UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Transformación Digital de Peluquerías: Implementación de un Sistema TPV para Gestión de Servicios, Productos y Clientes

Trabajo Fin de Grado

Grado en Tecnología Digital y Multimedia

AUTOR/A: Parcet Torre, Raquel Belén

Tutor/a: Arce Vila, Pau

Cotutor/a: García Valls, María Soledad

CURSO ACADÉMICO: 2023/2024

Resumen

En la actualidad, con el continuo desarrollo de la digitalización en todos los ámbitos de la vida, existe una clara necesidad de que los negocios tradicionales se adapten y modernicen para seguir siendo competitivos y eficientes. En este sentido, el presente Trabajo Final de Grado tiene como objetivo principal desarrollar una aplicación web orientada a la transformación digital de una peluquería.

Se propone la creación de una plataforma que, considerando la importancia de la digitalización en la optimización de procesos y la mejora de la experiencia del cliente, permita llevar un registro completo de los servicios ofrecidos, la gestión de clientes y la facturación de manera integrada y eficiente, con el objetivo de agilizar sus tareas diarias y optimizar la gestión de recursos.

La aplicación web se desarrollará teniendo en cuenta las necesidades específicas de este sector, así como las tendencias tecnológicas aprendidas a lo largo del grado. Se buscará implementar funcionalidades que permitan una administración intuitiva, sencilla y eficaz de los servicios ofrecidos, la base de datos de clientes y el registro de transacciones comerciales. Para ello, se utilizarán tecnologías web como HTML, CSS y JavaScript para la interfaz de usuario, mientras que Docker facilitará la gestión del servidor. Además, se emplearán bases de datos SQL para almacenar la información necesaria de manera segura y eficiente.

Palabras clave: transformación digital; aplicación web; peluquería; gestión de servicios; clientes; facturación; HTML; CSS; JavaScript; Docker; SQL.

Resum

En l'actualitat, amb el continu desenvolupament de la digitalització en tots els àmbits de la vida, existix una clara necessitat que els negocis tradicionals s'adapten i modernitzen per a continuar sent competitius i eficients. En este sentit, el present Treball Final de Grau té com a objectiu principal desenvolupar una aplicació web orientada a la transformació digital d'una perruqueria.

Es proposa la creació d'una plataforma que, considerant la importància de la digitalització en l'optimització de processos i la millora de l'experiència del client, permeta portar un registre complet dels servicis oferits, la gestió de clients i la facturació de manera integrada i eficient, amb l'objectiu d'agilitzar les seues tasques diàries i optimitzar la gestió de recursos.

L'aplicació web es desenvoluparà tenint en compte les necessitats específiques d'este sector, així com les tendències tecnològiques apreses al llarg del grau. Es buscarà implementar funcionalitats que permeten una administració intuïtiva, senzilla i eficaç dels servicis oferits, la base de dades de clients i el registre de transaccions comercials. Per a això, s'utilitzaran tecnologies web com a HTML, CSS i JavaScript per a la interfície d'usuari, mentres que Docker facilitarà la gestió del servidor. A més, s'empraran bases de dades SQL per a emmagatzemar la informació necessària de manera segura i *eficiente.

Paraules clau: transformació digital; aplicació web; perruqueria; gestió de servicis; clients; facturació; HTML; CSS; JavaScript; Docker; SQL.

Abstract

Currently, with the ongoing development of digitization in all aspects of life, there is a clear need for traditional businesses to adapt and modernize to remain competitive and efficient. In this regard, the present Final Degree Project aims to develop a web application focused on the digital transformation of a hair salon.

The creation of a platform is proposed that, considering the importance of digitization in optimizing processes and improving the customer experience, allows for a comprehensive record of services offered, customer management, and integrated and efficient billing, with the aim of streamlining daily tasks and optimizing resource management.

The web application will be developed considering the specific needs of this sector, as well as the technological trends learned throughout the degree. The aim is to implement functionalities that enable an intuitive, simple, and effective management of the services offered, the customer database, and the recording of commercial transactions. To achieve this, web technologies such as HTML, CSS, and JavaScript will be used for the user interface, while Docker will facilitate server management. Additionally, SQL databases will be employed to store the necessary information securely and efficiently.

Keywords: digital transformation; web application; hair salon; service management; customers; billing; HTML; CSS; JavaScript; Docker; SQL.

