CONSEJERÍA DE EDUCACIÓN
Comunidad de Madrid

IES ENRIQUE TIERNO GALVAN

Parla

CFGS DESARROLLO DE APLICACIONES WEB

Curso 2024/2025

Proyecto DAW

TITULO: *RoyList*

Alumno**:** José Rodrigo Santamaría Gallardo

Tutor**:** Francisco Isidoro Jiménez Caro

Junio 2025

ÍNDICE

1. INTRODUCCIÓN

1.1. Resumen del proyecto

He creado *RoyList* para ayudar a organizar, de manera sencilla, las listas de la compra. Desarrollada con Laravel 12 y Tailwind 4, para dar una experiencia rápida, cómoda e intuitiva para el usuario. Una de las ventajas que tiene esta aplicación, y una de las razones por las que me he decido a crearla, es que ya no vas a tener que estar con papel y boli apuntando los productos que necesitas en el Post-it de la nevera. Ya que, adaptándose a los tiempos de ahora, en los que todo el mundo tiene un dispositivo, ya sea un smartphone, un ordenador, una tablet, etc. Vas a poder, en el momento y lugar que te acuerdes de que necesitas un producto, apuntarlo en la aplicación con solo unos clics.

Además, se podrá explorar por los productos del supermercado separados por categorías como si se estuviese en persona en el supermercado. Ya que obtiene los productos de una API no-oficial de Mercadona con foto, precio y categoría. Aunque en esta primera versión de la aplicación solo está disponible el supermercado Mercadona, la idea es ir ampliando próximamente a otras cadenas como Ahorra Más, Dia, Carrefour, Lidl, Aldi, etc. También se prevé añadir más funcionalidades como la comparación de precios, notificaciones de ofertas/bajadas de precio y más.

1.2. Justificación del proyecto

Este proyecto surgió de una necesidad. Y vi la oportunidad de solucionarla. En mi familia muchas veces cuando íbamos a comprar, y llegábamos al supermercado, se nos olvidaba la lista de la compra que habíamos ido haciendo durante la semana o el mes en casa. Y muchas veces no recordábamos lo que habíamos escrito, o lo que necesitábamos. También muchas veces cuando mi padre o mi madre volvían de trabajar se pasaban por el supermercado antes de pasa por casa, y me pedían a mi o a mi hermana una foto de la lista de la compra y a veces, no estábamos en casa, por lo que tenían que comprar otra vez de memoria. Esto seguramente les pase a muchas personas y otras tantas familias. Entonces a principios de curso, me propuse hacer una aplicación que solucionara este problema.

Así que hice una aplicación en PHP puro. Con dos clases y una BD en MySQL. Ahora mismo está alojada en InfinityFree y se puede acceder a través de este enlace comprasanga.wuaze.com. La aplicación cumplía con los requisitos básicos, pero no era gran cosa. Ya que en el momento en el que la hice no tenía mucho conocimiento de PHP. Más adelante se me fueron ocurriendo funcionalidades sobre cómo mejorarla. Así es como surgió RoyList.

Creo que esta aplicación puede aportar mucho a los usuarios, porque les ahorrará tiempo y preocupaciones al tener siempre la lista actualizada en el móvil. Permite compartirla con la familia, evitando tener que ir a comprar de memoria. Además, al mostrar precios reales, puedes planificar

mejor la compra con el presupuesto que consideres y, pronto, comparar ofertas entre supermercados.

1.3. Objetivos

- Facilitar que la gestión de listas de la compra.
- Permitir la creación y edición de listas en tiempo real.
- Garantizar el acceso de la lista desde cualquier dispositivo.
- Proteger la información del usuario mediante un sistema robusto de autenticación.
- Optimizar el presupuesto familiar.
- Escalar la aplicación para añadir nuevas cadenas de supermercados.

1.4. Tecnologías usadas

Las tecnologías empleadas en el proyecto han sido:

- **Laravel 12**: Framework PHP para la estructura el backend.
- **Blade**: Motor de plantillas para las vistas.
- **Tailwind 4**: Framework CSS para el frontend.
- **Node.js + Express**: Plataforma y framework que levantan la API de los productos.
- **MySQL**: Gestor de Base de Datos relacional.
- **PHPMyAdmin**: Aplicación web para poder administrar la Base de Datos de forma visual.
- **Vite**: Herramienta que compila los archivos del frontend y actualiza la página al instante mientras se desarrolla.
- **Javascript**: Controla algunos datos entre la API y el backend.
- **XAMPP**: Software para poder levantar los servicios MySQL y Apache.

2. IDENTIFICACIÓN DE NECESIDADES DEL SECTOR

2.1. Público al que va dirigido

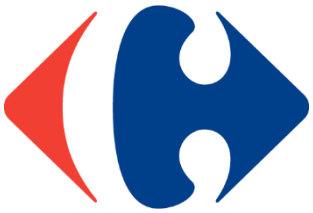
RoyList va dirigido, sobre todo, a familias y hogares con varios miembros, en los que quieren compartir una misma lista de la compra en el móvil para que cada persona pueda añadir los productos que necesite y que los demás integrantes lo puedan ver. También puede ser una buena solución para aquellas personas con poco tiempo que quieren una herramienta sencilla para preparar la compra desde cualquier lugar a cualquier hora, solamente desde su dispositivo móvil u ordenador. Para usuarios que busquen ahorrar lo máximo. Que quieran ver los precios reales antes de ir al supermercado. Es una perfecta aplicación para aquellos jóvenes que quieren dejar la costumbre de apuntar los productos en una hoja, post-it o lo que sea y quieran tener la lista más a mano.

2.2. Competencia principal

- **Apps de supermercados**

Mercadona, *Carrefour*, *Lidl*, etc. Permite comprar directamente del catálogo de cada tienda, pero solo sirven para esa cadena y no ofrecen una lista colaborativa entre familia.

Logo Lidl



Logo Carrefour

- **Aplicaciones de listas genéricas**

AnyList, *Bring!*, *Listonic* (populares fuera de España). Dejan crear listas y compartirlas, pero no conectan con los precios reales de los supermercados españoles.

Logo AnyList

Logo Listonic

- **Comparadores de precios online**

FindItApp, *Soysuper.com* muestran donde encontrar producto al mejor precio, pero no guardan listas colaborativas ni facilitan la planificación familiar

Logo FindItApp

Logo Soysuper.com

2.3. ¿Qué puede ofrecer RoyList de diferente?

- **Listas a la carta por supermercado:** Cada lista está vinculada a un establecimiento (Mercadona, y pronto otros.
- **Datos reales de precios:** Al conectar con el catálogo de Mercadona los usuarios ven precios y evitan pagar más de la cuenta.
- **Compartir en tiempo real:** Todos los de la casa pueden ver lo que comprar y eliminar desde su propio dispositivo.
- **Funciones sencillas:** Solo se necesita abrir la web (Aplicación móvil en un futuro), iniciar sesión y empezar a explorar y añadir productos a las listas sin complicaciones.
- **Comparación de precios:** Cuando se añadan más supermercados, la aplicación ayudará a comparar precios entre productos similares de distintos supermercados y ahorrar en la compra

2.4. Oportunidades

- **Número creciente de compras online:** Cada vez más gente se anima a gestionar la compra desde el móvil y valora ir más preparado al supermercado
- **Pocas opciones que sumen colaboración + precios reales:** Las aplicaciones actuales suelen ofrecer o una cosa o la otra, pero en pocas ocasiones las dos juntas.
- **Interés por ahorrar y planificar:** Con la subida de los precios y del nivel de vida, los usuarios buscan mejores formas de planificar sus gastos.

2.5. Amenazas

- **Lealtad a las apps de las grandes cadenas:** Muchos clientes prefieren entrar directamente en la app de Mercadona o Carrefour porque ya están acostumbrados.
- **Cambios en catálogos online:** Si Mercadona u otras cadenas cambian la forma de mostrar sus productos, puede complicarse mantener la información actualizada.
- **La competencia de otras apps que también evolucionan:** Si *AnyList, Bring!* u otras aplicaciones empiezan a conectar con supermercados españoles, RoyList tendrá que ofrecer más valor para destacar.

