HopInk Studio

Alejandra López Hidalgo

2025

Índice

1. Introducción y presentación del proyecto

1.1 Nombre del proyecto

Hopink Studio

1.2 Descripción general

Hopink Studio es una plataforma web dedicada a la venta de productos especializados para tatuajes y piercings, así como a la gestión de presupuestos personalizados para servicios en estudios de tatuaje. Este proyecto nace como una solución integral que permita tanto a profesionales del arte corporal como a clientes interesados, acceder fácilmente a un catálogo de productos, gestionar sus compras, enviar solicitudes de servicios, y mantenerse conectados con el estudio.

El sistema ha sido desarrollado utilizando una arquitectura moderna compuesta por Angular en el frontend, PHP como lenguaje del backend, y MySQL como sistema de gestión de base de datos. Esta combinación garantiza una aplicación ágil, escalable y segura, con una separación clara entre cliente y servidor.

Hopink Studio representa una solución profesional y adaptable para pequeños estudios que desean contar con una tienda online personalizada, sin depender de plataformas externas ni realizar grandes inversiones en desarrollo propietario.

1.3 Objetivos del proyecto

El objetivo general es crear un sistema web completo que permita gestionar productos, usuarios, presupuestos y servicios relacionados con tatuajes y piercings, en un entorno atractivo, rápido y funcional.

Entre los objetivos específicos encontramos diseñar una tienda web responsive que se adapte a móviles, tablets y ordenadores.

Permitir el registro y autenticación segura de usuarios.
Implementar funcionalidades como carrito de compras, wishlist y presupuestos personalizados.

Gestionar los datos mediante un backend sencillo y eficaz, conectado con una base de datos estructurada en MySQL.

Posibilitar la ampliación futura con panel de administración, pasarela de pagos, notificaciones automáticas, y más.

1.4 Público objetivo

Este proyecto está dirigido a tres tipos principales de usuarios:

Clientes finales: personas que desean adquirir productos de tatuaje/piercing o solicitar un presupuesto para servicios personalizados.

Tatuadores y artistas corporales: profesionales que buscan un canal de venta directo para sus productos y servicios.

Administradores del estudio: encargados de gestionar el contenido de la tienda, los presupuestos recibidos y el estado de los pedidos.

La plataforma ha sido diseñada con un enfoque inclusivo y moderno, buscando facilitar tanto la experiencia de compra como la gestión del negocio.

1.5 Justificación y necesidad del proyecto

La industria del tatuaje y el piercing ha experimentado un notable crecimiento, tanto en profesionalización como en demanda. Sin embargo, muchos estudios siguen careciendo de soluciones digitales personalizadas que les permitan vender productos o gestionar citas de forma eficiente.

Hopink Studio responde a esa necesidad, ofreciendo una solución que mejora la visibilidad y acceso online del estudio, automatiza procesos como presupuestos o gestión de stock., aumenta el nivel de profesionalización digital de estudios pequeños o medianos y, además, su desarrollo modular permite evolucionar el sistema con nuevas funciones sin necesidad de rehacer la plataforma desde cero.

1.6 Tecnologías utilizadas

Tecnología	Función
Angular 19	Desarrollo del frontend y componentes interactivos
PHP 8+	Lógica de negocio en el backend y conexión a base de datos
MySQL	Almacenamiento estructurado de datos (usuarios, productos, presupuestos, etc.)
XAMPP	Entorno local para desarrollo con Apache y MySQL
HTML/CSS/TypeScript	Estructura, estilo y comportamiento del sitio
phpMyAdmin	Administración gráfica de la base de datos

2. Diseño Físico y Lógico de la Red

2.1 Entorno de Desarrollo

Durante la fase de desarrollo del proyecto Hopink Studio, se ha utilizado un entorno de red local basado en el software XAMPP, que incluye el servidor web Apache y el sistema gestor de bases de datos MySQL. Esta solución permite ejecutar el backend PHP y gestionar la base de datos desde el mismo equipo, simulando un entorno de producción básico para pruebas y depuración.

Componentes del entorno local:

- Servidor web local: Apache (incluido en XAMPP)
- Base de datos local: MySQL (gestionada con phpMyAdmin)
- Cliente frontend: Navegador web (Chrome, Firefox, Edge)
- IDE: Visual Studio Code con extensiones de Angular y PHP
- Conexiones: Peticiones HTTP desde Angular hacia los scripts PHP del backend, alojados en la carpeta htdocs del servidor local.
- Diagrama lógico del entorno de desarrollo:

2.2 Infraestructura de Red Propuesta para Producción

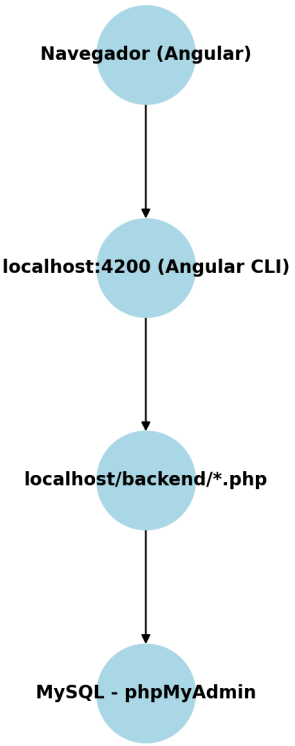
Actualmente, el proyecto Hopink Studio se ejecuta exclusivamente en entorno local, utilizando el servidor Apache y la base de datos MySQL integrados en XAMPP. Todo el desarrollo y pruebas se han llevado a cabo en este entorno, lo que ha permitido validar su funcionalidad sin necesidad de una infraestructura en la nube o servidor dedicado.

No obstante, con vistas a una posible fase de publicación y explotación real del proyecto, se plantea una infraestructura básica de red que permitiría el despliegue de la aplicación en producción. Este sería el siguiente paso natural tras finalizar el desarrollo y las pruebas locales.

Infraestructura propuesta para publicar Hopink Studio:

- Servidor Web (VPS o alojamiento compartido):
- Compatible con PHP 8+ y MySQL.
- Soporte para Angular (entregando archivos dist/ como contenido estático).
- Dominio propio (ej. hopinkstudio.com) apuntando al servidor.
- Subida del frontend Angular compilado (usando ng build --prod) al directorio /public_html o similar.
- Subida de scripts PHP y configuración de rutas de backend al servidor.
- Base de datos remota MySQL, accesible solo desde el servidor web.
- Certificado SSL (HTTPS) para asegurar las comunicaciones.

Diagrama lógico del entorno de desarrollo local

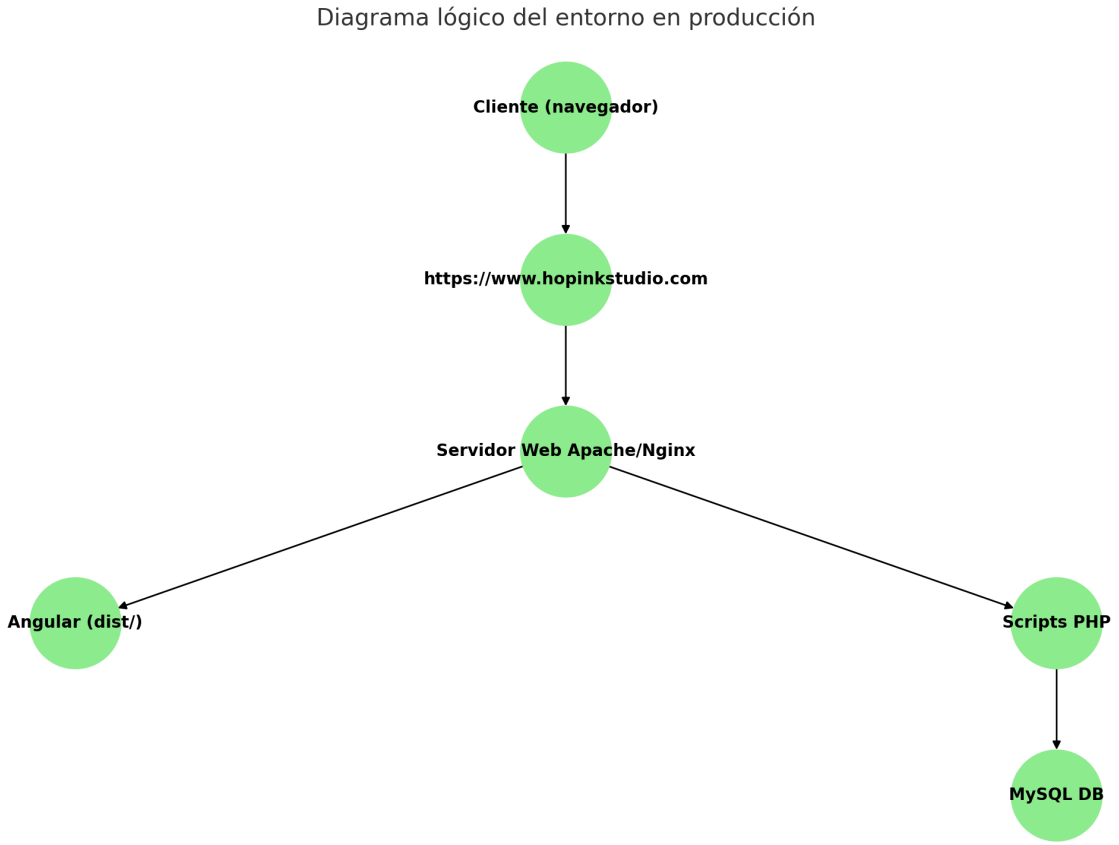


- Acceso FTP o SFTP para subir archivos y realizar mantenimiento.
- Copias de seguridad automáticas de archivos y base de datos.

Pasos para la publicación:

- Contratar un hosting compatible con PHP y MySQL (ej. Hostinger, OVH, IONOS, etc.).
- Registrar un dominio y configurarlo en el panel del proveedor.
- Subir el frontend Angular compilado al hosting.
- Subir el backend PHP a un subdirectorio protegido (/backend, por ejemplo).
- Importar la base de datos MySQL desde el archivo .sql generado en el entorno local.
- Configurar los scripts PHP para que apunten a la base de datos remota.
- Probar todo el sistema online asegurando que Angular, PHP y MySQL funcionen correctamente.

Éste sería el diagrama lógico en producción:



2.3 Ampliación de la Red

Para adaptar el sistema a una red real de producción y facilitar la administración, el proyecto contempla incluir:

- Un servidor dedicado o VPS que albergue tanto la web como la base de datos.
- Acceso FTP/SFTP para subir cambios fácilmente desde el equipo de desarrollo.
- Copias de seguridad automáticas de la base de datos y archivos PHP/Angular.
- Herramientas de monitoreo como UptimeRobot o Grafana para asegurar disponibilidad.
- Política de actualizaciones periódicas de PHP, Angular, Node.js y librerías utilizadas.

2.4 Seguridad Básica en Red

- Cifrado de contraseñas con password_hash() en PHP.
- Validación del lado del servidor para evitar inyecciones SQL y accesos indebidos.

- Protección de rutas restringidas mediante control de sesión ($_SESSION).
- Separación clara entre lógica de cliente y servidor, evitando exposición directa de la base de datos.

3. Requisitos Hardware y Software

En esta sección se describen los requisitos necesarios para ejecutar correctamente el proyecto Hopink Studio, tanto en entornos de desarrollo como en una posible implementación en producción. Se detallan los sistemas operativos compatibles, versiones de software utilizadas y especificaciones mínimas de hardware recomendadas.

