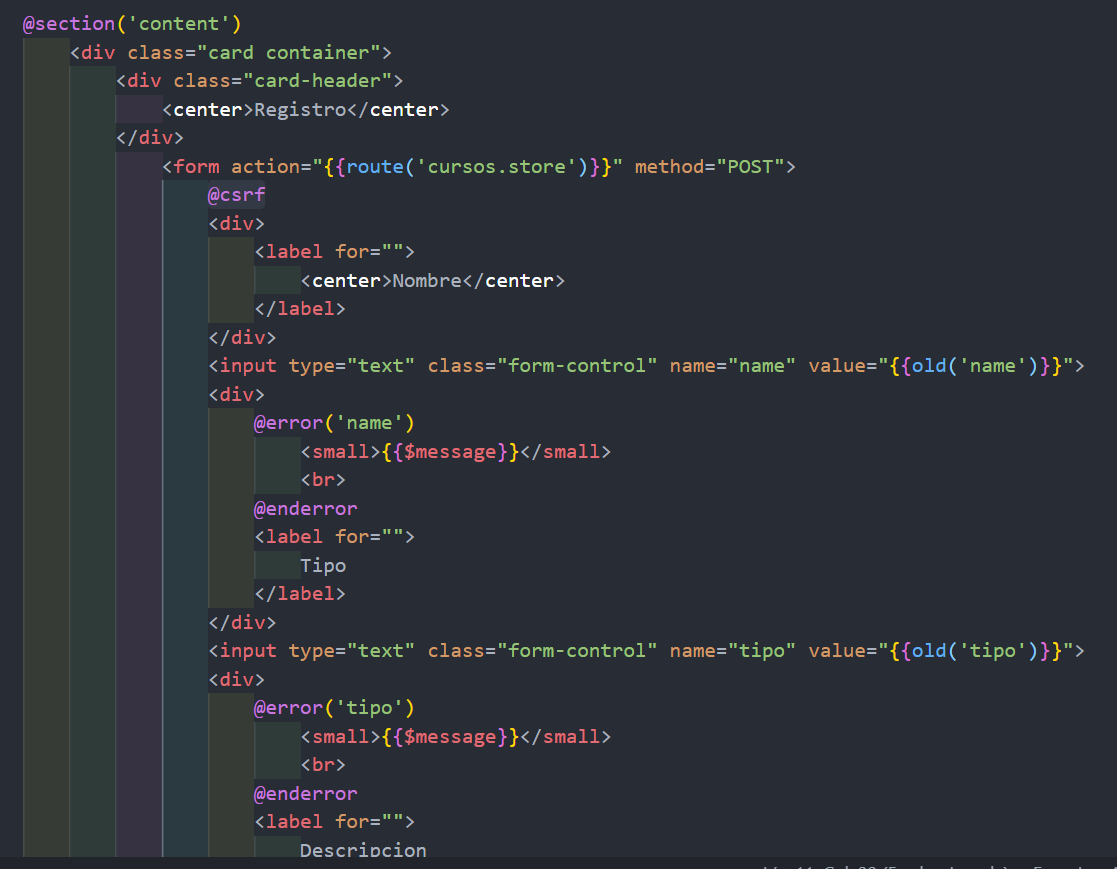
* **Insertar**

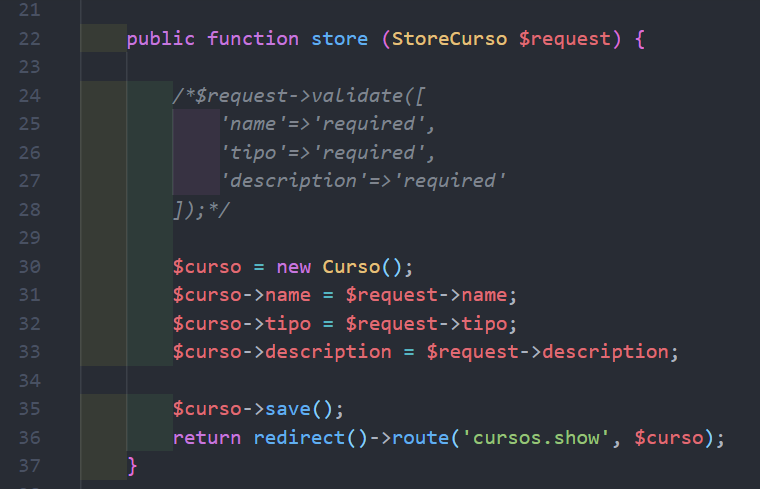
Creamos la ruta donde realizaremos el método store que se encargara de guardar los cursos nuevos que vayamos registrando.