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

RESUMEN EJECUTIVO

La memoria del TFG del Grado en Tecnología Digital y Multimedia debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la tecnologías digitales y multimedia

CONCEPT (ABET)	**CONCEPTO (traducción)**	**¿Cumple? (S/N)**	**¿Dónde? (páginas)**
1. IDENTIFY:	1. IDENTIFICAR:		
1.1. Problem statement and opportunity	1.1. Planteamiento del problema y oportunidad	S	1-2
1.2. Constraints (standards, codes, needs, requirements & specifications)	1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones)	S	5-6
1.3. Setting of goals	1.3. Establecimiento de objetivos	S	2-3
2. FORMULATE:	2. FORMULAR:		
2.1. Creative solution generation (analysis)	2.1. Generación de soluciones creativas (análisis)	S	7-12
2.2. Evaluation of multiple solutions and decision-making (synthesis)	2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)	S	12
3. SOLVE:	3. RESOLVER:		
3.1. Fulfilment of goals	3.1. Evaluación del cumplimiento de objetivos	S	52
3.2. Overall impact and significance (contributions and practical recommendations)	3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)	S	52

ÍNDICE

Capítulo 1. INTRODUCCIÓN Y CONTEXTO *1*

1.1 Introducción **1**

1.2 Motivación **1**

1.3 Objetivo **2**

1.4 Estado del arte **3**

Capítulo 2. METODOLOGÍA *4*

2.1 Metodología elegida. **4**

2.2 Fases **4**

2.3 Requisitos funcionales **5**

2.4 Requisitos de diseño **6**

Capítulo 3. TECNOLOGÍAS DISPONIBLES *7*

3.1 Tecnologías *front-end* **7**

3.1.1 HTML y CSS 7

3.1.2 JavaScript 7

3.1.3 Bibliotecas/Librerías de JavaScript 7

3.1.4 Frameworks 8

3.1.5 Comparación y elección 8

3.2 Tecnologías *back-end* **8**

3.2.1 Node.js 8

3.2.2 Django 9

3.2.3 Flask 9

3.2.4 FastAPI 9

3.2.5 Comparación y elección 9

3.3 Base de datos **9**

3.4 Otras herramientas empleadas **10**

3.4.1 GitHub – GitHub Desktop 10

3.4.2 Canva 11

3.4.3 Docker 12

3.4.4 Heidi SQL 12

3.4.5 Jinja2..12

3.5 Funcionamiento 12

Capítulo 4. DISEÑO BASE DE DATOS Y CREACIÓN DEL DOCKER 13

4.1 Estructura y definición de base de datos necesaria 13

4.1.1 Definición base de datos..13

4.1.2 Esquema base de datos...16

4.2 Creación del Docker para almacenar la base de datos 16

Capítulo 5. DISEÑO Y DESARROLLO DEL FRONT-END 18

5.1 Diseño de la interfaz.. 18

5.1.1 Paleta de colores y tipografía..18

5.1.2 Logotipo...19

5.1.3 Prototipo...19

5.1.3.1 Bonos 20

5.1.3.2 Mantenimiento...21

5.1.3.3 Caja 23

5.1.4 Flujo de navegación 24

5.2 Desarrollo del *front-end* 25

5.2.1 Header, botón atrás, logotipo y botón logout..25

5.2.2 Grupo 1 – Páginas de navegación 28

5.2.3 Grupo 2 – Páginas para interactuar con la base de datos 30

5.2.3.1 Funciones para crear, editar o eliminar...34

5.2.4 Grupo 3 – Visualización de datos en un periodo de tiempo 34

5.2.5 Grupo 4 – Páginas independientes 36

5.2.6 Conexión *front-end* y *back-end* 37

Capítulo 6. LÓGICA Y DESARROLLO DEL BACK-END 38

6.1 Estructura del proyecto 38

6.2 Configuración del proyecto 38

6.2.1 Modelos y esquemas...38

6.2.2 Conexión a la base de datos 42

6.2.3 Autenticación y autorización...42

6.2.4 Envío de correos electrónicos.....43

6.2.5 Funciones CRUD.....45

6.3 Implementación de *endpoints*.....49

6.3.1 Configuración de la API.....50

6.3.2 *Endpoints* creados.....50

6.3.2.1 *Endpoints* autenticación.....52

6.3.2.2 *Endpoints* páginas protegidas.....54

Capítulo 7. CONCLUSIONES Y LINEAS FUTURAS.....55

7.1 Conclusiones.....55

7.2 Líneas futuras.....55

Capítulo 8. REFERENCIAS.....57

ÍNDICE DE FIGURAS

Figura 1. Captura de pantalla interfaz gráfica GitHub Desktop 10