3.1 Requisitos de Software

- Sistema Operativo: El proyecto es multiplataforma y puede desarrollarse o ejecutarse correctamente en los siguientes sistemas:
    - Windows 10/11
    - Linux (Ubuntu, Debian o derivados)
    - macOS (con Homebrew para gestión de paquetes)
- Entorno de Desarrollo Local: Para trabajar en local se han utilizado los siguientes software:
    - XAMPP versión 8.2.0 o superior como servidor Apache + MySQL local
    - PHP versión 8.0+ como backend y lógica del servidor
    - MySQL versión 5.7+ o MariaDB para almacenamiento de datos
    - phpMyAdmin versión 5.2+ para la gestión visual de la base de datos
    - Node.js versión 18+ , que es requerido por Angular CLI
    - Angular CLI versión 19 para el desarrollo del frontend
    - Visual Studio Code última versión como editor de código fuente
    - Navegador moderno (Chrome, Firefox, Edge) para pruebas y visualización
- Dependencias del Proyecto Angular
    - Angular Core, Forms, Router, HttpClient
    - Bootstrap o Tailwind (según estilos usados)
    - Otros paquetes opcionales como ngx-toastr o FontAwesome

- Navegadores Compatibles:
    - Google Chrome (última versión)
    - Mozilla Firefox
    - Microsoft Edge
    - Safari (en versión moderna)

3.2 Requisitos de Hardware

Para ejecutar el sistema correctamente en desarrollo o entorno básico de pruebas, se recomienda el siguiente hardware mínimo:

- Procesador 2 GHz Dual-Core
- Memoria RAM 4 GB
- Espacio en disco 2 GB libres (mínimo para proyecto, XAMPP y dependencias)
- Resolución de pantalla 1280x720 o superior
- Conexión a Internet para dependencias y actualizaciones

Requisitos recomendados (desarrollo cómodo y fluido):

- Procesador 2.5+ GHz Quad-Core
- Memoria RAM 8 GB o más
- Disco SSD 10 GB disponibles
- Resolución Full HD o superior
- Conectividad Internet estable para trabajo colaborativo y documentación

3.3 Requisitos para Producción (Opcionales)

Aunque el proyecto aún no se ha desplegado en producción, para un futuro despliegue se recomienda:

- Servidor Web VPS con Apache o Nginx, soporte PHP y MySQL
- RAM del servidor de 2 GB mínimo

- Espacio en disco de 5 GB libres mínimo
- Sistema operativo del servidor Ubuntu Server 20.04 o superior
- Certificado SSL Let's Encrypt o comercial
- Panel de control cPanel o Plesk para facilitar la gestión

4. Análisis y Diseño de la Aplicación

4.1 Análisis de Requisitos

El sistema Hopink Studio ha sido concebido como una tienda web dinámica y moderna, capaz de cubrir tanto la venta de productos como la solicitud de servicios personalizados. Para lograrlo, se ha realizado un análisis detallado de los requisitos funcionales y no funcionales del sistema. Además, se incluyen fragmentos reales del código desarrollado, que representan la lógica principal del sistema y su implementación técnica.

4.1.1 Requisitos Funcionales

Los requisitos funcionales definen las acciones específicas que los usuarios pueden realizar dentro de la aplicación. En Hopink Studio, se han identificado los siguientes:

- Gestión de usuarios
    - Registro de nuevos usuarios, con validación de campos requeridos (email, contraseña…).
    - Inicio de sesión seguro mediante el envío de credenciales al backend PHP.
    - Sesión persistente gracias al uso de almacenamiento local (localStorage) en Angular.
    - Cierre manual de sesión y punto verde en el header que indica sesión activa.
- Gestión de productos
    - Catálogo de productos de tatuajes y piercings cargado desde base de datos.
    - Visualización en tarjetas: nombre, precio, descripción e imagen.
    - Posibilidad de añadir productos a: carrito de compra y wishlist (favoritos)
- Solicitud de presupuestos personalizados
    - Formulario dividido por tipo de servicio (tatuaje/piercing).
    - Selección de tamaño, zona del cuerpo, artista y comentario adicional.

    - Cálculo automático del precio según lógica definida (precio por hora, suplemento por zona).

    - Envío de la solicitud al backend y almacenamiento en base de datos.

- Panel de usuario
    - Visualización del historial de presupuestos enviados.

    - Acceso directo a productos guardados y carrito activo.

    - Posibilidad de cerrar sesión y volver a registrarse.

4.1.2 Requisitos no funcionales

Estos requisitos no implican funciones directas, pero afectan a la calidad, usabilidad y robustez del sistema:

- Usabilidad
    - Interfaz clara, con navegación basada en componentes Angular y menús accesibles.

    - Diseño responsive, adaptado a móviles, tablets y escritorio.

    - Textos, botones y formularios intuitivos.

- Rendimiento
    - Carga inicial del sitio en menos de 3 segundos.

    - Respuestas del backend rápidas (< 2 segundos en la mayoría de solicitudes).

- Seguridad
    - Encriptación de contraseñas usando password_hash() en PHP.

    - Validación del lado del cliente (Angular) y del servidor (PHP).

    - Uso de $_SESSION para proteger rutas sensibles.

- Escalabilidad
    - Backend PHP preparado para dividirse por funcionalidades.

    - Frontend dividido en componentes reutilizables (Angular Standalone Components).

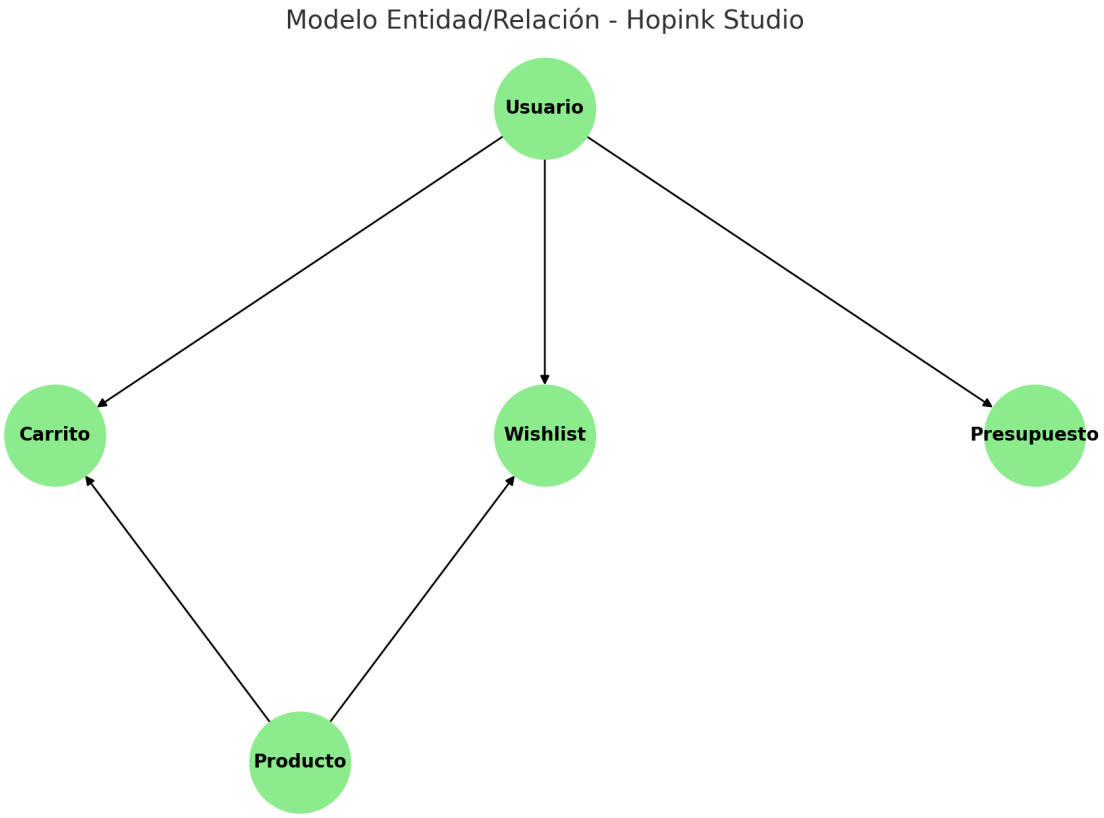
    - Posible integración futura con Stripe para pagos reales.

- Mantenibilidad
    - Código limpio, con comentarios y nombres descriptivos.

- Conexión con base de datos desacoplada en archivo de configuración (config.php).
- Uso de servicios (services) en Angular para agrupar llamadas HTTP.

4.2 Diseño del modelo de datos

4.2.1 Modelo Entidad/Relación



El modelo entidad-relación define las estructuras de datos principales del sistema y las relaciones entre ellas. En el caso de Hopink Studio, estas estructuras están orientadas a gestionar usuarios, productos, presupuestos personalizados, listas de deseos y carritos de compra.

Entre las entidades más importantes del sistema se encuentran:

- **Tabla: users**
  - Representa a los clientes que utilizan la plataforma.

  - Puede:
    - Añadir productos al carrito.
    - Guardar productos en la wishlist.
    - Enviar solicitudes de presupuesto.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**id**	int(11)			No	*Ninguna*		AUTO_INCREMENT
2	**name**	varchar(100)	utf8mb4_general_ci		No	*Ninguna*		
3	**email**	varchar(255)	utf8mb4_general_ci		No	*Ninguna*		
4	**password**	varchar(255)	utf8mb4_general_ci		No	*Ninguna*		
5	**token**	char(32)	utf8mb4_general_ci		No	*Ninguna*		
6	**status**	tinyint(1)			No	0		

- **Table: products**
  - Incluye los artículos disponibles en la tienda online.

  - Contiene: Nombre, descripción, precio, categoría e imagen.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**id**	int(11)			No	*Ninguna*		AUTO_INCREMENT
2	**name**	varchar(255)	utf8mb4_general_ci		No	*Ninguna*		
3	**description**	text	utf8mb4_general_ci		No	*Ninguna*		
4	**price**	decimal(10,2)			No	*Ninguna*		
5	**category**	varchar(50)	utf8mb4_general_ci		No	*Ninguna*		
6	**imageUrl**	varchar(255)	utf8mb4_general_ci		No	*Ninguna*		

- **Tabla: wishlist**
  - Es una lista de productos favoritos que un usuario guarda para revisar o comprar más adelante.

  - Se relaciona directamente con: users (quién lo guarda) y products (qué producto guarda)

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
☐	1	**id**	int(11)			No	*Ninguna*		AUTO_INCREMENT
☐	2	**user_id**	int(11)			No	*Ninguna*		
☐	3	**product_id**	int(11)			No	*Ninguna*		

- **Tabla: budgets**
  - Es un presupuesto único creado por un usuario o invitado.
  - Contiene el ID del presupuesto, el ID del usuario, y los datos necesarios para la creación del presupuesto.

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
☐	1	**id**	int(11)			No	*Ninguna*		AUTO_INCREMENT
☐	2	**user_id**	int(11)			No	*Ninguna*		
☐	3	**name**	varchar(100)	utf8mb4_general_ci		Sí	*NULL*		
☐	4	**email**	varchar(100)	utf8mb4_general_ci		Sí	*NULL*		
☐	5	**artist**	varchar(50)	utf8mb4_general_ci		Sí	*NULL*		
☐	6	**service**	varchar(20)	utf8mb4_general_ci		Sí	*NULL*		
☐	7	**height**	float			Sí	*NULL*		
☐	8	**width**	float			Sí	*NULL*		
☐	9	**message**	text	utf8mb4_general_ci		Sí	*NULL*		
☐	10	**budget**	float			Sí	*NULL*		
☐	11	**created_at**	datetime			Sí	*NULL*		

- **Tabla: cart**
  - Es un carrito único creado por un usuario o invitado.
  - Contiene el ID de usuario (si está logueado), estado del carrito (abierto, enviado, cancelado), etc.

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
☐	1	**id**	int(10)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT
☐	2	**user_id**	int(11)			Sí	*NULL*		
☐	3	**status**	enum('open', 'ordered', 'cancelled')	utf8_general_ci		No	open		
☐	4	**quantity**	int(11)			No	1		

- **Tabla: cart_items**
    - Son los productos añadidos a ese carrito.

    - Contiene el ID del producto, cantidad, y relación con el carrito (cart_id).

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
☐	1	**id**	int(10)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT
☐	2	**cart_id**	int(10)		UNSIGNED	No	*Ninguna*		
☐	3	**product_id**	int(10)		UNSIGNED	No	*Ninguna*		
☐	4	**quantity**	int(11)			No	1		
☐	5	**price**	decimal(10,2)			No	*Ninguna*		

Encontramos dos tablas en relación con el carritro dado que una sirve para abrir un carrito, ya sea de un usuario o de un invitado (en este caso la tabla cart) , y otra tabla para los items que lleva el carrito. A la tabla cart se añade un registro al abrir el carrito, y a la tabla cart_item se le añadirían tantos registros como productos.

La separación consta de ventaja:

- Eficiencia: puedes consultar los ítems de un carrito sin cargar toda la información del usuario.

- Escalabilidad: puedes añadir más productos sin cambiar la estructura de cart.

- Normalización: no se duplican datos como nombre o precio del producto.

- Control de estado: la tabla cart puede indicar si el carrito está en proceso, ya fue ordenado, o fue cancelado.

Relación visual entre tablas

- Un usuario puede tener muchos carritos (aunque normalmente uno está activo).

- Un carrito puede tener muchos productos, cada uno con su propia cantidad.

- Cada ítem del carrito pertenece a un solo carrito y está vinculado a un solo producto.

