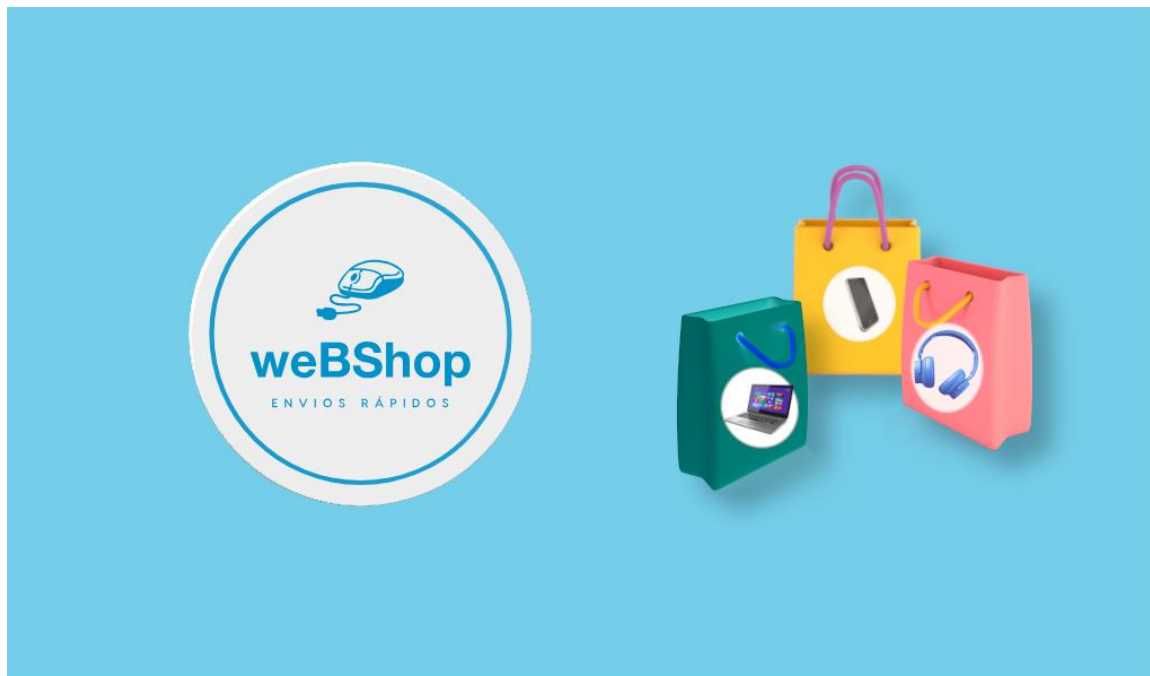


DSI Documento de diseño técnico

webshop

Versión 1.0.0



Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

HOJA DE CONTROL DOCUMENTAL

Realizado por	Aitor Logedo Ordoñez	Fecha	10/11/2024
Revisado por		Fecha	
Aprobado por		Fecha	

CONTROL DE VERSIONES

Versión	Fecha	Descripción
1.0	10/11/2024	Versión inicial del proyecto (Alpha)

LISTA DE DISTRIBUCIÓN

Nombre	Rol	Organización

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Índice

1. OBJETIVOS DEL DOCUMENTO	4
2. DOCUMENTOS RELACIONADOS	4
3. ARQUITECTURA DEL SISTEMA	4
4. DISEÑO DE ALTO NIVEL	6
4.1. ESQUEMA GENERAL DEL DISEÑO DE ALTO NIVEL	6
4.2. DISEÑO DEL MODELO LÓGICO DE DATOS	8
5. DISEÑO DE BAJO NIVEL	9
5.1. DISEÑO DE LA CAPA DE PRESENTACIÓN	9
5.2. DISEÑO DE LA CAPA DE NEGOCIO	11
5.3. DISEÑO DE LA CAPA DE DATOS	12
6. OTRAS CARACTERÍSTICAS	14
6.1. SEGURIDAD.....	14
6.2. GESTION DE LOGS Y ERRORES	16
6.3. Backup.....	17
7. ANEXO I Comandos Composer	17
8. Anexo II Comandos npm	18
9. Anexo III comandos artisan de laravel.....	20

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

1. OBJETIVOS DEL DOCUMENTO

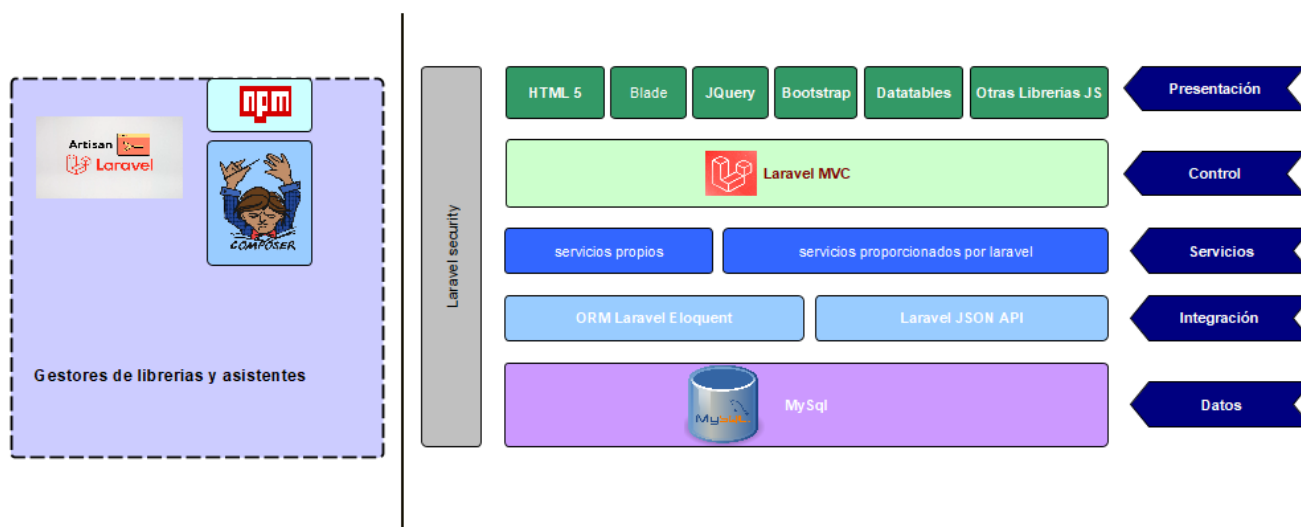
Este documento describe el diseño del sistema, generado a partir de la documentación de Análisis Funcional, y considerando para su elaboración el conjunto de posibilidades y restricciones específicas de la plataforma tecnológica a utilizar.