Figura 2. Boceto página "Mantenimiento" 11
Figura 3. Boceto página "Clientes" 11
Figura 4. Boceto página "Caja" 11
Figura 5. Boceto página "Caja formato ticket" 11
Figura 6. Esquema base de datos 16
Figura 7. Code snippet comandos empleados para creación del contenedor en Docker 16
Figura 8. Datos introducidos en HeidiSQL 17
Figura 9. Paleta de colores empleada 18
Figura 10. Tipografía Figtree de Google Fonts 18
Figura 11. Code snippet fuente embebida en HTML 18
Figura 12. Code snippet definición de fuente en CSS 18
Figura 13. Logotipo aplicación 19
Figura 14. Formulario página principal, inicio de sesión 19
Figura 15. Página principal 20
Figura 16. Botones página bonos 20
Figura 17. Calendarios para seleccionar periodo 20
Figura 18. Ticket de caja total en un periodo de tiempo establecido 20
Figura 19. Ticket de caja total por empleados en un periodo de tiempo 20
Figura 20. Captura de cómo se ve el PDF que descargamos del ticket 21
Figura 21. Botones página mantenimiento 21
Figura 22. Estilo formulario/modal común para todas las páginas 21
Figura 23. Página principal de clientes 22
Figura 24. Ejemplo de visualización de datos. 22
Figura 25. Vista página empleados con empleados activos 22
Figura 26. Vista página empleados con un empleado inactivo 22
Figura 27. Página servicios 23
Figura 28. Página productos 23
Figura 29. Página caja 23
Figura 30. Formulario para seleccionar datos asignados al servicio/producto 24
Figura 31. Formulario para seleccionar método de pago y enviar correo. 24
Figura 32. Email con ticket recibido 24
Figura 33. Contenido del email de ticket. 24

Figura 34. Diagrama de rutas 25
Figura 35. Estructura de carpetas front-end 25
Figura 36. Ejemplo de header común en las templates 26
Figura 37. Code snippet de botón atrás y logotipo 27
Figura 38. Code snippet función goBack() 27
Figura 39. Code snippet del archivo logout.js 27
Figura 40. Code snippet de la estructura común en las páginas del grupo 1. 28
Figura 41. Code snippet ejemplo de CSS 1 28
Figura 42. Code snippet ejemplo CSS 2 28
Figura 43. Ejemplo de visualización con CSS aplicado 29
Figura 44. Ejemplo de HTML con "id" asignados a los botones 29
Figura 45. Ejemplo de JS, cómo se accede a los botones con su "id" y agregar el evento 29
Figura 46. Esquema funcionamiento visualización de Servicios en la BBDD. 30
Figura 47. Code snippet código Jinja2 para visualizar datos de la BBDD. 30
Figura 48. Code-snippet ejemplo de modal para crear un nuevo servicio 31
Figura 49. Code snippet función abrirModalEditar(id) 31
Figura 50. Datos del símbolo para cerrar 32
Figura 51. Code snippet función para ordenar alfabéticamente 32
Figura 52. Code snippet función buscar 33
Figura 53. Code snippet función autocompletado 33
Figura 54. Code snippet función fetchData 35
Figura 55. Code snippet calendarios definidos en HTML 36
Figura 56. Code snippet definición calendarios con librería flatpickr en JavaScript 36
Figura 57. Estructura archivos back-end 38
Figura 58. Code snippet clase cliente 40
Figura 59. Code snippet clase cita 40
Figura 60. Code snippet ejemplo schema Cita 40
Figura 61. Code snippet database.py 42
Figura 62. Esquema traducido de [35] 43
Figura 63. Code snippet email_config.py 44
Figura 64. Code snippet conexión con el servidor SMTP 45
Figura 65. Code snippet ejemplo función para recuperar un cliente por su ID 45
Figura 66. Code snippet ejemplo función para recuperar lista clientes 45
Figura 67. Code snippet ejemplo función crear nuevo cliente. 46
Figura 68. Code snippet ejemplo función para actualizar un cliente 46