4.2.2 Análisis de las Formas Normales (hasta 3FN)

La normalización es un proceso que tiene como objetivo eliminar redundancias y dependencias innecesarias en una base de datos. A continuación se analiza cómo las tablas de Hopink Studio cumplen con las tres primeras formas normales (1FN, 2FN y 3FN), que son las más relevantes para un diseño relacional correcto.

- Primera Forma Normal (1FN)

Una tabla está en 1FN si todos los atributos contienen valores atómicos (no repetidos ni multivaluados) y cada registro es único. En Hopink Studio, todas las tablas cumplen esta condición:

En products, cada campo (name, description, price, etc.) contiene valores individuales, no listas ni conjuntos.

En cart_items, los campos como product_id o quantity son atómicos.
No se usan arrays ni múltiples valores en un solo campo.

- Segunda Forma Normal (2FN)

Una tabla está en 2FN si está en 1FN y todos los campos no clave dependen completamente de la clave primaria (no de una parte de ella). En Hopink Studio:

  - En cart_items, la clave principal compuesta sería (cart_id, product_id) si se quisiera evitar duplicados.
  - Aun así, el campo quantity depende de ambos: es la cantidad de ese producto en ese carrito.
  - En wishlist, tanto user_id como product_id definen la entrada, y no hay campos que dependan solo de uno.
  - Todas las dependencias son completas, no parciales. Cumple con 2FN.

- Tercera Forma Normal (3FN)

Una tabla está en 3FN si está en 2FN y no contiene dependencias transitivas (es decir, los campos dependen únicamente de la clave primaria, no de otros campos no clave).En Hopink Studio

- En users, por ejemplo, email, password, token, status → todos dependen directamente del id, no de entre sí.

- En budgets, campos como name, email, message dependen solo de id, aunque se podrían optimizar dividiendo artista en una tabla aparte si se quisiera más adelante.

- No hay dependencias transitivas importantes. Cumple con 3FN.

4.2.3 Representación de la Base de Datos en el SGBD (MySQL)

Una vez definido y normalizado el modelo lógico de datos, se procedió a implementar la base de datos en un Sistema Gestor de Bases de Datos (SGBD) real. En el caso del proyecto Hopink Studio, se ha utilizado MySQL, gestionado mediante la interfaz web phpMyAdmin, incluida en el entorno de desarrollo local XAMPP.

A continuación, se describe cómo se ha trasladado el diseño conceptual a la realidad del sistema.

- Elección de MySQL como SGBD: he elegido MySql porque
  - Es uno de los SGBD relacionales más utilizados en el mundo, especialmente en desarrollo web.

  - Totalmente compatible con PHP.

  - Fácilmente gestionable con herramientas gráficas como phpMyAdmin.

  - Permite exportar/importar fácilmente la estructura o datos para futuras migraciones o respaldos.

  - Alto rendimiento en aplicaciones de tamaño medio como la que plantea este proyecto.

- Proceso de implementación: se realizó en los siguientes pasos:
  - Creación de la base de datos hopink_db desde phpMyAdmin.

  - Creación de tablas utilizando sentencias SQL o el constructor visual.

  - Definición de claves primarias, tipos de datos y restricciones (ej. NOT NULL, AUTO_INCREMENT).

  - Establecimiento de relaciones lógicas mediante claves foráneas cuando ha sido necesario.

    - Pruebas de integridad insertando registros de ejemplo y relacionando tablas.

- Representación visual en phpMyAdmin

Se han capturado y documentado las estructuras de cada tabla directamente desde la interfaz de phpMyAdmin (ver imágenes anteriores). Esto permite visualizar de forma clara los campos, tipos, relaciones, y restricciones aplicadas, así como las claves primarias y foráneas.

Gracias al uso de phpMyAdmin y MySQL, se ha logrado una implementación fiel al modelo relacional previamente diseñado. La base de datos es funcional y coherente, escalable y segura y está preparada para entornos reales de producción

Esta representación práctica cierra el proceso de diseño y validación de datos, dejando lista la estructura para integrarse con el backend PHP y el frontend Angular de manera eficiente.

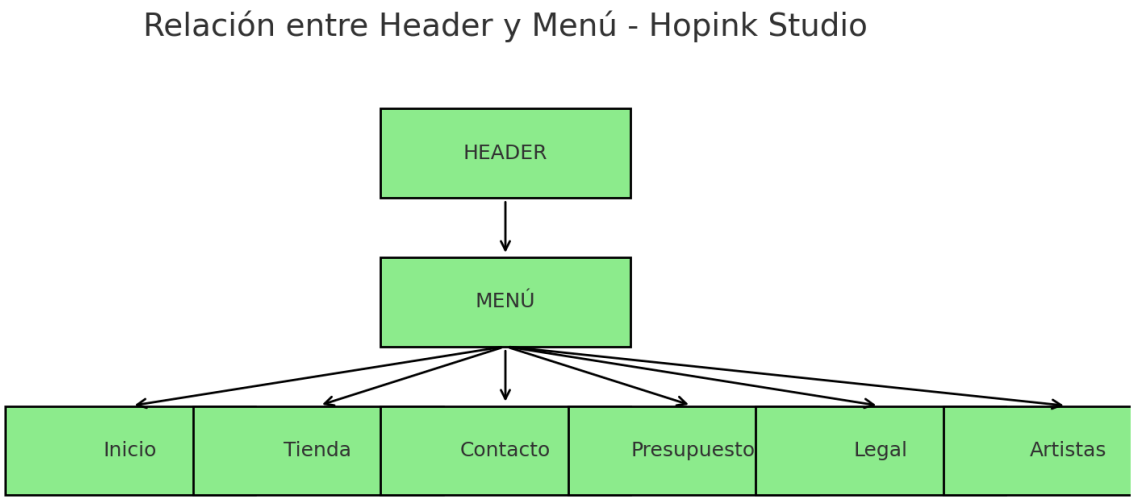
4.3 Diseño del Sitio Web

El diseño del sitio web de Hopink Studio tiene como propósito ofrecer una experiencia de usuario sencilla, visualmente atractiva y completamente funcional. Basado en el framework Angular 19, el sitio está compuesto por rutas bien definidas, componentes modulares y servicios que se comunican con el backend en PHP, lo que permite una arquitectura clara, escalable y fácil de mantener.

La estructura sigue un enfoque SPA (Single Page Application): el usuario navega entre secciones sin recargar la página, obteniendo un comportamiento más fluido y profesional.

4.3.1 Mapa del Sitio Web

El mapa del sitio representa la jerarquía de navegación y rutas del proyecto:



4.3.2 Estructura Modular con Angular

La estructura del proyecto en Angular se divide en:

- Componentes reutilizables (/components/)

Estos componentes están pensados para ser visibles en múltiples páginas. Se inyectan directamente en app.component.html.

- ***header***: Contiene el menú principal, el logo, acceso a cuenta y carrito. Se adapta al tamaño de pantalla.
- ***footer:*** Información adicional, enlaces legales, redes sociales, etc.
- ***menu:*** Menú hamburguesa o desplegable con navegación por secciones.
- ***cookies:*** Aviso legal de uso de cookies (cumplimiento RGPD).
- ***legal:*** Contiene subcomponentes: aviso legal, privacidad, cookies, condiciones.

Estos componentes no dependen de rutas, sino que están presentes en toda la app.

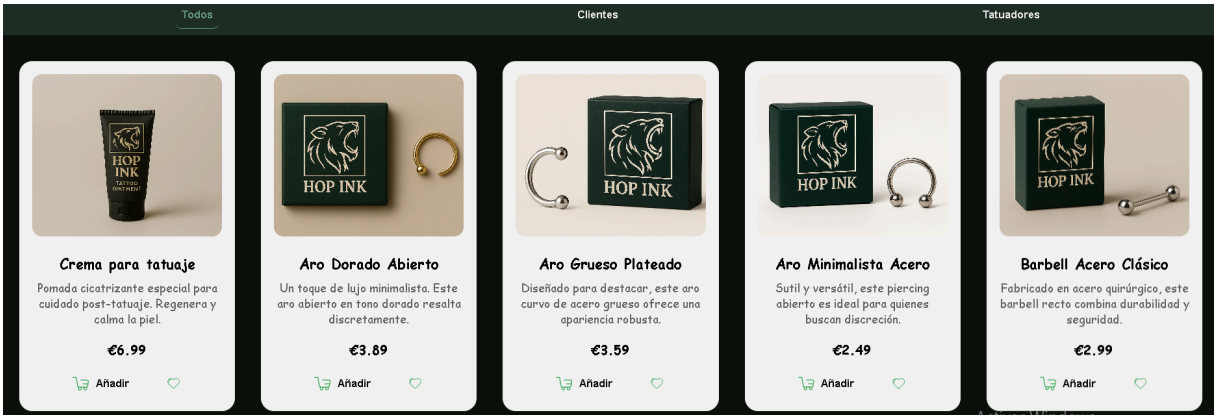
- Páginas funcionales (/pages/)

Cada página es accesible mediante una ruta específica. A continuación, se explican las principales páginas del sitio:

- /home – Página de bienvenida
  - Es el primer contacto del usuario con la aplicación.
  - Presenta el estudio, su estilo visual y enlaces hacia otras secciones.

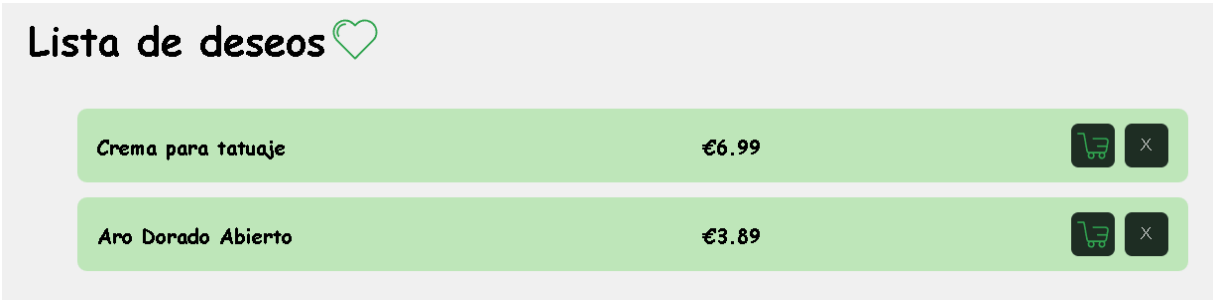
- Ideal para destacar promociones o productos destacados.

- /shop-page – Catálogo de productos
  - Muestra los productos disponibles en la tienda.
  - Los datos se cargan dinámicamente desde el backend (get_products.php).
  - Cada producto incluye nombre, imagen, descripción y precio.
  - Permite añadir productos al carrito o a la wishlist mediante botones.
  - El diseño es adaptable y modular: los productos se muestran en tarjetas (card).



- /cart – Carrito de compras
  - Lista de productos que el usuario ha decidido comprar.
  - Muestra la cantidad seleccionada de cada uno y el precio total.
  - Posibilidad de modificar cantidades o eliminar productos.
  - Enlace a pasarela de pago o simulación (Stripe o PayPal, en desarrollo).

- /account/wishlist – Lista de deseos
  - Productos marcados como favoritos por el usuario.
  - No tienen valor comercial inmediato, pero mejoran la experiencia.
  - Se guarda por usuario (requiere sesión iniciada).

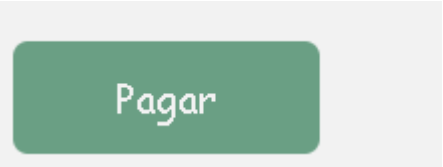


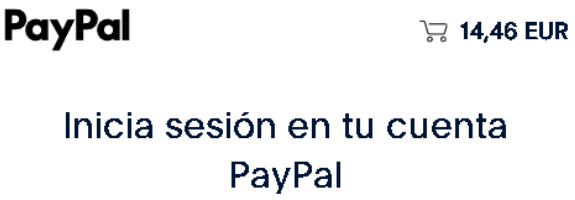
- /budget – Solicitud de presupuesto
  - Formulario donde el usuario puede:
  - Elegir si desea un tatuaje o piercing.
  - Indicar el tamaño, zona del cuerpo, artista preferido.
  - Añadir medidas aproximadas y un mensaje personalizado.
  - Se calcula automáticamente el precio aproximado, según lógica interna en Angular.
  - El formulario envía los datos al backend (save_budget.php), guardando la solicitud en la base de datos.