2. DOCUMENTOS RELACIONADOS

Nombre	Descripción
DRF	Documento de requisitos funcionales
ASI	Documento de análisis funcional

3. ARQUITECTURA DEL SISTEMA

El desarrollo del sistema se basa en la arquitectura basada en el patrón MVC basándonos en el framework Laravel en su versión 11



Conforme a las capas que se muestran en la figura:

Capa de presentación

Para el desarrollo de esta capa se va a utilizar html5 junto con el motor de plantillas **Blade** de Laravel. Asimismo, nos apoyaremos en **JQuery** para darle un contenido dinámico a las vistas y en **bootstrap** con el objetivo de obtener un diseño responsivo que se adapte a los diferentes tipos de dispositivos. También usaremos el componente **datatables** que nos facilitara mucho el desarrollo de pantallas con la visualización de tablas de una manera eficiente y visualmente atractiva. Se incluirán otras librerías javascript para dar soporte a otras necesidades como pueden ser la creación de gráficos, presentación de mensajes de confirmación tipo Toast, etc.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Capa de control

Para el desarrollo de la capa de controlador se va a utilizar Laravel con desarrollo de controladores en PHP como tecnología de referencia.

Capa de Servicios

por un lado, utilizaremos los servicios de seguridad, copias de seguridad, inyección, etc. proporcionados por Laravel así como servicios propios para dar soporte al acceso a datos y atender peticiones json que desde las vistas se puedan requerir.

Capa de Integración

En esta capa nos apoyaremos en el ORM (object Request Mapping) Eloquent de Laravel para la persistencia de datos y en el api Laravel json api para el mapeo de tablas a formato JSON

Capa de datos

Como almacén de datos el sistema debe utilizar MySql de la compañía Oracle como motor de base de datos relacional en su versión community

Capa de Seguridad

Para la seguridad utilizaremos el framework Brezel de Laravel basado en autenticación sobre base de datos y autorización basada en Json Token

Gestores de dependencias

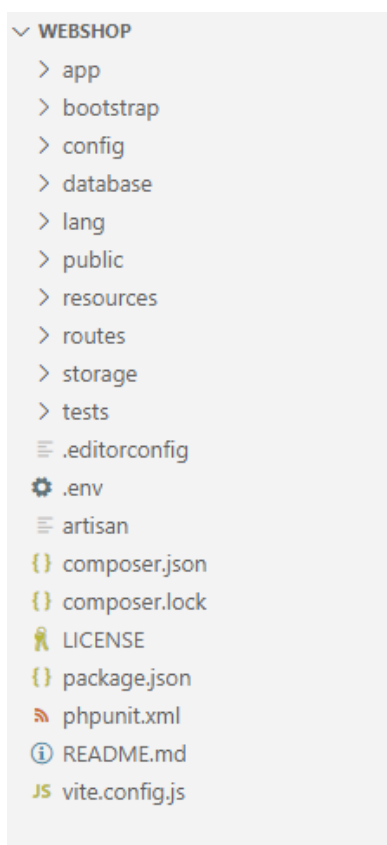
La gestión de dependencias del proyecto se efectuará con npm para las librerías javascript y con composer para librerías de módulos php

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

4. DISEÑO DE ALTO NIVEL

4.1. ESQUEMA GENERAL DEL DISEÑO DE ALTO NIVEL

No se contempla el desglose del sistema en subsistemas menores. el proyecto se adapta a una estructura típica de carpetas de un proyecto Laravel que pasamos a describir a continuación:



El directorio **app** contiene el código principal de tu aplicación (modelos, controllers, middleware, etc). Exploraremos este directorio en más detalle

El directorio **bootstrap** contiene el archivo `app.php` que inicializa el framework. Este directorio también alberga un directorio `cache` que contiene archivos generados por el framework para la optimización del rendimiento, como los archivos de caché de rutas y servicios.

El directorio **config**, como su nombre implica, contiene todos los archivos de configuración de tu aplicación.

El directorio **database** contiene tus migraciones de base de datos, fábricas de modelos y semillas.

El directorio **lang** contiene diferentes ficheros para la internacionalización de los componentes utilizados en la aplicación.

El directorio **public** contiene el archivo **index.php**, que es el punto de entrada para todas las solicitudes que ingresan a tu aplicación y configura la carga automática. Este directorio también alberga tus recursos como imágenes, JavaScript y CSS.

El directorio **resources** contiene tus vistas así como tus activos sin compilar, como CSS o JavaScript.