2.6. Comparativa directa

App	Qué hace	En qué se parece	Diferencia clave
AnyList	Listas colaborativas con autocompletado y categorías.	Colaborativa.	No muestra precios ni catálogos de supermercados. Solo disponible en inglés.
Bring!	Listas compartidas con recetas. Tarjetas de fidelidad.	Colaborativa y multiplataforma. Sugerencias visuales.	No conecta con precios reales. No separa listas por supermercados.
Listonic	Sugiere productos basados en compras previas. Sincronizado entre dispositivos.	Colaborativa. Edición en tiempo real.	No integra catálogos reales.
Out Of Milk	Gestiona listas, inventario de despensa y tareas, comparte listas y funciona sin conexión.	Colaborativa	No separa listas por supermercados. Y carece de precios de supermercados españoles
App Mercadona	Compra online y añade productos a la cesta desde el catálogo oficial de Mercadona	Productos y precios reales de Mercadona	Solo sirve para un único supermercado y no permite listas colaborativas externas
Mi Carrefour	Crea múltiples listas, repite compras, escanea códigos y gestiona cupones de fidelidad	Productos y precios de catálogo propio	Ligada solo a Carrefour no ofrece integración con otros supermercados.

2.7. Modelo de negocio

Para asegurar el futuro y la escalabilidad de RoyList, seria mas que recomendable definir modelos de ingresos que mantengan la satisfacción de los usuarios. Estos son algunas ideas que podrían aplicarse según el crecimiento de la plataforma:

1. **Modelo freemium con suscripción**

- Versión gratuita:
    - Listas limitadas
    - Sin alertas de bajada de precio
- Versión premium:
    - Listas ilimitadas
    - Alertas personalizadas
    - Historial de compra y estadística de gasto
    - Acceso prioritario a nuevas funcionalidades

2. **Comisiones por afiliación o acuerdos con supermercados**

- Establecer convenios con cadenas de supermercados a cambio de llevar tráfico o ventas desde RoyList a su e-commerce.

3. **Publicidad o promociones patrocinadas**

- Anuncios dentro de la app: Banners o recomendaciones patrocinadas al añadir productos.
- Colaboraciones puntuales: ofertas especiales de determinadas marcas o supermercados.

4. **Venta de datos anónimos**

- Recopilar de forma anónima, patrones de compra (categoría mas buscada, horas pico de uso, preferencias de productos). Y ofrecer estos informes a distribuidores o estudios de mercado interesados en entender el comportamiento del consumidor.

5. **Funciones avanzadas para empresas o negocios pequeños**

- Licencias o suscripciones B2B: ofrecer una versión personalizada de RoyList adaptada a pequeñas tiendas o cadenas locales que quieran gestionar pedidos online de clientes
- Servicio de integración de catálogo personalizado: se podría cobrar a un supermercado por configurar y mantener su propio catálogo dentro de RoyList.

3. DISEÑO

3.1. Análisis de los requisitos

En este apartado se definirán los objetivos que tiene RoyList, y a partir de ahí extraeremos los requisitos funcionales y no funcionales que van a servir de guía para el diseño e implementación de la aplicación.

3.1.1. Objetivos del proyecto

- Facilitar la gestión de listas de la compra personalizadas por supermercado
- Permitir la creación y edición colaborativa de listas en tiempo real entre varias personas
- Integrar datos reales de productos y precios mediante una API externa actualizada
- Garantizar el acceso a las listas y productos desde cualquier dispositivo, siempre con conexión a internet
- Proteger la información del usuario mediante un sistema robusto de autenticación
- Optimizar el presupuesto de la familia ofreciendo, en el futuro, comparativas de precio entre supermercados

3.1.2. Requisitos Funcionales

La aplicación RoyList ofrece un sistema robusto y seguro que permita al usuario iniciar sesión y registrarse con correo electrónico y contraseña, esta que tenga al menos 8 caracteres, siendo uno un carácter especial, una mayúscula, una minúscula y un número. El usuario también debe poder cerrar sesión cuando lo desee.

Tras el inicio de sesión, el usuario necesita crear y gestionar listas de la compra asociadas a un supermercado en concreto. Para ello, la aplicación debe ofrecer la posibilidad de crear varias listas con nombres descriptivos (ej. “Fruta Mercadona”), cada una etiquetada con el supermercado correspondiente y una fecha de creación. Después de crear una lista el usuario debe poder eliminarla si así lo desea.

Dentro de cada lista, por el momento, el usuario solo podrá añadir productos que vengan directamente del catálogo online de Mercadona (vía API). Cuando el usuario seleccione una categoría (ej. lácteos), la aplicación mostrará todos los productos disponibles de esa categoría, incluyendo nombre, imagen, enlace a la página oficial del producto y precio actual. Cuando el usuario pulse en el botón de añadir un producto, saltara un ‘modal’ en el que se pueden elegir tanto la cantidad que queremos añadir como un apartado de ‘notas’ (ej. Comprar los aguacates maduros). Aunque sí que prevé que se pueda en un futuro, no se pueden añadir productos personalizados ni introducir datos a mano.

Una vez haya añadido uno o más productos a la lista, esta los mostrará en la pestaña de "Mis productos" con los siguientes datos: Precio, unidades si hay más de dos, notas, si las hay e imagen. Esta pestaña también mostrará el precio total de la lista. El usuario puede eliminar cualquier producto de la lista cuando el desee.

Las contraseñas están cifradas en Base de Datos con Bcrypt. Herramienta utilizada por Laravel para hashearlas utilizando cifrado **AES-256** y **AES-128**. Cada usuario podrá acceder única y exclusivamente a las listas creadas por este mismo. Tampoco va a poder acceder a otras cuentas de usuario.

3.1.3. Requisitos no Funcionales

En términos de usabilidad, la interfaz debe ser muy clara e intuitiva: menús y botones de un tamaño correcto y unos colores visibles con contraste. Un diseño responsive para que se pueda usar cómodamente en cualquier tipo de dispositivo, ya sea una pantalla de un móvil en vertical como en un monitor de ordenador.

La aplicación debe ser rápida y eficiente, las peticiones a la API del catalogo no deben superar los 3 segundos.

En cuanto a la escalabilidad, otro aspecto clave, el diseño en Laravel 12 es más que correcto debido a su alta compatibilidad con otros frameworks de frontend. Como pueden ser Blade, React, etc. Permitiendo el añadir nuevos supermercados sin cambiar la arquitectura principal. El código debe estar estructurado en capas (servicios, controladores, etc.) para poder ampliar funcionalidades sin refactorizar grandes cantidades de código.

El código también deberá de ser mantenible y portable. Debe poder desplegarse en una maquina que tenga PHP 8 o superior, MySQL 8, Node.js 16 o superior y Laravel 12.

La aplicación tiene que estar disponible un mínimo de 99% del tiempo en las horas que consideramos críticas de uso (7h – 23h). Cualquier mantenimiento planificado se tendrá que realizar fuera de ese horario y se comunicará con un mensaje por la aplicación o por correo con antelación. Tendrá copias de seguridad automáticas cada día de la base de datos.

En lo que respecta a seguridad de la aplicación, debe impedir ataques comunes. Todas las entradas de formulario (registro, inicio de sesión, etc.) se validarán para evitar inyecciones SQL o ataques XSS. Laravel tiene protección automática contra CSRF (Cross-Site Request Forgery) para evitar el envío de formularios maliciosos. La aplicación exigirá una contraseña de mínimo 8 caracteres, combinar letras mayúsculas, minúsculas, números y un carácter especial (@, #, &, $, etc.)

3.2. Diseño de Base de Datos

3.2.1. Modelo Entidad-Relación

En el modelo entidad relación se identifican las identidades relevantes del dominio y las relaciones que existen entre ellas, sin entrar aun en detalles de atributos o tipos de datos concretos. Para esta aplicación he llegado a la conclusión de que este era el modelo correcto:

1. **Usuario**

Representa a cada persona que se registra la aplicación y puede crear listas y añadir productos a estas.