- /account – Autenticación y perfil
  - Permite registrarse con nombre, email y contraseña.
  - Permite iniciar sesión, validando los datos en el backend (login.php).
  - Guarda sesión en localStorage y muestra un punto verde en el icono del perfil si está activa.

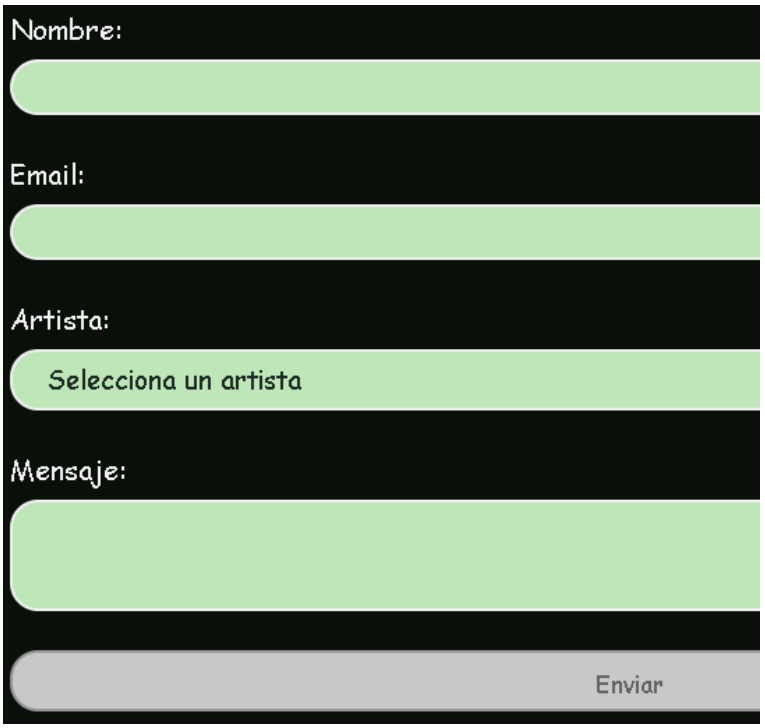
  - Puede mostrar un panel con presupuestos enviados u otras opciones en versiones futuras.

- /paypal – Integración de pagos (futuro)
  - Página de prueba para integrar Stripe Checkout o PayPal.
  - Permite simular una compra en modo de prueba (sandbox).
  - Esta sección es ampliable para completar la funcionalidad de tienda online real.





- /contact – Contacto
  - Página con datos del estudio (email, teléfono, ubicación).
  - Puede incluir un formulario de contacto básico .
  - Ideal para resolver dudas o recibir sugerencias de los usuarios.



- /legal – Documentación legal
  - Aviso legal.
  - Política de privacidad.

- Política de cookies.
- Términos y condiciones de uso.

5. Manual del Usuario

Este manual está destinado a explicar con detalle cómo puede cualquier usuario utilizar la plataforma web Hopink Studio. He diseñado esta aplicación con el objetivo de que tanto nuevos visitantes como clientes habituales puedan navegar, comprar o pedir presupuestos de forma rápida, intuitiva y sin conocimientos técnicos.

He procurado que la interfaz sea clara, adaptada a móviles y con navegación fluida, empleando Angular como framework SPA (Single Page Application). Cada funcionalidad ha sido pensada en base a cómo interactuaría un cliente real en una tienda física de tatuajes y piercings, pero llevada al entorno digital.

5.1 Acceso a la aplicación

Actualmente, el proyecto se encuentra en entorno local. Para acceder, es necesario tener XAMPP instalado (o un entorno similar con Apache y MySQL), y colocar el backend en la carpeta htdocs. Tambien se debe tener instalado node.js y Angular 19.

Una vez lanzado el servidor Apache y la base de datos MySQL desde el panel de control de XAMPP, debemos lanzar el front desde la terminal de angular con “ng serve”. Por último, basta con acceder desde el navegador a http://localhost/hopinkstudio .
En un futuro, este enlace cambiará por un dominio real como https://hopinkstudio.com.

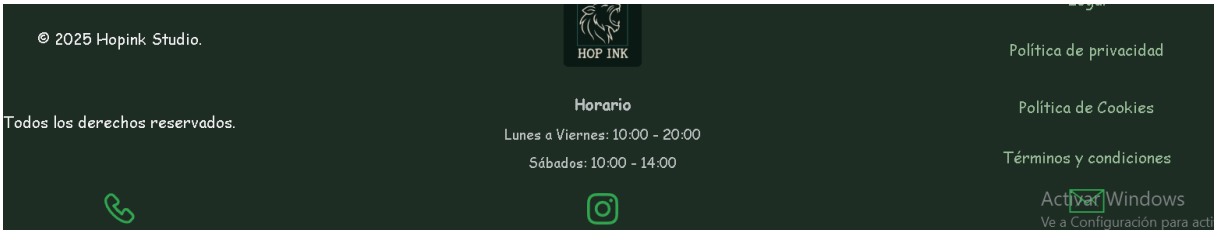
5.2 Estructura visual y navegación

He dividido la interfaz en tres bloques constantes que ayudan al usuario a orientarse desde cualquier sección:

- Header (parte superior): contiene la barra de navegación rápida con el buscador, acceso al perfil, carrito y menú lateral.

- Menú lateral: se abre desde un icono en el header, dando acceso a todas las secciones principales de la web.

- Footer (pie de página): con acceso a contacto, redes sociales y páginas legales.



He optado por dejar siempre accesibles los accesos más importantes: cuenta, carrito y búsqueda, porque son las acciones que más repiten los usuarios en tiendas online.

5.3 Uso de la tienda

El corazón de la web es la sección de tienda. Desde el menú o desde el header(en el apartado de búsqueda), el usuario puede acceder a un catálogo de productos clasificados como:

- Material de tatuaje (tintas, grips, agujas, etc.)

- Material de piercing

- Productos personalizados Hopink

Cada producto tiene su tarjeta individual con imagen, nombre, precio y botones para añadir al carrito o a la lista de deseos. El botón de añadir responde al instante y actualiza el estado del carrito.

He optado por esta organización visual para que sea lo más parecida posible a una tienda real: catálogo en grilla, visual, rápida y funcional.

5.4 Buscador de productos

En la parte superior, en el header, incluí un buscador para filtrar por nombre de producto. Esto permite al usuario localizar más rápido lo que necesita sin tener que navegar por toda la tienda. El filtro se aplica en tiempo real, mejorando la experiencia.

5.5 Carrito de compras

Desde cualquier parte de la web, se puede acceder al carrito. Esta sección muestra todos los productos añadidos, permite cambiar la cantidad de cada uno o eliminarlos individualmente. Además, se calcula automáticamente el total de la compra.

No se requiere estar logueado para usar el carrito, porque he querido que el usuario pueda explorar libremente antes de decidir registrarse.

5.6 Registro e inicio de sesión

Cuando el usuario decide registrarse, puede hacerlo desde la sección "Mi cuenta". Solo se pide nombre, email y contraseña. En el backend, las contraseñas se guardan de forma segura con password_hash().

Una vez logueado, el header muestra un punto verde junto al icono de perfil, indicando sesión activa. También cambia el menú lateral, dando acceso al panel de usuario.

He implementado este sistema para que la experiencia sea personalizada, pero sin obligar al usuario a iniciar sesión si solo quiere mirar productos.

5.7 Panel de usuario

Desde el apartado "Mi cuenta", los usuarios registrados tienen acceso a:

- Mi perfil: datos personales.
- Mis pedidos: historial de compras.
- Mis presupuestos: historial de formularios enviados.
- Devoluciones: espacio para gestionar posibles devoluciones (futuro).
- Lista de deseos: productos guardados.
- Cerrar sesión: finaliza la sesión activa.

Este panel está estructurado en secciones para que sea cómodo encontrar lo que se necesita.

5.8 Lista de deseos

Una funcionalidad pensada especialmente para usuarios frecuentes es la wishlist. Permite guardar productos que les gustan para volver más tarde. Se puede acceder desde el perfil, y los ítems se guardan incluso si se cierra sesión (gracias al localStorage).

5.9 Presupuesto personalizado

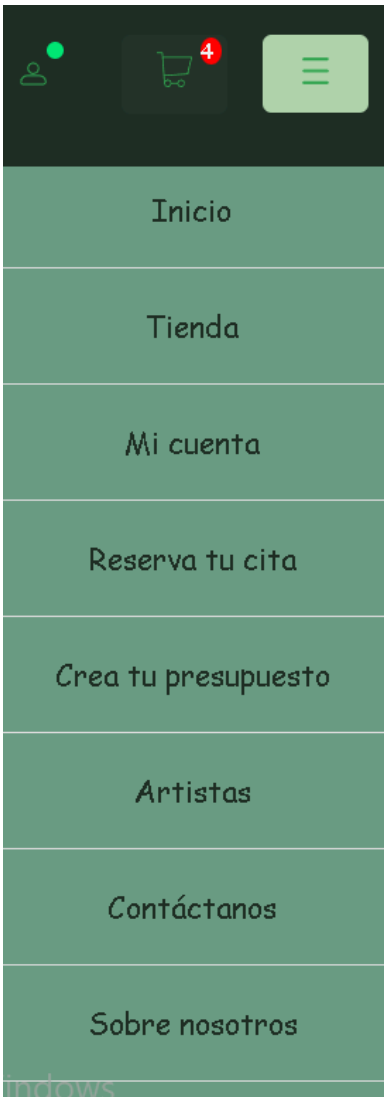
Una parte fundamental del sitio es el formulario para pedir presupuestos. He querido replicar el proceso real que se da en los estudios de tatuaje cuando un cliente se acerca y pide un presupuesto aproximado.

El formulario pide:

- Tipo de servicio (tatuaje o piercing)
- Tamaño en centímetros
- Zona del cuerpo
- Artista
- Datos personales

A partir de esta información, se calcula automáticamente un precio orientativo. Esta lógica está programada en TypeScript y PHP, y permite personalizar el trato sin necesidad de contacto inmediato.

El usuario puede enviar el presupuesto, y este se almacena en la base de datos para poder enseñarlo a la hora de pedir cita. Esto nos evita la conversación previa entre tatuador/cliente.



5.10 Secciones informativas

Desde el menú también se puede acceder a:

- Artistas: información sobre los tatuadores colaboradores.
- Contáctanos: formulario de contacto y datos del estudio.
- Sobre nosotros / Historia: breve recorrido del origen del proyecto.
- Legal: Aviso legal, política de privacidad y cookies.

Estas páginas han sido incluidas para cumplir con la normativa y reforzar la confianza del usuario.

5.11 Footer y redes

En el footer he incluido:

- Teléfono del estudio
- Correo electrónico
- Enlace a Instagram
- Documentación legal

Así, el usuario tiene siempre visible una vía de contacto rápida.

6. Manual Técnico

A continuación se presentan algunos de los fragmentos de código que considero más importantes y técnicamente significativos dentro del desarrollo de Hopink Studio. Estos fragmentos reflejan tanto decisiones clave en la arquitectura como implementaciones que han requerido un análisis más profundo, ya sea por su lógica, su conexión entre tecnologías o por la necesidad de mantener seguridad y eficiencia en el sistema.

Conexión con la base de datos (MySQL): diseño del modelo, consultas SQL optimizadas, normalización de datos.

Muchos de estos bloques han requerido planificación y pruebas, ya que el objetivo ha sido construir un sistema funcional, reutilizable, mantenible y seguro.

Los ejemplos seleccionados también ilustran cómo se implementa la comunicación entre el frontend y el backend, cómo se gestiona la sesión del usuario, y cómo se estructuran operaciones fundamentales como el carrito, los presupuestos o la validación de formularios.

Todos los fragmentos han sido comentados para facilitar su lectura, y se han incluido notas explicativas para contextualizar su uso dentro del sistema.

He elegido mostrar e interpretar el código dividido en las distintas secciones del proyecto:

CONCEPTOS GENERALES:

- En todos los archivos PHP de mi backend he añadido estas líneas al principio:

```
//CORS
header('Access-Control-Allow-Origin: http://localhost:4200');
header('Access-Control-Allow-Credentials: true');
header('Access-Control-Allow-Methods: GET, POST, DELETE, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type');
```

Esto es necesario porque, al estar en local, mi proyecto usa Angular (frontend) y PHP (backend), y ambos se ejecutan en sitios diferentes.

Angular funciona en http://localhost:4200/hopinkstudio , y PHP corre en http://localhost/Hopink-Studio/backend/.. (servidor Apache en XAMPP).