El método usado por lo general para agregar información es el método **post,** por eso se puede apreciar en la imagen que la ruta en este caso usa dicho método.

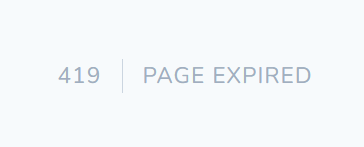
Lo siguiente en ir al respectivo controlador y crear este nuevo método, el **store** o el nombre que desees, en este metodo lo que hacemos guardar toda la información que recibamos de nuestro formulario.





En las imágenes anteriores se puede observar el formulario donde se pide la información, como dentro de la etiqueta <form> le indicamos el metodo post, en el atributo action le pasamos la ruta donde se procesará la información para ser guardada y también se aprecia la siguiente función **@csfr.** Te preguntaras ¿para qué sirve ese @crsf ?, resulta que laravel nos exige que cada vez que vayamos a enviar información por el metodo post se envie un token por cuestiones de seguridad y evitar usuarios maliciosos, entonces al colocar el @crsf en nuestro formulario automáticamente se genera un token cada vez que enviamos información.

Nota: Si omites colocar @crsf, lo que pasara es que al enviar el formulario aparecerá el siguiente error.

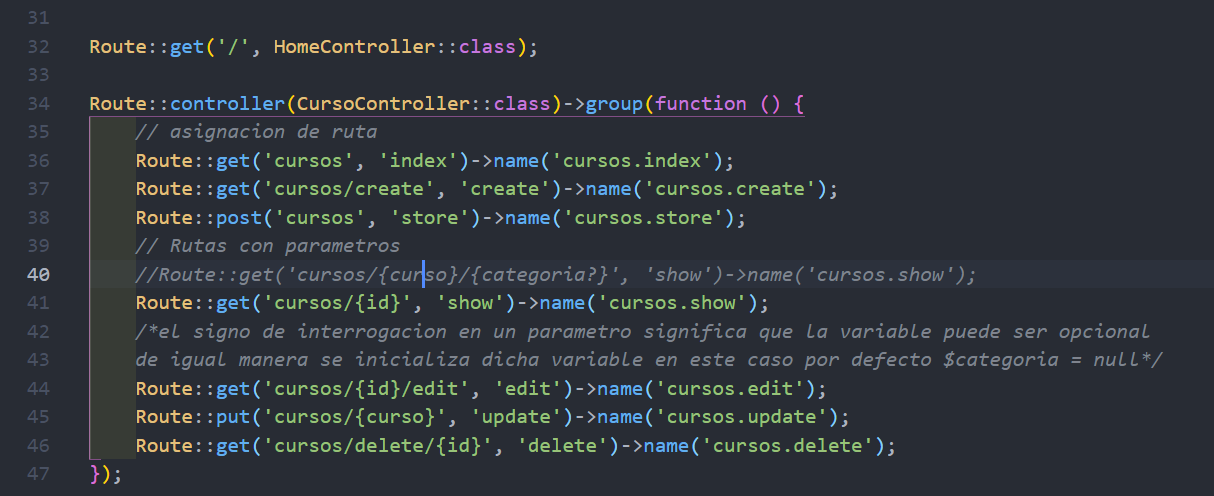


Esto sucede por que laravel nos exige que se coloque si o si por motivos de seguridad.

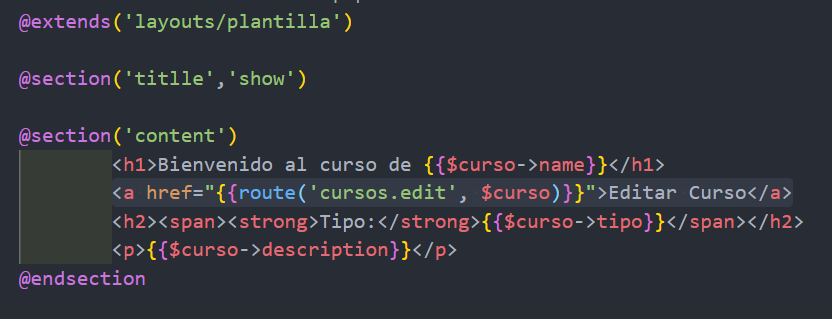
En la otra imagen se mostro el metodo store en nuestro controlador y como se hizo para guardar la información.

* **Editar**

Para editar la información de un curso procedemos a crear la ruta con el metodo que llamare update donde realizaremos el proceso para actualizar la información.