El directorio **routes** contiene todas las definiciones de rutas para tu aplicación. Por defecto, se incluyen dos archivos de rutas con Laravel: `web.php` y `console.php`. El archivo `web.php` contiene rutas que Laravel coloca en el grupo de middleware `web`, que

proporciona estado de sesión, protección CSRF y cifrado de cookies. Si tu aplicación no ofrece una API RESTful sin estado, entonces es muy probable que todas tus rutas estén definidas en el archivo `web.php`. El archivo `console.php` es donde puedes definir todos tus comandos de consola basados en funciones anónimas. Cada función anónima está vinculada a una instancia del comando, lo que permite un enfoque simple para interactuar con los métodos de IO de cada comando. Aunque este archivo no define rutas HTTP, define puntos de entrada basados en consola (rutas) en tu aplicación. También puedes programar tareas en el archivo `console.php`. Opcionalmente, puedes instalar archivos de ruta adicionales para las rutas de API (`api.php`) y los canales de transmisión (`channels.php`), a través de los comandos:

Artisan install:api y install:broadcasting.

El archivo `api.php` contiene rutas que están diseñadas para ser sin estado, por lo que las solicitudes que ingresan a la aplicación a través de estas rutas están destinadas a ser autenticadas a través de tokens y no tendrán acceso al estado de la sesión. El archivo `channels.php` es donde puedes registrar todos los canales de broadcasting de eventos que tu aplicación admite.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

El directorio **storage** contiene tus registros, plantillas Blade compiladas, sesiones basadas en archivos, cachés de archivos y otros archivos generados por el framework. Este directorio se divide en directorios app, framework y logs. El directorio app puede usarse para almacenar cualquier archivo generado por tu aplicación. El directorio framework se utiliza para almacenar archivos y cachés generados por el framework. Finalmente, el directorio logs contiene los archivos de registro de tu aplicación. El directorio storage/app/public se puede utilizar para almacenar archivos generados por el usuario, como avatares de perfil, que deben ser de acceso público. Debes crear un enlace simbólico en public/storage que apunte a este directorio. Puedes crear el enlace utilizando el comando `Artisan php artisan storage:link`.

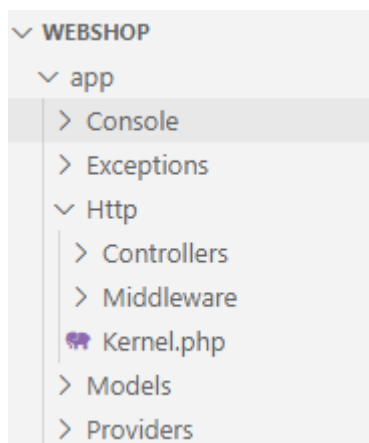
El directorio **tests** contiene tus pruebas automatizadas. Se proporcionan ejemplos de pruebas unitarias y pruebas de características de Pest o PHPUnit desde el principio. Cada clase de prueba debe tener el sufijo Test. Puedes ejecutar tus pruebas utilizando los comandos:

/vendor/bin/pest o /vendor/bin/phpunit.

O, si deseas una representación más detallada y bonita de los resultados de tus pruebas, puedes ejecutar tus pruebas utilizando el comando `Artisan php artisan test`.

El directorio **vendor** contiene tus dependencias de Composer.

Profundizando en el directorio **app** encontramos:



La mayoría de tu aplicación se encuentra en el directorio app. Por defecto, este directorio está en un espacio de nombres bajo App y se carga automáticamente con Composer utilizando el estándar de autoloading PSR-4. Por defecto, el directorio app contiene los directorios Http, Models y Providers. Sin embargo, con el tiempo, se generarán una variedad de otros directorios dentro del directorio app a medida que uses los comandos Artisan make para generar clases. Por ejemplo, el directorio app/Console no existirá hasta que ejecutes el comando `Artisan make:command` para generar una clase de comando.

Tanto los directorios Console como Http se explican más a fondo en sus respectivas secciones a continuación, pero piensa en los directorios Console y Http como proveedores de una API en el núcleo de tu aplicación. El protocolo HTTP y la CLI son ambos mecanismos para interactuar con tu aplicación, pero no contienen realmente la lógica de la aplicación. En otras palabras, son dos formas de emitir comandos a tu aplicación. El directorio Console contiene todos tus comandos Artisan, mientras que el directorio Http contiene tus controladores, middleware y solicitudes.

El directorio **broadcasting** contiene todas las clases de canal de transmisión para tu aplicación. Estas clases se generan utilizando el comando `make:channel`. Este directorio no existe por defecto, pero se creará para ti cuando crees tu primer canal. Para aprender más sobre los canales

El directorio **console** contiene todos los comandos Artisan personalizados para tu aplicación. Estos comandos se pueden generar utilizando el comando `make:command`.

El directorio **events** no existe por defecto, pero será creado para ti por los comandos `Artisan event:generate` y `make:event`. Alberga clases de eventos. Los eventos pueden usarse para alertar a otras partes de tu aplicación que se ha producido una acción dada, proporcionando una gran flexibilidad y desacoplamiento.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

El directorio **exceptions** contiene todas las excepciones personalizadas para tu aplicación. Estas excepciones se pueden generar utilizando el comando `make:exception`.

El directorio **http** contiene tus controladores, middleware y solicitudes de formulario. Casi toda la lógica para manejar las solicitudes que ingresan a tu aplicación se colocará en este directorio.

El directorio **jobs** no existe por defecto, pero se creará para ti si ejecutas el comando `Artisan make:job`. Los jobs pueden ser encolados por tu aplicación o ejecutarse de manera síncrona dentro del ciclo de vida de la solicitud actual.