Aunque ambos están en mi ordenador, son considerados dominios distintos por el navegador, así que cuando Angular intenta pedir algo al backend (por ejemplo, iniciar sesión o cargar productos), el navegador bloquea la solicitud por seguridad.

Eso se llama CORS (Cross-Origin Resource Sharing).

¿Qué hace cada una?

- ➔ Access-Control-Allow-Origin: le dice al navegador que sí se permite que Angular (desde localhost:4200) haga peticiones.
- ➔ Access-Control-Allow-Credentials: permite que Angular envíe y reciba cookies o sesiones, como las de login.
- ➔ Access-Control-Allow-Methods: indica qué tipos de peticiones están permitidas (GET para leer, POST para enviar datos, DELETE para borrar, OPTIONS para comprobar permisos).
- ➔ Access-Control-Allow-Headers: permite que Angular pueda enviar encabezados como Content-Type, necesario para enviar datos en JSON.

Sin estas cabeceras, el navegador bloquearía todas las conexiones entre Angular y PHP.

- En app.routes.ts, debo añadir la ruta relativa de todos mis componentes, pues así le digo a Angular qué componente debe mostrar.

El archivo app.routes.ts, es el mapa de rutas de la aplicación.

Por ejemplo:

```
import { Routes, provideRouter, withInMemoryScrolling, withRouterConfig
} from '@angular/router';
import { HomeComponent } from './pages/home/home.component';
```

```
import { ArtistsComponent } from './pages/artists/artists.component';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'artists', component: ArtistsComponent },
```

Esto significa que si el usuario va a /artists, Angular tiene que cargar el componente que muestra la cuenta del usuario.

Si no añado esta línea, aunque tenga hecho el componente, Angular no sabe que existe, y no lo puede mostrar.

- En muchos de los archivos de TypeScript de los componentes importo CommonModule o RouterModule

En Angular, los componentes no lo tienen todo por defecto. Algunas funciones comunes, como *ngIf, *ngFor o los enlaces de navegación (routerLink), no vienen incluidos si no los importas explícitamente.

```
<div class="accountPage" *ngIf="!isLoggedIn">
```

Por tanto, en su archivo .ts , nos fijamos en los imports:

```
import { CommonModule } from '@angular/common';
```

```
@Component({
  selector: 'app-account',
  standalone: true,
    imports:  [CommonModule,  ReactiveFormsModule,  HttpClientModule,
RouterModule],
  templateUrl: './account.component.html',
  styleUrls: ['./account.component.css']
})
```

¿Qué es CommonModule?

Es un módulo que contiene funciones básicas como:

➔ *ngIf: para mostrar u ocultar cosas según una condición.

➔ *ngFor: para repetir elementos como listas de productos.

➔ Pipes como date, currency, etc.

Si no importas CommonModule en tu componente (especialmente en los standalone), Angular no sabe qué es *ngIf o *ngFor y da error.

¿Qué es RouterModule?

Es el módulo que permite usar rutas internas en HTML, como:

```
  <a routerLink="/cart" class="cart">
    <span *ngIf="itemCount>0" class="cartNumber">{{ itemCount }}</span>
  </a>
```

Como se puede observar, el enlace tipo “a” ya no es “*a href=”* sino “*a routerLink*” , y éste necesita tener el RouterModule entre sus imports en el archivo .ts del componente.

```
import { CommonModule } from
'@angular/common';
import { RouterModule, Router, NavigationEnd }

@Component({
  selector: 'app-header',
  standalone: true,
  imports: [CommonModule, RouterModule, HttpClientModule, FormsModule],
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
…
```

Si no importas RouterModule, los routerLink tampoco funcionan, aunque las rutas estén bien definidas.

LÓGICA DEL CARRITO:

- cart.service.ts – Servicio Angular para gestionar el carrito

Este archivo se encarga de guardar, leer y modificar los productos del carrito. Funciona como una memoria interna de la tienda. No se ve, pero cualquier parte de la web puede usarlo para trabajar con el carrito.

Principales funciones explicadas:

➔ getCart(): Devuelve el contenido del carrito. Si no hay nada, crea un array vacío. Sirve para leer lo que ya está guardado, usando localStorage (una memoria del navegador que no se borra al recargar la página).

➔ addToCart(product): Añade un producto. Primero revisa si ese producto ya está en el carrito: Si sí está, aumenta su cantidad y si no está, lo agrega con cantidad 1.

```
  addToCart(product: { id: number }): void {
    this.http
      .post<{ success: boolean; items: CartItem[] }>(
        this.cartUrl,
        { product_id: product.id, quantity: 1 },
        this.httpOptions
      )
      .subscribe({
        next: itemsResponse => {
          this.cart = itemsResponse.items;
          this.cartSubject.next(this.cart);
        },
        error: err => {
          console.error('Error añadiendo al carrito', err);
          console.log('>>> Respuesta bruta del servidor:', err.error);
        }
      });
  }
```

➔ removeFromCart(product): Reduce la cantidad de un producto. Si llega a 0, lo elimina del carrito.

➔ clearCart(): Borra completamente todo el carrito.

➔ getTotal(): Recorre todos los productos y suma sus precios multiplicados por la cantidad. Devuelve el total de la compra.

➔ saveCart() / loadCart(): Se encargan de guardar y cargar el carrito desde localStorage, para que se conserve aunque se cierre la pestaña.

```
loadCart(): void {
    this.http
          .get<{ success: boolean; items: CartItem[] }>(this.cartUrl,
this.httpOptions)
      .subscribe(
        itemsResponse => {
          this.cart = itemsResponse.items;
          this.cartSubject.next(this.cart);
        },
        err => {
          console.error('Error cargando carrito', err);
        }
```

```
      );
  }
```

Este servicio es usado desde otros archivos para centralizar la lógica del carrito.

- cart.component.ts – Controlador Angular del carrito

Este archivo es el cerebro que maneja la vista del carrito. Conecta el HTML con el servicio (cart.service.ts) y define qué pasa cuando el usuario hace clic en botones como "+" o "vaciar carrito".

Principales partes:

➔ ngOnInit(): Esta función se ejecuta automáticamente cuando se carga el componente. Llama a getCart() para mostrar los productos actuales y calcular el total.

```
ngOnInit(): void {
    // Cargar el carrito desde el backend
    this.cartService.loadCart();
    // Suscribirse a los cambios del carrito:
     this.cartSub = this.cartService.getCartObservable().subscribe(items
=> {
      this.cart = items;
      this.total = this.cartService.getTotal();
    });
  }
```

➔ more(product): Aumenta la cantidad de un producto usando el método addToCart del servicio. Luego recalcula el total.

```
  more(item: CartItem): void {
    this.cartService.addToCart({ id: item.id });
  }
```

➔ decrement(product): Reduce la cantidad. Si llega a 0, lo elimina.

```
  decrement(item: CartItem): void {
    if(item.quantity<=1){
      this.removeProduct(item);
    }
    else{
      this.cartService.decreaseQuantity(item.id);
    }

  }
```

➔ removeProduct(product):Lo borra directamente del carrito.

➔ clearCart(): Borra todo el carrito y reinicia la vista.

➔ getTotal(): Llama al servicio para calcular cuánto cuesta el carrito.

➔ openCheckout(): Muestra un pequeño formulario para simular el paso de compra (aún sin conexión real con servidor).

```
openCheckout(): void {
     this.isCheckout = true;       // muestra <div class="checkoutBox"
*ngIf="isCheckout">
  }
```

➔ processPayment(): Envía los datos introducidos en el formulario y simula el pago. Esta función está preparada para enviar los datos a un backend en producción más adelante.

```
processPayment(): void {
    //Comprueba que todos los datos del envío esten completos
    if (
      !this.shippingData.fullName.trim() ||
      !this.shippingData.address.trim() ||
      !this.shippingData.city.trim() ||
      !this.shippingData.postalCode.trim() ||
      !this.shippingData.country.trim()
    ) {
      alert('Por favor, completa todos los campos de envío.');
      return;
    }
```

Este archivo se comunica con el servicio (cart.service.ts) para realizar todas sus acciones.

El cart.service.ts usa algo llamado localStorage, que es una especie de memoria del navegador.Gracias a eso, si el usuario recarga la página, el carrito no se borra.

Esta lógica no tiene nada que ver con lo que se ve, por eso está mejor en el servicio, no en el componente.

- cart.component.html – Vista del carrito

Este archivo define lo que el usuario ve en pantalla cuando entra a la página del carrito.

```
<div *ngIf="cart.length > 0">
    <ul class="cartList">
      <li *ngFor="let item of cart" class="cartItem">
        <span class="name">{{ item.name }}</span>
        <span class="qty">x{{ item.quantity }}</span>
                <span  class="price">{{  item.price  *  item.quantity  |
currency:'EUR' }}</span>
        <div class="removeButtons">
```

```
          <button class="more" (click)="more(item)">+</button>
          <button class="decrease" (click)="decrement(item)">-</button>
                                                <button     class="remove"
(click)="removeProduct(item)">✕</button>
        </div>
      </li>
    </ul>

     <p class="total"><strong>Total:</strong> {{ total | currency:'EUR'
}}</p>

    <div *ngIf="cart.length > 0" class="buttons">
                  <button   class="empty"    (click)="clearCart()">Vaciar
carrito</button>
      <button class="pay" (click)="openCheckout()">
        Finalizar compra
      </button>

      <div class="checkoutBox" *ngIf="isCheckout">
  <h3>Datos de envío</h3>
  <form (ngSubmit)="processPayment()">
    <div class="formGroup">
      <label for="fullName">Nombre completo</label>
      <input
        type="text"
        id="fullName"
        [(ngModel)]="shippingData.fullName"
        name="fullName"
        required
      />
    </div>
    <div class="formGroup">
      <label for="address">Dirección</label>
      <input
        type="text"
        id="address"
        [(ngModel)]="shippingData.address"
        name="address"
        required
      />
    </div>
    <div class="formGroup">
      <label for="city">Ciudad</label>
```

```
      <input
        type="text"
        id="city"
        [(ngModel)]="shippingData.city"
        name="city"
        required
      />
    </div>
    <div class="formGroup">
      <label for="postalCode">Código postal</label>
      <input
        type="text"
        id="postalCode"
        [(ngModel)]="shippingData.postalCode"
        name="postalCode"
        required
      />
    </div>
    <div class="formGroup">
      <label for="country">País</label>
      <input
        type="text"
        id="country"
        [(ngModel)]="shippingData.country"
        name="country"
        required
      />
    </div>

    <div class="buttonsCheckout">
                            <button     type="button"     class="cancel"
(click)="cancelCheckout()">Cancelar</button>
      <button type="submit" class="pay">Pagar</button>
    </div>
  </form>
```

Estructura:

- ➔ Un *ngFor recorre todos los productos y muestra: su nombre,su precio, su cantidad actual, botones para aumentar, reducir o eliminar

- ➔ Abajo hay: El total de la compra, un botón de “vaciar carrito” y un botón de “finalizar compra”

Si el usuario decide comprar, se muestra un formulario donde escribe su nombre, correo, dirección, etc.

Este HTML está totalmente conectado con las funciones del archivo .ts. Por ejemplo, cuando haces clic en el botón "+" en pantalla, Angular llama a more(product) y eso actualiza la cantidad.

- cart.php – Entrada del backend en PHP

Este archivo está pensado para cuando el usuario finalice su compra y se quiera guardar el pedido en una base de datos real.

```
$userId = $_SESSION['user']['id'] ?? null;

if (is_int($userId)) {
    // Usuario logueado
    $cartId = getOrCreateCartByUserId($userId);
} else {
    // Invitado
    if (empty($_SESSION['cart_id'])) {
        $_SESSION['cart_id'] = createGuestCart();
    }
    $cartId = $_SESSION['cart_id'];
}

$result = [];
switch ($_SERVER['REQUEST_METHOD']) {
    case 'GET':
        $result = getCartItemsByCartId($cartId);
        break;

    case 'POST':
         $body = json_decode(file_get_contents('php://input'), true) ?:
[];
        if (isset($body['product_id'], $body['quantity'])) {
            addOrUpdateCartItem(
              $cartId,
              (int)$body['product_id'],
              (int)$body['quantity']
            );
        }
        $result = getCartItemsByCartId($cartId);
        break;

    case 'DELETE':
        //lee el raw body como JSON
```

```
        $body = json_decode(file_get_contents('php://input'), true);
        //si existe, borra
        if (isset($body['product_id'])) {
            removeCartItem($cartId, (int)$body['product_id']);
        }
        //devuelve el carrito actualizado
        $result = getCartItemsByCartId($cartId);
        break;

    default:
        http_response_code(405);
        echo json_encode(['error' => 'Método no permitido']);
        exit;
}
```

Este código es el encargado de manejar todas las acciones relacionadas con el carrito de compra en la parte del servidor, es decir, en el backend. No es algo que el usuario vea directamente, pero cada vez que interactúa con el carrito desde la web (como añadir un producto, eliminarlo o ver lo que ya tiene), este archivo se encarga de responder y gestionar esos datos.