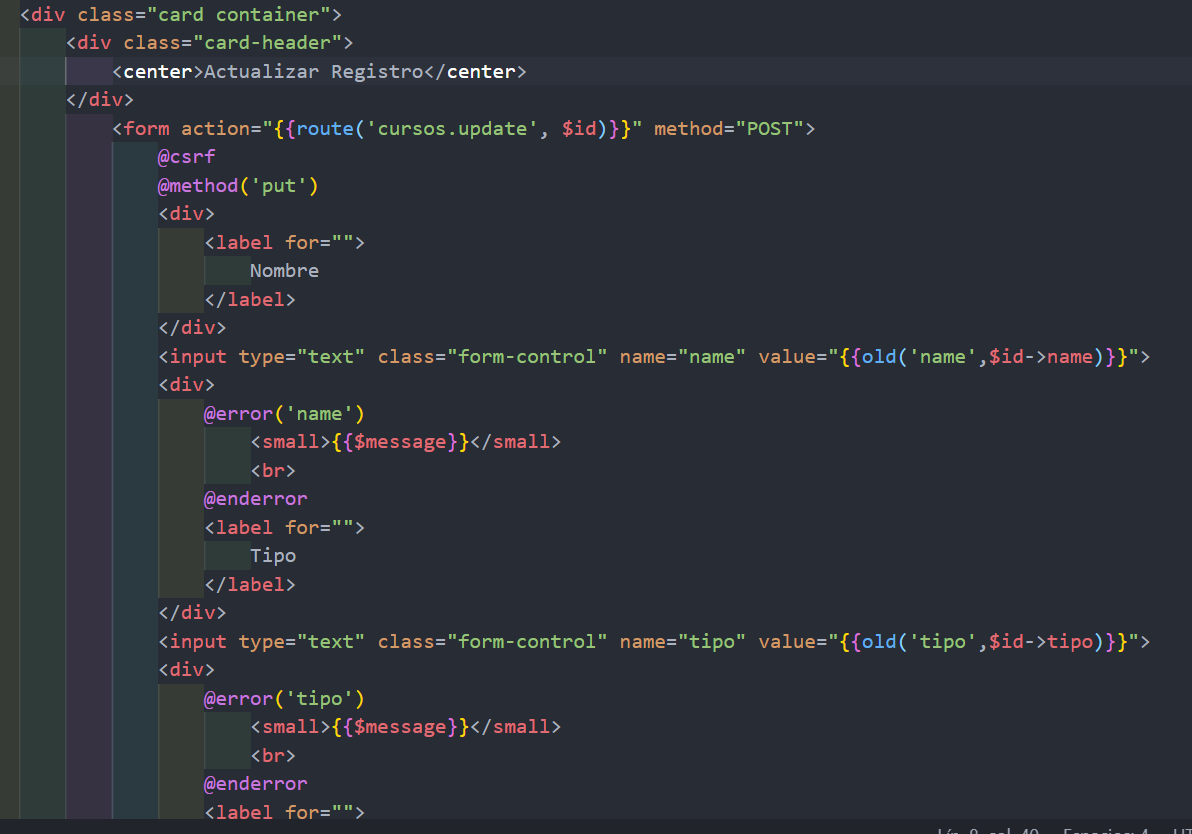


En este caso laravel nos proporciona del metodo **put** para cuando se vaya a cambiar o actualizar información de un formulario, en este caso nuestro formulario se encuentra en la vista que administrada por la ruta con el metodo edit como se aprecia en la imagen.



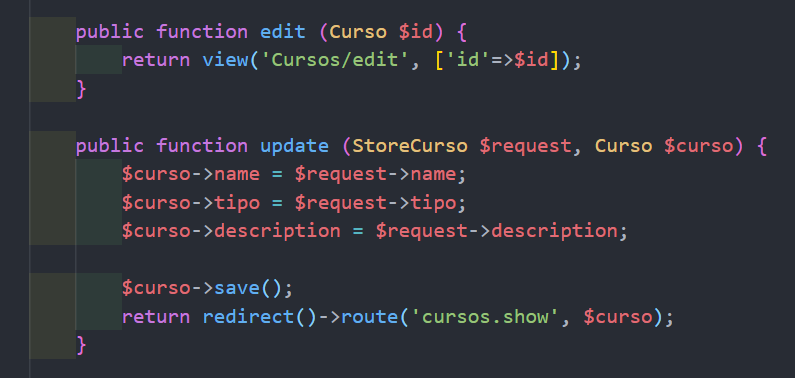
En la imagen de arriba estamos dentro de la vista show y te preguntaras el motivo, bueno la respuesta es por que dentro de esta vista agregare una etiqueta <a> donde podremos oprimir para ir a la vista donde podremos editar la información del curso en que nos encontremos, además de pasar la ruta de la vista editar también pasamos el parámetro que si recordamos dicha ruta lo requiere para mostrar la información del curso en el formulario de la vista editar.