El directorio **listeners** no existe por defecto, pero se creará para ti si ejecutas los comandos `Artisan event:generate` o `make:listener`. El directorio `Listeners` contiene las clases que manejan tus eventos. Los oyentes de eventos reciben una instancia de evento y realizan lógica en respuesta a que se dispare el evento. Por ejemplo, un evento `UserRegistered` podría ser manejado por un oyente `SendWelcomeEmail`.

El directorio **mail** no existe por defecto, pero se creará para ti si ejecutas el comando `Artisan make:mail`. El directorio `Mail` contiene todas tus clases que representan correos electrónicos enviados por tu aplicación. Los objetos `Mail` te permiten encapsular toda la lógica de construcción de un correo electrónico en una sola clase simple que se puede enviar utilizando el método `Mail::send`.

El directorio **models** contiene todas tus clases de modelo Eloquent. El ORM Eloquent incluido con Laravel proporciona una hermosa y simple implementación de `ActiveRecord` para trabajar con tu base de datos. Cada tabla de base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla. Los modelos te permiten consultar datos en tus tablas, así como insertar nuevos registros en la tabla.

El directorio **notifications** no existe por defecto, pero se creará para ti si ejecutas el comando `Artisan make:notification`. contiene todas las notificaciones "transaccionales" que envía tu aplicación, como notificaciones simples sobre eventos que ocurren dentro de tu aplicación. La función de notificaciones de Laravel abstrae el envío de notificaciones a través de una variedad de controladores como correo electrónico, Slack, SMS, o almacenadas en una base de datos.

El directorio **policies** no existe por defecto, pero se creará para ti si ejecutas el comando `Artisan make:policy`. Contiene las clases de política de autorización para tu aplicación. Las políticas se utilizan para determinar si un usuario puede realizar una acción dada contra un recurso.

El directorio **providers** contiene todos los proveedores de servicios para tu aplicación. Los proveedores de servicios inician tu aplicación vinculando servicios en el contenedor de servicios, registrando eventos o realizando cualquier otra tarea para preparar tu aplicación para las solicitudes entrantes. En una nueva aplicación Laravel, este directorio ya contendrá el `AppServiceProvider`. Puedes añadir tus propios proveedores a este directorio según sea necesario.

El directorio `rules` no existe por defecto, pero se creará para ti si ejecutas el comando `Artisan make:rule`. contiene los objetos de regla de validación personalizados para tu aplicación a aplicar sobre los modelos

4.2. DISEÑO DEL MODELO LÓGICO DE DATOS

Vease documento anexo `modelo.pdf`

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

5. DISEÑO DE BAJO NIVEL

5.1. DISEÑO DE LA CAPA DE PRESENTACIÓN

En este apartado se debe hacer una descripción desde el punto de vista técnico de las pantallas y módulos de soporte (validación, control de necesidad de campos, etc.) a la interacción del usuario con el sistema. El diseño de la capa de presentación deberá estar gobernado por los principios generales de desarrollo y especialmente por los aspectos de usabilidad


Nos apoyaremos en las características de plantillas blade de laravel. Una de las principales ventajas de Blade es su organización modular del código. Blade ayuda a organizar tu código en módulos reutilizables que puedes añadir, eliminar o actualizar fácilmente sin afectar al resto de tu aplicación.

La encapsulación del código es otra de las ventajas de Blade. Blade ayuda a encapsular funciones, haciendo que las pruebas, la depuración y el mantenimiento del código sean más manejables. Este enfoque beneficia a las aplicaciones más grandes, ya que las aplicaciones desorganizadas pueden convertirse rápidamente en un reto de gestión.


En nuestro caso particular definiremos un layout para la pagina de bienvenida, otro para la navegacion por productos de usuarios no autenticados y otro para usuarios autenticados (administradores o no). En este ultimo layout se mostrara un menu lateral donde ocultaremos opciones de menu en funcion del rol que tenga asignado el usuario.


Estos layouts constaran de cabecera, contenido, pie y menu lateral en su caso


F



aquí nombre del usuario







home

desconectar

MENU

administración

tablas maestras

[métodos de pago](#)

[métodos de envío](#)

[estados del pedido](#)

...

[categorías](#)

[productos](#)

[usuarios](#)

[clientes](#)

[proveedores](#)

[pedidos](#)

[devoluciones](#)

Utilidades

[logs de login usuario](#)

[copia de seguridad](#)

Número	Cliente	Fecha	País	Estado	Acciones
20240001	Fulanito	13/05/2024	España	Pendiente de envío	[Ver] [Editar]
20240001	Menganito.....	14/05/2024	Francia	Pagado	[Ver] [Editar]
20240001	Zutanito.....	15/05/2024	Italia	Posibles devoluciones	[Ver] [Editar]
.....	[Ver] [Editar]

datos de webshop

[política de ventas](#)

[devoluciones](#)

[privacidad](#)

[cookies de terceros](#)

[contacta](#)

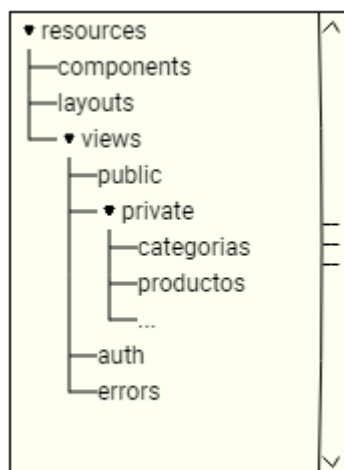
Vease apartado 5 interfaz de usuario del documento de analisis donde fueron definidas las diferentes pantallas.