Lo primero que hace el código es comprobar si el usuario ha iniciado sesión. Si lo ha hecho, se obtiene su identificador (ID) desde la sesión. Ese ID permite asociarle un carrito permanente en la base de datos. Si no está logueado (es decir, si es un visitante anónimo), se crea un carrito temporal específico para él, que se guarda en la sesión del navegador. Así, tanto usuarios registrados como invitados pueden usar el carrito.

Una vez que ya se sabe a quién pertenece el carrito (usuario o invitado), se guarda su identificador de carrito en una variable para usarla más adelante.

Después, el código se prepara para responder dependiendo del tipo de acción que se le haya pedido desde el frontend. Esto se hace revisando el tipo de petición que ha hecho el navegador (lo que se llama método HTTP). Puede ser de tres tipos: GET, POST o DELETE.

Si la petición es de tipo GET, significa que el usuario solo quiere ver lo que hay en su carrito. El código busca en la base de datos todos los productos guardados en ese carrito, y se los devuelve en formato que Angular pueda leer.

Si la petición es de tipo POST, significa que el usuario quiere añadir un producto nuevo al carrito o cambiar la cantidad de uno que ya tenía. Para eso, Angular le manda al servidor el ID del producto y la cantidad deseada. El código entonces actualiza el carrito: si el producto ya estaba, cambia la cantidad; si no estaba, lo añade. Después de actualizarlo, vuelve a devolver el contenido completo del carrito.

Si la petición es de tipo DELETE, el usuario está indicando que quiere quitar un producto del carrito. Angular le dice al servidor qué producto es, y el código lo elimina de la base de datos. Al terminar, devuelve otra vez el carrito, ya sin ese producto.

Por último, si alguien intenta usar un método que no es ninguno de esos tres (por ejemplo PUT o PATCH), el código responde con un mensaje de error que indica que esa acción no está permitida.

En resumen, este código se encarga de todo lo relacionado con el carrito desde el lado del servidor: crea un carrito si no existe, añade productos, los actualiza, los elimina y devuelve siempre el estado actualizado del carrito para que la web pueda mostrarlo al usuario. Está preparado para funcionar tanto con usuarios registrados como con invitados, y es esencial para que la experiencia de compra sea fluida, coherente y personalizada.

Está estructurado para recibir datos por POST y trabajar con funciones definidas en cart_helpers.php.

- cart_helpers.php – Funciones de carrito en el servidor

Este archivo contiene las funciones que interactúan directamente con la base de datos.

Por ejemplo:

➔ insertOrder():Inse ta un nuevo pedido en la tabla orders (con nombre, correo, dirección, fecha…).

➔ insertOrderItems($order_id, $cart): Guarda cada producto comprado en la tabla cart_items, relacionándolo con su pedido correspondiente.

Estas funciones no se llaman directamente desde Angular, sino desde cart.php, que será el encargado de recoger los datos enviados cuando se active la conexión real.

```
function getOrCreateCartByUserId(int $uid): int {
    global $conn;
    //Intentar recuperarlo
    $stmt = $conn->prepare("
      SELECT id
        FROM cart
       WHERE user_id = ?
         AND status = 'open'
       LIMIT 1
    ");
    $stmt->bind_param('i', $uid);
    $stmt->execute();
    $stmt->bind_result($cid);
    if ($stmt->fetch()) {
        $stmt->close();
```

```
        return $cid;
    }
    $stmt->close();
    //Si no existe, lo crea
    $stmt = $conn->prepare("
      INSERT INTO cart (user_id, status)
           VALUES (?, 'open')
    ");
    $stmt->bind_param('i', $uid);
    $stmt->execute();
    $newId = $stmt->insert_id;
    $stmt->close();
    return $newId;
}

//Crear carrito de invitado
function createGuestCart(): int {
  global $conn;
  $stmt = $conn->prepare("
    INSERT INTO cart (user_id, status)
         VALUES (NULL, 'open')
  ");
  $stmt->execute();
  $newId = $stmt->insert_id;
  $stmt->close();
  return $newId;
}


//Devuelve productos del carrito
function getCartItemsByCartId(int $cid): array {
    global $conn;
    $sql = "
      SELECT p.id, p.name, p.price, ci.quantity
        FROM cart_items ci
   LEFT JOIN products    p ON p.id = ci.product_id
       WHERE ci.cart_id = ?
    ";
    $stmt = $conn->prepare($sql);
    $stmt->bind_param('i', $cid);
    $stmt->execute();
    $res = $stmt->get_result();
    $items = $res->fetch_all(MYSQLI_ASSOC);
```

```
    $stmt->close();
    return $items;
}
```

Cómo se relaciona todo:

- ➔ El usuario interactúa con botones en el HTML (cart.component.html).
- ➔ Angular reacciona y ejecuta funciones en cart.component.ts.
- ➔ Estas funciones usan cart.service.ts para modificar o leer el contenido del carrito.
- ➔ Los cambios se reflejan en pantalla, y se guardan en localStorage para persistencia.
- ➔ Cuando se implemente la funcionalidad de compra real, Angular enviará los datos a cart.php.
- ➔ cart.php usará cart_helpers.php para guardar todo correctamente en la base de datos MySQL.

La lógica de la wishlist es similar a la del carrito, aunque ésta no es un componente como tal, sino que se encuentra dentro del componente account.

LÓGICA DEL INICIO DE SESIÓN:

El sistema de login en Hopink Studio está hecho para que el usuario pueda registrarse, iniciar sesión, y luego acceder a su cuenta personalizada, donde puede ver sus pedidos, presupuestos, datos personales, etc. Aunque parece algo simple desde fuera (poner el correo y contraseña y entrar), por detrás hay varios archivos trabajando juntos, tanto en el frontend como en el backend.

Voy a explicarlo paso por paso y archivo por archivo, como lo pensé y lo construí para que todo funcione correctamente.

- account.component.html

Esta es la página visual de “Mi cuenta”, donde el usuario ve dos formularios:

Uno para iniciar sesión (email y contraseña) y otro para registrarse (nombre, email, contraseña)

```
<div class="accountPage" *ngIf="!isLoggedIn">
  <div class="accountForms">
```

```
                <form    [formGroup]="loginForm"    (ngSubmit)="login()"
class="loginForm">
      <h3>Iniciar sesión</h3>
             <input  type="email"               formControlName="email"
pattern="^[a-zA-Z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"
placeholder="Email" required>
              <input  type="password"         formControlName="password"
placeholder="Contraseña" required>
      <button type="submit" class="next">Siguiente</button>
    </form>

            <form   [formGroup]="registerForm"   (ngSubmit)="register()"
class="registerForm">
      <h3>Registrarse</h3>
             <input  type="text"                 formControlName="name"
placeholder="Nombre completo" required>
             <input  type="email"               formControlName="email"
placeholder="Email"
pattern="^[a-zA-Z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$" required>
              <input  type="password"         formControlName="password"
placeholder="Contraseña" required>
      <button type="submit">Crear cuenta</button>
    </form>
  </div>
</div>
```

Cuando se pulsa el botón de enviar en cualquiera de los dos, no se recarga la página, sino que se llama a una función en el archivo TypeScript que está conectado: account.component.ts.

- account.component.ts

Aquí es donde ocurre la lógica del cliente. Cuando el usuario intenta iniciar sesión:

Angular recoge los datos del formulario.

Ejecuta la función login().

```
login() {
    const { email, password } = this.loginForm.value;
    this.http.post<ApiResponse>(
      `${this.API}/login.php`,
      { email, password },
      { withCredentials: true }
```

```
    ).subscribe({
      next: res => {
        if (res.status === 'success' && res.user) {
         this.isLoggedIn = true;
         this.user       = res.user;
         this.cartService.loadCart();
         this.loadBudgets();
         this.router.navigate(
           ['/account'],
           { fragment: 'wishlist' }
         );
       } else {
         alert(res.message || 'Email o contraseña incorrectos');
       }
      },
      error: err => {
        console.error('login error', err);
        alert('Contraseña o email incorrectos');
      }
    });
  }
```

Esa función envía una solicitud HTTP POST al archivo login.php del backend, con el correo y la contraseña.

Si el backend responde que el login es correcto, el componente guarda los datos del usuario y cambia el diseño para mostrar la cuenta activa (nombre, correo, pedidos, presupuestos…).

Si la sesión ya estaba iniciada, la función ngOnInit() llama a getUser() que pregunta al servidor si hay una sesión abierta (comprobando el archivo getUser.php). Así, aunque el usuario recargue la página, sigue dentro de su cuenta sin tener que volver a iniciar sesión.

- login.php

Este archivo es el que recibe los datos del login enviados por Angular. Hace lo siguiente:

Toma el email y la contraseña del usuario.

```
$body     = json_decode(file_get_contents('php://input'), true) ?: [];
$email    = trim($body['email']    ?? '');
$password =            $body['password'] ?? '';
```

Busca en la base de datos si existe un usuario con ese email.

```
$stmt = $conn->prepare("
    SELECT id, name, password
```

```
      FROM users
     WHERE email = ?
     LIMIT 1
");
$stmt->bind_param('s', $email);
$stmt->execute();
$res = $stmt->get_result();
```

Si lo encuentra, verifica que la contraseña es correcta usando password_verify() (que compara la contraseña cifrada).

```
if ($user = $res->fetch_assoc()) {
    // Verificar contraseña
    if (password_verify($password, $user['password'])) {
        session_start();
        $_SESSION['user'] = [
            'id'    => (int)$user['id'],
            'name'  => $user['name'],
            'email' => $email
        ];
```

Si todo está bien, crea una sesión PHP, y guarda dentro de ella los datos del usuario (ID, nombre, email).

```
$_SESSION['user_id'] = (int)$user['id'];
```

Luego, responde al frontend con los datos básicos del usuario.

Esto permite que el usuario esté “conectado” y que Angular pueda usar su información para mostrar su perfil, pedidos, etc.

- getUser.php

Este archivo es muy sencillo, pero muy útil. Solo se encarga de:

Comprobar si ya hay una sesión iniciada ($_SESSION['user'])

Si existe, devuelve los datos del usuario.

```
  session_start();
  if (isset($_SESSION['user'])) {
    echo json_encode(['status'=>'success','user'=>$_SESSION['user']]);
  } else {
    echo json_encode(['status'=>'error']);
  }
?>
```

Esto es lo que usa Angular al cargar la página para saber si ya hay alguien conectado.

- logout.php

Cuando el usuario hace clic en “Cerrar sesión”, Angular llama a este archivo.

Este archivo simplemente borra la sesión del servidor.

Eso significa que el usuario ya no está logueado, y al recargar la página, ya no verá su cuenta.

```
    session_start();
    session_unset();
    session_destroy();
    echo json_encode(['status'=>'success']);
```

- register.php

Cuando un nuevo usuario se quiere registrar:

```
    require_once __DIR__ . '/db.php';

    $raw  = file_get_contents('php://input');
    $body = json_decode($raw, true);
    if (!$body) {
    http_response_code(400);
       echo json_encode(['status'=>'error','message'=>'JSON inválido o
vacío']);
    exit;
    }

    $name  = $conn->real_escape_string(trim($body['name']  ?? ''));
    $email = $conn->real_escape_string(trim($body['email'] ?? ''));
    $pass  =           trim($body['password']    ?? '');
```

```
        if (!$name || !filter_var($email, FILTER_VALIDATE_EMAIL) ||
strlen($pass) < 4) {
    http_response_code(400);
    echo json_encode(['status'=>'error','message'=>'Datos inválidos']);
    exit;
    }

    $hash  = password_hash($pass, PASSWORD_DEFAULT);
    $token = bin2hex(random_bytes(16));

    $stmt = $conn->prepare(
    "INSERT INTO users (name,email,password,token,status)
    VALUES (?, ?, ?, ?, 0)"
    );
    $stmt->bind_param('ssss', $name, $email, $hash, $token);
    if ($stmt->execute()) {
            echo   json_encode(['status'=>'success','message'=>'Usuario
creado']);
    } else {
    http_response_code(409);
            echo   json_encode(['status'=>'error','message'=>'Email   ya
registrado']);
    }
    $stmt->close();
    $conn->close();
```

Angular recoge los datos y los envía a register.php.