2. **Supermercado**

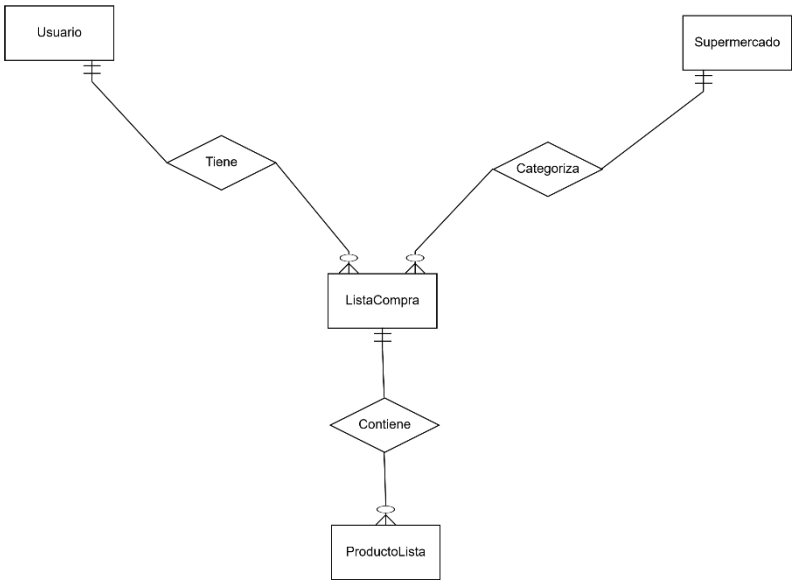
Almacena las cadenas de supermercados disponibles (ahora mismo Mercadona) aunque en un fututo habrá más.

3. **Lista de la compra**

Cada lista tiene un nombre (ej. Compra semanal), una fecha de creación y está vinculada a un usuario y a un supermercado.

4. **Productos**

Representa cada elemento que un usuario añade a una lista de la compra. Esta entidad contiene el identificador de la lista a la que pertenece, el nombre real del producto, la cantidad, etc.



Esquema Entidad Relación

3.2.2. Correspondencia

- Un **Usuario** puede tener varias **Listas de la compra** (1:N).
- Un **Supermercado** puede estar asociado a varias **Listas de la compra** (1:N).
- Una **Lista de compra** puede contener muchos **Productos** (1:N).

3.2.3. Cardinalidad

1. Usuario – Lista de la compra
    - Usuario → Lista de la compra: 0..N
        - Un usuario puede no tener ninguna lista (0) o tener muchas (N)
    - Lista de la compra → Usuario: 1..1
        - Cada lista debe pertenecer a un único usuario

2. Supermercado – Lista de la compra
    - Supermercado → Lista de la compra: 0..N
        - Un supermercado puede no tener listas asignadas (0) o tener muchas (N)
    - Lista de la compra → Supermercados: 1..1
        - Cada lista debe asociarse a un único supermercado

3. Lista de la compra – Productos
    - Lista de la compra → Producto: 0..N
        - Una lista de la compra puede comenzar sin productos (0) o contener muchos (N)
    - Producto → Lista de la compra: 1..1
        - Cada producto pertenece exactamente a una lista

3.2.4. Modelo Lógico

En el modelo lógico se definen las tablas, sus columnas, tipos de datos y restricciones principales (clave primaria, foránea, índices, etc.). A continuación, se describen cada una de las tablas de la base de datos de RoyList:

- Tabla: **users**

Descripción: Almacena la información de los usuarios registrados en RoyList.

Campos y tipos:

- **id** (bigint unsigned, PK, AUTO_INCREMENT): identificador único de cada usuario.
- **name** (varchar (255), NOT NULL): nombre o alias del usuario.
- **email** (varchar (255), NOT NULL, UNIQUE): correo electrónico, sirve como login.
- **email_verified_at** (timestamp, NULLABLE): fecha y hora en la que el usuario verificó su email.
- **password** (varchar (255), NOT NULL): contraseña cifrada (bcrypt).
- **remember_token** (varchar (100), NULLABLE): token para recorder el inicio de sesión.
- **created_at** (timestamp, NULLABLE): fecha de creación del registro.
- **updated_at** (timestamp, NULLABLE): fecha de la última actualización del registro.

- Tabla: **supermercados**

Descripción: Lista de cadenas de supermercados que se pueden elegir al crear una lista

Campos y tipos:

- **id** (bigint unsigned, PK, AUTO_INCREMENT): identificador único de cada supermercado.
- **nombre** (varchar (255), NOT NULL, UNIQUE): nombre de la cadena (ej. Mercadona)
- **created_at** (timestamp, NULLABLE): fecha de creación del registro.
- **updated_at** (timestamp, NULLABLE): fecha de la última actualización del registro.

- Tabla: **lista_compra**

Descripción: almacena cada una de las listas que los usuarios crean para gestionar la compra

Campos y tipos:

- **id** (bigint unsigned, PK, AUTO_INCREMENT): identificador único de cada supermercado.
- **user_id** (bigint unsigned, NOT NULL, FK → users.id): referencia al usuario propietario de la lista.
- **Supermercado_id** (bigint unsigned, NOT NULL, FK → supermercados.id): referencia al supermercado asociado a esta lista.
- **nombre** (varchar (255), NOT NULL): nombre de la lista.
- **created_at** (timestamp, NULLABLE): fecha de creación del registro.
- **updated_at** (timestamp, NULLABLE): fecha de la última actualización del registro.

- Tabla: **producto_lista**

Descripción: guarda cada producto que se ha agregado a una lista de compra determinada

Campos y tipos:

- **id** (bigint unsigned, PK, AUTO_INCREMENT): identificador único de cada supermercado.
- **lista_compra_id** (bigint unsigned, NOT NULL, FK → lista_compra.id): referencia a la lista a la que pertenece este producto
- **nombre_producto** (varchar (255), NOT NULL): nombre del articulo.
- **cantidad** (int, NOT NULL, DEFAULT 1): número de unidades que se desea adquirir.
- **precio** (decimal (8,2), NULLABLE): precio (en euros) del producto
- **notas** (text, NULLABLE): campo de texto libre para que el propietario añada comentarios
- **imagen** (varchar (500), NULLABLE): URL de la imagen del producto (tal y como devuelve la API del Mercadona)
- **created_at** (timestamp, NULLABLE): fecha de creación del registro.
- **updated_at** (timestamp, NULLABLE): fecha de la última actualización del registro.

Aquí esta un diagrama en el que se resume visualmente todo lo anterior, las tablas y las relaciones descritas:

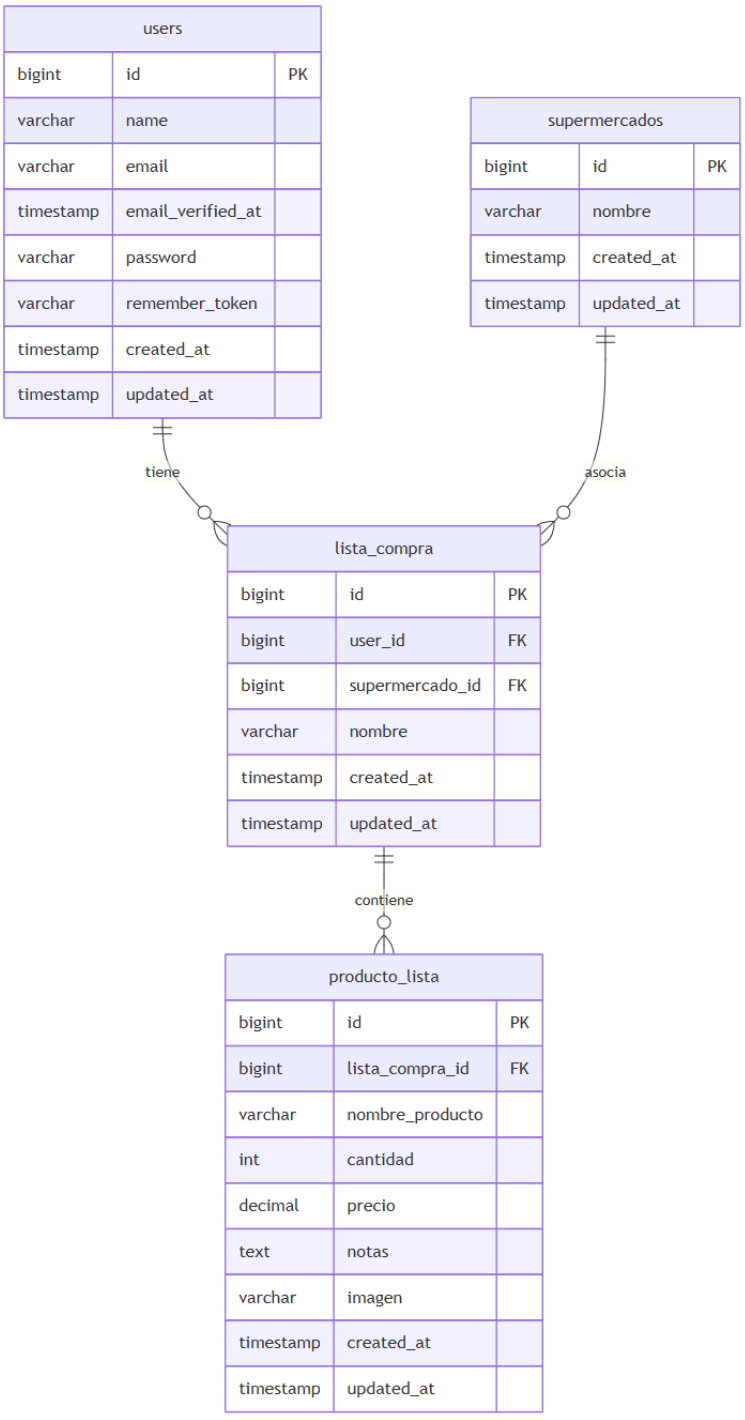


Diagrama Lógico Relacional

3.3. Diseño de la aplicación

En este apartado voy a agrupar todos los diagramas que se han hecho para diseñar la aplicación. Diagramas UML y de navegación que estructuran la arquitectura de la aplicación, el comportamiento y la interfaz. Antes de cada diagrama hay un resumen de este que explica su alcance y los componentes que refleja.

3.3.1. Diagrama de Clases

Modelo

El objetivo es representar las entidades que mapean las tablas de la base de datos y las relaciones que existen entre ellas, mostrando tanto los atributos que corresponden a las columnas de MySQL como los métodos para la manipulación de datos.

Las clases User, Supermercado, ListaCompra y ProductoLista representan los objetos, que se asocian a una tabla de la BD. Cada clase tiene atributos y métodos. Las flechas entre las clases muestran las relaciones de cardinalidad entre ellas.

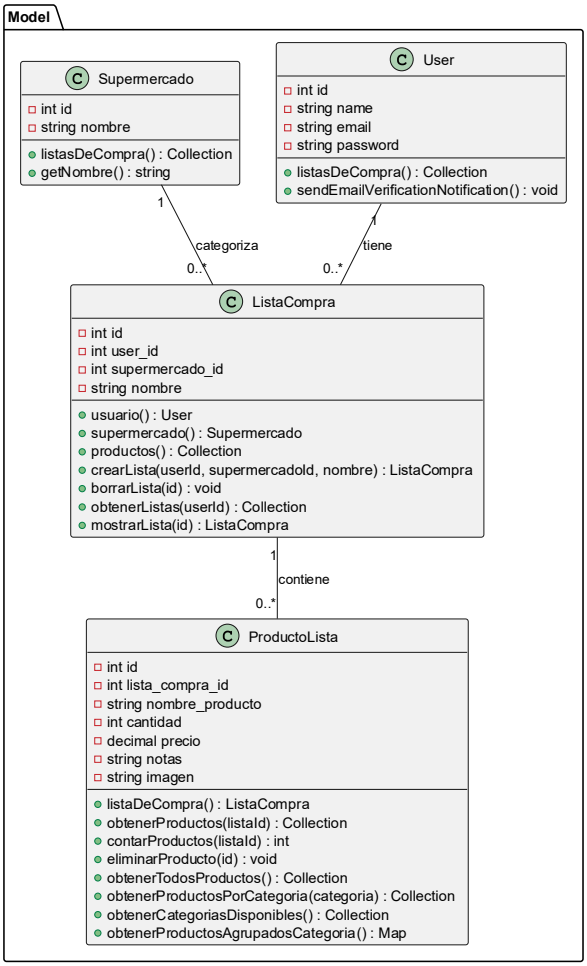


Diagrama del Modelo

Controlador

Este diagrama busca mostrar los controladores que gestionan a las entidades y coordinar el flujo de datos entre la capa de Vista y la del Modelo. Se incluyen cinco controladores: RegisterController y LoginController que gestionan el registro de autenticación y la autenticación de usuarios, y los demás que se encargán de la lógica de negocio principal. Cada controlador extiende de la clase base Controller y emplea los modelos correspondientes para llevar a cabo sus acciones.

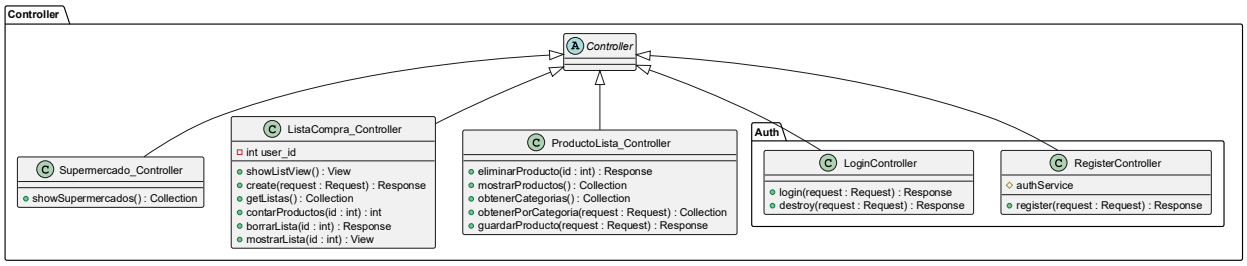
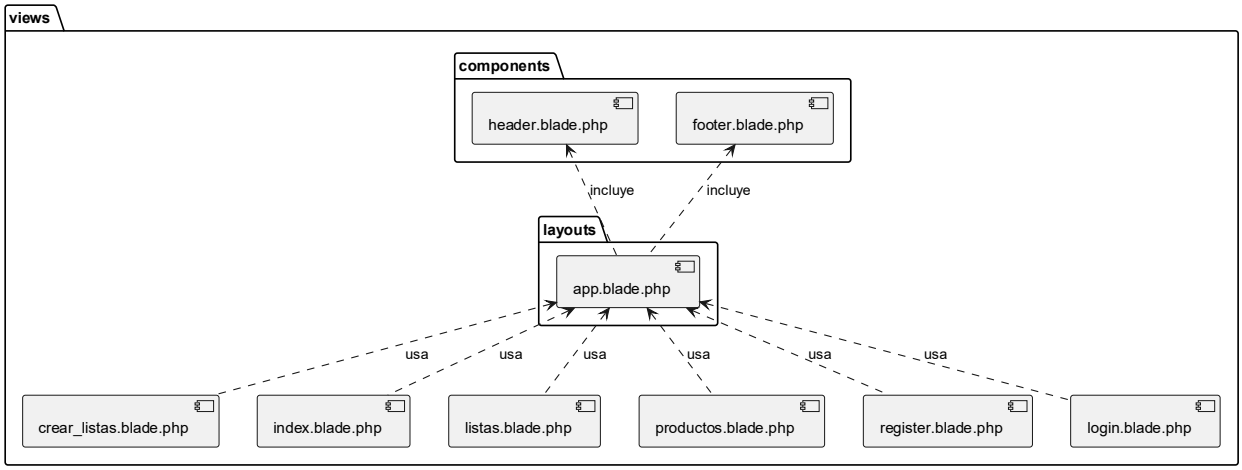


Diagrama del Controlador

Vista

En el siguiente diagrama se reflejan las plantillas de Blade que conforman la interfaz de usuario. La plantilla app.blade.php funciona como layout base, de la cual extienden las otras vistas principales como index.blade.php, etc. Para el header y el footer se han usado componentes para no repetir el código en todas las vistas. Se han indexado en app.blade.php.



Diseño de la Vista

3.3.2. Diagrama de Componentes

El diagrama de componentes muestra cómo se estructuran y se comunican las diferentes partes de la aplicación a un alto nivel.