Figura 69. Code snippet ejemplo función para eliminar un producto 47
Figura 70. Code snippet ejemplo función para obtener un cliente por nombre y apellidos 47
Figura 71. Code snippet función ejemplo añadir producto a ticket 47
Figura 72. Code snippet comandos para configurar la API 50
Figura 73. Code snippet ejemplo de endpoint 52
Figura 74. Diagrama de flujo endpoint POST /login 52
Figura 75. Diagrama de flujo endpoint POST logout 53
Figura 76. Diagrama de flujo endpoint GET login 53
Figura 77. Code snippet ejemplo de endpoint protegido 54

Capítulo 1. INTRODUCCIÓN Y CONTEXTO

1.1 Introducción

En el contexto actual, la digitalización ha dejado de ser una opción para las pymes (Pequeña y Mediana empresa), y se ha convertido en una necesidad marcada para su crecimiento y supervivencia. La digitalización consiste en el proceso de convertir procesos físicos y analógicos en digitales, haciendo uso de tecnologías que optimizan la eficiencia operativa, mejoran la experiencia del cliente y también, abren nuevas oportunidades de negocio.

Las pymes representan un porcentaje considerado del sector empresarial global y son fundamentales para el desarrollo económico. En España, según datos del Ministerio de industria, comercio y turismo, las pymes suponen el 99.84% de las empresas y conforman una parte muy importante de la economía de los 28 países de la Unión Europea [1].

Según las estadísticas de la Eurostat, la UE ha fijado dos objetivos principales para la transformación digital de las empresas de aquí a 2030, más del 90% deben alcanzar al menos un nivel básico de transformación digital y el 75% de las empresas deben utilizar servicios de computación en la nube, realizar análisis de macrodatos o utilizar inteligencia artificial. En España, en 2022, el 31.7% de las empresas alcanzó un nivel de transformación muy bajo y un 40% un nivel de transformación bajo [2].

En España, desde el Ministerio de Economía, Comercio y Empresa, se ha lanzado la *Agenda España Digital*, la hoja de ruta para la transformación digital del país. Así pues, uno de los planes y estrategias es el *Plan de Impulso a la Digitalización de pymes*. El objetivo es acelerar la digitalización de las pymes mediante la adopción de nuevas tecnologías y el fomento del emprendimiento digital [3].

1.2 Motivación

Teniendo en cuenta la situación mencionada en el punto anterior, nace la idea de este proyecto final de grado. Ayudar a la transformación digital de mi madre y su peluquería. La empresa ha ofrecido su servicio desde 2013, se trata de un pequeño negocio familiar, y desde entonces, emplea un sistema para administrar las citas, clientes, productos, servicios, caja diaria, entre otras. A pesar de que hasta la fecha funciona de manera correcta, se ha quedado anticuado y presenta una gran cantidad de funciones innecesarias.

Cuando tuve que elegir de qué trataría mi proyecto, tenía claro que quería realizar algo útil, provechoso y efectivo, así que gracias a mi conexión con la necesidad que presentaba el negocio familiar, se me planteó la posibilidad de llevar a cabo este trabajo con las necesidades particulares y personalizadas para lo que esta peluquería necesita.

1.3 Objetivo

El objetivo de este trabajo es desarrollar desde cero una aplicación web de tipo plataforma TPV (Terminal Punto de Venta), optimizando los procesos operativos y permitiendo una gestión integral y eficiente de los servicios ofrecidos, la base de datos de clientes y la facturación. Esto permitirá agilizar los procesos de la empresa, así como actualizarse digitalmente.

El proyecto se centrará en el desarrollo *full-stack* de la aplicación, abarcando tanto la parte del cliente y diseño (*front-end*) como la parte del servidor y lógica (*back-end*).

Se llevará a cabo un estudio preliminar de las diferentes tecnologías disponibles en la actualidad, y se compararán, para después elegir las más adecuadas para los requisitos del proyecto.

Las tareas a realizar para alcanzar el objetivo del proyecto son las siguientes:

- Diseñar y desarrollar una aplicación web para una peluquería que agilice los procesos de administración y gestión de esta.
- Recopilar las funcionalidades necesarias que debe tener la aplicación para que su interfaz sea simple y eficaz.
- Crear una base de datos que recopile: clientes, empleados, productos, servicios, citas de los clientes y cantidad monetaria de la caja diaria.
- Hacer que la aplicación interactúe con la base de datos.
- Hacer que la aplicación sea fácil de usar.
- Estudiar las diferentes tecnologías actuales para elegir la que más se adecúe a las necesidades del proyecto.

