

Table of Contents

Introduction	1.1
Introducción	1.2
Capítulo 1. Instalación	1.3
Capítulo 2. PSR-4 y namespaces	1.4
Capítulo 3. Conexión con base de datos	1.5
Capítulo 4. Estructura de un proyecto en Laravel	1.6
Capítulo 5. JSON	1.7
Capítulo 6. Migraciones y Seeders	1.8
Capítulo 7. Modelos y uso de Eloquent	1.9
Capítulo 8. Model factories (Poblar base de datos con faker)	1.10
Capítulo 9. Entrutamiento básico	1.11
Capítulo 10. Vistas y motor de plantillas Blade	1.12
Capítulo 11. Controladores	1.13
Capítulo 12. Validaciones en Laravel	1.14
Capítulo 13. Middlewares	1.15
Anexo A. HTML5	1.16
Anexo B. CSS	1.17
Anexo C. CRUD con Laravel	1.18
Anexo D. Componente Datatable	1.19

Libro | Drive | 5

Introducción a Laravel 5

Laravel es un framework para aplicaciones web con sintaxis expresa y elegante. Creemos

que el desarrollo debe ser una experiencia agradable y creativa para que sea verdaderamente enriquecedora. Laravel busca eliminar el sufrimiento del desarrollo

facilitando las tareas comunes utilizadas en la mayoría de los proyectos web, como la autenticación, enrutamiento, sesiones y almacenamiento en caché.

Laravel es un framework para el lenguaje de programación PHP. Aunque PHP es conocido por tener una sintaxis poco deseable, es fácil de usar, fácil de desplegar y se le puede encontrar en muchos de los sitios web modernos que usas día a día. Laravel no solo ofrece

atajos útiles, herramientas y componentes para ayudarte a conseguir el éxito en tus proyectos basados en web, si no que también intenta arreglar alguna de las flaquesas de PHP.

Laravel tiene una sintaxis bonita, semántica y creativa, que te permite destacar entre la gran cantidad de frameworks disponibles para el lenguaje. Hace que PHP sea un placer, sin sacrificar potencia y eficiencia. Es sencillo de entender, permite mucho la modularidad de código lo cual es bueno en la reutilización de código.

Beneficios de Laravel

- 1. Incluye un ORM:** A diferencia de CodeIgniter, Laravel incluye un ORM integrado. Por lo cual no debes instalar absolutamente nada.
- 2. Bundles:** existen varios paquetes que extienden a Laravel y te dan funcionalidades increíbles..



3. Programas de una forma elegante y eficiente: No más código basura o spaghetti que no se entiende, aprenderás a programar 'con clase' y ordenar tu código de manera de que sea lo más re-utilizable posible.

4. Controlas la BD desde el código: Puedes tener un control de versiones de lo que haces con ella. A esto se llaman migrations, es una excelente herramienta, porque puedes manejar todo desde tu IDE, inclusive montar datos en tus tablas.

5. Da soporte a PHP 5.3.

6. Rutas elegantes y seguras: Una misma ruta puede responder de distinto modo a un método GET o POST.

7. Cuenta con su propio motor de plantillas HTML.

- 8. Se actualiza facilmente desde la línea de comandos:** El framework es actualizable utilizando composer update y listo, nada de descargar un ZIP y estar remplazando.
- 9. Cuenta con una comunidad activa que da apoyo rápido al momento de que lo necesitas.**

Requerimientos iniciales

Para empezar a trabajar con Laravel es necesario cumplir con los siguientes requisitos iniciales:

- Un entorno de desarrollo web: Apache, IIS, Nginx PHP 5.3 o superior
- Base de datos: MySQL, Sqlite, Postgresql o sqlserver
- Librerías php : Mcrypt

Composer es una herramienta para administración de dependencias en PHP. Te permite declarar las librerías de las cuales tu proyecto depende o necesita y éste las instala en el proyecto por ti.

Composer no es un administrador de paquetes. Sí, él trata con "paquetes" o "librerías", pero las gestiona en función de cada proyecto y no instala nada globalmente en tu equipo, por lo cual solo administra las dependencias del mismo.

Composer usa un archivo dentro de tu proyecto de Laravel para poder administrar las dependencias el cual se llama: **composer.json**. Este usa un formato JSON el cual se explicará más adelante, un ejemplo de él se muestra e está imagen:



Ahora, composer no se limita a su uso únicamente con proyectos Laravel, sino que en Laravel el uso de composer nos facilita el control de dependencias y en la actualización de cada una como se explicó anteriormente. Para este curso se trabajará con este archivo pues es el que se va a crear al momento de instalar Laravel.

En este archivo podemos observar cierto orden en el acomodo de la información.

- **"name"**: En esta sección se describe el nombre del usuario propietario del proyecto seguido del nombre del repositorio que aloja el proyecto separados por una barra(/).
- **"description"**: Sirve para facilitar una breve descripción del paquete. Debemos ser muy claros y breves si deseamos colocar una descripción de nuestro paquete.
- **"keywords"**: Estas palabras claves son una matriz de cadenas usadas para representar tu paquete. Son similares a etiquetas en una plataforma de blogs y, esencialmente, sirven al mismo propósito. Las etiquetas te ofrecen metadatos de búsqueda para cuando tu paquete sea listado en un repositorio.
- **"homepage"**: La configuración de la página es útil para paquetes que van a ser de código libre. Puedes usar esta página para el proyecto o quizás para la URL del repositorio. Lo que creas que es más informativo.
- **"license"**: Si tu paquete está pensado para ser redistribuido, querrás ofrecer una licencia con él. Sin una licencia muchos programadores no podrán usar el paquete por restricciones legales. Escoge una licencia que se ajuste a tus requisitos, pero que no sea muy restrictiva para aquellos que esperan usar tu código. El proyecto de Laravel usa la licencia MIT que ofrece gran libertad.
- **"authors"**: ofrece información sobre los autores del paquete, y puede ser útil para aquellos usuarios que quieran contactar con el autor o autores. Ten en cuenta que la sección de autores permite una matiz de autores para paquetes colaborativos.

Aprender más sobre HTML5

Para profundizar un poco más en HTML5 es recomendable el tutorial de [w3schools](http://w3schools.com).

Con esto le indicamos a composer que debe descargar nuestras dependencias y las dependencias de estas dependencias para satisfacer las necesidades de nuestro proyecto. Para más información sobre composer, sus campos y su forma de uso podemos consultar su página oficial <https://getcomposer.org/doc/> la cual se encuentra en inglés.

Gestor de dependencias

Una de las opciones interesantes del archivo composer.json es el campo **"require"**, en el se agregan como un arreglo el nombre de los paquetes que queremos incluir en nuestro proyecto seguido de la versión de cada dependencia.

Al final cuando se han agregado todas las dependencias que queremos para nuestro proyecto entonces solo basta con usar el siguiente comando en nuestra consola:

```
composer install
```

Preparando nuestro entorno de trabajo.

Laravel necesita un servidor web. No importa cuál sea pero la mayoría de la comunidad usa Apache o Nginx y hacer lo mismo te pondrá las cosas más fáciles a la hora de buscar ayuda si la necesitas.

Instalación de XAMPP (Windows)

XAMPP es un programa que nos ofrece una distribución de Apache, MySQL, PHP y Perl muy simple de instalar, administrar y utilizar. Podemos descargarlo [aquí](#).

Instalación de LAMP (Linux)

LAMP es el conjunto de aplicaciones Apache, MySQL, PHP o Python en entornos Linux que nos facilitan el desarrollo de sistemas.

En Ubuntu o derivadas podemos instalarlo con los siguientes comandos:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install lamp-server^
sudo apt-get install php5-mcrypt
```

Después de tener instalado nuestro Servidor web, es necesario instalar composer el cuál es un gestor de dependencias php muy útil y del cuál se hablará más tarde.

Instalación de Laravel

Existen diferentes formas de instalar laravel en nuestra computadora.

- Podemos clonar el repositorio [Laravel](#) de github.
- Usando el instalador:


```
composer global require "laravel/installer=~1.1"
laravel new Proyecto
```
- Usando composer:


```
composer create-project laravel/laravel --prefer-dist Proyecto
```

Una vez instalado laravel es recomendable situarse en la raíz del proyecto y ejecutar:

```
composer update
php artisan key:generate
php artisan app:name Curso
```

Instalación de composer (Linux)

La forma más sencilla de instalar Composer en tu ordenador Windows consiste en descargar y ejecutar el archivo [Composer-Setup.exe](#), que instala la versión más reciente de Composer y actualiza el PATH de tu ordenador para que puedas ejecutar Composer simplemente escribiendo el comando composer.

En ubuntu bastará con ejecutar los siguientes comandos en la terminal.

```
sudo apt-get install curl
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
sudo echo 'PATH=$PATH:~/.composer/vendor/bin' >> ~/.profile
```

PSR-4 y namespaces

¿Qué es PSR-4?

Es una especificación para la auto carga de clases desde la ruta de los archivos. Describe dónde se encuentran ubicados los archivos que serán autocargados. PSR-4 hace uso de namespaces para distinguir una clase de otra, esto es de gran ayuda cuando ocupamos librerías de terceros porque en muchas ocasiones existirán clases con el mismo nombre que las nuestras y podrían sobreescribirse o usar una que no queremos.

PSR-4 fue creada por el grupo de interoperabilidad de PHP, ellos han trabajado en la creación de especificaciones de desarrollo para este lenguaje para que estandarizemos diferentes procesos, como es en este caso el como nombrar las clases de nuestro proyecto y hacer uso de ellas.

Usar especificaciones PSR-4 no es obligatorio y su uso puede ser completo o parcial, aunque es recomendable no omitirlo porque a Composer le permite cargar nuestras clases automáticamente.

¿Qué es un autoloader?

Aparecieron desde la versión de PHP5 y nos permite encontrar clases para PHP cuando llamamos las funciones new() o class_exists(). De esta forma no tenemos que seguir haciendo uso de require() o include().

PSR-4 nos permite definir namespaces de acuerdo a la ruta de los archivos de las clases, es decir, si tenemos una clase "Pdf" en el directorio Clases/Templates/, ese será su namespace. Podemos hacer un similar con el import de java.

El namespace de Clases/Templates quedaría de la siguiente forma:

Clases/Templates\Pdf.php

Para usar PSR-4 en composer podemos definir el namespace de nuestra aplicación y el directorio donde serán alojadas las clases, por ejemplo:

```
{
    "autoload": {
        "psr-4": {
            "taller\\": "app/"
        }
    }
}
```

Para usar los namespaces dentro de nuestros archivos php basta con referenciarlos de la siguiente forma:

```
use Taller\Clase;
```

¿Qué es classmap?

Es un autoloader que nos permite registrar nuestras clases para poder ocupárlas sin necesidad de un namespace, la desventaja respecto a PSR-4 es la colisión de clases con mismo nombre, la principal ventaja es la rapidez de autocarga de clases. Otro inconveniente de usar classmap es que debemos ejecutar constantemente el comando "composer dump-autoload" por cada clase nueva en el directorio que indiquemos o tengamos registrado en el archivo "composer.json".

Ejemplo:

```
{
    "classmap": [
        "database"
    ],
}
```

Conexión con bases de datos

Laravel tiene soporte para los motores de bases de datos más populares como:

- MySQL
- Postgresql
- SQLite3
- SQL Server

Véremos como utilizar MySQL con laravel.

Dentro del archivo `database.php` en el directorio `config` configuraremos el driver de la conexión, por defecto vendrá con mysql, si queremos cambiarlo por otro motor de base de datos tendremos que cambiar el valor `mysql` por sqlite, pgsql, sqlsrv.

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

Tendremos que configurar el archivo `.env` ubicado en la raíz del proyecto.

```
DB_HOST=localhost
DB_DATABASE=curso
DB_USERNAME=root
DB_PASSWORD=12345
```

Una vez que tengamos todo configurado, nos dirigimos a la terminal y ejecutamos el comando `php artisan migrate` para crear las migraciones, si todo ha salido bien tendremos que ver las tablas:

- migrations
- password_resets
- users

Si eres una persona curiosa habrás notado que el nombre de las tablas en Laravel siempre son escritas en plural, esto no es por puro capricho, es parte de una convención:

Convención de la configuración, dicha convención le permite a Laravel hacer magia por nosotros, nos evita realizar configuración y pasos extras de la asociación de Modelos con tablas entre otras cosas.

Estructura de un proyecto en Laravel

Todos los proyectos nuevos en Laravel 5.1 tienen la siguiente estructura de directorios:

- app/
- bootstrap/
- config/
- database/
- public/
- resources/
- storage/
- tests/
- vendor/
- .env
- .env.example
- .gitattributes
- .gitignore
- artisan
- composer.json
- composer.lock
- gulpfile.js
- package.json
- phpspec.yml
- phpunit.xml
- readme.md
- server.php

A continuación describiremos los directorios y archivos más importantes para que nos ayuden a entender más el funcionamiento del framework.

El directorio app

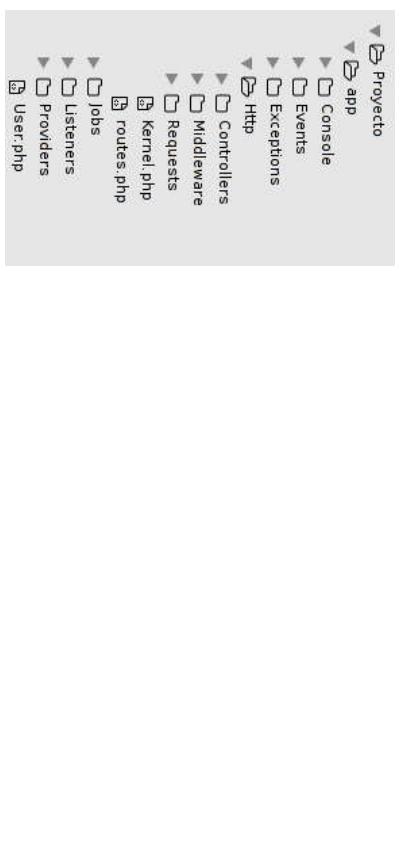
App es usado para ofrecer un hogar por defecto a todo el código personal de tu proyecto.

Eso incluye clases que puedan ofrecer funcionalidad a la aplicación, archivos de configuración y más. Es considerado el directorio más importante de nuestro proyecto ya que es en el que más trabajaremos.

El directorio app tiene a su vez otros subdirectorios importantes pero uno de los más utilizados es el directorio **Http** en el cuál ubicaremos nuestros `controllers`, `Middlewares` y `Requests` en sus carpetas correspondientes, además dentro del subdirectorio **Http**

encontraremos también el archivo `routes.php` donde escribiremos las rutas de la aplicación.

A nivel de la raíz del directorio `app` encontraremos el modelo `User.php`, los modelos comúnmente se ubicarán a nivel de la raíz de la carpeta `app` aunque igual es posible estructurarlo de la forma que queramos, por ejemplo, en una carpeta llamada `Models`.



El directorio public

Dentro de este directorio colocaremos todos los recursos estáticos de nuestra aplicación, es decir, archivos css, js, imágenes y fuentes.

Es recomendable crear una carpeta por cada tipo de recurso.

El directorio resources

Dentro de este directorio se encuentran los subdirectorios:

- assets : Aquí se ubican todos los archivos less de nuestra aplicación (útil para desarrolladores front-end).
- lang : Aquí se encuentran todos los archivos de internacionalización, es decir, los archivos para poder pasar nuestro proyecto de un idioma a otro. Normalmente habrá una carpeta por cada idioma, ejemplo:
 - en : idioma inglés
 - es : idioma español
- views : Aquí ubicaremos nuestras vistas en formato php o php.blade, es recomendable crear una carpeta por cada controlador, además agregar una carpeta **templates** para las plantillas. Una plantilla es una vista general, que tiene segmentos que pueden ser reemplazados mediante la herencia de plantillas, más adelante se hablará de este tema.

El directorio database

Aquí se encontrarán los archivos relacionados con el manejo de la base de datos. Dentro de este directorio se encuentran los subdirectorios:

- factories : Aquí escribiremos nuestros model factories.
- migrations : Todas las migraciones que creamos se ubican en este subdirectorío.
- seeds : Contiene todas las clases de tipo seed.

El directorio storage

Cuando Laravel necesita escribir algo en el disco, lo hace en el directorio `storage`. Por este motivo, tu servidor web debe poder escribir en esta ubicación. Aquí podemos encontrar otros directorios entre los cuales el más relevante es el subdirectorío `framework`, es ahí

donde se almacena el cache y las vistas compiladas.