1. **Cliente (Navegador Web)**

Representa la interfaz que utiliza el usuario final para interactuar con la aplicación. A través del navegador el usuario hace peticiones HTTP, envía formularios y visualiza los resultados. Este componente es pasivo en cuanto a lógica de negocio: solo envía peticiones y renderiza la respuesta que recibe del servidor.

2. **Servidor Web (Laravel/PHP)**

Es el que se encarga de recibir las solicitudes de los clientes y responderles con el contenido solicitado. A través de los diferentes servicios de la aplicación.

3. **Servidor de Base de Datos (MySQL)**

Aquí se almacenan todas las entidades persistentes de la aplicación: usuarios, supermercados, etc. Cada modelo Eloquent escribe y lee estas tablas. El motor InnoDB permite transacciones y soporte de transacciones, lo que asegura que las operaciones críticas sean atómicas y seguras.

4. **API**

La API es un microservicio independiente, construido en Node.js y Express que se despliega en el puerto 3000. Su única responsabilidad es ofrecer de manera eficiente, el catálogo de productos para cada categoría solicitada. Cuando el controlador de Laravel necesita obtener los productos disponibles para una categoría, envía una petición HTTP. La API procesa la solicitud y recupera los datos desde la fuente correspondiente y devuelve un array JSON con la información necesaria.

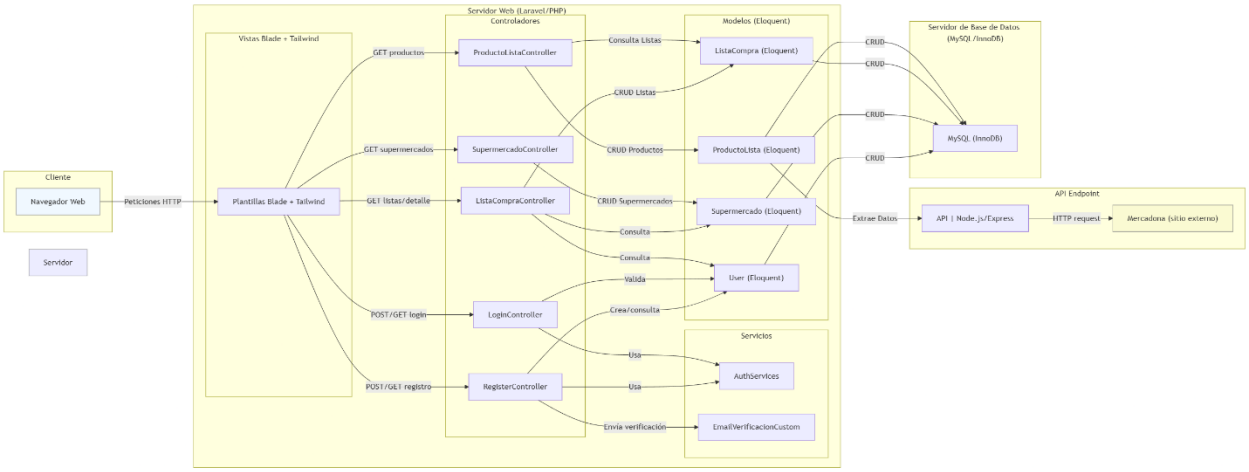


Diagrama de Componentes

3.3.3. Diagramas de Actividad

A continuación, se muestran los diagramas de actividad que describen, paso a paso, los procesos principales de RoyList. Cada diagrama utiliza dos columnas para distinguir las acciones del usuario de las reacciones internas de la aplicación, de modo que quede claro cómo fluye la interacción: desde que el usuario inicia una acción hasta que la aplicación valida datos, persiste información en la base de datos o envía notificaciones. Estos gráficos permiten visualizar tanto las decisiones como el orden en que ocurren cada uno de los pasos dentro del sistema.

Registro de un usuario

Este es el diagrama de actividad que representa el flujo de registro y verificación de un nuevo usuario en RoyList. En la primera columna el usuario completa el formulario con sus datos y envía la solicitud. En la segunda columna, la aplicación recibe esos datos, valida la información, crea el registro en base de datos y envía un correo de verificación. Finalmente, el usuario hace clic en el enlace recibido y la aplicación confirma el token, actualizando el estado de verificación.

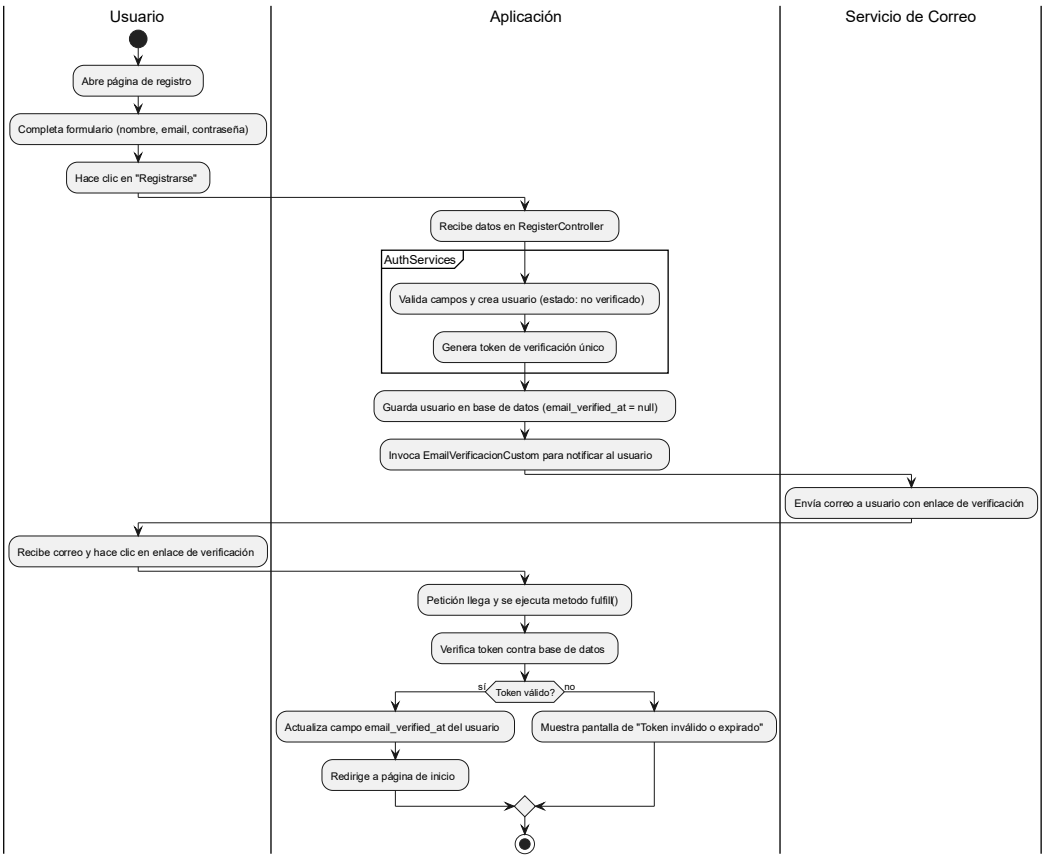


Diagrama de actividad de registrar un nuevo usuario

Crear una lista de la compra

El siguiente diagrama de actividad detalla el proceso de creación de una lista de la compra en RoyList. En la primera columna se describen las acciones del usuario y en la segunda columna se muestra cómo la aplicación responde. Este flujo refleja de manera clara cada paso necesario para que un usuario pueda dar de alta una lista de la compra y verla inmediatamente en su panel.