En la siguiente imagen observaremos la vista donde tenemos el formulario para editar la información del curso selecionado.



Te preguntaras por que en nuestro <form> pasamos el metodo post si ya habíamos dicho que para actualizar laravel nos proporcionaba el metodo put.

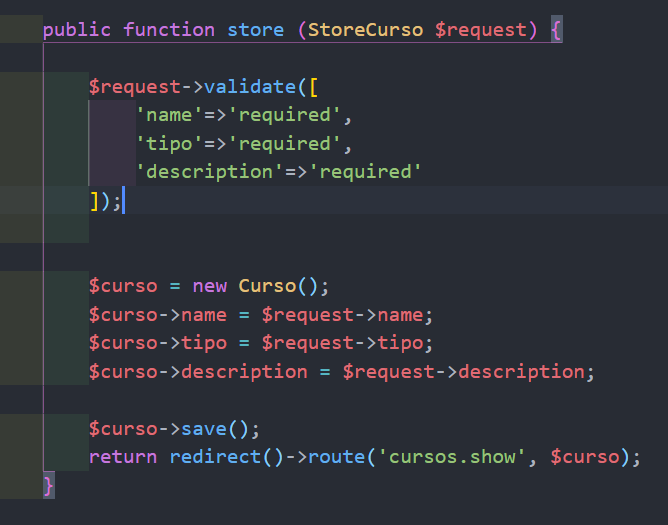
Lo que sucede el lo siguiente y es que el atributo method de la etiqueta form solo reconoce los métodos **get** y **post,** por ello colocamos en este caso el metodo post ya que lo que se va hacer es enviar información, pero de igual manera tenemos que indicar que el metodo que se esta usando es el put de la ruta con el metodo update, para ello laravel nos regala la función @method() señalada en la imagen anterior y esta función recibe como parámetro el metodo que indiquemos en este caso el put, también enviamos como parámetro el objeto a la ruta update.



Como muestra se muestra en la imagen, ya estando en el metodo update recibimos como parámetro la información nueva que viene del formulario y también el objeto con la información anterior del curso seleccionado el cual fue el que se envió desde el action en el <form>. Y con esos datos lo que hacemos es asignarle al objeto la información nueva que viene del formulario guardarla y redirigir a la vista show y observar los cambios.

* **Como validar formularios en Laravel**

Laravel nos permite validar los campos de nuestros formularios por medio del objeto Request. A continuación, veremos como validar que no se envié ningún campo vacío, para este ejemplo lo hare desde el controlador y mas adelante les mostrare como hacerlo de una forma mas organizada y no ensuciar mucho el código.



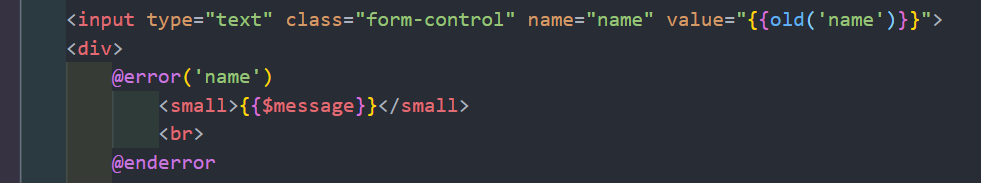
En la anterior imagen validamos los campos del formulario para crear un nuevo registro desde el metodo store() que es donde recibimos la información para ser almacenada ya que por lógica la información antes de ser almacenada primero se valida que cumpla con las condiciones que agreguemos haciendo uso del metodo validate() que nos proporciona la clase Request. Dentro del metodo validate() pasamos los name identificativos de cada campo y las validaciones que queramos agregarle, en la imagen se pasa en forma de array por que son varios campos y cada uno lleva su validación.

Para agregar más de una validación a un campo en especifico se hace uso del símbolo | por ejemplo supongamos que además de que queremos que el campo tipo sea obligatorio también queremos que solo admita máximo 10 caracteres, quedaría de la siguiente manera:

* ‘tipo’ => ‘required|max:10’,

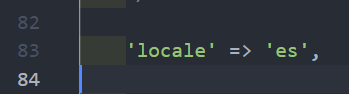
Lo que sucede cuando no se cumple alguna de estas validaciones es que se vuelve a retornar el formulario y no se procesa la información hasta que se cumplan todas las reglas de validación puestas.