```

    <storage>
      <app>
        > debugbar
      <framework>
        <cache>
          > sessions
        > views
        > ignore
      > logs
    
```

El directorio tests

Aquí escribiremos los archivos de pruebas que serán ejecutadas posteriormente por phpuunit.

El archivo .env y .env.example

El archivo .env no existe cuando instalamos laravel, en este archivo se configurará el modo en que se ejecuta nuestra aplicación, por defecto será el modo debug, además podemos configurar la conexión a la base de datos y la conexión con el servidor de correo electrónico. El archivo .env lo creamos copiando el archivo .env.example y renombrando la copia como .env.

Por motivos de seguridad de la base de datos el archivo .env nunca se sube cuando hacemos un push en nuestro repositorio. Es por eso que aparece escrito dentro del archivo .gitignore en la raíz de nuestro proyecto.

```
{
  "Frutas": [
    {
      "Nombre": "Manzana",
      "Cantidad": 20,
      "Precio": 10.50,
      "Podrida": false
    },
    {
      "Nombre": "Pera",
      "Cantidad": 100,
      "Precio": 1.50,
      "Podrida": true
    }
  ]
}
```

JSON

JSON es un acrónimo de JavaScript Object Notation, un formato ligero originalmente concebido para el intercambio de datos en Internet. JSON nos permite representar objetos, arrays, cadenas, booleanos y números.

La ventaja de usar JSON para la transferencia de información es que puede ser parseada por varios lenguajes y es un formato estandarizado, es decir que cualquier lenguaje puede intercambiar datos con otro mediante JSON.

Por defecto, JSON se guarda sin espacios entre sus valores lo cual lo puede hacer un poco más difícil de leer. Esto se hace normalmente para ahorrar ancho de banda al transferir los datos, sin los espacios en blanco adicionales, la cadena JSON será mucho más corta y por tanto habrá menos bytes que transferir.

Sin embargo, JSON no se inmuta con los espacios en blanco o saltos de linea entre las claves y valores, así que podemos hacer uso de ellos para hacerlo un poco más legible. JSON es un formato de transferencia de dato y no un lenguaje.

Debemos tener siempre en cuenta que en el formato JSON las cadenas siempre van en comillas dobles, además, los elementos clave y valor debes estar separadas con dos puntos (:), y las parejas clave-valor por una coma (,).

Por ejemplo:

Los tipos de valores aceptados por JSON

Los tipos de valores que podemos encontrar en JSON son los siguientes:

- Numéricos (entero o flotante)
- Strings o cadenas (entre comillas dobles)
- Booleanos (true o false)
- Arrays o arreglos (entre corchetes [])
- Objetos (entre llaves {})
- Null

¿Por qué aprender JSON?

JSON es utilizado ampliamente en:

- El archivo composer.json de proyectos PHP
- Intercambio de información
- Representación de una base de datos
- AJAX
- Web Services

Validación de JSON

JSON es un formato para el intercambio de información muy rígido y estricto, si tenemos un error de sintaxis, obtendremos un error y no podremos parsear el JSON. Para solucionar este tipo de problemas, existen en internet un gran número de herramientas que nos ayudan a escanear y encontrar posibles errores en la formación de nuestro JSON.

Podemos ocupar [JSONLint](#) que es una muy buena opción, bastará con copiar y pegar nuestro JSON en el área de texto y a continuación dar click en el botón "Validate".

JSONLint nos informará si es correcto el formato o en caso contrario nos mostrará los errores sintácticos de nuestro JSON.

Migraciones

Cuando creamos nuestras bases de datos solemos crear diagramas que nos facilitan la abstracción de como se va a almacenar nuestra información, pero la forma de llevarlo a la realidad en algún gestor de bases de datos, como por ejemplo: [MySQL](#), [SQLite](#), [PostgreSQL](#), [SQL Server](#), etc., lo más común es meternos al lenguaje de script encargado de implementar nuestra idea de la BD y ejecutar dicho script, o incluso ocupar programas más avanzados que nos sirven como interfaz para crearlas de una forma más gráfica y sin la necesidad de profundizar demasiado en el lenguaje, como [Workbench](#) o [Navicat](#).

En Laravel se lleva a otro contexto esta situación, puesto que visto de la forma tradicional si se requieren cambios en la base de datos tenemos que meternos ya sea a otro programa para cambiar el diagrama de la base o a un archivo SQL con una sintaxis usualmente complicada o difícil de leer y ejecutar los cambios para reflejarlos en el proyecto, sin embargo, con esto no contamos con un control de los cambios (control de versiones) sobre la base de datos, si necesitamos consultar un cambio anterior o de repente la solución previa o inicial era la que se necesita al momento debemos re-escribir todo otra vez, cosa que con la [migraciones](#) se soluciona instantáneamente.

Las migraciones son archivos que se encuentran en la ruta `database/migrations/` de nuestro proyecto Laravel, por defecto en la instalación de Laravel 5 se encuentran dos migraciones ya creadas, `create_users_table` y `create_password_resets_table`.

Para crear nuestras migraciones en Laravel se usa el siguiente comando:

```
php artisan make:migration nombre_migracion
```

que nos crea el archivo limpio para escribir nuestra migración, o bien el comando:

```
php artisan make:migration nombre_migracion --create=nombre_tabla
```

que nos agrega una plantilla de trabajo básica para empezar a trabajar.

Como ejemplo del curso se tomara este comando:

```
php artisan make:migration crear_table_pasteles --create=pasteles
```

el cual nos dará este resultado:

```
created Migration: 2015_06_23_054801_crear_tabla_pasteles
```

Y nos creará además el siguiente archivo:

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CrearTablaPasteles extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('pasteles', function (Blueprint $table) {
            $table->increments('id');
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('pasteles');
    }
}
```

En el ejemplo observamos que ya tenemos el campo `'id'` de tipo `increments` que es equivalente a un campo en SQL así:

```
create table pasteles (id int auto_increment);
```

en nuestra base de datos, esta recibe dos parámetros, el primero es el nombre que va a recibir la tabla que si no mal recuerdan es el que se le dio en el comando y por lo cual ya se encuentra en su lugar, y el segundo parámetro es una función closure o función anónima que lo que hace es definir las columnas de nuestra tabla, a su vez esta función anónima recibe como parámetro un objeto de tipo **Blueprint** que se agregó dentro del namespace con la palabra **use** en la cabecera del archivo, el objeto `$table` es con el que vamos a trabajar para definir los campos, como se ve en la imagen anterior esto se logra escribiendo `$table->tipo_dato('nombre');`, y esto puede variar dependiendo el tipo de dato que se use y para ello podemos revisar la documentación oficial de Laravel [aquí](#) para poder ver todos los tipos de campos con los que contamos.

Ahora bien se puede observar que el archivo como tal no se llama simplemente `crear_tabla_pasteles` sino `2015_06_23_054801_crear_tabla_pasteles`, esto pasa porque Laravel al crear una migración agrega como prefijo la fecha y hora en la que fué creada la migración para poder ordenar qué migración va antes que otra, por lo cual si tu ejecutas este comando, obviamente el nombre de tu archivo será diferente pues la fecha y hora no pueden ser las mismas que la mia al crear este ejemplo. Además las migraciones que vienen por defecto en Laravel también se encuentran con este formato por lo cual podemos observar que estos dos archivos si tienen el mismo nombre.

Dentro de la estructura del archivo podemos ver dos funciones, una llamada `up()` y otra llamada `down()`, la primera función es en donde vamos a especificar la estructura de nuestra tabla, inicialmente y gracias al comando se encuentran ya algunas cosas escritas como los son la clase **Schema** en la cual se llama al método `create`, el cual nos permite crear la tabla

en nuestra base de datos, esta recibe dos parámetros, el primero es el nombre que va a recibir la tabla que si no mal recuerdan es el que se le dio en el comando y por lo cual ya se encuentra en su lugar, y el segundo parámetro es una función closure o función anónima que lo que hace es definir las columnas de nuestra tabla, a su vez esta función anónima recibe como parámetro un objeto de tipo **Blueprint** que se agregó dentro del namespace con la palabra **use** en la cabecera del archivo, el objeto `$table` es con el que vamos a trabajar para definir los campos, como se ve en la imagen anterior esto se logra escribiendo `$table->tipo_dato('nombre');`, y esto puede variar dependiendo el tipo de dato que se use y para ello podemos revisar la documentación oficial de Laravel [aquí](#) para poder ver todos los tipos de campos con los que contamos.

Si bien cada función realiza una tarea en específico, ¿Cuando es que se usan? o ¿Como se mandan a llamar?. Bueno para esto iremos nuevamente a nuestra linea de comandos.

Para correr o iniciar nuestras migraciones usamos el comando:

```
php artisan migrate
```

Con esto si es la primera vez que se ejecuta este comando se creará en nuestra base de datos la tabla **migrations** que es la encargada de llevar el control de que migraciones que ya han sido ejecutadas, con el fin de no correr el mismo archivo más de una vez si el comando se usa nuevamente.

Entonces si creamos nuestra migración `crear_tabla_pasteles` y usamos el comando `php artisan migrate` como resultado en nuestra base de datos se agregará la tabla `pasteles` y en la tabla `migrations` se añadirá el registro de la migración recién ejecutada.

Pero, ¿si quisiera eliminar la tabla con la función `down` de la migración `crear_tabla_pasteles`?

Esto se puede resolver de dos formas básicamente:

1. Con el comando `php artisan migrate:rollback` que lo que hará es deshacer la última migración ejecutada y registrada en la base de datos.
2. Con el comando `php artisan migrate:reset` que lo que hará es deshacer todas las migraciones de la base de datos.

- **Nota:** Un comando extra que nos permite actualizar las migraciones es el comando `php artisan migrate:refresh`, el cual es equivalente a usar `php artisan migrate:reset` y después `php artisan migrate`.

En el caso que necesitáramos agregar más campos a la tabla `pasteles`, podríamos simplemente ir a la migración `crear_tabla_pasteles` y en la función `up` poner la nueva columna, pero con esto perderíamos la primera versión de la tabla, entonces para poder ejemplificar como se agregan columnas con las migraciones crearemos una nueva que se llame `agregar_c Campos_tabla_pasteles` con los comandos que ya hemos visto:

1. Primero ejecutamos el comando: `php artisan make:migration agregar_c Campos_tabla_pasteles`, para crear la migración simple sin la plantilla.

2. Dentro de la función `up` agregamos los campos que necesitamos, en este caso solo agregaremos el nombre y el sabor.

3. Despues como la función `down` hace lo opuesto que la función `up`, dentro de esta eliminaremos los campos recién agregados.

Ahora el archivo resultante quedaría así:

```
2015-06-23 064353_agregar_c Campos_tabla_pasteles.php
1 <?php
2
3 use Illuminate\Database\Schema\Blueprint;
4 use Illuminate\Database\Migrations\Migration;
5
6 class AgregarCamposTablaPasteles extends Migration
7 {
8     /**
9      * Run the migrations.
10     *
11     * @return void
12     */
13     public function up()
14     {
15         Schema::table('pasteles', function(Blueprint $table)
16         {
17             $table->string('nombre', 60);
18             $table->enum('sabor', ['chocolate', 'vainilla', 'cheesecake']);
19         });
20     }
21
22     /**
23      * Reverse the migrations.
24      *
25      * @return void
26     */
27     public function down()
28     {
29         Schema::table('pasteles', function(Blueprint $table)
30         {
31             $table->dropColumn(['nombre', 'sabor']);
32         });
33     }
34 }
35 
```

Para poder agregar más columnas a las tablas desde Laravel en vez de llamar al método `create` llamamos al método `table` de la clase `Schema` pasandole como primer parámetro a que tabla se va a agregar los campos y como segundo parámetro la función anónima donde definimos que columnas se agregarán.

Y en la función `down` para eliminar columnas que vendría siendo lo opuesto de agregarlas, se llama al método `table` y dentro de la función anónima del objeto `$table` se usa el método `dropColumn()` que recibe como parámetro ya sea el nombre de una sola columna o un arreglo con todas las columnas que se desean eliminar.

Y ¡listo!, con esto podemos tener una idea inicial de como usar las migraciones, lo que para este ejemplo podría continuar sería agregar más columnas a la tabla `pasteles` y probar los comandos necesarios para poder deshacer los cambios de la primera vez que se corrió la migración con una nueva versión, ya sea sobre el mismo archivo o sobre otro nuevo.

Beneficios

- Tenemos un mayor control de las versiones de la base de datos.

- Podemos con un simple comando ver reflejados los cambios de nuestra base de datos.
- El lenguaje en el cual se trabaja sigue siendo PHP, por lo cual no se diferencia tanto de lo que ya nos acostumbraremos con Laravel.
 - La ultima versión de nuestra base siempre estará actualizada para todos los miembros del equipo de trabajo si usamos un control de versiones como **Git**.
 - Provee de portabilidad para diferentes gestores, usando el mismo código.

Seeders

Los Seeders por otra parte son archivos que nos van a permitir poblar nuestra base de datos para no tener que perder el tiempo escribiendo de forma manual todos los datos, un ejemplo, imagina llenar 15 tablas con 100 registros cada una y piensa en que entre cada tabla deben existir registros que se relacionan entre sí, eso suena de verdad horrible y tedioso, por lo cual Laravel nos salva con estos archivos Seeders.

Un Seeder se ubica en la carpeta `database/seeds/` de nuestro proyecto de Laravel y para poder crear un nuevo Seeder se usa el comando:

```
php artisan make:seeder nombre_seeder
```

Esto nos creará un archivo en la carpeta `database/seeds/` que tendrá el nombre que le demos en el comando, por ejemplo crearemos uno retomando el ejemplo anterior de las migraciones, se llamará `PastelesSeeder`, por lo cual el comando quedaría de la siguiente forma:

```
php artisan make:seeder PastelesSeeder
```

Con esto ya tenemos el archivo pero no es todo lo que necesitamos para poder trabajar con datos autogenerados, para ello usaremos un componente llamado **Faker** el cual se encargará de generar estos datos, por defecto el boilerplate del proyecto de Laravel 5.1 que estamos trabajando viene ya con **Faker** dentro del composer.json por lo cual ya debe estar instalado dentro de nuestro proyecto, ahora bien si estamos trabajando con una instalación Laravel 5.0 sin el componente **Faker** basta con ir al archivo composer.json y agregar en el `"require-dev"` las dependencias y para tener una idea más clara podemos ir a la página de [Packagist](#) donde se encuentra **Faker** o a su [Repositorio](#) en Github y ahí nos muestra que es lo que se debe agregar.

Quedando el archivo de la siguiente forma:

Al final solo se ocupa el comando `composer update` para actualizar las dependencias y descargar **Faker** al proyecto.

Una vez ya teniendo **Faker** iremos a nuestro archivo `PastelesSeeder` y dentro podremos observer que se encuentra una función llamada `run()` que es donde nosotros vamos a usar **Faker** para poblar, ahora bien antes de todo debemos agregar la clase de **Faker** a nuestro Seeder, para esto agregamos al inicio del archivo la linea:

```
use Faker\Factory as Faker;
```

Después crearemos una variable llamada `$faker` que nos servirá para poblar la base de datos, ahora bien usando la clase DB, si bien dentro del ejemplo queremos crear 50 pasteles vamos a crear un `for` para que ejecute nuestro código de inserción 50 veces y el componente de **Faker** en cada pasada cambiará los valores del registro que se va a agregar, quedando de esta forma:

```
$faker = Faker::create();
for ($i=0; $i < 50; $i++) {
    \DB::table('pasteles')->insert(array(
        'nombre' => $faker->firstNameFemale,
        'sabor' => $faker->randomElement(['chocolate', 'vainilla', 'cheesecake']),
        'created_at' => date('Y-m-d H:i:s'),
        'updated_at' => date('Y-m-d H:i:s')
    ));
}
```

Creamos nuestro objeto Faker, el cual puede generar información falsa para nuestra base de datos y ahora usamos la clase `DB` el método `table` para llamar la tabla donde se va a insertar la información y se le concatena el método `insert()` el cual recibe por parámetro un arreglo `clave => valor` con los campos de la tabla.

Faker tiene muchas variedades de datos, los cuales podemos consultar en su [Repositorio de Github](#) así como su uso básico.

En este ejemplo usamos una propiedad que se llama `firstNameFemale` para darle nombre al pastel y la propiedad `randomElement` que de un arreglo que se le da asigna un elemento de ese arreglo aleatoriamente.

Y ahora lo que sigue es abrir un archivo llamado `DATABASESeeder.php`, en este archivo se mandan a llamar todos los seeders en el orden que los necesitemos, en este archivo se agregará la linea:

```
$this->call('PastelSeeder');
```

que en si mandará a llamar nuestro seeder recién creado y para ejecutar este archivo se usa el comando:

```
php artisan db:seed
```

Y con esto queda poblada la tabla `Pastel` y lo puedes verificar en tu gestor de base de datos.

Cuando trabajamos con Migraciones y Seeder por primera vez puede parecer un poco más complicado que a lo que estamos acostumbrados pero las ventajas que nos da superan por mucho a la forma convencional, además de ser una forma más profesional de trabajar.