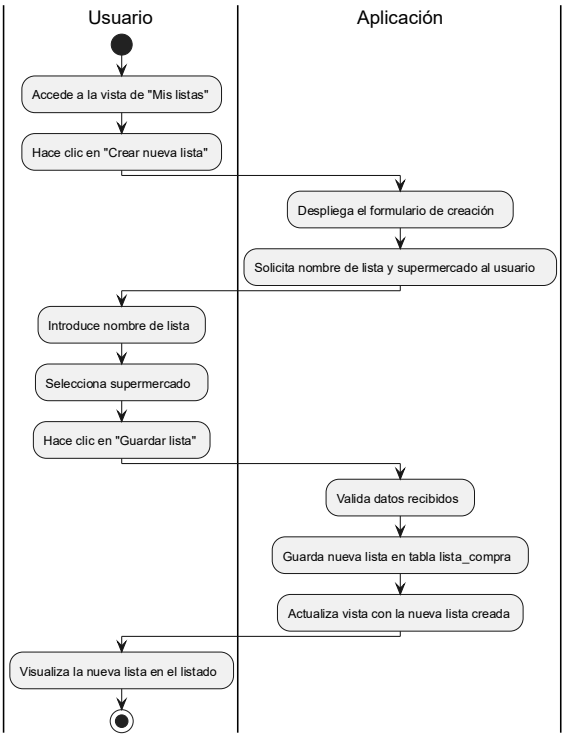


Diagrama de actividad de crear una lista de la compra

Añadir un producto a una lista

A continuación, se presenta el diagrama que describe el proceso de añadir un producto a una lista de la compra en la aplicación. En la columna izquierda se describen las acciones del usuario, la central las acciones de la aplicación y la de la derecha la de la API. Este diagrama muestra cómo interactúan usuario y sistema desde que se decide incluir un producto hasta que éste aparece reflejado en la lista.

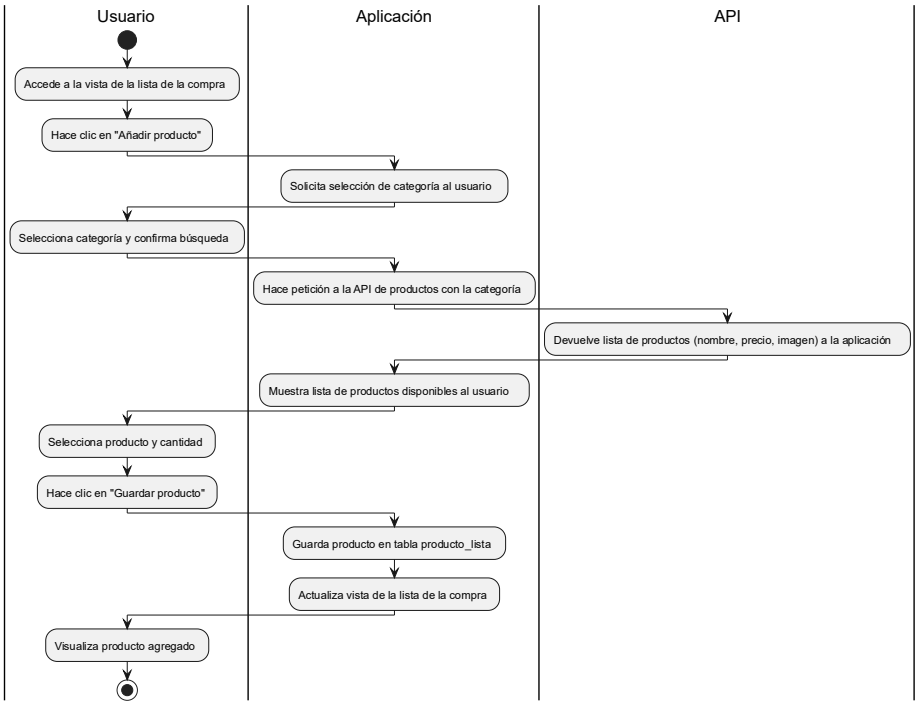
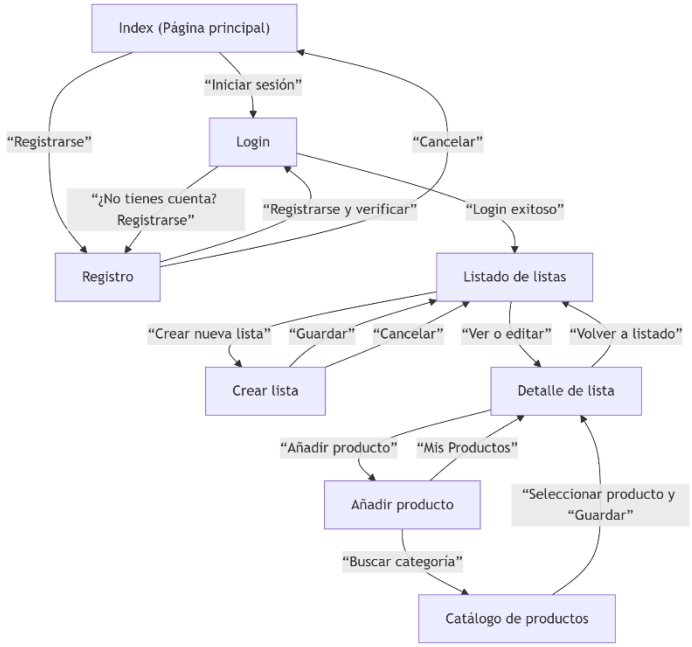


Diagrama de actividad de agregar un producto a una lista

3.3.4. Diagrama de Navegación

El diagrama de navegación muestra de forma esquemática las pantallas principales de la aplicación y las rutas que puedes seguir el usuario. En él aparecen los nodos Index, Login, Registro, Listas, Crear listas, Detalle de lista, Añadir producto y catálogo de productos. Las flechas indican acciones que puede hacer el usuario navegando por la aplicación. De esta manera, se visualiza rápidamente la estructura general.



Diseño de Navegación

4. DESARROLLO

A continuación, se detalla cómo está construido RoyList, explicando cómo se ha organizado el código y que criterios se han seguido para desarrollar la aplicación en Laravel 12. Se ofrece una vista de las piezas que tiene la aplicación y las herramientas empleadas.

4.1. Arquitectura de la aplicación

RoyList se ha desarrollado sobre Laravel 12, siguiendo el patrón MVC (Modelo-Vista-Controlador). La estructura general se divide en 3 capas

1. Capa de presentación (MVC - Vista)

Se ha usado **Blade**, que es un motor de plantillas de Laravel, que permite crear vistas dinámicas y reutilizables en el framework. Para el diseño de la página y el comportamiento 'responsive' se ha decidido utilizar **Tailwind 4**, que es un framework CSS para facilitar los diseños personalizados de forma rápida y eficiente.

Todas las vistas extienden de un único layout: **views/layouts/app.blade.php**

```
1  <!DOCTYPE html>
2  <html lang="es">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="csrf-token" content="{{ csrf_token() }}">
8      <meta http-equiv="X-UA-Compatible" content="ie=edge">
9      <title>@yield('title')</title>
10     <link rel="shortcut icon" href="{{ asset('images/LogoTrans.ico') }}" type="image/x-icon">
11     @vite(['resources/css/app.css', 'resources/js/app.js'])
12 </head>
13
14 <body class="">
15     <x-header />
16
17     <main>
18         @yield('content')
19     </main>
20
21     <x-footer />
22 </body>
23 </html>
```

Código íntegro de app.blade.php

Donde el contenido se ejecutará, como se verá después, en la línea 18 '**@yield('content')**' dentro de la etiqueta **</main>**

Con dos componentes, que son el header y el footer:

- **views/components/header.blade.php**
- **views/components/footer.blade.php**.

En estos archivos simplemente se codificarán con la etiqueta correspondiente **</header>** y **</footer>**, porque como veremos después, se anexarán a las vistas principales.

Las vistas principales como: registro, inicio de sesión, listas, inicio, etc. Estarán ubicadas en la raíz de la carpeta **/resources/views/**. Extenderán de **app.blade.php** y tendrán anexadas el header y el footer. La parte dinámica que cambia en cada una ira dentro de unas etiquetas especiales de Blade. Que es la etiqueta **@section**. Como veremos en el siguiente ejemplo:

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

@section('title', 'Ejemplo - RoyList')

@section('content')

    <section class="bg-white py-16">
        <div class="container mx-auto px-6">
            Aquí irá el contenido dinámico de cada página.
        </div>
    </section>

@endsection
```

Ejemplo de páginas dinámicas con Blade

Donde:

- **@extends** indica el archivo del que, como su nombre indica, extiende
- **@section ('title')** indica que el segundo parámetro que se le pasa será el título de esa página
- **@section ('content')** / **@endsection,** lo que haya entre estas dos etiquetas será el contenido dinámico de la página

2. Capa de controladores y servicios (MVC - Controlador)

Hay 5 controladores principales en el proyecto, dos que se encargan exclusivamente del registro y del login de la aplicación. Y tres que son los que manejan las operaciones "CRUD" y comunican la parte del modelo con la vista.

RegisterController.php / LoginController.php

Ubicados en **app/Http/Controllers/Auth/.** Como he dicho antes, gestionan el registro y el login de los usuarios. También controlan la lógica de cerrár sesión

- **RegisterController.php**

Solamente contiene un método llamado register() que recoge por parámetro la petición del formulario de la vista, la valida y si tiene los requisitos correctamente crea un nuevo usuario en la base de datos. A continuación, empieza la validación de email. Gestionada por otro servicio.