Cuando no se cumple alguna regla, estas vienen con mensajes predeterminados que podemos mostrar en nuestras nuestras vistas para notificar al usuario; esto lo hacemos de la siguiente manera usando la directiva de Blade @error() @enderror que nos proporciona Laravel, en al siguiente imagen lo veremos de forma mas clara.



En la imagen se puede aprecia el uso de la directiva de Blade @error() @enderror la cual recibe como parámetro el name identificativo que hayamos puesto en el campo y dentro de la directiva escribimos $message y listo en el momento que falle una validación se mostrara su respectivo mensaje.

Estos mensajes por defecto vienen en ingles así que si queremos tenerlos en español lo primero es indicarle a Laravel y esto lo hacemos dirigiéndonos a la carpeta config, abrimos el archivo app.php, ubicamos la línea donde dice ‘locale’ => ‘en’ y cambiamos el en por ‘es’.



Luego nos dirigimos al siguiente link <https://github.com/Laraveles/spanish> entramos donde dice resources/lang y en al carpeta es encontraremos varios archivos, los copiamos y luego vamos nuestro proyecto entramos a la ruta resources/Lang y dentro creamos la carpeta es y dentro agregamos los archivos copiados anteriormente los cuales contienes los mensajes de validación ya traducidos.

Ahora que pasa si queremos al momento de que una validación falle y se recargue el formulario, pero sin limpiar la información que habíamos puesto, ¿Cómo lo hacemos? Pues fácil esto podemos lograrlo haciendo uso del metodo old() que recibe como parámetro en identificativo name de cada campo, este metodo lo usamos dentro del atributo value como veremos a continuación.



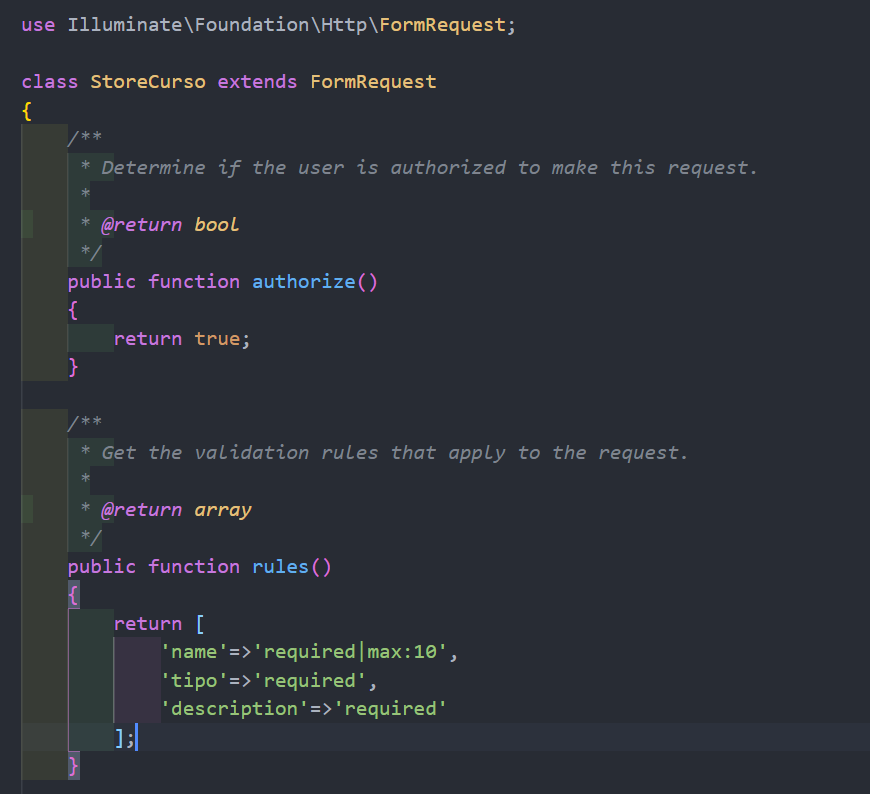
* **Form Request**

Para escenarios de validación más complejos, es posible que desee crear una "Form Request". Las solicitudes de formulario son clases de solicitud personalizadas que encapsulan su propia lógica de validación y autorización. Para crear una clase de solicitud de formulario, puede usar el make:request comando Artisan CLI:

* **php artisan make:request StoreRequest**

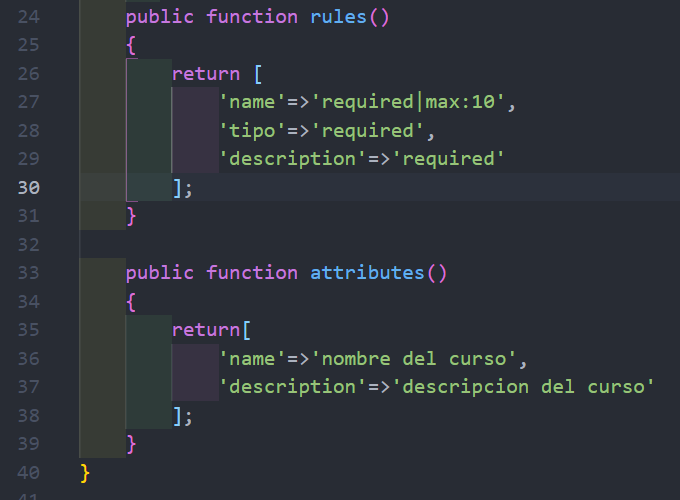
La clase Form Request generada se colocará en el app/Http/Requestsdirectorio. Si este directorio no existe, se creará cuando ejecute el make:request comando. Cada solicitud de formulario generada por Laravel tiene dos métodos: authorize y rules.

Como habrás adivinado, el authorize método es responsable de determinar si el usuario actualmente autenticado puede realizar la acción representada por la solicitud, mientras que el rules método devuelve las reglas de validación que deben aplicarse a los datos de la solicitud



Como se observa en la imagen al crear un Form Request ya no es necesario validar desde el controlador y con esto nuestro controlador queda con el código mas limpio y el código en general se va viendo de forma mas ordenada con las validaciones en una clase FormRequest.

Otra de las cosas que podemos realizar en nuestro Form Request es modificar los mensajes de las validaciones, para ser más específico los name identificativos de cada campo recordemos que al momento de mostrar los mensajes de error el name de cada campo se coloca tal cual, entonces para esto se hace uso del método **attributes()** donde podemos modificar el mensaje para que el identificativo se Muestre como deseemos.

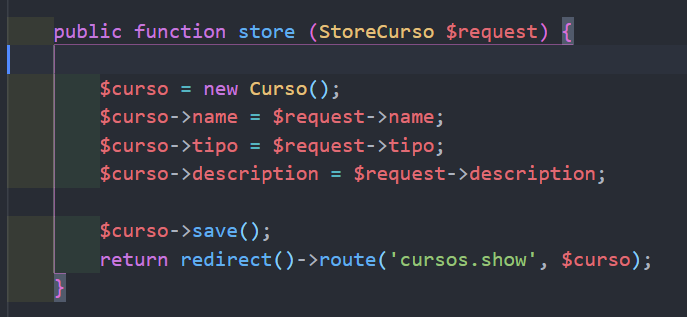


Se que ahora te preguntaras y como hago para que las validaciones se aplique a nuestros formularios al momento de procesar la información desde el controlador, bueno esto lo logramos de la siguiente forma nos dirigimos al controlador en este caso siguiendo ejemplo de la clase curso nos dirigimos al controlador respectivo, luego buscamos el metodo store(), puede ser también el update() o cualquier otro donde estemos procesando información y hacemos lo que se mostrara en la siguientes imágenes.

Lo primero al abrir el controlador es importar la clase FormResquest que creamos



Después en el metodo store() en este caso en ves de recibir un objeto de tipo Request como lo veníamos haciendo lo que hacemos cambiar a un objeto del tipo **StoreCurso** que fue el que creamos y al importarlo podemos usar los métodos de este.