Unos comandos extras que nos pueden ser útiles son:

```
php artisan migrate --seed
```

El comando anterior lo que hace es realizar una combinación entre los comandos `php artisan migrate` y `php artisan db:seed`.

```
php artisan migrate:refresh --seed
```

El comando anterior lo que hace es realizar una combinación entre los comandos `php artisan migrate:refresh` y `php artisan db:seed`.

Modelos y uso de Eloquent

Eloquent

En Laravel podemos hacer uso de un **ORM** llamado Eloquent, un **ORM** es un **Mapeo Objeto-Relacional** por sus siglas en inglés (Object-Relational mapping), que es una forma de mapear los datos que se encuentran en la base de datos almacenados en un lenguaje de script SQL a objetos de PHP y viceversa, esto surge con la idea de tener un código portable con el que no tengamos la necesidad de usar lenguaje SQL dentro de nuestras clases de PHP.

Eloquent hace uso de los **Modelos** para recibir o enviar la información a la base de datos, para esto analizaremos el modelo que viene por defecto en Laravel, este es el modelo **User** que se ubica en la carpeta `app/`, los modelos hacen uso de PSR-4 y namespaces, un modelo nos ayuda a definir que tabla, atributos se pueden llenar y que otros se deben mantener ocultos.

Los modelos usan convenciones para que a Laravel se le facilite el trabajo y nos ahorre tanto líneas de código como tiempo para relacionar más modelos, las cuales son:

- El nombre de los modelos se escribe en singular, en contraste con las tablas de la BD que se escriben en plural.
- Usan notación UpperCamelCase para sus nombres.

Estas convenciones nos ayudan a detectar automáticamente las tablas, por ejemplo: el modelo **User** se encuentra en **singular** y con notación **UpperCamelCase** y para Laravel poder definir que tabla es la que esta ligada a este modelo le es suficiente con realizar la conversión a notación **underscore** y **plural**, dando como resultado la tabla: **users**.

Y esto aplica para cuando queremos crear nuestros modelos, si tenemos una tabla en la base de datos con la que queremos trabajar que se llama **user_profiles**, vemos que se encuentra con las convenciones para tablas de bases de datos (plural y underscore), entonces el modelo para esta tabla cambiando las convenciones sería: **UserProfile** (singular y **UpperCamelCase**).

Retomando el ejemplo que vimos en el [Capítulo 6](#) sobre la migración de pasteles, crearemos ahora un modelo para poder trabajar con esa tabla, el cual recibira el nombre de **Pastel** y el comando para poder crear nuestro modelos es:

```
php artisan make:model Pastel
```

Con esto se generara el archivo en donde ya se encuentra el modelo **User** en la carpeta `app/` y dentro de el vamos a definir la tabla que se va a usar con esta linea:

```
protected $table = 'pasteles';
```

¿Pero no se suponia que Laravel identificaba automáticamente que tabla usar?

Si lo hace pero si cambiamos las convenciones del modelo **Pastel** el resultado sería **pastels** y nuestra tabla se llama **pasteles**, esto es un problema para nosotros por el hecho del uso del lenguaje español porque la conversion de singular a plural no es la misma que la forma en que se hace en inglés, debido a esto nos vemos forzados a definir el nombre de la tabla.

Bien una vez creado nuestro modelo pasaremos a crear una ruta de tipo **get** en nuestro archivo **routes.php**, posteriormente estudiaremos el enrutamiento básico en Laravel en el Capítulo 9, por el momento solo seguiremos el ejemplo, que quedaría de la siguiente forma:

```
Route::get('pruebasPastel', function() {
});
```

Dentro de esta ruta de prueba vamos a usar nuestro modelo, pero como estamos usando la especificación PSR-4 debemos incluir el namespace del modelo al inicio del archivo, que sería igual a esto:

```
use Curso\Pastel;
```

Con esto estamos diciendo que incluya la clase **Pastel** que es nuestro modelo, y con esto podemos ya hacer consultas a nuestra BD y mapear a objetos PHP. En la [documentación oficial](#) de Laravel podemos ver todas las opciones que nos permite **Eloquent**, unas de las instrucciones básicas de este son **get()** que nos regresa todos los registros de la BD y **first()** que nos regresa el primer registro de una selección.

A su vez podemos unir esto a más filtros de selección SQL, como por ejemplo seleccionar el primer pastel de vainilla, la sintaxis de Eloquent sería la siguiente:

```
$pastel = Pastel::where('sabor', 'vainilla')->first();
```

Esto nos va a dar el primer pastel sabor vainilla, pero si quisieramos todos los pasteles de vainilla cambiariamos el método **first()** por el metodo **get()** para obtener todos.

Y si queremos ver el resultado de esto y que de verdad estamos haciendo lo correcto podemos usar la función `dd()` para mostrar en pantalla el valor de una variable, con esto entonces nuestra ruta le agregaríamos lo siguiente:

```
Route::get('pruebasPastel', function(){
    $pastelles = Pastel::where('sabor', 'vainilla')->get();
    dd($pastelles);
});
```

Y en el navegador deberíamos ver algo como esto:

```
Collection {#253 ▶
#items: array:18 [▶
0 => Pastel {#254 ▶
1 => Pastel {#255 ▶
2 => Pastel {#256 ▶
3 => Pastel {#257 ▶
4 => Pastel {#258 ▶
5 => Pastel {#259 ▶
6 => Pastel {#260 ▶
7 => Pastel {#261 ▶
8 => Pastel {#262 ▶
9 => Pastel {#263 ▶
10 => Pastel {#264 ▶
11 => Pastel {#265 ▶
12 => Pastel {#266 ▶
13 => Pastel {#267 ▶
14 => Pastel {#268 ▶
15 => Pastel {#269 ▶
16 => Pastel {#270 ▶
17 => Pastel {#271 ▶
] ▶
```

Esto es la función `dd($pastelles)` mostrando el contenido de la variable `$pastelles`. Ahora bien si tuvieramos la necesidad de realizar siempre un mismo filtro, Eloquent nos provee de una herramienta llamada **scopes** que lo que realizan son consultas en específico encapsulándolas dentro de funciones en el modelo, por ejemplo si quisieramos que el modelo **Pastel** tuviera una función que me diera todos los pasteles de vainilla, otra de chocolate y otra función mas para cheesecake, entonces podría crear un scope para cada una.

Con el ejemplo de la ruta `pruebasPastel` para el sabor vainilla:

```
public function scopeVainilla($query){
    return $query->where('sabor', 'vainilla');
}
```

Los scopes en la función se debe iniciar el nombre de la función con la palabra **scope** y seguido en notación camelCase el nombre con el cual se va a llamar el scope. Y su equivalente dentro de la ruta sería la siguiente:

```
Route::get('pruebasPastel', function(){
    $pastelles = Pastel::vainilla()->get();
    dd($pastelles);
});
```

También podemos crear scopes dinámicos de la siguiente forma:

```
public function scopeSabor($query, $sabor){
    return $query->where('sabor', $sabor);
}
```

Esto nos daría una función genérica para obtener los pasteles de cierto sabor y su implementación sería así:

```
Route::get('pruebasPastel', function(){
    $pastelles = Pastel::sabor('vainilla')->get();
    dd($pastelles);
});
```

Además con Eloquent también podemos insertar, actualizar o eliminar registros, por ejemplo:

Para insertar la sintaxis sería la siguiente:

```
$pastel = new Pastel;
$pastel->nombre = 'Pastel Richos Style';
$pastel->sabor = 'chessecake';
$pastel->save();
```

Para actualizar sería la siguiente:

```
$pastel = Pastel::find(51);
$pastel->sabor = 'chocolate';
$pastel->save();
```

Para eliminar sería la siguiente:

```
$pastel = Pastel::find(51);
$pastel->delete();
```

o bien podríamos destruir el registro directamente con el modelo si tenemos su ID:

```
Pastel::destroy(51);
```

Los model factories son una excelente forma de poblar nuestra base de datos con datos de prueba generados automáticamente. Laravel en su versión 5.1 incorpora este nuevo componente por defecto, en versiones anteriores a Laravel 5.1 era necesario agregar el componente faker en nuestro **composer.json** y realizar el proceso de manera manual en los archivos seeders, para más información sobre este proceso puedes visitar el link de github del componente [Faker](#).

Los model Factories en realidad también trabajan con el componente Faker, esto lo podemos confirmar si miramos nuestro **composer.json**, sin embargo, nos ofrecen una manera más elegante y ordenada de trabajar.

Laravel 5.1 trae un ejemplo de como usar este nuevo componente, lo podemos encontrar en el archivo **database/factories/ModelFactory.php**.

El método `$factory->define()` regresa un array con los datos del modelo que se va a poblar, recibe como primer parámetro el modelo con el que deseamos trabajar y como segundo parámetro una función que recibe como parámetro un objeto `$faker`.

Ejemplo:

```
$factory->define(App\User::class, function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => Str::random(10),
        'remember_token' => Str::random(10),
    ];
});
```

El método `$factory->defineAs()` regresa un array con los datos del modelo que se va a poblar, recibe como primer parámetro el modelo con el que deseamos trabajar, como segundo parámetro un tipo específico de poblado y como tercer parámetro una función que recibe como parámetro un objeto `$faker`.

Ejemplo:

Model Factories

```
// Creamos un model factory para poblar usuarios de tipo administrador
$factory->defineAs(AppUser::class, 'administrador', function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'type' => 'administrador',
        'remember_token' => str_random(10),
    ];
});
```

```
// Creamos un model factory para poblar usuarios de tipo encargado
$factory->defineAs(AppUser::class, 'encargado', function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'type' => 'encargado',
        'remember_token' => str_random(10),
    ];
});
```

En el ejemplo de arriba hemos creado dos tipos de poblado para el modelo **User**, uno será para poblar usuarios de tipo "administrador" y otro para usuarios de tipo "encargado".

Una vez creados los Model Factories, debemos ir al archivo **database/seeds/DatabaseSeeder.php** y ejecutar el poblado dentro del método `run` como en el siguiente ejemplo:

```
public function run()
{
    Model::unguard();
    factory('cursoUser', 100)->create();
    // Opcionalmente podemos agregar el tipo de poblado
    factory('cursoUser', 'administrador', 100)->create();
    factory('cursoUser', 'encargado', 100)->create();
    // $this->call('UserTableSeeder');
    Model::guard();
}
```

El objeto `factory` recibe como parámetros el nombre del modelo, el tipo de poblado como parámetro opcional y el número de registros que deseamos crear. Con el método `create` realizamos el poblado de datos.

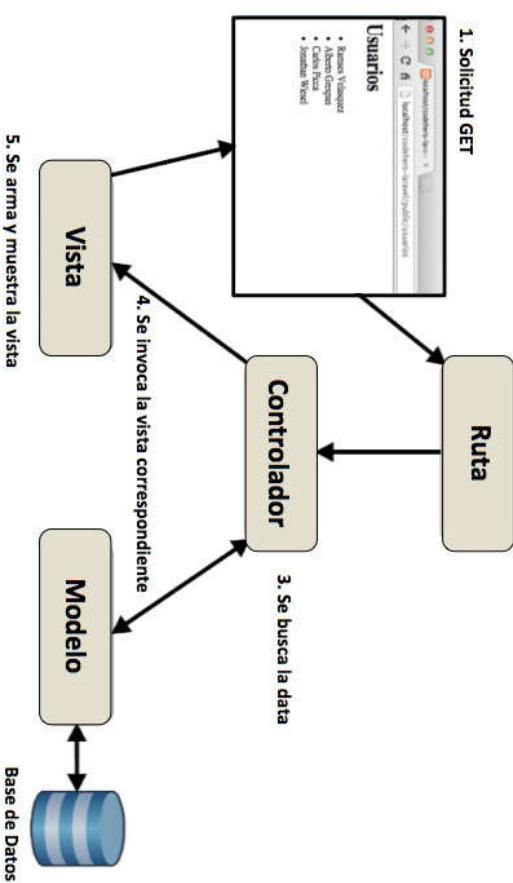
Ejemplos:

```
factory('cursoUser', 100)->create();
// Opcionalmente podemos agregar el tipo de poblado
factory('cursoUser', 'administrador', 100)->create();
```

Enrutamiento básico

La siguiente imagen muestra el proceso que se realiza cuando ingresamos a una URL. Además muestra la arquitectura del patrón **MVC** que utiliza laravel para el desarrollo de proyectos.

1. Se dirige la petición al controlador



Cuando ingresamos a una url directamente desde el navegador lo hacemos mediante una petición http de tipo GET, esta solicitud se envía al archivo routes.php ubicado dentro de app/Http/routes.php, en caso de no existir nos dará un error, si la ruta existe, nos llevará a un controlador en el cuál se encuentra la lógica , el controlador interaccionará con un modelo (opcionalmente) para recuperar información de una base de datos. Esta información llega al controlador y desde el controlador invitamos una vista, las vistas se encuentran en el directorio resources/views, finalmente la vista se carga y se muestra en el navegador.

Así es como funciona el modelo MVC (Model-View-Controller).

Supongamos que queremos ingresar a la siguiente URL `http://dominio.com/saludo` y desplegar una página con el mensaje "Bienvenido :)". En laravel la porción /saludo pertenecería a una ruta que regresa una respuesta o una vista dependiendo lo complejo que llegue a ser lo que queramos mostrar. La parte de dominio.com pertenecería a localhost

si lo andamos probando de manera local. En nuestro ejemplo lo que mostraremos es un mensaje muy simple por lo cual no es necesario hacer mostrar una vista. Para lograrlo haremos lo siguiente:

```
Route::get('saludo', function () {
    return "Bienvenido :)";
});
```

Lo que debería mostrar un mensaje similar a este:

Tipos de rutas por encabezado Http

Las rutas están siempre declaradas usando la clase Route . Eso es lo que tenemos al principio, antes de ':'. La parte get es el método que usamos para 'capturar' las peticiones que son realizadas usando el verbo 'GET' de HTTP hacia una URL concreta.

Como verás, todas las peticiones realizadas por un navegador web contienen un verbo. La mayoría de las veces, el verbo será GET , que es usado para solicitar una página web. Se envía una petición GET cada vez que escribes una nueva dirección web en tu navegador.

Aunque no es la única petición. También está POST , que es usada para hacer una petición y ofrecer algunos datos. Normalmente se usa para enviar un formulario en la que se necesita enviar los datos sin mostrarlo en la URL.

Hay otros verbos HTTP disponibles. He aquí algunos de los métodos que la clase de enrutado tiene disponible para ti:

```
Route::get();
Route::post();
Route::any();
Route::delete();
Route::put();
```

Cualquier método de la clase Route recibe siempre dos argumentos, el primero es la URI con la que queremos hacer coincidir la URL y el segundo es la función a realizar que en este caso es un Clousure que no es otra cosa que una función anónima, es decir, que no tiene un nombre.

Rutas de tipo get

En este caso ocuparemos el método estático **get** para escribir una ruta que responda a una petición de este tipo, las rutas de tipo **get** son las más usadas. El método estático **get** recibe como primer parámetro un string indicando la url con la cual vamos a ingresar, el string `"/alumnos"` responderá a la solicitud `http://localhost:8000/alumnos`, el string `"/"` equivale a `http://localhost:8000`, es decir, la ruta por defecto. Como segundo parámetro el método estático **get** recibe un `closure` (una función sin nombre) que puede devolver una `view` o un `string`.

```
// ruta de tipo GET que devuelve una vista
Route::get('/', function () {
    return view('welcome');
});

// ruta de tipo GET que devuelve un simple string
Route::get('/', function () {
    return "Hola mundo";
});
```

El método **view** dentro del closure recibe como parámetro el nombre de una vista sin la extensión. En el ejemplo de arriba la vista `welcome` se encuentra ubicada en

`resources/views/welcome.blade.php` si escribimos `view('pasteles.lista_pasteles')`

estamos indicando que regresará el archivo `/lista_pasteles.blade.php` ubicado en

`resources/views/pasteles/lista_pasteles.blade.php`. Las vistas las veremos en el [capítulo 10](#).

Las rutas pueden ser relacionadas con métodos de un controlador. En el siguiente ejemplo, la ruta `http://localhost:8000/home` regresará lo que indiquemos en el método `index` del **Controller HomeController**.

```
Route::get('home', 'HomeController@index');
```

Parámetros en las rutas de tipo get

Los parámetros de las rutas pueden ser utilizados para introducir valores de relleno en tus definiciones de ruta. Esto creará un patrón sobre el cual podamos recoger segmentos de la URL y pasártolos al gestor de la lógica de la aplicación. Para dejarlo un poco más claro pondremos unos ejemplos.

```
Route::get('Libros/{genero}', function($genero){
    switch ($genero) {
        case 'amor':
            return "Libros de Amor";
            break;
        case 'terror':
            return "Libros de Terror";
            break;
        case 'drama':
            return "Libros de Drama";
            break;
        case 'aventura':
            return "Libros de Aventura";
            break;
        default:
            return "No existe esa categoría de libros";
            break;
    }
});
```

De igual forma es posible restringir rutas por medio de expresiones regulares como por ejemplo:

```
Route::get('/posts/{numero?}', function($numero=1){
    return "Vista número {$numero}";
})->where('numero', '[0-9]+');
```

En la imagen anterior podemos ver dos conceptos nuevos, el uso de valores por default lo cual logramos con el símbolo (?) después del nombre de la variable y en la función asignandole un valor por defecto, en este caso el entero 1.