Asi mismo utilizaremos los layouts y vistas proporcionados por el framework de seguridad de laravel para la gestión de usuarios

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Cuando el contenido html a incluir en la vista sea reutilizable se creara un "componente" que se incorporaran a las diferentes las diferentes vistas y layouts mediante etiquetas @include a modo de ejemplo podemos ver como los botones de temas diurno y nocturno son reutilizados en varios layouts por lo que se definira un componente para ello

La estructura de la carpeta resources, será la siguiente:



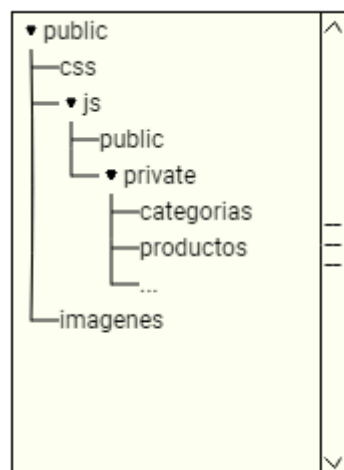
Components: contendrá todos los componentes utilizables por las vistas y plantillas.

Layouts: contendra todas las plantillas.

Vistas: dentro de la carpeta public pondremos todas las vistas que no requieren de una autenticación de usuario. Dentro de la carpeta private pondremos las vistas que requieren de autenticación por parte del usuario creando una carpeta por cada funcionalidad. Continuando con la imagen de ejemplo dentro de la carpeta categorias encontraremos las vistas de listado de categorias y detalle de una categoria. En la carpeta auth encontramos las vistas relacionadas con la gestion de usuarios y que son generadas automaticamente por laravel. Finalmente y tambien generados por laravel encontramos las vistas relacionadas con los "erores"

asociados a los estados de las repuestas http (según codigo de estado de respuesta).

En cuanto a la carpeta public encontraremos el siguiente contenido:



Css: ubicaremos todos los ficheros de hojas de estilo

js: ubicaremos todos los ficheros javascript. La carpeta private contendra los ficheros javascript por cada una de las vistas siguiendo la misma logica que la carpeta views. Es decir, si una vista tiene una ruta resources/views/xxx/yyy.blade.php y requiere un fichero javascript de apoyo este deberá estar localizado bajo la ruta public/js/xxx/yyy.js. si requiriese mas de un archivo por estructuracion de codigo estos contendran indices yyy_1.js yyy_2.js yyy_n.js

imágenes: dentro de la carpeta imágenes, encontraremos todos los recursos gráficos de la aplicación. Damos libertad para estructurar la carpeta como se estime más oportuno.

En cuanto al enrutado crearemos tres grupos de rutas uno para url publicas, otro para las rutas accesibles para usuarios autenticados y finalmente un conjunto de rutas para administradores. El grupo tendra los middleware establecidos para no dar acceso si no se dispone del rol.

rutas navegables por todos los usuarios

```
Route->group(function () {...})
```

rutas navegables por todos los usuarios autenticados

```
Route::middleware('auth')->group(function () {...}) rutas
```

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

rutas navegables por todos los usuarios autenticados con rol administrador
Route::middleware('auth', 'can:admin')-> group(function () {...}

Cada una de las rutas estará habilitada para su correspondiente verbo http, en el siguiente ejemplos e muestra la ruta para ver el detalle de un determinado usuario

```
Route::get('/user/{id}', [UserController::class, 'show']);
```

Como puede observarse esta url solo esta disponible bajo una petición get http y nunca sera visible bajo un post, delete , put o trace

Como norma general ninguna vista será devuelta por el fichero de rutas y todas las rutas redirigiran la petición a un controlador que será el encargado de retornar la vista conveniente. Continuando con el ejemplo anterior podemos ver como la ruta redirige a un metodo del controlador UserController.

Las rutas siempre seguiran un patron relativo al raiz de la aplicación y bajo una nomenclatura semantica no permitiendose el uso de querystring. Asi, a modo de ejemplo

http://localhost/webshop/usuario?id=xxx **PROHIBIDO**
/usuario/{id} **CORRECTO**

Con el objeto de facilitar el mantenimiento de la aplicación daremos un nombre a cada una de las rutas usando el metodo name

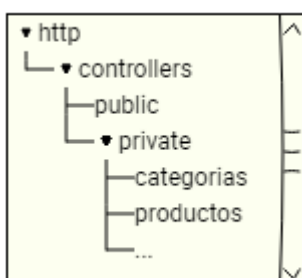
```
Route::get('/user/{id}', [UserController::class, 'show']->name('userShow');
```

5.2. DISEÑO DE LA CAPA DE NEGOCIO

La capa de negocio está formada básicamente por los controladores de laravel. Estos se crean con el comando

```
php artisan make:controller UserController
```

La estructura de esta carpeta será la siguiente:



Haremos coincidir la estructura de carpetas con las de las vistas. De tal forma que si queremos ver las vistas relacionadas con el controlador categorías visitaremos la carpeta equivalentedentro de resources:

resources/views/private/categorías

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

todos los controladores serán designados bajo la siguiente nomenclatura EntidadController, de esta forma el controller que administra las llamadas relacionadas con la entidad categorías se denominara CategoriaController usando siempre el singular y anexándole el sufijo Controller

Los controladores destinados a dar el soporte crud de las entidades contendrán como mínimo una función para cada uno de los métodos, index, create, store, show y delete.

No es necesario agregar middlewares a los controllers ya que están en las rutas.