Este archivo los recibe, comprueba que el email no esté repetido, cifra la contraseña con password_hash(), y guarda al nuevo usuario en la base de datos.

Después, también inicia sesión automáticamente y devuelve sus datos al frontend.

Cómo se relaciona todo:

Todo esto está pensado como un sistema conectado entre frontend y backend:

- ➔ El usuario rellena el formulario → Angular lo recoge. (account.component.html/.ts)
- ➔ Angular envía los datos al servidor (PHP) → login.php o register.php.
- ➔ El servidor verifica o guarda datos → si son válidos, crea una sesión PHP.
- ➔ Angular pregunta constantemente a getUser.php si la sesión sigue activa.

- ➔ Si el usuario hace logout, se llama a logout.php y la sesión se elimina.
- ➔ Gracias a las sesiones, el servidor “recuerda” quién es el usuario, incluso si navega por otras páginas o recarga la web.

¿Por qué lo hice así?

Porque separar el frontend (Angular) del backend (PHP) me permite tener un sistema limpio, seguro y organizado.

Usar sesiones PHP me permite mantener el login activo sin pedirle al usuario que vuelva a conectarse cada vez.

Uso funciones de PHP como password_hash() y password_verify() para proteger las contraseñas de los usuarios.

Angular controla toda la parte visual y muestra solo lo que el usuario puede ver si está logueado.

LÓGICA DE PRESUPUESTOS

- budget.component.html

En este componente encontramos un formulario con campos como email con validación, que muestra un error si no cumple la misma, un select para la selección de artista, según el cual variará el precio por hora, botones tipo radio, mediante los que se puede elegir el tipo de servicio a presupuestar. Si eliges tatuaje, se habilitarán los campos de ancho y alto, y selector de imagenes, que determinarán las horas estimadas que durará el tatuaje

Bajo el formulario, tenemos una sección donde, de forma reactiva, se muestra el presupuesto estimado en euros siempre que el usuario haya seleccionado artista y tipo de servicio. Esta cifra se recalcula al instante conforme se cambian los valores de dimensiones o tipo de servicio, gracias a la vinculación entre plantilla y clase.

Para calcular el presupuesto, el código indica que debe estar relleno el campo de artista y de servicio, los que darán un precio base, y al añadir el ancho u alto irá subiendo. Para ello, utiliza la función “getTotalPrice” , la cual veremos detenidamente en budget.component.ts y

que, como he mencionado anteriormente, depende del servicio, artista, alto y ancho del formulario.

```
<p *ngIf="form.service && form.artist">
      <strong>Presupuesto estimado:</strong>
            {{  getTotalPrice(form.service,  form.artist,  form.height,
form.width) }} €
    </p>
```

Un botón de tipo “submit” para enviar el formulario, que al pulsarlo, se invoca el método de la clase que valida de nuevo y, si todo está correcto, prepara y envía la petición POST al servidor.

- save_budget.php

Al comenzar el script, se llama a la función que inicia o retoma la sesión de PHP (session_start). Gracias a esto, el script puede acceder a variables de sesión que guardan la información del usuario.

Se comprueba la existencia de $_SESSION['user_id']. Si no está definido, significa que el usuario no ha iniciado sesión, por lo que se envía un código de estado HTTP 401 (no autorizado) y un JSON con un mensaje de error, quedando ahí interrumpida la ejecución.

El script lee el contenido bruto de la petición (php://input) y lo decodifica con json_decode para obtener un array asociativo de PHP con las claves: nombre, correo, artista, servicio, altura, ancho, mensaje y presupuesto.

Se asigna cada valor a variables locales, aplicando un valor predeterminado si alguna clave no viene presente.

```
$userId = $_SESSION['user_id'];
    $data = json_decode(file_get_contents('php://input'), true);

    $name     = $data['name']     ?? '';
    $email    = $data['email']    ?? '';
    $artist   = $data['artist']   ?? '';
    $service  = $data['service']  ?? '';
    $height   = $data['height']   ?? 0;
    $width    = $data['width']    ?? 0;
    $message  = $data['message']  ?? '';
    $budget   = $data['budget']   ?? 0;
```

Se prepara una consulta SQL parametrizada (usando prepare y bind_param) para insertar un nuevo registro en la tabla budgets.

```
     $stmt = $conn->prepare("INSERT INTO budgets (user_id, name, email,
artist, service, height, width, message, budget, created_at)
```

```
                                    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,
NOW())");
      $stmt->bind_param("issssiisd", $userId, $name, $email, $artist,
$service, $height, $width, $message, $budget);
```

Los campos que se insertan incluyen: identificador del usuario, nombre, correo, artista, servicio, altura, ancho, mensaje, presupuesto numérico y la marca de tiempo de creación.

Se ejecuta la consulta y se comprueba si se guardó correctamente.

En caso de éxito, se devuelve un JSON indicando { "success": true }. Si hay error al ejecutar la sentencia, se responde con código de estado 500 (Error interno de servidor) y un JSON con el mensaje de fallo.

```
    if ($stmt->execute()) {
        echo json_encode(['success' => true]);
    } else {
        http_response_code(500);
                echo json_encode(['error' => 'Error al guardar el
presupuesto']);
    }
```

- get_budgets.php

Igual que en el script anterior, se invoca session_start y se configuran las mismas cabeceras CORS.

Se comprueba que exista un user_id en la sesión; si no, se retorna HTTP 401 con un JSON de error.

```
if (!isset($_SESSION['user_id'])) {
        http_response_code(401);
                echo json_encode(['ok' => false, 'message' => 'No
autenticado']);
        exit;
    }
```

Se prepara una consulta SQL parametrizada que selecciona, de la tabla budgets, las columnas relevantes (identificador del presupuesto, artista, servicio, importe de presupuesto, altura, ancho, mensaje y fecha de creación) donde user_id coincide con el identificador de la sesión.

Se ejecuta la consulta, se recorre el conjunto de resultados y se acumulan en un array de registros (cada registro es un array asociativo con los campos mencionados).

```
$userId = $_SESSION['user_id'];

    $stmt = $conn->prepare("SELECT id, artist, service, budget, height, width, message, created_at FROM budgets WHERE user_id = ?");
    $stmt->bind_param("i", $userId);
    $stmt->execute();
    $result = $stmt->get_result();

    $budgets = [];
    while ($row = $result->fetch_assoc()) {
        $budgets[] = $row;
```

Finalmente, se emite un objeto JSON con una clave “ok” a true, y otra clave “budgets” que contiene la lista de presupuestos encontrados.

```
    echo json_encode(['ok' => true, 'budgets' => $budgets]);
```

En caso de error en la consulta, se podría capturar y devolver “ok” a false con un mensaje de error, pero en la implementación estándar se asume que si la sesión es válida y la consulta no falla, siempre se devuelve la lista.

- delete_budget.php

Igual que en los anteriores, se inicia la sesión y se comprueban credenciales.

Se recibe por POST un parámetro “id” (identificador del presupuesto a eliminar).

Se prepara una sentencia parametrizada (“DELETE FROM budgets WHERE id = ? AND user_id = ?”), de tal forma que solo se elimine si el registro pertenece al usuario autenticado.

```
      $stmt = $conn->prepare("DELETE FROM budgets WHERE id = ? AND user_id = ?");
    $stmt->bind_param("ii", $id, $userId);
```

Se ejecuta, y se comprueba el número de filas afectadas. Si al menos una fila fue eliminada, se considera éxito; en caso contrario, se asume que el registro no existía o no pertenecía a ese usuario, por lo que podría devolverse un JSON con “ok: false” y un mensaje de error.

Se responde con un JSON que indica si la operación fue satisfactoria o no. En caso de éxito, envía { "ok": true }; si hubo error de permisos o de consulta, envía { "ok": false, "message": "Error al eliminar" } y un código HTTP 400 o 500 según el caso.

```
if ($stmt->execute()) {
    echo json_encode(['ok' => true]);
    } else {
        echo json_encode(['ok' => false, 'message' => 'Error al
eliminar']);
    }
```

- account.component.ts

En ella, se declaran variables que sirven para almacenar el estado de autenticación del usuario y los datos asociados:

➔ **isLoggedIn: boolean** , que indica si el usuario ha iniciado sesión. Se inicializa en false y se actualiza durante el proceso de verificación de sesión (checkSession()).

➔ **user: any | null** , objeto que contendrá, al menos, las propiedades email y name del usuario autenticado. Se inicializa en null y se rellena con la respuesta de getUser.php cuando se confirma sesión.

➔ **loginForm: FormGroup** y **registerForm: FormGroup** , son estructuras de formularios reactivos para el proceso de inicio de sesión y registro.

En la misma clase AccountComponent se declaran variables dedicadas a gestionar la lista de presupuestos y su eliminación:

➔ **budgets: any[] = [] ,** es un array vacío que, una vez cargado, contendrá objetos de presupuesto recuperados desde el servidor (campos típicos: id, artist, service, budget, height, width, message, created_at).

➔

➔ **selectedBudget: any = null ,** objeto que almacena temporalmente el presupuesto seleccionado en la interfaz cuando el usuario hace clic en “Eliminar”. Su contenido es un objeto con, al menos, la propiedad id (junto a artist, service, etc.), obtenidos de la lista que vino del servidor.

➔ **showConfirmDelete: boolean = false ,** es un booleano que controla la visibilidad del cuadro modal o sección emergente de confirmación de borrado. Cuando el usuario pulsa “X” en alguna tarjeta de presupuesto, esta variable pasa a true y la plantilla muestra el modal de confirmación. Si se cancela o se borra, vuelve a false.

El constructor de AccountComponent recibe instancias de varios servicios y utilidades necesarias:

- ➔ **HttpClient** (para realizar peticiones HTTP a los scripts PHP).
- ➔ **FormBuilder** (para construir los formularios reactivos de login y registro).

Se llama a this.checkSession(), un método que realiza una petición GET a getUser.php, para ver si hay sesión activa.

```
  private checkSession() {
              this.http.get<ApiResponse>(`${this.API}/getUser.php`,    {
withCredentials: true })
      .subscribe({
        next: res => {
          if (res.status === 'success' && res.user) {
            this.isLoggedIn = true;
            this.user       = res.user;
            this.cartService.loadCart();
            this.loadBudgets(); // Carga presupuestos
          } else {
            this.isLoggedIn = false;
          }
        },
        error: () => {
          this.isLoggedIn = false;
        }
      });
  }
```

Si la respuesta indica status: 'success' y contiene un objeto user, asigna this.isLoggedIn = true y this.user = res.user.

Si no hay sesión válida o ocurre error, deja this.isLoggedIn = false.

Cuando se confirma isLoggedIn = true, se invoca inmediatamente this.loadBudgets() para poblar la lista de presupuestos.

```
  loadBudgets(): void {
    const email = this.user?.email;
    if (!email) return;


this.http.post('http://localhost/Hopink-Studio/backend/get_budgets.php'
, { email })
      .subscribe({
        next: (res: any) => {
```

```
          if (res.ok) {
            this.budgets = res.budgets;
          } else {
            console.error('Presupuestos no cargados correctamente');
          }
        },
        error: err => {
          console.error('Error al cargar presupuestos', err);
        }
      });
  }
```

En caso de error, permanece isLoggedIn = false, y la sección de “Mis presupuestos” no se mostrará al usuario.

Cuando se recibe la respuesta del servidor (un JSON con, al menos, { ok: boolean, budgets: [...] }):

Si res.ok es true, entonces res.budgets es un array de objetos.

Se asigna this.budgets = res.budgets.

En caso de que res.ok sea false o que la llamada HTTP falle (error de red, error 500, etc.), se registra en consola un mensaje de error y, opcionalmente, podría mostrarse una alerta al usuario.

Tras asignar el array a this.budgets, la plantilla HTML (account.component.html) se actualizará automáticamente gracias al data binding de Angular, y se mostrará cada presupuesto en pantalla.

- ➔ **confirmDelete(budget: any) ,** tiene un parámetro de entrada: el objeto budget correspondiente a la tarjeta que el usuario quiere borrar.

Asigna this.selectedBudget = budget.

Pone this.showConfirmDelete = true para activar la visualización del modal de confirmación en la plantilla.

```
confirmDelete(budget: any) {
  this.selectedBudget = budget;
  this.showConfirmDelete = true;
}
```

Resultado visual: al actualizar showConfirmDelete a true, la plantilla (account.component.html) renderiza el cuadro emergente de confirmación preguntando “¿Estás seguro de eliminar este presupuesto?”.