Lo segundo que vemos es el uso del método `where` el cual nos permite establecer expresiones regulares a las variables que usamos en la construcción de las URIs.

Vistas y Blade

Las vistas en Laravel son la parte pública que el usuario de nuestro sistema va a poder ver, se escriben en HTML junto con un motor de plantillas llamado **Blade**. Las vistas se encuentran ubicadas en la carpeta `resources/views/` y Laravel por defecto trabaja con la idea de que tenemos que escribir la menor cantidad de código repetido, modularizar nuestro código en donde mas se pueda, y si esto lo aplicamos en nuestros modelos, controladores y demás partes de nuestro proyecto, entonces, **¿Por que no hacerlo tambien en las vistas?**.

Laravel usa unos archivos que se llaman **plantillas** o **templates** que suelen ser nuestros archivos principales, que tienen los segmentos de código que se repiten en mas de una vista, como por ejemplo la barra de navegación, un menú de opciones, la estructura del acomodo de nuestro proyecto, etc. y como deben de estar prácticamente presentes en todos lados, no tiene sentido estarlos repitiendo en todas las vistas. Por defecto Laravel contiene un **template** llamado **app.blade.php**, usualmente los **templates** contienen el head del HTML, las ligas del **CSS** del sistema y una sección exclusiva para los archivos Javascript.

Además de los **templates**, se cuentan con archivos que se llaman **partials**, estos archivos son pequeños segmentos de código que suelen ser usados comúnmente en partes del sistema en específico, como los formularios o secciones de mensajes, estos archivos surgen por el código que es mas pequeño que repetimos mucho pero no es lo suficientemente grande como para considerarlo un **template**.

Esto hace que las vistas de cada parte del proyecto, que suelen ser llamadas por una ruta o controlador sean mucho mas pequeñas que usando otro tipo de frameworks para desarrollo Web, y para poder unir todos estos archivos o piezas del rompecabezas usamos el motor de plantillas de Laravel llamado **BLADE**.

Antes de ver mas sobre el motor de plantillas **Blade**, veremos como trabajar con las Vistas y llamarlas desde una ruta, crearemos un vista simple con un archivo nuevo en la carpeta `resources/views/` llamado **saludo.blade.php** con el siguiente código:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
</head>
<title>Vista ejemplo</title>
<body>
<h1>Hola mundo desde Laravel</h1>
</body>
</html>
```

Es un HTML simple con un título 1, ahora vamos a crear una ruta que nos muestre esta vista:

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

De esta forma con la función `view()` le estamos diciendo a Laravel que busque dentro de la carpeta `resources/views/` la vista **saludo.blade.php**, por convención las vistas Laravel no necesita que especifiquemos la extensión **.blade.php**, sino solo su nombre. Una vez hecho esto debemos ver este resultado o uno similar:



Hola mundo desde Laravel

Continuando con el ejemplo de los Pasteles vamos a mandar a la vista el nombre de un pastel, dentro de la ruta `saludo` vamos a obtener el primer Pastel de chocolate de nuestra BD y a poner ese nombre en vez del mensaje. Para esto podemos usar el scope de sabor

para obtener los pasteles de chocolate y después decirle que con el método `first()` nos regrese el primer pastel y eso guardarlo en una variable, dejando la ruta de la siguiente forma:

```
Route::get('saludo', function(){
    $pastel = Pastel::sabor('chocolate')->first();
    return view('saludo')->with('pastel', $pastel->nombre);
});
```

De esta forma estamos diciendo a la ruta que nos regrese la vista `saludo.blade.php` con una variable llamada `pastel`, que es el nombre del pastel, pero esto por si solo no lo va a mostrar el navegador, solo va a mandar la variable, para que el navegador la muestre debemos agregar un título donde este esa variable de esta forma:

```
<h2>{{ $pastel }}</h2>
```

Esta linea va justo abajo del mensaje **Hola mundo desde Laravel**, y ahora si debemos de ver algo parecido a esto ya que nuestras BD tienen cosas diferentes y gracias a Faker ninguno de nuestros resultados debería ser igual:



Hola mundo desde Laravel

Pastel Elisa Stracke

Esto es el equivalente a `<?= $pastel ?>` y aunque con un ejemplo tan sencillo no se ve demasiada diferencia, con lo siguiente podremos verificar la potencia de este motor de plantillas.

Usualmente las estructuras de control que conocemos las usamos en los archivos PHP dedicados al Back-end (lado del servidor), pero blade nos da una sintaxis muy comoda para este tipo de estructuras que con PHP plano son muy sucias e incomodas de usar.

Para cada una de estas estructuras como lo son **If**, **else**, **elseif**, **for**, **foreach**, etc., se antepone un `@` para usar estas estructuras y listo!! eso es suficiente, pero a diferencia de como estamos a costumbres de encapsular un grupo de sentencias o líneas de código con llaves `{}`, en blade definimos el fin de una estructura con un `@end` seguido del nombre de la estructura que usamos, por ejemplo:

```
<h1>Lista de pasteles</h1>
@foreach($pasteles as $pastel)
    <h2>{{ $pastel->nombre }}</h2>
@endforeach
```

Ahora si bien usamos los caracteres de las dobles llaves y no sabemos bien que son, esto es parte de la sintaxis que ahora veremos con **Blade**.

Blade

Blade nos provee de muchas ventajas (así como casi todo en Laravel), además de modularizar nuestras vistas de una forma sorprendente, tambien nos permite usar estructuras de control y variables de PHP directamente en ellas, aunque esto ya era posible antes usando las etiquetas de php, por ejemplo:

```
<?php echo $var ?>
<?= $var ?>
```

Pero esto ademas de ser un poco incomodo de escribir deja nuestras vistas mucho más difíciles de entender y sucias por la mezcla de tanto código.

Entonces para el ejemplo anterior usamos el siguiente código:

```
{{ $pastel }}
```

Entonces en la ruta donde regresamos solo un nombre de un pastel, podríamos regresar todos los pasteles y escribir una lista de todo los pasteles de un cierto sabor e imprimirla en la vista.

Un ejemplo para el **if** sería:



```

@if( $pastel->count() > 10 )
    <h1>Hay muchos Pasteles</h1>
@endif
<h1>Lista de pasteles</h1>
@foreach($pastel as $pastel)
    <h2>{{ $pastel->nombre }}</h2>
@endforeach

```

El if nos dice, si el numero de pasteles que recibimos es mayor a 10 entonces escribe el título **Hay muchos Pasteles**.

Esto aplica para todas las estructuras, su lógica es la misma pero solo cambia un poco su sintaxis, pero hace que el HTML quede mas limpio que si incrustaramos PHP plano dentro de nuestra vista.

Templates y Partials

Anteriormente hablábamos de **templates y partials**, describirímos un poco de como se trabaja con esta estructuras de Blade y sus beneficios:

Templates

Estos archivos como se menciona al principio del capítulo son **plantillas** que nos ahorran mucho código o leguaje HTML, y para usar un **template** se usa la sentencia:

```
@extends('template')
```

Claramente se tendría que sustituir la palabra **template** dentro de la sentencia **extends** por el nombre de la vista que va a funcionar como **template** o **plantilla**.

Un template es una vista como las demás, simplemente que dentro de ella se usan otras sentencias que nos va a permitir definir areas del archivo que se van a poder sustituir mas adelante dentro de otra vista si es que lo deseamos. Para esto se ocupa la sentencia:

```
@yield('nombre_sección')
```

Para declarar una sección que se va a rellenar en otro lugar:

```
@section('nombre_sección')
```

que funciona de la misma forma que **yield()** con la diferencia que en la sección puedes definir HTML por defecto en caso de no definir la sección con un nuevo HTML.

Definiremos nuestra vista recien creada **saludo.blade.php** para que use un template, por defecto Laravel trae uno que se llama **app.blade.php**, ese es el que usaremos para este ejemplo.

El template **app** por defecto tiene definida un **yield** llamado **content** que significa contenido, por lo cual la lista de pasteles que tenemos la vamos a agregar en esta parte y la vista quedaría de la siguiente forma:

```

@extends('app')

@section('content')
    <h1>Lista de pasteles</h1><br>
    @if( $pastel->count() > 10 )
        <h2>Hay muchos Pasteles</h2><br>
    @endif
    @foreach($pastel as $pastel)
        <h4>{{ $pastel->nombre }}</h4>
    @endforeach
    @stop

```

Ahora nuestra vista ya no tiene el encabezado HTML normal ni las etiquetas **<body>** ni **<html>**, sino que estamos diciendo que vamos a extender del template **app** y que el yield **content** lo vamos a sustituir por nuestro propio contenido, cabe mencionar que aunque en el template se uso la sentencia **yield('content')**, al momento de sustituirlo la vamos a cambiar por **section('content')**, por lo cual en todas las vistas hijas del template solo se va a definir secciones y el fin de esa sección se va a declarar con la sentencia **@stop**.

Ahora el resultado sería algo parecido a esto:



Lista de pasteles

Hay muchos Pasteles

```
<h1>Lista de pasteles</h1><br>
@if( $pasteles->count() > 10 )
    <h2>Hay muchos Pasteles</h2><br>
@endif
<ul>
    @foreach($pasteles as $pastel)
        <li>{ $pastel->nombre }</li>
    @endforeach
</ul>
```

Pastel Bernice Battenberg
Pastel Etta Stacks
Pastel Gail Beer
Pastel Gertie Jacobs
Pastel Queen Tamar
Pastel Linda Stacks
Pastel Lydia Mitchell
Pastel Marlene Tiel
Pastel Connie McDermott

Y nuestra vista **saludo.blade.php** quedaría de esta forma una vez que ya incluyamos nuestro partial:

```
@extends('app')

@section('content')
    @include('pastelles.partials.lista')
@stop
```

Continuaremos con los partials, basicamente es lo mismo que ya hemos visto pero con una sentencia mas que se llama `include('nombre.partial')`, la cual esta incluyendo o incrustando un archivo de HTML, podemos hacer un siml con los **use** de PSR-4 o los **import** de Java, a diferencia de que esto lo incluye justo en el lugar donde lo definimos.

Vamos a verlo con un ejemplo práctico.

Dentro la actual vista **saludo.blade.php**, vamos a quitar todo el HTML Blade que definimos para crear esta lista pequeña de pasteles y lo vamos a guardar en nuevo archivo, para esto vamos a crear una carpeta llamada **pasteles** y dentro otra carpeta llamada **partials**, donde vamos a guardar la vista de nuestro nuevo partial, quedando la ruta de la siguiente forma: `resources/views/pasteles/partials/`.

Ahi vamos a crear un archivo llamado **lista.blade.php** y dentro de este archivo vamos a cortar el código de nuestra vista saludo, quedando así:

- **@yield('nombre')**: Esta sentencia nos permite declarar un futuro **section** de nuestro HTML que se definira en las vistas que son heredadas y no puede agregarse algun tipo de contenido por defecto, este sólo se usa en archivos que toman el rol de **Template**.
- **@section('nombre')**: Esta sentencia tiene dos usos dependiendo de que queremos declarar, el primero es que nos permite declarar como su nombre lo dice una sección dentro del template que puede tener un contenido por defecto que si no es redefinido en la vista que herede el template entonces aparecera; el segundo nos permite asignar

Resumen, Anotaciones e información adicional

Blade es un motor de plantillas potente que nos permite modularizar y estilizar a un gran nivel nuestro HTML.

Como recordatorio listaremos algunas de las sentencias de **Blade** junto con su función:

- **@extends('nombre_template')**: Esta sentencia nos ayuda a decirle a una vista cual va a ser el template que se va a usar.

- **@yield('nombre')**: Esta sentencia nos permite declarar un futuro **section** de nuestro HTML que se definira en las vistas que son heredadas y no puede agregarse algun tipo de contenido por defecto, este sólo se usa en archivos que toman el rol de **Template**.
- **@section('nombre')**: Esta sentencia tiene dos usos dependiendo de que queremos declarar, el primero es que nos permite declarar como su nombre lo dice una sección dentro del template que puede tener un contenido por defecto que si no es redefinido en la vista que herede el template entonces aparecera; el segundo nos permite asignar

el contenido en una sección que fue declarada en nuestro **template**, es decir esta palabra **section** se usa tanto en el template como en las vistas hijas, una diferencia mas es que si se usa en el **template** entonces la sección termina con un **@show**, pero si se usa en una vista hija entonces termina la sección con un **@stop**.

- **@show:** Esta sentencia se usa para decir donde termina el **section** definido en el **template**.
- **@parent:** Esta sentencia nos ayuda a cargar el contenido por defecto de un **section** del template, esto podemos usarlo cuando queremos agregar mas contenido dentro pero sin alterar el contenido por defecto, es decir agregarle mas HTML, esta sentencia se usa dentro de un **section**, podemos hacer un simil con el **super()** de Java que sirve para llamar al constructor de la superclase de la que se hereda.
- **@stop:** Esta sentencia nos permite decir donde termina un **section** cuando se usa el section dentro de las vistas hijas.

- **@include('ruta.nombre')**: Esta sentencia nos agrega en el lugar donde sea usada un archivo blade.php que contiene un **partial** o fragmento parcial de HTML, si ese partial se encuentra en la raíz de las vistas no necesita mas que el nombre sin la extensión **blade.php**, pero si está dentro de, por ejemplo, la carpeta "**views/admin/users/**" llamado **table.blade.php** para poder ser incluido se usaria la ruta junto con el nombre quedando como `@include('admin.users.table')`, views no se contempla pues es la raíz de las vistas.

Para mas información de **Blade** podemos ir a la documentación oficial de Laravel sobre templates.

Los controladores nos ayudan a agrupar estas peticiones en una clase que se liga a las rutas, en el archivo `app/Http/routes.php`, para esto usamos un tipo de ruta llamada resource:

```
Route::resource('pasteles', 'PastelesController');
```

Asociando los métodos de la siguiente forma:

- **GET:** index, create, show, edit.
- **POST:** store.
- **PUT:** update.
- **DELETE:** destroy.
- **PATCH:** update.

Esta ruta nos creara un grupo de rutas de recursos con las peticiones que estas mencionadas arriba: **index, create, show, edit, store, update, destroy**. Estas son las operaciones mas usadas en una clase y para no tener que crear una ruta para cada método es que Laravel agrupa todo esto con una ruta de tipo resource que se liga a un controlador.

Estos métodos significan:

- **index:** Es el método inicial de las rutas resource, usualmente lo usamos para mostrar una vista como página principal que puede contener un catálogo o resumen de la información del modelo al cual pertenece o bien no mostrar información y solo tener la función de página de inicio.

Controladores

En lugar de definir en su totalidad la lógica de las peticiones en el archivo routes.php, es posible que deseé organizar este comportamiento usando clases tipo Controller. Los **Controladores** puede agrupar las peticiones HTTP relacionada con la manipulación lógica en una clase. Los **Controladores** normalmente se almacenan en el directorio de aplicación `app/Http/controllers/`.

Un controller usualmente trabaja con las peticiones:

- **GET.**

- **POST.**

- **PUT.**

- **DELETE.**

- **PATCH.**

- **create:** Este método lo podemos usar para direccional el sistema a la vista donde se van a recolectar los datos(probablemente con un formulario) para después almacenarlos en un registro nuevo, usualmente redirigir al index.
- **show:** Aquí podemos hacer una consulta de un elemento de la base de datos o de todos los elementos o registros por medio del modelo para realizar una descripción.
- **edit:** Este método es similar al de **create** porque lo podemos usar para mostrar una vista que recolecta los datos pero a diferencia de **create** es con el fin de actualizar un registro.
- **store:** Aquí es donde se actualiza un registro en específico que proviene del método **create** y normalmente redirige al index.
- **update:** Al igual que el **store**, solo que en vez de provenir de **create** proviene de **edit** y en vez de crear un nuevo registro, busca un existente y lo modifica, también suele redirigir al index.
- **destroy:** En este método usualmente se destruye o elimina un registro y la petición puede provenir de donde sea siempre y cuando sea llamado con el método **DELETE**, después puede redirigir al index o a otro sitio dependiendo si logro eliminar o no.

Ahora esto no quiere decir que un controlador necesariamente debe ejecutar estas peticiones obligatoriamente, podemos omitirlas o incluso agregar mas.

Para crear controladores en Laravel usamos artisan con el siguiente comando:

```
php artisan make:controller NameController
```

El comando anterior creara un controlador en la carpeta `app/Http/controllers/` que por defecto va a tener todos estos métodos dentro de si, entonces agregaremos la ruta de tipo resource anterior al archivo de rutas y correremos el siguiente comando en la consola:

```
php artisan make:controller PastelesController
```

Con esto vamos a poder trabajar para cada método del controlador una ruta y las funciones internas son las que se van a ejecutar, el archivo creado se verá de la siguiente manera:

```

x  □ - ~/Projects/Curso-Laravel/Projectof/app/Http/Controllers/PastelesController.php
File Edit Selection Find View Goto Tools Project Preferences Help
◀ ▶ chapter11.md
1 <?php
2
3 namespace Curso\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 use Curso\Http\Requests;
8
9 class PastelesController extends Controller
10 {
11     /**
12      * Display a listing of the resource.
13      *
14      * @return Response
15     */
16     public function index()
17     {
18         /**
19          * Show the form for creating a new resource.
20          */
21         /**
22          * Store a newly created resource in storage.
23          *
24          * @return Response
25         */
26         public function create()
27         {
28             /**
29             */
30         }
31         /**
32          * Store a newly created resource in storage.
33          *
34          * @return Response
35         */
36         public function store()
37         {
38             /**
39             */
40         }
41         /**
42          * Display the specified resource.
43         */
44     }
}
2 lines, 6 characters selected

```

En la linea de comandos podemos ver todas las rutas que nuestro proyecto tiene registradas:

```
php artisan route:list
```

Este comando nos va a mostrar en la consola un resultado similar a esto:

Domain	Method	URL	Name	Action	Middleware
localhost:8000	GET, HEAD	/		Closure	
localhost:8000	GET, HEAD	/saludo	home	Closure	
localhost:8000	POST	/pasteleras	pasteleras	Closure	
localhost:8000	GET, HEAD	/pasteleras/create	pasteleras.create	Closure	
localhost:8000	POST	/pasteleras	pasteleras.store	Closure	
localhost:8000	PUT, PATCH	/pasteleras/{pasteleras}	pasteleras.update	Closure	
localhost:8000	DELETE	/pasteleras/{pasteleras}	pasteleras.destroy	Closure	
localhost:8000	GET, HEAD	/auth/login	auth.login	Closure	
localhost:8000	POST	/auth/logout	auth.logout	Closure	
localhost:8000	GET, HEAD	/auth/register	auth.register	Closure	
localhost:8000	POST	/auth/register	auth.register	Closure	
localhost:8000	GET, HEAD	/probando/sel	probando.select	Closure	
localhost:8000	GET, HEAD	/debugbar/open	debugbar.open	Closure	
localhost:8000	GET, HEAD	/debugbar/assets/{asset}	debugbar.assets	Closure	
localhost:8000	GET, HEAD	/debugbar/assets/{asset}/script	debugbar.assets.script	Closure	

aquí podemos ver el nombre de nuestras rutas, de que tipo son, si es que reciben parámetros y como se llaman, esta información es muy útil para poder asociar los métodos del controlador con las rutas y también como es que las vamos a usar en el navegador.

Por ejemplo la ruta `/pasteleras/{pasteleras}` de tipo **GET** con el nombre **pasteleras.index**, se asocia a la función **index()** del controlador **PastelerasController** y por consecuente lo que hagamos en esa función lo podremos ver en el navegador.

Los controladores son un tema complicado y extenso así como el enrutamiento aunque en el curso solo vimos **enrutamiento basico**, por lo cual dejamos los links de la documentación oficial de [Controladores](#) y de [Enrutamiento](#) en la versión 5.1 de Laravel.

Validaciones en Laravel

Existen varias formas de validar nuestra aplicación para cubrir aspectos de seguridad como SQL Injection, ataques XSS o CSRF, algunas de ellas son:

- Validación de lado del cliente (Javascript y etiquetas HTML).
- Validación a nivel de base de datos (Migraciones y modelos).
- Validación de formularios (Request).

Validación del lado del cliente:

Podemos validar que los campos de un formulario sean requeridos al agregar el atributo `required`.

```
<form action="demo_form.asp">
  username: <input type="text" name="username" required>
  <input type="submit">
</form>
```

El atributo `required` es un atributo booleano. Cuando esta presente, este especifica que un campo debe ser rellenado antes de ser enviado el contenido del formulario.

El atributo `required` trabaja con los siguientes tipos de input:

- text
- search
- url
- tel
- email
- password
- data pickers
- number
- checkbox

El atributo *pattern*

Con se menciono anteriormente, con `required` solo se necesita de cualquier valor en el elemento `<input>` para ser válido, pero utilizando el atributo `pattern` en conjunto, se logra que se verifique no solo la presencia de un valor, sino que este valor debe contener un

formato, una longitud o un tipo de dato específico. Esto último se logra definiendo un patrón con expresiones regulares.

```
<label for="tel">Teléfono 10 dígitos empezando por 228</label>
<input type="text" pattern="^228\d{8}$">
```

Para utilizar el atributo `pattern` es recomendable utilizar el `type="text"` y no un `type` de los predefinidos en HTML5 que ya cuentan con patrones de validación en el propio navegador. Mezclar ambos puede llevar a resultados inesperados.

Validación de formularios con plugins JQuery

El mejor plugin JQuery para validar formularios es [formvalidation](#). Sin embargo `formvalidation` es un plugin que tiene un costo dependiendo la licencia que queramos ocupar.

`Formvalidation` es compatible con los formularios de los frameworks css más populares:

- Bootstrap
- Foundation
- Pure
- Semantic UI
- Otros

[Smoke](#) es el más completo plugin JQuery diseñado para trabajar con bootstrap 3, además es open source e incluye las siguientes características:

- Validación de formularios.
- Sistema de notificaciones.
- Progressbar.
- Soporte para fullscreen.
- Agregar funcionalidad extra a los paneles de bootstrap.
- Helpers para conversión de tipos.

Para incluir `smoke` en nuestro proyecto sólo tenemos que descargar el plugin de [aquí](#).

Extraer los archivos del zip, y colocar los archivos CSS y JS dentro de la carpeta `public/assets` de nuestro proyecto en Laravel:

Lo primero que debemos hacer es crear un request para el método `store` de `PastelController` ya que necesitamos validar que los datos enviados en el formulario para crear un nuevo pastel sean válidos.

Ejemplo:

```
public/
assets/
css/
smoke.css
smoke.min.css
js/
smoke.js
smoke.min.js
lang/
es.js
es.min.js
```

Una vez hecho esto, debemos hacer referencia a los estilos y scripts desde nuestro documento html en la sección de head y body:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Ejemplo Smoke</title>
{!! Html::style('assets/css/smoke.css') !!}
</head>
<body>
{!! Html::script('assets/js/smoke.js') !!}
{!! Html::script('assets/lang/es.js') !!}
</body>
</html>
```

Los ejemplos sobre validaciones, notificaciones, progressbar, etc los encontrarás en la página oficial de [Smoke](#).

Validación del lado del servidor (Request)

Laravel permite validar los datos enviados por un formulario de forma muy sencilla ocupando un Mecanismo llamados "Requests". Veamos un ejemplo ocupando el controller `PastelesController` visto en el capítulo anterior, dandole uso a los métodos `store` y `update`, el funcionamiento y lógica puedes verlo en el [Anexo C. CRUD con Laravel](#) para comprender su funcionamiento:

Lo primero que debemos hacer es crear un request para el método `store` de `PastelController` ya que necesitamos validar que los datos enviados en el formulario para crear un nuevo pastel sean válidos.

```
php artisan make:request CrearPastelesRequest
```

Con este comando Crearemos el Request CrearPastelesRequest ubicado en:

```
app/Http/Requests/CrearPastelesRequest
```

Su contenido es el siguiente:

```
<?php

namespace Curso\Http\Requests;

use Curse\Http\Requests\Request;

class CrearPastelesRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nombre' => 'required|string|max:60',
            'sabor' => 'required|in:chocolate,vainilla,cheesecake'
        ];
    }
}

public function authorize()
{
    return true;
}
```

Lo siguiente que haremos será cambiar el valor que regresa el método `authorize()` de false a true para permitir que el Request lo pueda ocupar cualquier usuario.

```
<?php

namespace Curse\Http\Requests;

use Curse\Http\Requests\Request;

class CrearPastelesRequest extends Request
{
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nombre' => 'required|string|max:60',
            'sabor' => 'required|in:chocolate,vainilla,cheesecake'
        ];
    }
}

public function authorize()
{
    return true;
}
```

Posteriormente en el método `rules()` agregaremos las reglas de validación del formulario para crear pasteles quedando así:

```
public function rules()
{
    return [
        'nombre' => 'required|string|max:60',
        'sabor' => 'required|in:chocolate,vainilla,cheesecake'
    ];
}
```

Al final el archivo CrearPastelesRequest deberá verse así:

```
<?php

namespace Curse\Http\Requests;

use Curse\Http\Requests\Request;

class CrearPastelesRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nombre' => 'required|string|max:60',
            'sabor' => 'required|in:chocolate,vainilla,cheesecake'
        ];
    }
}
```

De igual forma crearemos el Request para el método update de PastelesController.

```
php artisan make:request EditarPastelesRequest
```

Y el archivo lo dejaremos como se muestra a continuación:

```
<?php

namespace Curso\Http\Requests;

use Curse\Curso\Requests\Request;

class EditarPastelesRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nombre' => 'required|string|size:60',
            'sabor' => 'required|in:chocolate,vainilla,cheesecake'
        ];
    }
}
```

Las reglas de validación ocupadas en el ejemplo anterior las podemos encontrar explicadas con mayor detalle en la página de [Validaciones](#) de Laravel.

Para ocupar el nuevo request en PastelesController debemos incluirlo:

```
use Curse\Curso\Http\Requests\CrearPastelesRequest;
use Curse\Curso\Http\Requests\EditarPastelesRequest;
```

En el modelo **Pastel** agregaremos una propiedad `$fillable` para indicar que atributos de la tabla pasteles podrán ser ocupados con el método `$request->all()`.

El modelo pasteles quedaría así:

```
{ $pasteles = Pastel::get();
return view('pasteles.index')->with('pasteles', $pasteles); }
```

Los métodos `store` y `update` recibirán como parámetro un objeto `Request` para aplicar las reglas de validación, al final el controlador `PastelesController` debe tener el siguiente aspecto:

```
<?php

namespace Curse\Curso\Http\Controllers;

use Illuminate\Http\Request;
use Curse\Curso\Http\Requests;
use Curse\Curso\Http\Controllers\Controller;

use Curse\Pastel;
use Curse\Curso\Http\Requests\CrearPastelesRequest;
use Curse\Curso\Http\Requests\EditarPastelesRequest;

class PastelesController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return Response
     */
    public function index()
    {
```

```

    /**
     * Show the form for creating a new resource.
     *
     * @return Response
     */
    public function create()
    {
        return view('pastel.create');
    }

    /**
     * Store a newly created resource in storage.
     *
     * @return Response
     */
    public function store(CreatePastelRequest $request)
    {
        $pastel = Pastel::create($request->all());
        return redirect()->route('pastel.index');
    }

    /**
     * Display the specified resource.
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        // ...
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param int $id
     * @return Response
     */
    public function edit($id)
    {
        $pastel = Pastel::find($id);
        return view('pastel.edit', [
            'pastel' => $pastel,
        ]);
    }

    /**
     * Update the specified resource in storage.
     *
     * @param int $id
     * @return Response
     */
    public function update(UpdatePastelRequest $request, $id)
    {
        $pastel = Pastel::find($id);
        $pastel->fill($request->all());
        $pastel->save();
        return redirect()->route('pastel.index');
    }
}

```

Sí te interesa ver el proceso con el cual se completaron los métodos te recomendamos ir al [Anexo C. CRUD con Laravel](#) para mayor información y así poder probar adecuadamente los Request.

Middlewares

En este punto debemos tener nuestro CRUD para la clase Pasteles, en caso de no tenerlo recomiendo ver el [Anexo C. CRUD con Laravel](#) para poder continuar. Ahora bien si ya podemos realizar las operaciones básicas no podemos pensar en llevar a un ambiente real o comercial un proyecto en este nivel. Aun si funciona correctamente no hemos contemplado todos los posibles casos o amenazas que se encuentran afuera en la Web, tales como son hackers, estafadores o incluso las fallas usuales de los usuarios finales.

Para solucionar esto Laravel utiliza los Middleware, que nos permiten proteger las rutas de accesos no autorizados, como su nombre lo indica (`middle`) se ubica en el **medio** de la petición (`Request`), entonces si deseamos agregar un nuevo nivel de seguridad a nuestro sistema los Middleware son la respuesta.

Primero vamos a analizar un Middleware para la autenticación o logueo de los usuarios en nuestras rutas. Por defecto en nuestro proyecto de Laravel debemos de contar con un middleware llamado **auth**, este middleware de lo que se encarga es de ver que el usuario se encuentre con una sesión activa, recuerden que en Laravel ya tenemos por defecto el manejo de sesiones junto con las tablas de la base de datos. Para decirle a nuestro proyecto que las rutas de nuestro controlador de pasteles van a estar protegidas por el middleware auth usamos el método `middleware('name')`; dentro del constructor de nuestra clase de la siguiente forma:

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

Recuerden que esto debe ubicarse dentro de nuestro controlador **PastelesController** como una función mas, usualmente esta función es la primera que se ve por lo cual recomiendo que al momento de agregar este código lo hagan el inicio de su clase.

Con este cambio se darán cuenta de que si ingresan a las rutas en las cuales ya antes podíamos ver nuestro crud los redirige al login, si crean una cuenta de usuario y lo intentan nuevamente verán que el acceso ya se les va a conceder, además el nombre y usuario con el que accedan se vera en la barra superior de navegación al lado derecho, comprobando las sesiones que Laravel nos da como un regalo.

Analizaremos un poco los archivos, siempre es importante saber como funciona lo que estamos usando y no quedarnos solo con la idea de que funciona sin tener la mas remota idea de lo que sucede. Si la autenticación ya está hecha esto lo podemos verificar en el

archivo `kernel.php` dentro de la ruta `app/Http/`, en este archivo vamos a ver un código similar a este:

```
1 1 <?php
2 2
3 3 namespace Curso\Http;
4 4 use Illuminate\Foundation\Http\Kernel as HttpKernel;
5 5
6 6 class Kernel extends HttpKernel
7 7 {
8 8
9 9     /**
10 10     * The application's global HTTP middleware stack.
11 11     *
12 12     * @var array
13 13 */
14 14 protected $middleware = [
15 15     \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
16 16     \Curso\Http\Middleware\EncryptCookies::class,
17 17     \Illuminate\Cookie\Middleware\Authenticate::class,
18 18     \Illuminate\Session\Middleware\StartSession::class,
19 19     \Illuminate\View\Middleware\ShareErrorsFromSession::class,
20 20     \Curso\Http\Middleware\VerifyCsrfToken::class,
21 21
22 22 /**
23 23     * The application's route middleware.
24 24     *
25 25     * @var array
26 26 */
27 27 protected $routeMiddleware = [
28 28     'auth' => \Curso\Http\Middleware\Authenticate::class,
29 29     'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
30 30     'guest' => \Curso\Http\Middleware\RedirectIfAuthenticated::class,
31 31 ];
32 32 }
33 33
34 34 }
```

en el podemos ver una variable protegida llamada `$routeMiddleware` donde están definidos los middleware del sistema, podemos observar tanto la ubicación como el nombre con el cual podemos usarlo, estos son **auth**, **auth.basic** y **guest**.

El archivo que hace referencia al middleware **auth** se llama **Authenticate.php** que se encuentra en `app/Http/Middleware/`, dentro de este archivo podemos observar la estructura de un Middleware normal, este tipo de archivos cuentan con una función llamada `handle()` que es la que se ejecuta cuando se llama al middleware, la función `handle` de este archivo es la siguiente:

```
35 35 public function handle($request, Closure $next)
36 36 {
37 37     if ($this->auth->guest()) {
38 38         if ($request->ajax()) {
39 39             return response('Unauthorized.', 401);
40 40         } else {
41 41             return redirect()->guest('auth/login');
42 42         }
43 43     }
44 44 }
45 45
46 46 return $next($request);
```

En ella podemos observar que realiza una serie de preguntas para saber que respuesta dar a donde redirigir, primero pregunta si el usuario esta logueado con la pregunta `if ($this->auth->guest())`, guest significa invitado y por lógica si esta invitado quiere decir que no cuenta con una sesión iniciada. Si ese fuera el caso entonces debemos verificar si el logueo se intenta realizar por medio de AJAX y si la petición es de este tipo entonces rechazarla y mandar una respuesta de acceso no autorizado, pero si no es de tipo AJAX entonces redirigir directo a la vista de logueo. En el caso de que no sea un invitado entonces quiere decir que si tiene iniciada su sesión por lo cual pasara la petición al siguiente middleware hasta que llegue al último y termine las verificaciones del sistema.