Queremos una aplicación web en la que podamos administrar una base de datos de clientes, recopilando en ella como ficha los datos del cliente, así como los tratamientos y servicios que se ha realizado, así como sus datos de contacto. La base de datos también recopilará los empleados que tiene la pyme, los servicios prestados e información relacionada y los productos que están a la venta. También dispondrá de un sistema que recoja a modo de “ticket” cada venta que hagamos y por quién ha sido realizada (como venta entendemos, cuando un cliente ha ido a la peluquería y se ha hecho una serie de servicios), esto quedará reflejado de manera que al final del día, podamos hacer una recopilación de la caja total, o también la caja total de un periodo de tiempo.

En conclusión, queremos buscar las tecnologías adecuadas para la correcta creación y desarrollo de la aplicación, para dar un paso en el futuro y entrar en el porcentaje de pymes digitalizadas dentro del plan estratégico propuesto en España.

1.4 Estado del arte

En este apartado se lleva a cabo la revisión de herramientas y softwares existentes. Actualmente existen diversas tecnologías utilizadas para gestionar todo lo expuesto previamente, la mayoría de ellas son de carácter general y algunas pueden ser complejas e incluir características adicionales.

Algunos softwares existentes:

- **SolverMedia Peluquería** [4]: es un sistema TPV muy completo para la gestión de peluquerías. Para salones de belleza grandes esta sería la opción de trabajo ideal. Es un software propietario que se instala localmente y requiere la compra de una licencia para su uso, pero no es necesario comprar un paquete de hardware específico para utilizarlo. Para nuestro objetivo, resulta complejo y poco intuitivo, tiene muchas características que realmente no se utiliza en nuestra pyme, como por ejemplo la agenda de citas, control de deudas y análisis varios entre otros.
- **Shortcuts Software de Gestión para Peluquerías** [5]: Se trata de una herramienta para gestionar todas las áreas de la peluquería. Es una solución en la nube que se ofrece bajo un modelo de suscripción, lo que te permite acceder al software desde cualquier lugar y dispositivo sin necesidad de comprar hardware específico. Nuevamente, se trata de una interfaz con muchas funciones, lo que lo hace poco intuitivo para aquellos que no tienen un gran dominio de las tecnologías.

En resumen, si revisamos los softwares existentes, encontraremos una gran cantidad de ellos que cumplen la mayoría de nuestros requisitos, pero todavía hay algunas áreas que no están abordadas, como pueden ser las fichas técnicas de los clientes; o áreas que, en nuestra empresa, no nos sirven, como puede ser la agenda virtual.

Al fin y al cabo, nuestra pyme es una peluquería en un pueblo, un pequeño negocio cuyos empleados necesitan un software personalizado y fácil de utilizar, y eso es lo que se pretende conseguir en este proyecto.

Capítulo 2. METODOLOGÍA

2.1 Metodología elegida.

En el campo de la programación, se utilizan metodologías de desarrollo de software [6], entre otras cosas, para facilitar el trabajo en equipo. Es el conjunto de técnicas y métodos que se utilizan para diseñar una solución. Actualmente, es indispensable trabajar con una metodología, es cuestión de organización. Asimismo, usar una metodología también nos sirve para controlar el desarrollo del trabajo, así podremos minimizar los márgenes de error y anticiparse a las situaciones.

En el ámbito de este trabajo, dado que es realizado por una única persona, resulta complejo emplear una única metodología, lo que implica que la metodología sea una mezcla de varias. Partimos de la premisa que, al utilizar metodologías *ágiles,* se adopta un enfoque cuya base está definida por una metodología tradicional, la metodología *Incremental* [7]. Esta consiste en construir el proyecto de manera progresiva, añadiendo funcionalidades en cada etapa, lo que permite visualizar los resultados de una forma más rápida y permite utilizar el software sin que esté completamente finalizado. Este método sirve como base para las siguientes dos metodologías *ágiles* que en este trabajo se combinan: *Kanban* y *Scrum*.

El método *Kanban*, consiste en tener tareas pequeñas y organizarlas en pendientes, en curso y finalizadas, mientras que *Scrum* se divide en bloques de tiempos cortos y fijos para conseguir un resultado completo de cada bloque, cada bloque se denomina sprint [7]

En este trabajo, he asumido la responsabilidad total de todas las fases y componentes del proyecto. Esto incluye desde la captación de requisitos, diseño de la aplicación hasta la programación, lo que, en un proyecto típico desarrollado por un equipo, cada una de estas fases o componentes estarían asignadas a una persona especializada en cada ámbito.