Todos los métodos de los controladores que reciban datos de formularios deberán aplicar reglas de validación coherentes con el diseño de la base de datos evitando errores en la capa de persistencia

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

5.3. DISEÑO DE LA CAPA DE DATOS

La capa se desarrollara mediante el framework eloquent de laravel con persistencia de la información en la base de datos MySQL

Con el fin de no seguir la convención de pluralles en los nombres de tabla. Todas las clases de modelo contendran la propiedad table

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Por defecto, Eloquent espera que las columnas `created_at`, `updated_at` y `deleted_at` existan en la tabla de base de datos correspondiente a tu modelo. Eloquent establecerá automáticamente los valores de estas columnas cuando se creen o actualicen modelos. Cuando no se precisen de estos valores estableceremos una propiedad `$timestamps` en tu modelo con un valor de `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

Los modelos contendrán definidas las relaciones de las bases de datos mediante el uso de etiquetas `HasOne`, `hasMany` y `BelongsTo`

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Preferiblemente se realizará una carga lazy de las entidades relacionadas

6. OTRAS CARACTERÍSTICAS

En este apartado se podrán incluir todos aquellos aspectos del diseño que apliquen y sean importantes para la construcción del sistema y que no hayan podido cubrirse en otros apartados de este documento.

6.1. SEGURIDAD

La seguridad se realiza a través del framework breeze de laravel basado en autenticación de usuarios en base de datos y mediante el uso de Json Tokens siguiendo el flujo oauth mediante el paquete sanctum

Laravel Sanctum destaca por proporcionar una solución ligera para la autenticación de aplicaciones de una sola página (SPA) y APIs. A diferencia de otras herramientas como Passport, Sanctum simplifica la autenticación mediante la gestión de tokens de acceso personales y cookies de sesión. Un enfoque que permite a los desarrolladores proteger tanto APIs como aplicaciones front-end sin una configuración compleja.

El proceso de login podría definirse en los siguientes pasos

- 1) El cliente intenta acceder a una página privada, el controlador tiene definida una middleware que en caso de no estar autenticado le redirige a la página de login
- 2) El cliente introduce su usuario u contraseña y al servidor llega una petición http atendida por ConfirmPasswordController que toma el email y la contraseña de la petición aplica un encriptado md5 con la función hash y con la clave definida en el archivo de configuración .env. A continuación obtiene el modelo usuario desde la base de datos y compara las contraseñas. En caso de no coincidencia redirige a login con adjuntado el error para que se muestre por pantalla. En caso de coincidencia el sistema genera un token cuyo payload está constituido por datos del usuario, el token se almacena en base de datos para su refresco cada cierto tiempo (a configurar en el archivo /config/auth.php) y es devuelto al cliente quien lo almacena en el almacenamiento persistente de sesión del navegador. Y es redirigido al home de la aplicación
- 3) A partir de ese momento en cada petición del cliente se utiliza el encabezado **Authorization: Bearer <token>** que el middleware('auth') verifica en cada controller que lo tenga activado
- 4) cuando el token expira, el navegador es redirigido a una url donde obtiene el nuevo token a emplear

A continuación se detallan los diferentes controllers de este framework

RegisterController

El RegisterController, actúa en combinación con la clase RegistersUsers. Contiene valores predeterminados sobre cómo mostrar a los nuevos usuarios un formulario de registro, cómo validar su entrada, cómo crear nuevos usuarios una vez que se valida su entrada y dónde redirigirles después.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

La propiedad `$redirectTo` define dónde se redirigirá a los usuarios después del registro. El método `validator()` define cómo validar los datos aportados de usuario. Y `create()` es el método que define cómo crear un nuevo usuario basado en un registro entrante.

La clase `RegistersUsers`, que importa `RegisterController`, administra algunas funciones principales para el proceso de registro. Primero, muestra a los usuarios el formulario de registro. A continuación, maneja el POST del formulario de registro con el método `Register()`. Este método pasa la entrada de registro del usuario al validador invocando al método `validator()`. A continuación invoca al método `create()` para su persistencia en base de datos. Finalmente, el método `redirectPath()` define dónde se debe redirigir a los usuarios después de un registro con éxito. Puedes definir este URI con la propiedad `$redirectTo` en su controlador.

LoginController

Es el controlador encargado de realizar la autenticación de los usuarios, trabaja importando la clase `AuthenticatedUsers`. `LoginController` dispone de una propiedad `$redirectTo` donde se almacena la ruta a redirigir a los usuarios tras un login con éxito.

Por su parte la clase `AuthenticatedUsers` es la responsable de mostrar a los usuarios la vista con el formulario de login, validar sus intentos de autenticación y administrar los logouts en esta clase podemos ver entre otros los siguientes metodos:

`showLoginForm()`: metodo que por defecto devuelve la vista `auth.login.blade` con la vista del formulario de login

`login()`: realiza la comprobacion de las credenciales de usuario. Admite solamente el verbo http post y valida los datos de entrada con las credenciales. El método `login()` acepta el POST del formulario de inicio de sesión. Valida la solicitud usando el método `validateLogin()`. Luego utiliza la clase `ThrottlesLogins` para rechazar usuarios con demasiados inicios de sesión fallidos (5 intentos por minuto). Y finalmente, redirige al usuario a la ruta deseada (si el usuario fue redirigido al inicio de sesión)

página al intentar visitar una página dentro de la aplicación) o la ruta definida por el `redirectPath()`, que devuelve su propiedad `$redirectTo`.

`authenticated()`: establece las credenciales de ese usuario como autenticadas

`Username()`: devuelve el nombre del usuario autenticado, en laravel por defecto es el campo email.