```php
public function register(Request $request)
{
    $request->validate([
        'name' => ['required', 'string', 'max:255'],
        'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],
        'password' => ['required', 'string', 'confirmed', Rules\Password::min(size: 8)->letters()->numbers()->symbols()],
        'terms' => ['accepted'],
    ]);

    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);

    event(new Registered($user));

    Auth::login($user);

    return redirect()->route('verification.notice');
}
```

Método de registro en el controlador

- **LoginController.php**

Simplemente obtiene la petición del formulario de login por parámetro que contiene el email y la contraseña. La valida y si es correcta usa el servicio AuthService (se detallará más adelante) para iniciar sesión y redirigir a la página principal con la sesión iniciada.

```php
public function login(Request $request)
{
    $credentials = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if ($this->authService->attemptLogin($credentials)) {
        return redirect()->route('index');
    } else {
        return back()->withErrors([
            'email' => 'Las credenciales proporcionadas no coinciden.',
        ])->onlyInput('email');
    }
}
```

Método de login en el controlador

Controladores Principales

He de aclarar que los métodos de los controladores NO tienen lógica o tienen la mínima, simplemente se instancia un objeto del modelo y ejecutan el método de éste, que es el que tiene toda la lógica de programación y el que se "habla" con la Base de Datos y demás

Ubicados en **app/Http/Controllers/.** Estos controladores gestionan a las entidades del modelo.

- **ListaCompra_Controller.php**

Se encarga de las operaciones CRUD sobre las listas de la compra. Tiene varios métodos:

- **showListView()** Simplemente devuelve la vista donde están las listas
- **create(Request $request)** Con los datos del formulario para crear una lista
- **getListas()** Devuelve una lista con todas las listas que tiene el usuario
- **contarProductos($id)** Cuenta los productos que tiene una lista
- **mostrarLista($id)** Muestra la vista 'productos' con la información de una sola lista
- **borrarLista($id)** Elimina una lista de compra por su id

- **ProductoLista_Controller.php**

También se encarga de las operaciones CRUD de los productos. Sus métodos:

- **mostrarProductos()** Devuelve todos los productos agrupados por categoría
- **obtenerCategorias()** Obtiene todas las categorías para mostrarlas en el select.
- **obtenerPorCategoria(Request $request)** Obtiene los productos de una sola categoría
- **guardarProducto(Request $request)** Coge los datos de un producto de la API y lo guarda en la base de datos
- **eliminarProducto($id)** Elimina un producto por su id

- **Supermercado_Controller.php**

No tiene mucha lógica, solamente consulta las cadenas de supermercados disponibles:

- **showSupermercados()** Devuelve una lista de supermercados.

Servicios

- **AuthService.php**

Ubicado en **app/Http/Services.** Contiene la lógica que he contado antes que usa el LoginController y RegisterController para validar, iniciar sesión y cerrar sesión a usuarios.

```php
class AuthService {

    public function attemptLogin($credentials){
        return Auth::attempt($credentials);
    }

    public function logout() {
        Auth::logout();
    }
}
```

AuthService.php

- **EmailVerificacionCustom.php**

Ubicado en **app/Notifications.** Se encarga de enviar correos de verificación al usuario tras el registro, usando el sistema de notificaciones de Laravel 12.

- **via()** Indica las vías por las que se enviará la notificación de verificación
- **toMail()** Personaliza el mensaje que se mostrará en la notificación
- **verificationUrl()** Genera el enlace de verificación firmado

3. Capa de modelos (MVC- Modelo)

Para gestionar la persistencia de datos he optado por el uso de **Eloquent**, el ORM (Object-Relational Mapping) que incluye Laravel de forma nativa. Un ORM es una técnica que permite mapear las tablas de una Base de Datos relacional a clases de un lenguaje POO (Orientado a objetos), en este caso PHP. Con Eloquent se puede trabajar con modelos como si fuesen objetos de PHP, sin necesidad de escribir consultas SQL manuales cada vez que necesito acceder a datos de la base de datos.

Cada entidad de la base de datos dispone de su modelo Eloquent correspondiente dentro de **app/Models.**

Modelo: User → Tabla: users

Modelo: ListaCompra → Tabla: lista_compra

Modelo: ProductoLista → Tabla: producto_lista

Modelo Supermercado → Tabla: supermercados

Al definir cada modelo como una clase que extiende de **Illuminate\Database\Eloquent\Model**, Eloquent asocia automáticamente el nombre de la tabla (plural, en mayúsculas) al modelo, aunque este comportamiento se puede personalizar si se usa un nombre distinto con el atributo **$table**, como es mi caso:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class ProductoLista extends Model
{
    protected $table = 'producto_lista';
}
```

Ejemplo modelo ProductoLista.php

4.2. Migraciones y base de datos

Para crear la estructura física de la base de datos en MySQL, se utilizan las migraciones de Laravel. Cada migración es un archivo .php que contiene métodos up() y down(), encargados de crear o eliminar las tablas y sus campos:

```php
return new class extends Migration {

    public function up()
    {
        Schema::create('producto_lista', function (Blueprint $table) {
            $table->id();
            $table->foreignId('lista_compra_id')->constrained()->onDelete('cascade');
            $table->string('nombre_producto');
            $table->integer('cantidad')->default(1);
            $table->text('notas')->nullable();
            $table->decimal('precio', 8, 2)->nullable();
            $table->string('imagen', 500)->nullable();
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('producto_lista');
    }
};
```

Ejemplo migración tabla producto_lista

Cuando se ejecuta el comando **php artisan migrate**, Laravel revisa qué migraciones no se han aplicado aún y ejecuta el método up() de cada una, construyendo así las tablas en MySQL. Si en algún momento se quiere revertir los cambios, **php artisan migrate:rollback** ejecuta el método down().

En los modelos de Eloquent, también se tiene que indicar las relaciones entre entidades. Estas relaciones hacen más fácil la obtención de datos relacionados entre tablas. Por ejemplo, en el modelo **ProductoLista** tiene un método llamado listaDeCompra() que vincula cada producto al registro de lista de compra correspondiente:

Ejemplo de relación entre modelos

De esta manera, con una sola línea como **$usuario→listas()→with('productos') →get();** Eloquent genera las consultas SQL necesarias para unir users, lista_compra y producto_lista, devolviendo un árbol de objetos PHP listo para usar en el controlador.

En resumen, la capa de datos de la aplicación se apoya en MySQL como sistema de gestión relacional y en Eloquent ORM para mapear esas tablas a objetos PHP. Las migraciones garantizan que la estructura de la base de datos sea consistente entre los distintos entornos (local, producción, etc.) y en distintos dispositivos si se requiere. Al definir las relaciones en los modelos, se simplifican las operaciones de lectura y escritura de registros relacionados, manteniendo el código limpio y fácil de mantener.

4.3. API

A continuación, voy a describir el funcionamiento de la API desarrollada con Node.js y Express, la cual expone los datos obtenidos mediante el scraping en forma de archivos JSON. Este apartado explica las tecnologías empleadas, la estructura de la API y el flujo interno que sigue cada petición para devolver la información de productos.

La API se implementa sobre Node.js (versión 22) y el framework Express. Toda la información se sirve a partir de dos fuentes de datos. Primero, existe un fichero principal llamado **producto_mapping.json** que contiene un array con todos los productos, cada uno representado mediante un objeto con los atributos (id, ean, slug, name). En segundo lugar, en la carpeta /products existen tantos ficheros JSON como productos. Cada fichero **{id}.json** incluye los detalles

completos del producto (precio, URL de la imagen, categoría, etc.). Estos archivos se generan diariamente mediante un script en bash que ejecuta un programa en PHP, que dejo el enlace del proyecto aquí, porque no es mío (Repositorio proyecto), que realiza un scraping a la web oficial del Mercadona y vuelca la información en el directorio correspondiente.