2.2 Fases

Podemos definir las siguientes fases del proyecto:

Fase 1 – Captación de los requisitos y funcionalidades necesarias que la aplicación debe tener.

Fase 2 – Diseñar un boceto/prototipo en canva, del resultado que queremos obtener.

Fase 3 – Estudio de las tecnologías disponibles para la creación y desarrollo de la aplicación para tomar una decisión de cuáles usar basándonos en el prototipo que queremos conseguir.

Fase 4 – Programar la aplicación.

Fase 5 – Probar la aplicación.

2.3 Requisitos funcionales

Estos son los requisitos que la aplicación web debe tener:

REQ 1. El programa debe estar dividido en 3 funciones básicas, caja, mantenimiento y bonos.

REQ 2. El programa debe tener un sistema de inicio de sesión, con un correspondiente usuario y contraseña, dónde algunos usuarios tendrán la condición "administrador" para tener acceso a todas las páginas y funciones, mientras que los usuarios con la condición "empleado" tendrán el acceso restringido.

Mantenimiento

REQ 3. El sistema debe almacenar en una base de datos los clientes de la peluquería, pudiendo crear, editar o eliminar.

REQ 4. El sistema debe almacenar para cada cliente, a modo de registro, las citas que han tenido con su correspondiente fecha y descripción.

REQ 5. El sistema debe almacenar en una base de datos los empleados de la peluquería, pudiendo crear o editar los empleados. Esta página estará restringida, únicamente los administradores podrán acceder.

REQ 6. El sistema debe almacenar en una base de datos los productos a la venta disponibles en la peluquería, pudiendo crear, editar o eliminar.

REQ 7. El sistema debe almacenar en una base de datos los servicios que se ofrecen en la peluquería, pudiendo crear, editar o eliminar.

Caja

REQ 8. El sistema debe tener una pantalla principal dónde podremos hacer un seguimiento de todos los "ticket" que se realizan a lo largo de la jornada, esta pantalla debe tener la opción de añadir servicios y productos al "ticket", eligiendo el empleado que realiza el servicio o la venta, y debe hacer una suma total que equivalga al ticket total.

REQ 9. El sistema debe dejar marcar si el "ticket" ha sido pagado con efectivo o tarjeta, y enviar un correo al cliente si lo desea con el ticket detallado.

REQ 10. En esta pantalla del sistema, debe ser posible la edición de los precios de manera "extraordinaria", (ejemplo: la cantidad de pelo de un cliente es mayor a lo normal, de manera que un tintado de pelo sube el precio por haber usado más producto).

REQ 11. El sistema debe almacenar en una base de datos el monto total de dinero efectuado cada día.

Bonos

REQ 12. El acceso a esta página debe estar restringido, solo los administradores podrán acceder.

REQ 13. Debe ser posible seleccionar la caja total realizada en un día, o en un periodo de tiempo.

REQ 14. Debe ser posible visualizar el monto total realizado por un empleado en un día o en un periodo de tiempo.

2.4 Requisitos de diseño

A la hora de crear una aplicación web, se debe tener en cuenta a los usuarios que en un futuro utilizarán el sistema. De manera que debemos elegir una tipografía correcta y estilos que sean simples, intuitivos y visuales.

REQ 1. El estilo general de la aplicación será minimalista, simple y visual.

REQ 2. Se debe utilizar una paleta de colores pastel, y únicamente el mismo tono.

REQ 3. No se sobrecargará las pantallas con demasiado texto, solo las palabras clave y necesarias para entender lo que hace cada parte.

REQ 4. Se debe utilizar la misma fuente para todo el proyecto, legible y elegante para que dé sensación de limpieza.

REQ 5. Se debe utilizar la misma temática para todo el proyecto.

REQ 6. En principio debe estar pensado para ser responsivo en pantallas de ordenador.

Capítulo 3. TECNOLOGÍAS DISPONIBLES

En el ámbito del desarrollo de aplicaciones web, la selección adecuada de tecnologías tiene un papel fundamental a la hora de construir un sistema que se adhiera a nuestros requisitos. En este apartado abordaremos las diversas opciones tecnológicas disponibles tanto para *front-end* como *back-end*. Analizaremos detalladamente las características y ventajas de las tecnologías disponibles actualmente.