Creando nuestros propios Middlewares

Preparativos

Ahora bien, esto no es la solución a todos nuestros problemas (*ain*), aunque el tener la autenticación hecha por defecto es una gran ayuda podemos requerir más protección, por ejemplo si manejamos roles en nuestra aplicación, digamos que el CRUD de pasteles solo lo podemos ver si tenemos una cuenta de administrador en el sistema pero no si tenemos una cuenta normal.

Vamos a modificar algunos archivos para poder aplicar nuestro nuevo middleware, primero la migración de usuarios llamada `create_users_table.php` en la carpeta de las migraciones, vamos a agregar un campo de tipo de esta forma:

```

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

```

Pero en el modelo de Users que viene por defecto con Laravel 5 se definen los campos que se pueden llenar y los ocultos, vamos a decirle al modelo que también puede llenar el campo type de esta forma:

```
protected $fillable = ['name', 'email', 'password', 'type'];
```

El modelo **User** se encuentra en `app/Http/`, ahora revisaremos el controlador de registro, es decir el controlador `AuthController` en la carpeta `app/Http/controllers/Auth/`, buscaremos el método `create` y vamos a dejarlo de esta forma.

```

protected function create(array $data)
{
    return User::create([
        'name'      => $data['name'],
        'email'     => $data['email'],
        'password'  => bcrypt($data['password']),
        'type'      => $data['type'],
    ]);
}

```

Hasta ahora solo estamos diciéndole al método que agregue otro elemento que va a provenir de la vista llamado **'type'**.

Ahora necesitamos poder crear un administrador o un usuario normal, por esto vamos a agregar un campo select en nuestra vista de registro para completar las modificaciones, vamos a modificar la vista `register.blade.php` dentro de la carpeta `resources/views/auth/`. Le vamos a agregar el campo y deberá verse de esta forma:

```

@extends('app')

@section('content')


Sign Up



{!! Form::open(['route' => 'auth/register', 'class' => 'form']) !!}


{!! Form::label('Name') !!}
{!! Form::text('name', '', ['class'=> 'form-control']) !!}



{!! Form::label('Email') !!}
{!! Form::email('email', '', ['class'=> 'form-control']) !!}



{!! Form::label('Type') !!}
{!! Form::select('type', [
    'admin' => 'Administrador',
    'user'  => 'Usuario normal'
]) !!}



{!! Form::password('password_confirmation', ['class'=> 'form-control']) !!}



{!! Form::password('password_confirmation', ['class'=> 'form-control']) !!}



{!! Form::submit('Send', ['class' => 'btn btn-primary']) !!}


```

Crear el middleware IsAdmin

Con lo anterior tendremos todos los cambios necesarios para administrar roles en nuestro sistema. Lo siguiente es crear el middleware `isAdmin`, para esto artisan no provee este comando:

```

php artisan make:middleware name

```

El comando para nuestro ejemplo sería igual a esto:

```

php artisan make:middleware IsAdmin

```

esto nos va a crear un archivo dentro de la carpeta `app/Http/Middleware/` con el nombre que le dimos con la función handle que explicamos anteriormente. Para poder verificar si el usuario logueado es un administrador debemos poder obtener el usuario por lo cual vamos a agregar una clase para poder obtener ese usuario, inyectaremos la dependencia, vamos a crear un atributo que será la autenticación que viene del middleware anterior y el constructor de esta forma:

```
<?php

namespace Curso\Http\Middleware;

use Illuminate\Contracts\Auth\Guard;

use Closure;

class IsAdmin
{
    protected $auth;

    public function __construct(Guard $auth)
    {
        $this->auth = $auth;
    }

    /**
     * Handle an incoming request.
     */
    public function handle($request, Closure $next)
    {
        if ($this->auth->user()->type != 'admin') {
            $this->logout();
            if ($request->javascript()) {
                return response('Unauthorized.', 401);
            } else {
                return redirect()->to('auth/login');
            }
        }

        return $next($request);
    }
}
```

Para la lógica podemos emular lo que pasa en el Middleware de Authenticate, vamos a preguntar el tipo de usuario y vamos a preguntar si la petición es AJAX para dirigir la petición a donde sea el caso, si el usuario es de tipo admin entonces vamos a redirigir la petición al siguiente Middleware, sino entonces vamos a cerrar la sesión y preguntaremos si la petición es AJAX, en caso de ser AJAX vamos a denegarla, si no entonces vamos a mandar al login. Usaremos la función `to()`; porque si nos quedamos con la función `guest()` eso guarda la ruta de destino a la que queremos llegar y nunca vamos a poder iniciar sesión con usuarios normales.

Ahora vamos a registrar nuestro middleware para poder usarlo ya que por si mismo no lo vamos a poder integrar al controlador, para esto tenemos que modificar el **Kernel.php**, lo único aquí es agregar en el **\$routeMiddleware** un nombre para nuestro Middleware y la ubicación del mismo, así es como se vería:

```
protected $routeMiddleware = [
    'auth' => \Curso\Http\Middleware\Authenticate::class,
    'auth,basic' => \Illuminate\Http\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \Curso\Http\Middleware\RedirectIfAuthenticated::class,
    'is_admin' => \Curso\Http\Middleware\IsAdmin::class,
];
```

Entonces con esto ya podemos ligarlo al controlador dentro de su constructor y para cada función dentro de este se va a mandar a llamar el Middleware **is_admin**:

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

Así podemos probar entrando a una ruta con una cuenta de usuario normal y de nuevo con una cuenta de administrador, entonces veremos que solo si usamos una cuenta de administrador podemos ver lo que ya teníamos al inicio pero de otro modo cerrará la sesión y no nos dejará entrar a ver el contenido.

Para mas información lean la [documentación oficial sobre Middlewares](#) de Laravel 5.

¿Qué es HTML5?

HTML5 es la nueva versión del lenguaje de marcado que se usa para estructurar páginas web, actualmente sigue en evolución, HTML5 incorpora características nuevas y modificaciones que mejorara significativamente la forma de construir sitios web.

HTML5 nos permite crear documentos HTML de una forma más simplificada y sencilla que sus versiones anteriores.

¿Qué hay de nuevo en HTML5?

La declaración DOCTYPE es ahora más simple:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Título de la página</title>
</head>
<body>
    Cuerpo del documento
</body>
</html>
```

En la sección de la cabecera `<head>` escribiremos:

- La codificación que ocuparemos para el documento, es recomendado usar UTF-8.
- El título de la página
- Los elementos `link` para utilizar los archivos CSS.

En la sección del cuerpo `<body>` escribiremos:

- Barrá de navegación
- Encabezados
- Secciones
- Parágrafos
- Elementos multimedia : audio, video, img
- Texto en negritas, cursiva y subrayado.
- Tablas
- Listas
- Formularios
- Hipervínculos
- etc.

Nuevas APIs

- HTML Geolocation
- HTML Drag and Drop
- HTML Local Storage
- HTML Application Cache
- HTML Web Workers
- HTML SSE

Encabezados

Los encabezados en html tienen 6 tamaños diferentes y se escriben de la siguiente forma:

Plantilla básica de un documento en HTML5

Cualquier documento en HTML5 debe contener la siguiente estructura básica.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Título de la página</title>
</head>
<body>
    Cuerpo del documento
</body>
</html>
```

HTML	Resultado
<body>	
<h1>Título h1</h1>	Título h1
<h2>Título h2</h2>	Título h2
<h3>Título h3</h3>	Título h3
<h4>Título h4</h4>	Título h4
<h5>Título h5</h5>	Título h5
<h6>Título h6</h6>	Título h6
</body>	

Secciones (divisiones)

Podemos dividir nuestro documento en secciones distintas con la etiqueta <div> para tener un mayor orden sobre nuestro documento y aplicar diferentes estilos según la sección.

HTML	Resultado
<body>	
<div class="encabezado">Sección 1</div>	Sección 1
<div class="cuerpo">Sección 2</div>	Sección 2
<div class="pie_pagina">Sección 3</div>	Sección 3
</body>	

Formularios

HTML	Resultado
<form>	Define un Formulario HTML.
<input>	Define un campo que puede ser de tipo: button, checkbox, color, date, datetime, datetime-local, email, file, hidden, image, month, number, password, radio, range, reset, search, submit, tel, text, time, url, week.
<textarea>	Define un textarea para guardar una gran cantidad de texto.
<button>	Define un botón
<select>	Define una lista desplegable
<option>	Define una opción en una lista desplegable
<label>	Define una etiqueta para un input

Atributos de los formularios

- Sección en desarrollo.

Podemos definir diferentes el formato del texto como: negrita, cursiva, subrayado, tipo de letra, tamaño de fuente, saltos de línea, párrafos, citas, etc.

Tablas

Ejemplo de una tabla básica:

```
<table>
  <thead>
    <tr>
      <th>cabecera</th>
      <th>cabecera</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <td>celda</td>
      <td>celda</td>
      <td>celda</td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>celda</td>
      <td>celda</td>
      <td>celda</td>
    </tr>
  </tbody>
</table>
```

Las tablas se escriben con la etiqueta `<table>`, dentro de la tabla tendremos filas y columnas, la etiqueta `<tr>` define las filas y la etiqueta `<td>` define las columnas.

Hipervínculos e imágenes

Las imágenes pueden ser de formato png, jpg o gif y se escriben con la etiqueta `` entre sus principales atributos tenemos:

- `src`: La URI a la imagen.
- `alt`: Texto que se desplegará en caso de que la imagen no sea desplegada.
- `width`: Ancho de la imagen, puede ser escrita en pixeles o en porcentaje.
- `height`: Alto de la imagen, puede ser escrita en pixeles o en porcentaje.

Ejemplo:

```

```

Los Hipervínculos o links son definidos con la etiqueta `<a>` que cuenta con los siguientes atributos:

- `href`: especifica la URI de destino.
- `target`: especifica en dónde se abrirá el nuevo documento del link.
 - `_blank`: Abre el nuevo documento en una nueva ventana o pestaña.
 - `_self`: Abre el nuevo documento en el mismo frame (acción por defecto).
 - `_parent`: Abre el nuevo documento en el frame padre.
 - `_top`: Abre el nuevo documento en todo el cuerpo de la ventana.

Ejemplo de un link:

```
<a href="https://www.facebook.com/oca159">Facebook</a>
```

Dentro de las etiquetas `<a>` puede ir un texto o una imagen.

```
<a href="default.asp">
  
</a>
```

Los hipervínculos también pueden redireccionar a un segmento específico de la página web.

Ejemplo: Primero creamos una sección con un atributo id.

```
<div id="Encabezado">Sección de encabezado</div>
```

Entonces agregamos un link que nos envíe a esa sección de la página. Para lograr este objetivo, agregamos en el atributo `href` el id de la sección precedido de un signo #.

```
<a href="#Encabezado">Visitar la sección de encabezado</a>
```

Aprender más sobre HTML5

Para profundizar un poco más en HTML5 es recomendable el tutorial de w3schools.

CSS3 (Hojas de estilo en cascada)

Es un lenguaje usado para definir y crear la presentación de un documento estructurado escrito en HTML o XML.

La idea es separar el contenido (Texto) de su presentación (formato).

La información de estilo puede ser definida en un documento separado o en el mismo documento HTML. En este último caso podrían definirse estilos generales en la cabecera del documento o en cada etiqueta particular mediante el atributo `<style>`.

Sintaxis

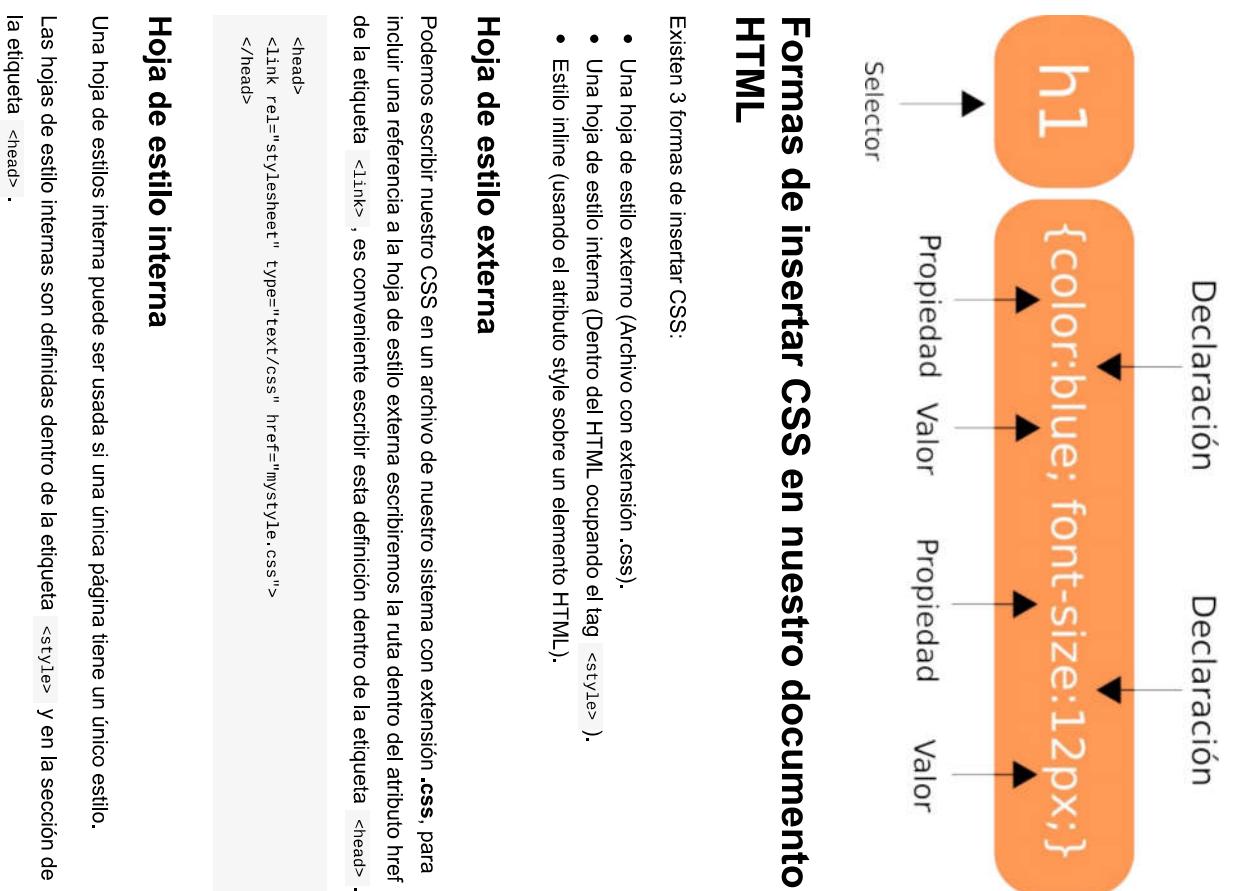
Una hoja de estilo se compone de una lista de reglas. Cada regla o conjunto de reglas consiste en uno o más selectores y un bloque de declaración (o «bloque de estilo») con los estilos a aplicar para los elementos del documento que cumplen con el selector que les precede. Cada bloque de estilos se define entre llaves, y está formado por una o varias declaraciones de estilo con el formato `propiedad: valor;`.

En el CSS, los **selectores** marcarán qué elementos se verán afectados por cada bloque de estilo que les siga, y pueden afectar a uno o varios elementos a la vez, en función de su tipo, nombre (name), ID, clase (class), posición dentro del Document Object Model, etcétera.

Abajo puede verse un ejemplo de una parte de una hoja de estilos CSS:

```
selector [ , selector2, ... ] [ ;pseudo-class][ ;pseudo-element] {  
    propiedad: valor;  
    propiedad2: valor2;  
    ...  
}  
  
/* comentarios */
```

Un conjunto de reglas consiste en un selector y un bloque de declaraciones.



```

<head>
<style>
body {
background-color: linen;
}

h1 {
color: maroon;
margin-left: 40px;
}

</style>
</head>

```

Estilos inline

Un estilo inline puede ser usado para aplicar un único estilo para un único elemento de una página.

Para usar estilos inline , agregamos el atributo `al elemento html.`

```
<h1 style="color:blue; margin-left:30px;">This is a heading.</h1>
```

Comentarios en CSS

Los comentarios son una forma de explicar y documentar el código, pueden ser de utilidad cuando queremos editar el código tiempo después. Los comentarios son ignorados por el navegador.

En CSS un comentario empieza con `/*` y termina con `*/`.

```

p {
  color: red;
  /* This is a single-line comment */
  text-align: center;
}

/* This is
   a multi-line
comment */

#dato1 {
  text-align: center;
  color: red;
}


```

El nombre de un id jamás empieza con un número, al igual que la definición de variables en los lenguajes de programación.

Selector por clase

El selector por clase selecciona todos los elementos HTML con un atributo class específico.