ResetPasswordController

`ResetPasswordController` simplemente usa la clase `ResetsPasswords`. Esta proporciona validación y acceso a vistas básicas de restablecimiento de contraseña y luego utiliza un instancia de la clase `PasswordBroker` de Laravel para el envío de la nueva contraseña. Tambien se utiliza para restablecer correos electrónicos. Contiene los siguientes metodos interesantes:

`showResetForm()`: muestra la vista de restablecimiento de contraseña. En concreto la vista `auth.passwords.reset.blade`

`resetPassword()`: cuando el controller recibe los datos del formulario los valida e invoca a este metodo para el reestablecimiento efectivo de la contraseña.

ForgotPasswordController

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Este controlador se apoya en la clase SendsPasswordResetEmails. Muestra el formulario auth.passwords.email.blade con el metodo showLinkRequestForm() y maneja el POST de ese formulario con el método sendResetLinkEmail() que simplemente envia un email con un enlace para cambiar la contraseña de un solo uso.

VerificationController

VerificationController importa la clase VerifiesEmails, que se encarga de verificar las direcciones de correo electrónico de los usuarios recién registrados. Puede personalizar la ruta a la que se redirigen los usuarios a después de la validación, normalmente una vista personalizada agradeciendo su registro

6.2. GESTION DE LOGS Y ERRORES

Nos apoyaremos en la funcionalidad de logs proporcionados por Laravel. En cada captura de excepción se pondrá un log a nivel "error", "critical", "atert" o "emergency" en función de la criticidad del mismo y pondremos diferentes logs a nivel "info" en los puntos más importantes de la aplicación, como al almacenar y recuperar información de la capa de datos,

Para hacer escritura de logs importamos la clase use `Illuminate\Support\Facades\Log`;

Que dispone de los siguientes métodos:

```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

Toda la configuracion de logs se realiza en la clase /config/logging.php en la cual estableceremos un fichero de log diario con una antigüedad maxima de 7 dias rotativo y en una ruta del servidor que se tomara del fihero .env

```
'daily' => [
    'driver' => 'daily',
    'path' => storage_path(env('LOG_LEVEL', 'logs/laravel.log')),
    'level' => env('LOG_LEVEL', error),
    'days' => env('LOG_DAILY_DAYS', 7),
    'replace_placeholders' => true,
],
```


Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

también dejaremos logs en base de datos mediante uso de eventos de laravel que permitan llevar la auditoria del registro de sesión de usuarios. Toda esta información deberá ser visible por los usuarios administradores en la propia aplicación mediante las vistas específicas.

6.3. BACKUP

La mejor herramienta sin duda para realizar copias de seguridad en Laravel es Spatie Backup, la cual es compatible desde Laravel 5.1.20 y Php 5.5.9. Este paquete está disponible para MySQL y PostgreSQL a través de mysqldump y pg_dump respectivamente.

Se configura desde el archivo /config/backup

Donde definiremos una serie de parametros como la compresion o no de los datos, la ubicación de los ficheros dump , si el dump deberá estar encriptado o no y las notificaciones que se deben realizar durante las diferentes operaciones que realiza.

Se creara un script php que programaremos en el kernel del sistema para que se ejecute diariamente y que cree una copia de seguridad de la base de datos y archivos

```
<?php
protected function schedule(Schedule $schedule) {
    // genera una copia de seguridad completa, base de datos y archivos cada día a las 04:00
    $schedule->command("backup:run")->dailyAt("04:00");
}
```

Asimismo se dispondra un menú de usuarios administradores que le permita crear un backup en cualquier momento.

7. ANEXO I COMANDOS COMPOSER

composer init: Este comando se utiliza para crear un archivo "composer.json" en un proyecto vacío. Pide al usuario que proporcione información básica sobre el proyecto y crea un archivo "composer.json" con la información proporcionada.

composer install: Este comando se utiliza para instalar las dependencias de un proyecto. Lee el archivo "composer.json" y descarga las dependencias especificadas en el archivo, así como sus dependencias recursivas. También genera el archivo "vendor/autoload.php" que se puede incluir en el código de la aplicación para cargar automáticamente las dependencias.

composer require <package>: Este comando se utiliza para instalar una nueva dependencia en un proyecto. Añade la dependencia al archivo "composer.json" y la descarga.

composer update: Este comando se utiliza para actualizar las dependencias de un proyecto a sus versiones más recientes. Lee el archivo "composer.json" y descarga las últimas versiones disponibles de las dependencias especificadas en el archivo, así como sus dependencias recursivas.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

`composer remove <package>`: Este comando se utiliza para eliminar una dependencia de un proyecto. Elimina la dependencia del archivo "composer.json" y elimina los archivos de la dependencia del directorio "vendor".

`composer search <query>`: Este comando busca paquetes en el repositorio central de Composer (Packagist) que coinciden con la consulta especificada.

`composer show`: Este comando muestra información sobre las dependencias instaladas en un proyecto, incluyendo sus nombres, versiones y autores.

`composer validate`: Este comando valida el archivo "composer.json" para asegurar que está bien formado y que las dependencias especificadas son válidas.

`composer why-not <package>`: Este comando muestra por qué un paquete determinado no puede ser instalado en el proyecto actual, incluyendo cualquier conflicto de versiones o dependencias.

`composer why <package>`: Este comando muestra qué paquetes hacen que se instale el paquete dado.

`composer self-update`: Este comando actualiza la instalación de Composer a la última versión disponible.

`composer outdated`: Este comando muestra las dependencias de un proyecto que están desactualizadas y las versiones más recientes disponibles.

`composer create-project`: Este comando se utiliza para crear un proyecto a partir de un paquete Composer existente. Por ejemplo, se puede utilizar para instalar un framework como Laravel o Symfony.

`composer global require <package>`: Este comando instala una dependencia globalmente en el sistema, lo que permite utilizarla en cualquier proyecto sin tener que instalarla manualmente.

`composer prohibits <package>`: Este comando muestra qué paquetes impiden que se instale el paquete dado.

`composer home <package>`: Este comando muestra la página del repositorio del paquete dado, si no se ofrece, en un proyecto Laravel por ejemplo se abrirá la página del repositorio de Laravel Framework.

`composer suggests`: Este comando muestra sugerencias de paquetes para nuestro proyecto.

`composer exec <binary>`: Este comando ejecuta un script binario del directorio vendor/bin.

`composer fund`: Este comando muestra cómo ayudar a financiar el mantenimiento de las dependencias utilizadas.

`composer licenses`: Este comando muestra información acerca del tipo de licencias de las dependencias utilizadas.

`composer status`: Este comando muestra una lista de paquetes modificados localmente.

8. ANEXO II COMANDOS NPM

Ver la versión actual de node:

```
$ node -v
```

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

Instalar dependencias de desarrollo:

```
$ npm install nombre_paquete -D
```

O versión extendida:

```
$ npm install nombre_paquete --save-dev
```

Comprobar si existen versiones más actuales de los paquetes instalados:

```
$ npm outdated
```

Actualizar todos los paquetes:

```
$ npm update
```

Actualizar solo el paquete nombrado:

```
$ npm update nombre_paquete
```

Desinstalar el paquete nombrado:

```
$ npm uninstall nombre_paquete
```

Revisar vulnerabilidades de seguridad:

```
$ npm audit
```

Corregir vulnerabilidades de seguridad:

```
$ npm audit fix
```

Actualizar npm globalmente a la última versión:

```
$ npm install npm@latest -g
```

Eliminar node_modules de tu proyecto y volver a instalar las dependencias de nuevo:

```
$ rm -rf node_modules
```

```
$ rm -rf package-lock.json
```

```
$ npm cache clean -f
```

```
$ npm install
```

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

9. ANEXO III COMANDOS ARTISAN DE LARAVEL

php artisan help Muestra la ayuda general de Artisan.

php artisan list Lista todos los comandos Artisan disponibles.

php artisan info Muestra información sobre tu entorno de Laravel.

php artisan version Muestra la versión de Laravel instalada.

php artisan cache:clear Limpia la caché de la aplicación.

php artisan config:cache Limpia la caché de configuración de la aplicación.

php artisan route:cache Limpia la caché de rutas de la aplicación.

php artisan optimize Optimiza la aplicación para producción.

php artisan migrate Ejecuta las migraciones pendientes.

php artisan migrate:reset Deshace todas las migraciones y las vuelve a ejecutar.

php artisan migrate:status Muestra el estado de las migraciones.

php artisan db:seed Ejecuta los seeders de la aplicación.

php artisan tinker Abre Tinker, una consola interactiva para PHP.

php artisan db:drop Elimina todas las tablas de la base de datos (¡con precaución!).

php artisan db:backup Crea una copia de seguridad de la base de datos.

php artisan db:restore Restaura una copia de seguridad de la base de datos.

php artisan make:auth Crea los controladores, modelos, vistas y migraciones necesarias para la autenticación de usuarios.

php artisan register Registra un nuevo usuario en la aplicación.

php artisan login Inicia sesión en la aplicación como un usuario existente.

php artisan logout Cierra la sesión del usuario actual.

php artisan password:reset Envía un correo electrónico con un enlace para restablecer la contraseña de un usuario.

php artisan make:model Crea un nuevo modelo de Eloquent.

php artisan make:controller Crea un nuevo controlador.

php artisan make:job Crea un nuevo trabajo en cola.

php artisan make:event Crea un nuevo evento.

php artisan make:listener Crea un nuevo listener para un evento.

php artisan make:middleware Crea un nuevo middleware.

php artisan make:request Crea una nueva clase de solicitud HTTP.

php artisan make:test Crea una nueva prueba unitaria.

php artisan composer require <paquete> Instala un paquete Composer en tu proyecto.

php artisan composer remove <paquete> Desinstala un paquete Composer de tu proyecto.

php artisan composer update Actualiza los paquetes Composer instalados en tu proyecto.

php artisan queue:work Procesa trabajos en cola.

Fecha	10/11/2024	Versión	1.0.0
Autor:	Aitor Logedo Ordoñez		
Documento:	DSI Diseño Técnico		
Proyecto:	webshop		

php artisan queue:failed Muestra los trabajos en cola que han fallado.

php artisan queue:retry Reintenta los trabajos en cola que han fallado.

php artisan queue:flush Elimina todos los trabajos en cola.

php artisan tinker Abre Tinker, una consola interactiva para PHP.

php artisan key:generate Genera una clave de aplicación aleatoria.

php artisan serve Inicia un servidor web local para tu aplicación.

php artisan down Pone la aplicación en modo mantenimiento.

php artisan up Saca la aplicación del modo mantenimiento.

php artisan storage:link Crea un enlace simbólico en la carpeta storage.

php artisan session:clear Limpia la caché de sesiones de la aplicación.

php artisan view:clear Limpia la caché de vistas de la aplicación.

php artisan route:clear Limpia la caché de rutas de la aplicación.

php artisan config:clear Limpia la caché de configuración de la aplicación.

php artisan cache:clear Limpia la caché de la aplicación.