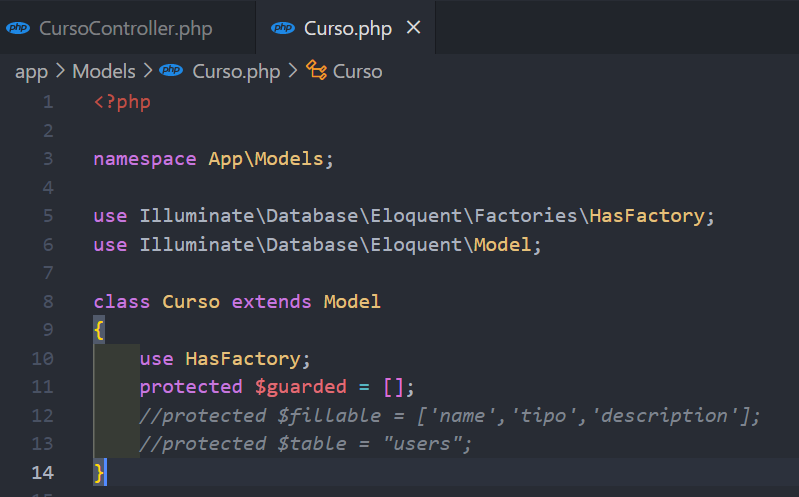


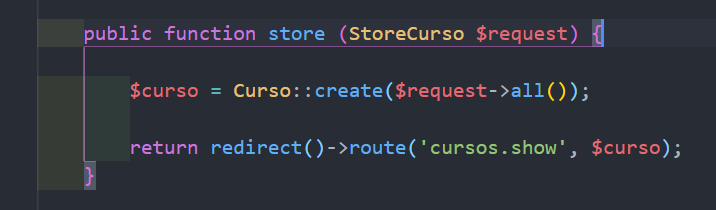
Haciendo esto, automáticamente se aplicarán las validaciones al formulario desde donde se está procesando información en este caso será al formulario de la vista crear cursos.

* **Asignación masiva**

Laravel nos brinda un método para guardar información diferente al que veníamos usando en el que simplemente creábamos una instancia del objeto le asignábamos a cada atributo lo que venía de nuestra vista y luego guardábamos la información con el método **save().**

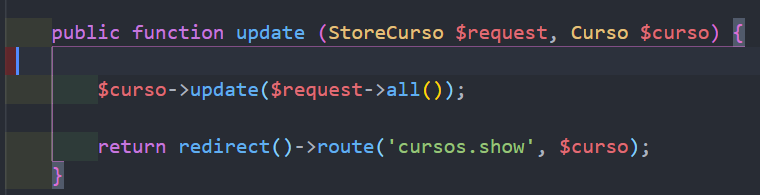
Con la asignación masiva se puede usar el metodo **create** para "guardar" un nuevo modelo usando una sola instrucción PHP. La instancia del modelo insertado le será devuelta por el método. Sin embargo, antes de usar el create método, deberá especificar una propiedad **fillable** o **guarded** en su clase de modelo. Estas propiedades son necesarias porque todos los modelos de Eloquent están protegidos contra vulnerabilidades de asignación masiva de forma predeterminada.





En las imágenes anteriores se observa con mas detalle lo ya dicho para usar la asignación masiva y como se usa, te preguntaras que diferencia hay entre usar el **$guarded** o **$fillable,** bueno lo que diferencia el uno del otro es que en el $fillable especificamos los campos o atributos de nuestro objeto con los que queremos que se genere el registro al intentar agregar este con asignación masiva ignorando los atributos que no están especificados en $fillable, mientras que al $guarded por el contrario se le especifica los campos que queremos que se ignoren al momento de generar un registro por asignación masiva. Esto con el fin de darle más seguridad a nuestra pagina evitando usuarios malintencionados que quieran modificar el formulario con inyección SQL y demás.

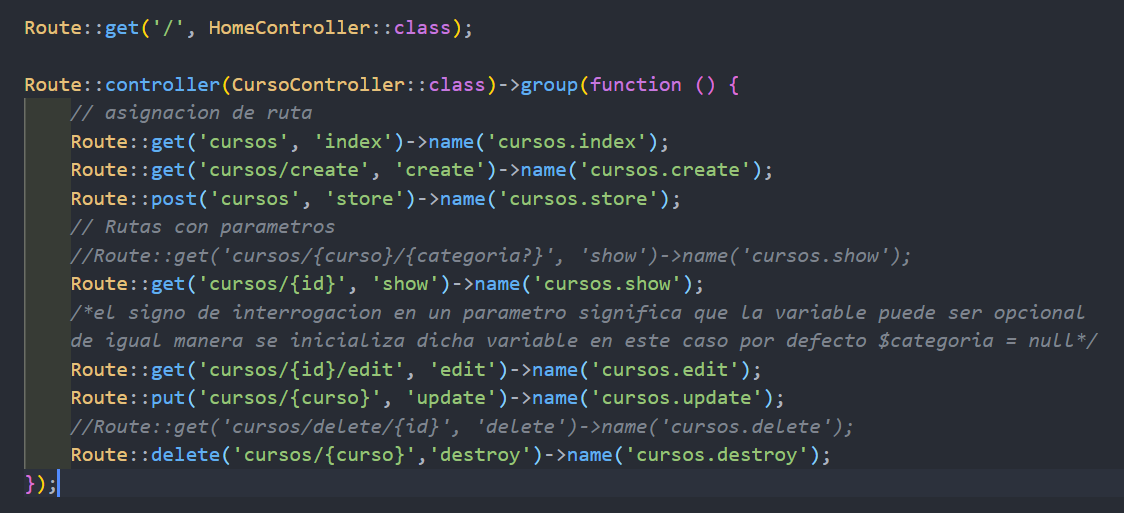
Aquí les dejo otro ejemplo en este caso haremos asignación masiva con el metodo **update()** del controlador CursoController.