Para seleccionar elementos con una clase específica, escribimos un punto seguido por el nombre el nombre de la clase.

Selectores

CSS3 puede aplicar diferentes estilos a un grupo de elementos HTML dependiendo el tipo de selector que ocupemos, veremos cada uno de ellos.

Selector por elemento

Selecciona elementos basándose en el nombre del elemento.

Por ejemplo para seleccionar todos los elementos `<>` de una página y aplicarles el estilo: texto alineado al centro y de color rojo.

```
p {
  text-align: center;
  color: red;
}
```

Selector por id

El selector por id utiliza el atributo id de un elemento HTML para especificar el elemento al que se le va a aplicar un estilo.

Cada id es un valor único dentro de una página HTML, por lo cuál la selección sólo se aplicará sobre un único elemento en particular.

Para seleccionar un elemento con un id específico, escribe un signo de numeral o gato seguido por el valor del id del elemento.

La siguiente regla se aplicará a un elemento HTML con el `id="dato1"`.

```
#dato1 {
  text-align: center;
  color: red;
}
```

En el ejemplo de abajo seleccionaremos todos los elementos HTML con `class="center"` y los alinearemos al centro.

```
.center {
    text-align: center;
    color: red;
}
```

De igual forma es posible seleccionar elementos de un tipo HTML en específico y luego aquellos con una clase en particular.

En el ejemplo de abajo seleccionaremos los párrafos con `class="center"` y los alinearemos al centro.

```
p.center {
    text-align: center;
    color: red;
}
```

Al igual que los ids, las clases no deben nombrarse empezando con un número.

Agrupar selectores

En el caso de que tuvieramos varios selecciones que apliquen el mismo estilo:

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

h2 {
    text-align: center;
    color: red;
}

p {
    text-align: center;
    color: red;
}
```

Podemos reducir el código agrupando los selectores separando cada selector por una coma.

```
h1, h2, p {
    text-align: center;
    color: red;
}
```

Para aprender más sobre todos as propiedades y valores existentes en CSS3 así como guías sobre diseño responsive y otros temas recomendamos visitar la página [w3schools](http://w3schools.com).

Aprender más sobre CSS3

Creacion del CRUD con Laravel desde 0

Este anexo del libro de Laravel 5 esta pensado para resumir el contenido de una forma aplicada, de modo que se puedan ver en conjunto todos los conocimientos adquiridos durante el curso.

Se explicara el proceso para dar altas, bajas, cambios y consultas de la tabla **pastelos** o **CRUD** que significa **C**reate, **R**ead, **U**pdate and **D**elete por sus siglas en inglés.

Primero retomaremos las [migraciones y los seeders](#), creando la migracion y un pequeño seeder para poblar nuestra BD.

Para crear la migracion con la plantilla basica usaremos el comando

```
php artisan make:migration crear_tabla_pastelos --create=pastelos
```

Ahora dentro de la migracion vamos a definir la estructura de la tabla que tendra solo cuatro campos que seran: id, nombre, sabor y timestamps (esto en BD es igual a created_at y updated_at).

Dando un resultado como lo siguiente:

```
class CrearTablaPastelos extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('pastelos', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre', 60);
            $table->enum('sabor', ['chocolate', 'vainilla', 'cheesecake']);
            $table->timestamps();
        });
    }

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

Ahora crearemos el seeder con el comando:

```
php artisan make:seeder PastelosSeeder
```

y vamos a usar el componente [Faker](#) para crear 50 pasteles de forma automatica, el archivo final quedara de la siguiente forma, recuerden que para usar [Faker](#) es necesario importar la clase con la instrucion **use**:

```
<?php
use Illuminate\Database\Seeder;
use Faker\Factory as Faker;

class PastelesSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $faker = Faker::create();

        for ($i=0; $i < 50; $i++) {
            \DB::table('pasteles')->insert(array(
                'nombre' => 'Pastel ' . $faker->firstNameFemale() . ' ' . $faker->lastName(),
                'sabor' => $faker->randomElement(['chocolate', 'vainilla', 'cheesecake']),
                'created_at' => date('Y-m-d H:m:s'),
                'updated_at' => date('Y-m-d H:m:s')
            ));
        }
    }
}
```

Con esto ya tendremos la estructura de la tabla y un seeder para poblar, con esto ahora usaremos el siguiente comando para crear la tabla en la BD y llenarla:

```
php artisan migrate --seed
```

entonces debemos definir al modelo con que tabla va a estar trabajando. Agregaremos dentro del modelo el atributo `protected $table = 'pasteles';`, y eso seria todo para el modelo, quedando de la siguiente manera:

```
<?php
namespace Curso;

use Illuminate\Database\Eloquent\Model;
class Pastel extends Model
{
    protected $table = 'pasteles';
}

Route::resource('pasteles', 'PastelesController');
```

Ya tenemos listo nuestro manejo de datos, ahora vamos a proceder a crear nuestras vistas para que desde el navegador podamos mandar la informacion y un controlador para poder definir nuestras operaciones, ademas de especificar las rutas de nuestro sistema.

Para comenzar vamos a crear nuestro controlador con el comando:

```
php artisan make:controller PastelesController
```

Laravel automaticamente crea un archivo dentro de la ruta `app/Http/controllers` con el nombre que especificamos y dentro de el las funciones para los metodos: **index**, **create**, **store**, **show**, **edit**, **update**, **delete**. Estos metodos los vamos a definir mas adelante, por el momento vamos a crear en nuestro archivo de rutas un grupo de rutas asociadas a cada uno de los metodos del controlador que acabamos de crear, esto se hace con una ruta de tipo **Resource** o recurso, modificando el archivo routes agregando el siguiente codigo:

```
Route::resource('pasteles', 'PastelesController');
```

Con esto ya tendremos nuestro controlador(aun vacio) y nuestras rutas del sistema, lo cual podemos verificar con el comando:

```
php artisan route:list
```

Ahora vamos a terminar de llenar nuestro controlador, debemos importar la clase `Pastel` a nuestro controlador para poder hacer uso del Modelo y asi trabajar con la base de datos, convenciones de Laravel por defecto esto seria lo unico necesario, pero debido a que nuestro lenguaje no convierte las palabras a plural de la misma forma que el ingles

Laravel nos recomienda seguir las convenciones para facilitarnos el trabajo, por lo que las tablas de la Base de datos deben encontrarse en notacion **underscore** y en **plural**, y los modelos deben encontrarse en notacion **UpperCamelCase** y en **singular**.

Con esto Laravel nos creara el archivo `Pastel.php` en la carpeta `app/`, si seguimos las convenciones de Laravel por defecto esto seria lo unico necesario, pero debido a que nuestro lenguaje no convierte las palabras a plural de la misma forma que el ingles

se describe en el [capítulo 11](#) los métodos responden a una ruta en específico de la ruta resource que agregamos en `routes.php`, para este caso vamos a definir cada uno de ellos en orden:

Index - Página de Inicio

Cuando entramos a una página principal de administración se pueden ver en ocasiones la información que se encuentra en la BD y acceso a las operaciones básicas del CRUD, por lo cual debemos ser capaces de recibir en el index los registros de la BD. Dentro el método `index()`, vamos a usar Eloquent para obtener todos los pasteles y enviarlos a una vista que vamos a definir más adelante, quedando de la siguiente forma:

```
public function index()
{
    $pasteles = Pasteles::get();
    return view('pasteles.index')->with('pasteles', $pasteles);
}
```

Eloquent nos facilita mucho las consultas a la BD y hace que sea portable nuestro código, en el método decimos que seleccionamos todos los pasteles y los envíe a una vista llamada `index` ubicada en la carpeta `resources/views/pasteles/`, como vimos en el [capítulo 10](#) las rutas en blade cambian la `/`(diagonal) por un `.`(punto), la función `view('pasteles.index')`; toma como carpeta raíz a `resources/views/` por lo que no tenemos la necesidad de agregarlo en la ruta. Además se está concatenando el método `with('nombre', $var')`; que como **primer** parámetro pide el nombre con el cual se va a poder usar una variable del lado de la vista, y como **segundo** parámetro recibe la variable que se va a mandar a la vista.

Create - Página de registro

Este método es muy sencillo puesto que solo va a devolver una vista sin ninguna variable ni uso de Eloquent, por lo cual queda de la siguiente manera:

```
public function create()
{
    return view('pasteles.create');
}
```

Store - Función de almacenamiento

Este método es donde después de haber entrado a `create` se reciben los datos y se guardan en la base de datos, para poder recibir la información en este ejemplo vamos a usar la clase `Request` que significa petición y es una clase que Laravel agrega por nosotros

cuando creamos el controlador, vamos a pasar por parámetro la petición en el método definiendo que es una variable de la clase `Request` y después de eso podemos recuperar por el nombre del campo del formulario(atributo `name`) la información enviada, entonces el método quedaría de la siguiente forma:

```
public function store(Request $request)
{
    $pastel = new Pasteles;
    $pastel->nombre = $request->input('nombre');
    $pastel->sabor = $request->input('sabor');
    $pastel->save();

    return redirect()->route('pasteles.index');

}
```

En la función estamos creando una instancia de un nuevo `Pasteles` y asignando los atributos de la clase que se llaman igual que los campos de la BD los valores del formulario con la variable `request` y el método `input('name')`; que recibe como parámetro el nombre del campo del formulario, para mas detalle revise la sección del [Anexo A de HTML](#) que habla sobre los atributos de los formularios.

Después de asignar los valores de la petición a la variable `$pastel`, se usa el método `save()`; para que el modelo se encargue de guardar los datos en la BD y finalmente redireccionar al index con los métodos encadenados: `redirect()->route('pasteles.index');`

Show - Página de descripción

- Lo sentimos, sección en desarrollo.

Edit - Página de edición

La función de `edit` es similar a la de `create` pues solo muestra una vista, con una pequeña diferencia, la cual es que se va a buscar el pastel que se quiere editar y se va a mandar a la vista, esto es obvio pues debemos poder ver la información que vamos a editar. La función quedaría de la siguiente forma:

```
public function edit($id)
{
    $pastel = Pasteles::find($id);
    return view('pasteles.edit')->with('pastel', $pastel);
}
```

Es muy claro, en una variable se guarda el pastel, gracias al modelo esto se soluciona facilmente con el metodo `find()`, el id del pastel se manda en la url, ahora bien si esto es preocupante puesto que el id se ve directamente en la URL recordemos que esto no modifica aun, solo nos manda a la pagina que va a poder hacer una nueva petición para actualizar.

Update - Funcion de actualización

Bien, despues de entrar en la pagina de edit vamos a poder editar la informacion y regresara al controlador para que efectue los cambios, dentro del metodo `update` vamos a recuperar nuevamente el `Pastel` por medio de su id que tambien va en la url y se recibe como parametro de la funcion, ademas vamos a agregar otro parametro que sera el Request al igual que en la funcion `create` para recuperar la informacion del lado del cliente en el controlador, dejando la funcion de esta forma:

```
public function update(Request $request, $id)
{
    $pastel = Pastel::find($id);
    $pastel->nombre = $request->input('nombre');
    $pastel->sabor = $request->input('sabor');
    $pastel->save();
    return redirect()->route('pasteles.index');
}
```

Esta funcion es similar a la de `create` lo unico que cambia es que en vez de crear un nuevo pastel vamos a recuperar uno existente y cambiar sus atributos.

Destroy - Funcion de borrado

Este metodo tiene la funcion de eliminar el registro de la BD, pero para efectuarlo tenemos dos opciones, la **primera** forma: crear una variable `$pastel` y despues usar el metodo `delete()` de Eloquent, o bien la **segunda**: directamente del modelo usar el metodo de `Eloquent destroy($id)`, que se encarga de directamente buscar y eliminar el registro, finalmente vamos a redirigir al index, el metodo al final quedara de la siguiente forma:

```
// Esta es la primera opcion
public function destroy($id)
{
    $pastel = Pastel::find($id);
    $pastel->delete();
    return redirect()->route('pasteles.index');
}

// Esta es la segunda opcion
public function destroy($id)
{
    Pastel::destroy($id);
    return redirect()->route('pasteles.index');
}
```

Vistas del CRUD

Estas son la ultima parte que vamos a crear, primero debemos preparar los directorios y los archivos que usaremos.

La estructura de la carpeta quedaria de la siguiente forma:

```
resources/
views/
pasteles/
partials/
    fields.blade.php
    table.blade.php
    create.blade.php
    edit.blade.php
    index.blade.php
```

Usaremos el `template` por defecto de Laravel llamado `app.blade.php` que fuimos modificando durante el curso por lo cual solo deberemos crear los archivos restantes.

Ahora en el archivo `app.blade.php` vamos a modificarlo para que el contenido este mejor acomodado usando `Bootstrap` y vamos a agregar los estilos y scripts para que la tabla donde vamos a mostrar el contenido funcione como un `DataTable`, dejando el archivo app de la siguiente forma:

Template App

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Laravel </title>
    @section('styles/laravel')
        {!! Html::style('assets/css/datatables-bootstrap.css') !!}
        {!! Html::style('assets/css/bootstrap.css') !!}
        <!-- Fonts -->
        <link href='//fonts.googleapis.com/css?family=Roboto:400,300' rel='stylesheet' type='text/css'>
    @show
    @yield('my_styles')
</head>
<body>
    @include('partials.layout.navbar')
    @include('partials.layout.errors')
    <div class="container">
        <div class="row">
            <div class="col-md-10 col-md-offset-1">
                @yield('content')
            </div>
        </div>
        <div>
            <!-- Scripts -->
            {!! Html::script('assets/js/jquery.js') !!}
            {!! Html::script('assets/js/dataTables.js') !!}
            {!! Html::script('assets/js/bootstrap.min.js') !!}
            {!! Html::script('assets/js/datatables-bootstrap.js') !!}
        </script>
        $(<#MyTable>).dataTable();
    </div>
    </div>
    @extends('app')
</body>
</html>

```

Los cambios mas notorios que podemos observar es que el `@yield('content')` se metio dentro de una columna con un offset, eso dentro de una fila y todo dentro de un contenedor. Asi nuestro contenido no lo vamos a tener todo pegado a la izquierda de nuestro navegador.

Nota: para poder hacer esto son necesarios los archivos que se incluyen con blade, si no los agregan la tabla se vera mas sencilla pero esto no quiere decir que no va a funcionar, ya solo es cuestion de estillo, si quieren obtener los archivos dejamos los links a continuacion

- Estilos para el DataTable: `datatable-bootstrap.css`.
 - Archivo JQuery: `jquery.js`.
 - Archivo JQuery para DataTable: `jquery.dataTables.js`.
 - Archivo JQuery para DataTable de bootstrap: `datatable-bootstrap.js`.
- Nota:** tambien les invito a ver el [anexo de DataTable](#) para mayor informacion.

Vista Index

Esta vista se refiere al archivo `index.blade.php` dentro de la carpeta `resources/views/pasteles/`, aqui vamos a mostrar la tabla y un boton para crear nuevos pasteles. Ahora bien para esto debemos tener nuestro **partial** de la tabla, mas adelante lo vamos a mostrar pero por el momento el archivo index quedaria de la siguiente forma:

```

@extends('app')