- ➔ **cancelDelete() ,** asigna this.selectedBudget = null y this.showConfirmDelete = false.

Efecto: oculta el modal de confirmación sin realizar ningún borrado y limpia la variable que almacenaba el presupuesto seleccionado.

```
cancelDelete() {
  this.selectedBudget = null;
  this.showConfirmDelete = false;
}
```

- ➔ **deleteBudget() ,** para borrar un presupuesto, this.selectedBudget debe contener un objeto con, al menos, la propiedad id (identificador numérico del presupuesto a borrar).

Extrae const id = this.selectedBudget.id;.

Realiza una petición HTTP POST a delete_budget.php, enviando { id } como JSON en el cuerpo.

En el subscribe, si la respuesta JSON contiene { ok: true }, se procede a:

Filtrar localmente el array this.budgets para eliminar el elemento cuyo id coincida con this.selectedBudget.id. Esto se hace sin recargar la página ni volver a llamar a loadBudgets().

Asignar this.showConfirmDelete = false para ocultar el modal.

Si el servidor responde { ok: false } o la petición falla, se registra el error en consola y, opcionalmente, se puede notificar al usuario (por ejemplo, mediante una alerta).

```
deleteBudget() {
  if (!this.selectedBudget?.id) return;


this.http.post('http://localhost/Hopink-Studio/backend/delete_budget.ph
p', {
    id: this.selectedBudget.id
  }).subscribe({
    next: (res: any) => {
      if (res.ok) {
                this.budgets = this.budgets.filter(b => b.id !==
this.selectedBudget.id);
        this.showConfirmDelete = false;
      }
```

```
    },
    error: err => {
      console.error('Error al eliminar presupuesto', err);
    }
  });
}
```

- account.component.html

Incluye una sección (por ejemplo, <section id="budgets">) que engloba todo lo relativo a “Mis presupuestos”.

Se utiliza la directiva *ngFor="let b of budgets" para recorrer cada elemento en el array budgets (asignado en account.component.ts).

```
<div *ngFor="let b of budgets" class="budget-card">
          <button class="close" (click)="confirmDelete(b)">X</button>
          <h4>
                  {{ getServiceLabel(b.service) }} con {{ b.artist |
titlecase }}
          </h4>

                        <p><strong>Presupuesto estimado:</strong> {{
getTotalPrice(b.service, b.artist, b.height, b.width) }} €</p>
          <div class="budgetContent">
                      <ng-container *ngIf="b.service !== 'piercing'"
class="budgetContent">
              <p>Base: {{ getBasePrice(b.artist) }} €/hora</p>
                <p *ngIf="b.service === 'color'">+ Suplemento por color:
30 €</p>
                    <p>Tiempo estimado: {{ getEstimatedHours(b.height,
b.width) }} hora(s)</p>
          </ng-container>
                    <p><em>Fecha del presupuesto: {{ b.created_at |
date:'dd/MM/yyyy HH:mm' }}</em></p>

          </div>
```

Por cada iteración, Angular crea un bloque HTML (por ejemplo, un) que representará de forma visual cada presupuesto individual.

Cada presupuesto consta de un botón situado en la esquina superior de la tarjeta con una “X” o icono de cierre.

Ese botón tiene (click)="confirmDelete(b)", de modo que al pulsarlo se invoca el método confirmDelete de AccountComponent, pasando el objeto b como parámetro.

Aunque el importe ya está guardado en la base de datos, la plantilla vuelve a invocar el método getTotalPrice(b.service, b.artist, b.height, b.width) para recalcular localmente el valor y mostrar el desglose transparente al usuario.

Esto no modifica el valor real almacenado; solo sirve para que el usuario entienda cómo se obtuvo su presupuesto.

Para el desglose de datos para tatuajes, utilizo *ngIf="b.service !== 'piercing'", y se muestra un bloque de texto con:

Precio base por hora: se invoca getBasePrice(b.artist) en el componente para conocer el precio por hora que corresponde a ese artista.

Suplemento por color: se muestra únicamente si b.service === 'color'. Visualiza “+ Suplemento por color: 30 €”.

Tiempo estimado: se invoca getEstimatedHours(b.height, b.width) para mostrar “X hora(s)”.

En caso de que b.service === 'piercing', este bloque no se renderiza y, en su lugar, el presupuesto mostrado es un valor fijo calculado directamente por getTotalPrice.

Fecha de creación del presupuesto

Se muestra b.created_at (que en la base de datos está guardado como DATETIME) utilizando el pipe de Angular: {{ b.created_at | date:'dd/MM/yyyy HH:mm' }}.

7. Propuesta de ampliación del proyecto

Hopink Studio, en su estado actual, constituye una plataforma web funcional que permite visualizar productos, gestionar carritos de compra, solicitar presupuestos personalizados y operar como punto de contacto entre el cliente y el estudio de tatuajes y piercings. Sin embargo, al tratarse de una versión local, se encuentra en una etapa inicial que sienta las bases técnicas y conceptuales para una futura evolución. Por ello, se plantean diversas líneas de ampliación orientadas a convertir el proyecto en un servicio web profesional y completo.

Una primera ampliación fundamental sería la publicación online del proyecto. Esto implicaría contratar un servicio de hosting con soporte para PHP y MySQL, configurar un dominio propio, implementar un certificado SSL y migrar los archivos del frontend (compilados con Angular) al entorno de producción. Este paso haría posible que cualquier usuario pudiera acceder a la plataforma desde cualquier dispositivo, dotando al proyecto de visibilidad real y permitiendo validar su utilidad desde el punto de vista comercial.

Junto a esta publicación, se contempla la integración de una pasarela de pago real, como Stripe o PayPal. Actualmente, el sistema simula la finalización de compras para efectos de prueba, pero no existe una transacción económica real, sino que utiliza una SandBox. Incluir este servicio no solo ampliaría la funcionalidad del proyecto, sino que lo convertiría en una tienda completamente operativa, permitiendo monetizar los productos ofrecidos.

Otra ampliación relevante es la implementación de un panel de administración protegido para usuarios con permisos especiales (por ejemplo, empleados o gestores del estudio). Este panel podría incluir gestión de stock, actualización de precios, alta y baja de productos, visualización de presupuestos y pedidos, así como análisis de comportamiento del usuario (productos más vistos, ventas mensuales, etc.).

En cuanto a los usuarios, se propone una ampliación del sistema de perfiles, permitiendo categorías como cliente, administrador y artista. Esto permitiría crear un área específica para artistas donde podrían mostrar su portafolio, gestionar sus citas o responder solicitudes de presupuesto.

A nivel de experiencia de usuario, se pueden incorporar mejoras como:

- ➔ Sistema de comentarios y valoraciones para productos y servicios.
- ➔ Sugerencias personalizadas en función del historial de navegación o compra.
- ➔ Integración de un blog o sección informativa.
- ➔ Módulo de contacto en tiempo real mediante chat.
- ➔ Optimización para dispositivos móviles.
- ➔ Posibilidad de registrar una cita o pedir presupuesto desde una app.

Finalmente, si el proyecto llegase a consolidarse, se podría integrar con soluciones de envío como Correos o MRW para automatizar la logística y el seguimiento de pedidos, e incluso sincronizar con sistemas ERP para controlar inventario y contabilidad. Todas estas ampliaciones permitirían posicionar Hopink Studio como una plataforma integral en el sector del tatuaje y el piercing, y podría llegar a funcionar como un marketplace donde diferentes estudios o artistas vendan productos y servicios a través de un sistema centralizado.

8. Estudio de viabilidad del proyecto

- ➔ Descripción de la idea de negocio

Hopink Studio es una plataforma de comercio electrónico orientada a un nicho específico dentro del mercado de la belleza y el arte corporal: los productos, servicios y profesionales del tatuaje y del piercing. Su propuesta de valor radica en facilitar la adquisición de insumos especializados (agujas, tintas, cremas, sprays, etc.), mostrar el portafolio de artistas del

estudio y permitir la solicitud de presupuestos personalizados, todo desde un entorno digital accesible y moderno.

El crecimiento sostenido de este sector, junto con la alta digitalización de su público objetivo, hacen de esta iniciativa una idea de negocio con potencial de penetración, fidelización y especialización. Además, el hecho de ser un e-commerce de nicho le da ventajas competitivas frente a otras plataformas más generales, como Amazon o eBay, que no están enfocadas a este segmento.

➔ Propuesta de forma jurídica

En una primera fase, lo más viable es que la explotación comercial del proyecto se realice bajo la figura de autónomo. Esto permite agilidad de gestión, menor carga burocrática y flexibilidad operativa. Posteriormente, si el volumen de operaciones y facturación lo requieren, podría migrarse a una Sociedad Limitada, lo cual reduciría la responsabilidad patrimonial directa, facilitaría la contratación de personal y permitiría el acceso a líneas de crédito o inversión.

Cualquiera de estas figuras puede operar con un CNAE relacionado con comercio al por menor de productos estéticos y artísticos o con actividades de venta online, lo que encaja con el objeto de Hopink Studio. A nivel legal, también se requerirá una adecuada redacción de los términos y condiciones, políticas de devolución y privacidad, especialmente si se integran pagos y recogida de datos personales.

➔ Plan de recursos humanos

Durante la fase de desarrollo, el proyecto puede ser mantenido por una sola persona o un pequeño equipo técnico. Sin embargo, para una fase comercial completa, se propone una estructura funcional mínima:

- ❖ Un desarrollador o equipo de mantenimiento técnico que se encargue de la seguridad, nuevas funciones y estabilidad de la plataforma.
- ❖ Un gestor de producto y marketing, encargado de definir campañas, mantener redes sociales, generar contenido y analizar el mercado.
- ❖ Un diseñador o ilustrador para la creación de gráficos promocionales, banners y adaptaciones de imagen corporativa.
- ❖ Personal de atención al cliente para resolver incidencias y asesorar en compras y presupuestos.
- ❖ Encargado de logística o almacén, si se gestiona stock propio.

Adicionalmente, si se vincula con un estudio físico real, se pueden integrar tatuadores y perforadores que interactúen directamente desde la web, recibiendo citas, respondiendo presupuestos o publicando contenido.

En caso de que la tienda comience a crecer en ventas, se podría considerar la subcontratación de servicios externos como logística, diseño web, SEO o asesoría fiscal, optimizando la operativa sin necesidad de ampliar en exceso la plantilla inicial.

9. Conclusiones

El desarrollo de Hopink Studio ha supuesto la creación de un sistema web completo, con funcionalidades integradas que permiten tanto la exploración de productos como la gestión de compras, presupuestos y cuentas de usuario. Se ha llevado a cabo utilizando tecnologías modernas y estándares de buenas prácticas, como el uso de Angular para la interfaz, PHP como motor de servidor, y MySQL como sistema de base de datos.

Desde su planteamiento inicial, el proyecto se ha centrado en ofrecer una experiencia fluida y adaptada al público objetivo. Cada funcionalidad ha sido diseñada con lógica y usabilidad: el carrito de compras mantiene el estado incluso si se cierra el navegador; los presupuestos se calculan automáticamente en base a reglas reales del sector; el sistema de login permite diferenciar entre usuarios registrados e invitados, conservando funcionalidades clave como el carrito.

Uno de los logros principales del desarrollo ha sido integrar correctamente un sistema de comunicación entre frontend y backend, superando los retos habituales de CORS, sesiones y seguridad de datos. A nivel visual, se ha procurado mantener una estética coherente con el sector del tatuaje, apoyada en una navegación clara, responsiva y enfocada a la conversión.

El sistema ya es operativo en local, y está preparado para ser desplegado en un entorno real con muy pocos ajustes. Las futuras ampliaciones permitirán transformar este prototipo funcional en una plataforma viva, conectada con clientes reales y adaptada a los requerimientos del día a día de un estudio profesional.

En definitiva, Hopink Studio demuestra que es posible desarrollar una tienda especializada, escalable, personalizable y con funcionalidades reales desde cero, con una estructura técnica bien definida y una clara orientación a negocio.

10. Bibliografía

Documentación oficial de Angular: https://angular.io/docs

Manual de PHP: https://www.php.net/manual

Guía de MySQL: https://dev.mysql.com/doc/

Bootstrap CSS Framework: https://getbootstrap.com

Curso Angular 19 & PHP de NorthStar

Apuntes de clase y recursos de formación profesional

Documentación de Paypal: https://developer.paypal.com/home/