Express levanta un endpoint (http://localhost:3000/) en el que se exponen dos rutas principales a través de este:

1. **GET** /api/data

Cuando un cliente realiza una petición GET a /api/data, el servidor comprueba que exista el archivo **producto_mapping.json**. Si no se encuentra, responde con un código 404 y un objeto JSON { error: "Archivo no encontrado" }. Si todo va bien y encuentra el archivo se lee de forma síncrona desde disco (fs.readFileSync) y se procesa el objeto a JavaScript. Finalmente, se envía el contenido completo como respuesta JSON. De esta manera cualquier consumidor puede obtener la lista de todos los productos con sus atributos básicos.

2. **GET** /api/productos/:id

Esta ruta recibe como parámetro de URL el identificador del producto. El servidor construye la ruta al fichero correspondiente y verifica su existencia. Si no se haya el fichero responde con el mismo error 404 que la otra ruta. Si el archivo si se encuentra, se lee con **fs.readFileSync**, se procesa y se envía como objeto JSON.

En resumen, la API basada en ficheros JSON, permite exponer en el endpoint localhost:3000 los datos. Las dos rutas cubren los dos casos de uso principales.

4.4. Pruebas

Las pruebas las he diseñado en base a lo aprendido en clase, una tabla de 3 columnas en la que la primera columna es:

- **Escenario:** Aquí se describe el entorne que tiene este caso de prueba. Describiendo el archivo/ruta en el que estamos y los datos de la base de datos o de la API que necesitamos conocer para verificar que la salida esté bien.
- **Entrada:** La acción que realiza el usuario en el escenario en el que estamos, ya sea hacer clic en algún botón o escoger una categoría en el selector.
- **Resultado esperado:** Se define el resultado que debería dar en base a los datos de la primera columna.

Ya que las pruebas generadas son muy grandes, las he hecho en un archivo aparte. Por lo que voy a dejar el archivo anexado aquí y en el apartado de ANEXOS

PRUEBAS

5. CONCLUSIONES

5.1. Futuras líneas de investigación

Como he ido contando a lo largo del proyecto, hay muchas funcionalidades que se pueden, y quiero, seguir desarrollando. Una de las principales es, obviamente, el expandir la aplicación a todos los supermercados posibles, sean del tamaño que sean, locales, o internacionales. En relación con eso, pretendo añadir la funcionalidad de comparación de precios entre productos parecidos de la misma categoría para poder ahorrar lo máximo en la lista. Así como la notificación cuando un producto baje de X precio, siendo X el precio que estime el usuario. Algo parecido a lo que es ahora mismo la aplicación “Idealo”.

En cuanto a dispositivos, también tengo pensado hacer soporte para iOS y Android. Así lograría llegar a un número de usuarios más amplio. Aunque esta idea es un poco más lejana que otras, ya que habría dos formas realistas de hacerlo. Una es hacer una PWA (Aplicación Web Progresiva), que es la aplicación ejecutándose como si estuviese en un navegador, pero comportándose como si fuese una app móvil. Esto haría que no se tenga acceso a funcionalidades nativas de los móviles como la cámara, NFC, sensores. Aunque sea más fácil la portabilidad, esto limitaría posibles funcionalidades que se quieran introducir en un futuro. Otra forma de hacer soporte de móviles es usar un contenedor tipo Capacitor. Esto requeriría portear todo el frontend a un framework como React o Vue. Aunque es sería más costoso en tiempo y esfuerzo, creo que el esfuerzo vale mucho la pena por los beneficios que trae. Aunque también tiene inconvenientes. Como la complejidad de compilar los dos proyectos más por separado, como son en Andorid (Java) e iOS (Swift). Para este últimos se necesita si o si la aplicación Xcode, la cual solo está disponible para dispositivos Apple como Macs o MacBooks.

Una de las funcionalidades que quiero implementar más a corto plazo son las listas compartidas entre usuarios. Hasta ahora cada usuario tenía un número de listas personalizado. Y cada lista iba ligada a un usuario. Para tener la misma lista en el dispositivo había que estar con la sesión iniciada con el mismo usuario. Pero ¿y si se quieren tener distintos usuarios dentro de la familia para más privacidad? Usando WebSockets se podría, además esto haría que el rendimiento de la aplicación sea mucho mejor. Se podrían compartir entre usuarios listas específicas que se sincronizarían en tiempo real y no se tendría que hacer una petición al servidor cada vez que se añadan productos o se borren.

Por último, y esto es utópico, la idea de monetizar la aplicación. Como he dicho en el apartado 2.7, habría varias formas de monetizar la aplicación en caso de éxito.

5.2. Problemas encontrados

Uno de los principales problemas que me he encontrado en el desarrollo del proyecto ha sido el de la eficiencia al hacer las peticiones a la API. Me di cuenta de que recuperar los datos de la API era un auténtico cuello de botella, porque cada vez que alguien pedía la lista completa de productos, el servidor se ponía a leer y parsear el archivo producto_mapping.json. Como todo funcionaba en lectura síncrona, y cuando llegaban varias peticiones al mismo tiempo, todo se ralentizaba. Al final la aplicación tardaba en responder y la experiencia de usuario se resentía.

Además, cada petición implicaba volver a leer el mismo fichero una y otra vez, lo que desperdiciaba recursos. Para mejorar el rendimiento, hice que el servidor guardase el resultado en caché durante una hora, así no tenia que volver a leer los ficheros en cada petición. Además, implementé un selector que únicamente carga los productos de la categoría solicitada, en lugar de toda la lista completa. Con estas dos modificaciones, reduje significativamente el tiempo de respuesta.

5.3. Opinión personal

En lo personal, he aprendido bastante, porque he hecho cosas que no había hecho antes. Como trabajar con una API, recoger datos, mostrarlos y guardarlos en base de datos. No había hecho, ni conocía en profundidad el framework Express de Node.js para hacer APIs. En cuanto a las cosas que ya había hecho en alguna ocasión o no conocía tan bien, este proyecto me ha ayudado a afianzarme con los conceptos o tecnologías como MVC, Eloquent o MySQL entre otras.

Y siendo honestos, creo que este proyecto cumple los requisitos mínimos, o poco por encima de los mínimos del que se espera de un proyecto así en este curso. Creo sinceramente que lo podría haber hecho bastante mejor, pero como ya he dicho, tengo pensado seguir con este proyecto y tomármelo como algo personal y profesional.

6. BIBLIOGRAFÍA

Referencias de código:

- *Laravel 12 (02/2025). Documentación oficial:* https://laravel.com/docs/12.x
- *Tailwind labs (01/2025):* Tailwind
- *Laravel 12 (02/2025). Eloquent ORM:* https://laravel.com/docs/12.x/eloquent
- *Laravel 12 (02/2025). Autenticación:* https://laravel.com/docs/12.x/authentication
- *Laravel 12 (02/2025). Verificación Email:* https://laravel.com/docs/12.x/verification
- *Tailwind Components. Componentes UI reutilizables:* https://www.creative-tim.com/twcomponents
- *Fetch API (MDN). Fetch API - Consumo de APIs en JavaScript:* https://developer.mozilla.org/es/docs/Web/API/Fetch_API

- *JavaScript DOM (MDN). Manipulación del DOM en JavaScript:* https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model

Diagramas

- *Mermaid*. *Diagramas en Markdown*. https://mermaid.js.org/
- *PlantUML. Herramienta de modelado UML.* https://plantuml.com/es/
- *ERDPlus. Entidad-Relación*. https://erdplus.com/

Otros

- *GitHub (josantonius, 20/04/2024). Repositorio del proyecto de scraping Mercadona*: https://github.com/josantonius/php-mercadona-importer
- *Mailtrap*. *Servicio/Emulación de correo:* https://mailtrap.io/

7. ANEXOS

- Manual de Administrador
- Manual de Usuario
- Pruebas

Repositorio del Proyecto:

- Proyecto principal: https://github.com/rodriiii94/RoyList
- API: https://github.com/rodriiii94/RoyList-Api