@section('content')
    <a class="btn btn-success pull-right" href="{{ url('/pasteles/create') }}" role="button">Nuevo pasteel</a>
    @include('pasteles.partials.table')
@endsection

```

Recuerden que gracias a blade nuestras vistas quedan de tamaños pequeños mas faciles de entender, aqui solo estamos heredando la plantilla `app` y definiendo la sección `content` con un `link` que le daremos estillo de boton con la ruta para mostrar la vista de crear pasteles, ademas de importar nuestra tabla, el archivo partial lo definiremos ahora.

Partials: table y fields

Table

Estos archivos los trabajaremos en partials por comodidad y porque son componentes de un sistema Web que suelen repetirse constantemente, empezaremos por el table.

Primero recordemos un poco del pasado, en nuestro controlador en el metodo `index` definimos que retornaria la vista `index` junto con una variable llamada `$pasteles` que contendria todos los pasteles del sistema, ahora bien esos pasteles los vamos a vaciar en la

para que los descarguen y los guarden dentro de la carpeta respectiva, es decir los **CSS** en `public/assets/css/` y los **JS** dentro de `public/assets/js/`:

`<meta name="viewport" content="width=device-width, initial-scale=1">`

`<title>Laravel </title>`

`@section('styles/laravel')`

`{!! Html::style('assets/css/datatables-bootstrap.css') !!}`

`{!! Html::style('assets/css/bootstrap.css') !!}`

`<!-- Fonts -->`

`<link href='//fonts.googleapis.com/css?family=Roboto:400,300' rel='stylesheet' type='text/css'>`

`= 'text/css'>`

`@show`

`@yield('my_styles')`

`</head>`

`</body>`

`@include('partials.layout.navbar')`

`@include('partials.layout.errors')`

`<div class="container">`

`<div class="row">`

`<div class="col-md-10 col-md-offset-1">`

`@yield('content')`

`</div>`

`</div>`

`<div>`

`<!-- Scripts -->`

`{!! Html::script('assets/js/jquery.js') !!}`

`{!! Html::script('assets/js/dataTables.js') !!}`

`{!! Html::script('assets/js/bootstrap.min.js') !!}`

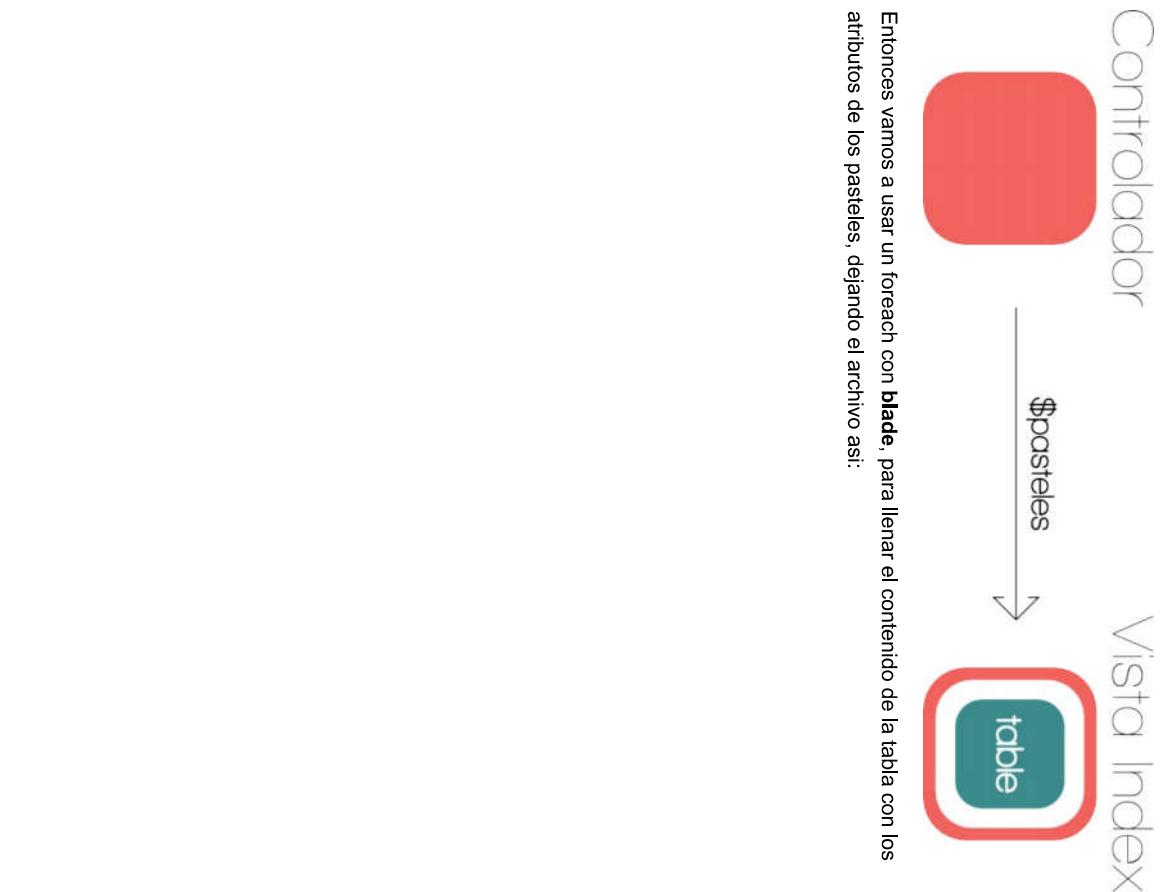
`{!! Html::script('assets/js/datatables-bootstrap.js') !!}`

`</script>`

`$(<#MyTable>).dataTable();`

`</div>`

tabla pues si bien no especificamos que esa variable va a llegar al partial **table** como lo estamos incluyendo en el index tambien comparte las variables que tenga index, entonces el envío puede verse de la siguiente manera:



Entonces vamos a usar un **foreach** con **blade**, para llenar el contenido de la tabla con los atributos de los pasteles, dejando el archivo así:

```

<h1 class="text-primary">Control de Pasteles</h1>
<table class="table table-bordered" id="MyTable">
    <thead>
        <tr>
            <th class="text-center">ID</th>
            <th class="text-center">Nombre</th>
            <th class="text-center">Sabor</th>
            <th class="text-center">Fecha</th>
            <th class="text-center">Acciones</th>
        </tr>
    </thead>
    <tbody>
        @foreach($pasteles as $pastel)
            <tr>
                <td class="text-center">{{ $pastel->id }}</td>
                <td class="text-center">{{ $pastel->nombre }}</td>
                <td class="text-center">{{ $pastel->sabor }}</td>
                <td class="text-center">{{ $pastel->created_at }}</td>
                {{!! Form::open(['route' => ['pasteles.destroy', $pastel->id], 'method' => 'DELETE']) !!}}
                    <td class="text-center">
                        <button type="submit" class="btn btn-danger btn-xs">
                            <span class="glyphicon glyphicon-remove" aria-hidden="true"></span>
                        </button>
                        <a href="{{ url('/pasteles/' . $pastel->id, 'edit') }}" class="btn btn-info btn-xs">
                            <span class="glyphicon glyphicon-edit" aria-hidden="true"></span>
                        </a>
                    </td>
                    {{!! Form::close() !!}}
                </td>
            @endforeach
        </tbody>
        <tfoot>
            <tr>
                <th class="text-center">ID</th>
                <th class="text-center">Nombre</th>
                <th class="text-center">Sabor</th>
                <th class="text-center">Fecha</th>
                <th class="text-center">Acciones</th>
            </tr>
        <tfoot>
    </table>

```

La estructura de la tabla la pueden ver en [este link](#), pero lo importante esta dentro del <body>, en donde con **BLADE** vamos a usar un foreach diciendo que para cada pastel dentro de la variable **\$pastel** que llego del controlador se van a vaciar sus datos dentro de la fila de la tabla, primero vamos a agregar su id, nombre y fecha de creacion, pero se va a agregar una columna de acciones que contenga dos botones, uno para eliminar ese registro y otro para editarlo, el boton de eliminar debe ser de tipo submit para enviar la petición **DELETE** y para esto es que se abre un formulario con la clase **Form::** de Laravel,

para que asi que de sintaxis mas legible, agregamos los campos necesarios que son la ruta y el metodo del formulario; para el boton de edit basta con un [link](#) <a> que con la funcion `url()` de Laravel la vamos a dirigir con el ID de cada pastel.

Nota: Bootstrap nos permite tener a disposicion iconos para que los botones de nuestras acciones se vean mas profesionales, por lo cual es lo que se agrega el tag , para mas informacion ir a la pagina oficial de [Bootstrap](#).

Con esto debemos ser capaces de poder ver ahora nuestra tabla como un `DataTable`(en caso de haber agregado lo necesario) llena con la informacion de pasteles:



```
<div class="form-group">
    {!! Form::label('nombre', 'Nombre', ['for' => 'nombre']) !!}
    {!! Form::text('nombre', null, ['class' => 'form-control', 'id' => 'nombre', 'placeholder' => 'Escribe el nombre del pastel...']) !!}
</div>

<div class="form-group">
    {!! Form::label('sabor', 'Sabor', ['for' => 'sabor']) !!}
    <select name="sabor" class="form-control">
        <option value="" disabled selected>Elige un sabor...</option>
        <option value="chocolate">Chocolate</option>
        <option value="vainilla">Vainilla</option>
        <option value="cheesecake">Cheesecake</option>
    </select>
</div>
```

Si bien la clase `Form::` no se ha explicado detalladamente dejó el [link](#) de la documentación oficial de Laravel, aunque se encuentra en su version 4.2 es debido a que para las versiones mas actuales no se encuentra explicada, pero sigue siendo completamente compatible con Laravel 5 y 5.1.

Con este partial vamos a poder llamar los campos de entrada para crear o editar un pastel.

Vista Create

En esta vista al igual que el index quedara muy corta:

```
@extends('app')

@section('content')
    {!! Form::open(['route' => 'pasteles.store', 'method' => 'POST']) !!}
    @include('pastelles.partials.fields')
    <button type="submit" class="btn btn-success btn-block">Guardar</button>
    {!! Form::close() !!}
@endsection
```

Fields

Ahora vamos a crear un partials con los campos que va a requerir nuestro proyecto, si bien sabemos es necesario pedir al usuario el nombre y sabor del pastel, pero la fecha de creacion y ultima actualizacion son campos que Laravel pone automaticamente cuando se ejecuta el metodo `save()` en el controlador, por lo cual nuestro partial solo debera tener dos campos de entrada y un boton para enviar la solicitud.

Entonces para este archivo solo vamos a agregar como su nombre lo indica los campos de entrada para un poste, dejandolo de la siguiente forma:

Extendemos del template **app**, definimos el contenido abriendo un formulario pero como se trata de un almacenamiento el metodo se va a trabajar con **store** y **POST**, dentro vamos a incluir el partial de **fields** para tener los campos de texto, el menu de opciones y un boton de tipo submit para mandar la petición.

Deberia quedar un resultado similar a este:

Laravel Home

Nombrar
Nombre: Editar el nombre del pastel...
Sabor: Elija un sabor...
Estado: Guardar cambios

Login Registro

Vista Edit

Al tener ya listos nuestros archivos HTML la vista de edit se crea de la misma forma que la de create con la diferencia de que en vez de abrir un formulario vamos a abrir un modelo, es decir vamos a abrir el objeto que se envio del controlador a la vista para poder editar los campos, como observacion podran notar cuando lo ejecuten en el navegador que un select no se asigna automaticamente en valor anterior, por el momento vamos a ver como quedaria la vista:

```
@extends('app')

@section('content')
    <h4 class="text-center">Editar Pastel: {{ $pastel->nombre }}</h4>
    {!! Form::model($pastel, [ 'route' => ['pastel.update', $pastel], 'method' => 'PUT']) !!}
        @include('pastel/partials/fields')
    <button type="submit" class="btn btn-success btn-block">Guardar cambios</button>
    >
    {!! Form::close() !!}
@endsection
```

Al igual que las demas vistas se esta heredando de app y se agrega un titulo para saber que pastel se esta editando, pero el Form::model() abre nuestra variable \$pastel que enviamos desde el controlador y crea un formulario lleno a partir de los valores del modelo, claro que esto solo para los campos que coincidan con los nombres de los atributos del modelo.

El resultado seria algo similar a esto:

Laravel Home

Nombrar
Nombre: Editar Pastel: Pastel Casandra Hermann
Sabor: Elija un sabor...
Estado: Guardar cambios

Login Registro

Y con esto quedarian nuestras vistas del sistema terminadas y el CRUD basico de los **Pasteles** finalizado tambien, para mas informacion pueden retomar los capitulos de este libro para analizar las diferentes opciones que tenemos para resolver este ejemplo pues esta es solo una propuesta, no una solucion definitiva.

nota: La seccion de show no se ha contemplado para este anexo con una vista, disculpen las molestias.

DataTable

Los DataTables son un plug-in para la librería JQuery de Javascript. Nos permiten un mayor control sobre un elemento `<table>` de HTML. Algunas de sus características principales son:

- Paginación automática
- Búsqueda instantánea
- Ordenamiento multicolumna
- Soporta una gran cantidad de datos fuente.
- DOM (Convertir un elemento HTML `<table>` en un DataTable).
- Javascript (Un arreglo de arreglos en Javascript puede ser el dataset de un DataTable).
- AJAX (Un DataTable puede leer los datos de una tabla de un json obtenido vía AJAX, el json puede ser servido mediante un Web Service o mediante un archivo.txt que lo contenga).
- Procesamiento del lado del servidor (Un DataTable puede ser creado mediante la obtención de datos del procesamiento de un script en el lado del servidor, este script comúnmente tendrá interacción con la base de datos).
- Habilitar temas CSS para el DataTable fácilmente.
 - Crear un tema
 - Tema JQuery UI
 - Tema Bootstrap
 - Tema Foundation
 - Amplia variedad de extensiones.
 - Altamente internacionalizable.
 - Es open source (Cuenta con una licencia MIT).

¿Cómo usar DataTables?

Para ocupar datables tenemos dos formas:

- Descargar el código fuente de los archivos js y css de datatables en el siguiente link.
- Ocupar un CDN
 - CSS `//cdn.datatables.net/1.10.7/css/jquery.dataTables.min.css`
 - JS `//cdn.datatables.net/1.10.7/js/jquery.dataTables.min.js`

Usar DataTable descargando el código fuente

Antes que nada debemos tener descargado jquery, el cuál podemos descargar de [aquí](#). Una vez que hemos descargado los archivos css y js de DataTables de este [link](#) debemos extraer el contenido y ubicar los archivos en la carpeta css y js correspondiente dentro del directorio **public** de Laravel.

Ejemplo:

```
public/
assets/
css/
  datatable-bootstrap.css
js/
  dataTable-bootstrap.js
  jquery.dataTables.js
  jquery.js
```

Lo siguiente será hacer una referencia de los script y archivos css dentro de nuestro documento HTML.

```
<body>
<!-- Contenido -->
<!-- Scripts -->
  {!! Html::script('assets/js/jquery.js') !!}
  {!! Html::script('assets/js/jquery.dataTables.js') !!}
  {!! Html::script('assets/js/bootstrap.min.js') !!}
  {!! Html::script('assets/js/dataTables.bootstrap.min.js') !!}
<script>
$(document).ready(function(){
  $('#MyTable').dataTable();
});
</script>
</body>
```

En el ejemplo de arriba estamos indicando que la tabla con el id `#MyTable` será un elemento de tipo dataTable. La obtención de datos la hacemos mediante el DOM.

Por otro lado, los archivos para temas del datatable los ubicaremos dentro de la etiqueta `<head>` de nuestro documento HTML .

```
<head>
<!-- Más archivos CSS -->
{!! Html::style('assets/css/dataTables.bootstrap.css') !!}
</head>
```

De esta forma hemos conseguido crear nuestro primer DataTable.

```
$(document).ready(function() {
    $('#demo').html('<table cellpadding="0" cellspacing="0" border="0" class="display"
" id="example"></table>');
}

$('#example').dataTable( {
    "data": dataset,
    "columns": [
        { "title": "Engine" },
        { "title": "Browser" },
        { "title": "Platform" },
        { "title": "Version", "class": "center" },
        { "title": "Grade", "class": "center" }
    ]
});
```

Existen diversas formas de llenar un DataTable con datos, veremos algunas de las más usadas.

Ocupando el DOM (etiqueta <table>)

Ocuparemos una tabla HTML con un id="example", posteriormente convertiremos la tabla en un DataTable mediante un script.

- HTML de la [tabla](#)

El código Javascript para convertir la tabla con id="example" en un DataTable es:

```
$('document').ready(function() {
    $('#example').dataTable();
})
```

Ocupando un dataset en Javascript

Un arreglo de arreglos en Javascript puede ser el dataset de un DataTable.

- Javascript del [dataset](#)

Una vez definido el dataset bastará con ejecutar la siguiente función para crear una table en un div con id="demo" en nuestro HTML.

Ocupando AJAX

Un DataTable puede leer los datos de una tabla de un json obtenido vía AJAX, el json puede ser servido mediante un Web Service o mediante un json.txt que lo contenga.

- El archivo [JSON](#)

El código Javascript para la creación de DataTable ocupando el json.

```
$(document).ready(function() {
    $('#example').dataTable(
        "ajax": 'json.txt'
    );
})
```

El archivo HTML contiene una tabla con un id="example".

```
<table id="example" class="display" cellspacing="0" width="100%>
  <thead>
    <tr>
      <th>Name</th>
      <th>Position</th>
      <th>Office</th>
      <th>Extn.</th>
      <th>Start date</th>
      <th>Salary</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th>Name</th>
      <th>Position</th>
      <th>Office</th>
      <th>Extn.</th>
      <th>Start date</th>
      <th>Salary</th>
    </tr>
  </tfoot>
</table>
```

Filtrar datos de un DataTable

Internacionalizar un dataTable

Es posible cambiar el idioma de las etiquetas de un datatable si bajamos el archivo de la traducción en el siguiente [link](#).

Para ocuparlo tenemos que hacer referencia en la función de la creación del DataTable:

```
<script type="text/javascript" src="jquery.dataTables.js"></script>
<script type="text/javascript">
$(document).ready(function() {
  $('#example').dataTable( {
    "language": {
      "url": "dataTables/spanish.lang"
    }
  });
</script>
```

Podemos ocupar el campo de texto de búsqueda del DataTable y buscar bajo diferentes criterios separando por un espacio en blanco.

Ejemplo:

- Para buscar un administrador cuyo nombre empiece con la letra O y su teléfono sea extensión 228, podemos poner administrador o 228