Las ventajas de usar asignación masiva primero es que ahorramos escribir código y que ¿te imaginas guardar información de un formulario con 20 campos? seria bastante tedioso hacerlo de la manera convencional con la estábamos trabajando hasta el momento, mientras que con el uso de la asignación masiva lo hacemos en una sola línea de código.

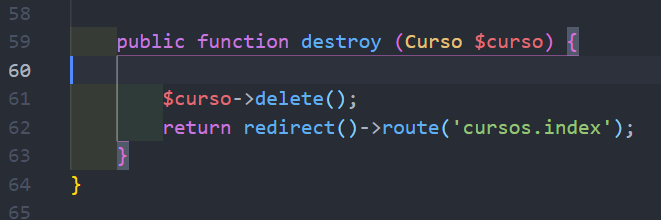
* **Como eliminar un registro**

Para eliminar un registro lo primero que debemos hacer es la creación de la ruta en el archivo web.php de la carpeta routes



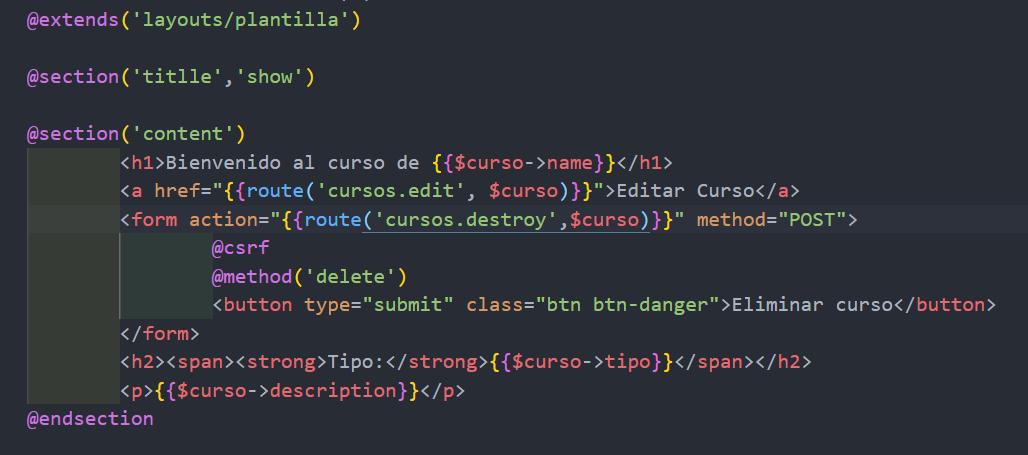
Laravel nos recomienda que al momento de eliminar un registro hagamos uso del método **delete()** así como hicimos con el **put()** al actualizar un registro. Como se observa en la anterior imagen las rutas **show**, **update** y **destroy** son iguales entonces te preguntaras ¿cómo hace laravel para reconocer cual es la que se va utilizar? La respuesta es simple gracias a que hacemos uso del método put y delete al crear las rutas ya le estamos indicando a laravel que cada una se utilizara para determinada acción en el caso de las etiquetas <a> siempre se tomara como método **get** al momento de enviar información, en caso de la update estamos haciendo referencia al método **put** para actualizar y en el **destroy** el método delete para eliminar gracias a esto laravel aunque las rutas sean iguales al usar dichos métodos y referéncialos desde el <form> ya reconoce la ruta para la acción realizar.

Lo siguiente es dirigirnos al controlador, crear la función destroy() y realizar la acción de eliminar el registro como se muestra en la siguiente imagen.



Como último paso lo que no falta es crear el botón eliminar, en este caso lo hare desde la vista **show.**

En la siguiente imagen les dejo como lo hice:



Y listo tenemos la función de eliminar un registro.

**Cómo proteger rutas con expresiones regulares**

**Laravel 10**

En Laravel, puede restringir el formato de los parámetros de ruta utilizando el método where() en una instancia de ruta. Este método acepta el nombre del parámetro y una expresión regular que define cómo se debe restringir el parámetro. Por ejemplo:

Route::get('/user/{name}', function (string $name) {

// ...

})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function (string $id) {

// ...

})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $name) {

// ...

})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);

Además, hay métodos auxiliares que le permiten agregar rápidamente restricciones de patrones a sus rutas. Algunos de los métodos auxiliares comunes son:

oute::get('/user/{id}/{name}', function (string $id, string $name) {

// ...

})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {

// ...

})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {

// ...

})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {

//

})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {

// ...

})->whereIn('category', ['movie', 'song', 'painting']);