3.1 Tecnologías *front-end*

Las tecnologías *front-end* son fundamentales para la creación de interfaces de usuario interactivas y atractivas. Entre las principales tecnologías disponibles se encuentran:

3.1.1 HTML y CSS

HTML (HyperText Markup Language) es el estándar para la estructura y el contenido de las páginas web [8], mientras que CSS (Cascading Style Sheets) se utiliza para definir su apariencia visual, nos describe como deben mostrarse los documentos HTML [9]. Estas tecnologías son básicas y esenciales, además de ampliamente usadas debido a su simplicidad y eficacia en la creación de interfaces de usuario.

3.1.2 JavaScript

Es el lenguaje de programación ligero que permite la creación de contenido dinámico e implementación de funciones complejas en las páginas webs. Es un lenguaje de programación basada en prototipos, dinámico y con soporte para programación orientada a objetos. Nos permite controlar multimedia, animar imágenes… A menudo se suele confundir con el lenguaje de programación Java, pero son muy diferentes [10]

3.1.3 Bibliotecas/Librerías de JavaScript

Las librerías de JavaScript, se trata de código reutilizable que facilita el desarrollo de sitios web y aplicaciones con características y funcionalidades variadas [11]

Una de las más famosas es React [12], es una biblioteca de JavaScript desarrollada por Facebook, para construir interfaces de usuario a partir de piezas individuales conocidas como componentes, y una gran ventaja a destacar es la capacidad de crear componentes reutilizables.

En este contexto también se valorará el uso de la librería Flatpickr [13], está dedicada a la creación de *datepickers* (elementos que ayudan al usuario a seleccionar una fecha de forma sencilla a través de un calendario visual). Es ligero, potente y ágil.

Otra librería para tener en cuenta es jsPDF, es una herramienta creada por Parallax, la agencia digital. Ésta nos permite la posibilidad de descargar un archivo PDF con información específica [14].

3.1.4 Frameworks

Existen una gran cantidad de ellos que facilitan el desarrollo *front-end*, algunos de los más famosos son: Angular y Bootstrap.

Angular [15] es un framework de JavaScript de código abierto desarrollado por Google, ofrece una estructura completa para el desarrollo de aplicaciones web robustas. Se creó para simplificar el desarrollo y las pruebas de las aplicaciones web con un marco de trabajo para las arquitecturas del lado del cliente.

Bootstrap [16] es un framework CSS desarrollado por Twitter para *front-end* utilizado para desarrollar aplicaciones web y móviles. Combina CSS y JavaScript para estilizar elementos de una página HTML, proporciona interactividad y presenta una serie de componentes que facilitan la comunicación con el usuario. Su principal objetivo es permitir construir sitios web responsive para dispositivos móviles.

3.1.5 Comparación y elección

He optado por emplear HTML, CSS y JavaScript puro en el desarrollo de *front-end*. HTML y CSS me proporcionan simplicidad y control directo sobre la estructura y estilo de la página, permitiéndome personalizar la interfaz sin necesidad de aprender la sintaxis y convenciones de un framework. JavaScript puro es suficiente para la interactividad específica de este proyecto, ya que la simplicidad de la aplicación no hace necesario el requerimiento adicional que frameworks como Bootstrap o Angular podrían aportar. Pero sí haremos uso de dos librerías, flatpickr para añadir calendarios visuales e intuitivos, y jsPDF para descargar PDF de partes y datos específicos de la aplicación.

3.2 Tecnologías *back-end*

El *back-end* es muy importante para el procesamiento de datos, la lógica y la comunicación con las bases de datos. Entre las tecnologías más relevantes se encuentran:

3.2.1 Node.js

Se trata de un entorno de ejecución para JavaScript del lado del servidor (permite ejecutar código JavaScript fuera de un navegador web). Este entorno nos brinda la posibilidad de desarrollar aplicaciones escalables y rápidas. Se le conoce por su capacidad para manejar operaciones asíncronas y su versatilidad para funcionar en múltiples plataformas [17].

3.2.2 Django

Es un framework web de alto nivel de desarrollo web para Python, gratuito y de código abierto, que ofrece una arquitectura robusta y una gran variedad de herramientas integradas para el desarrollo rápido de aplicaciones seguras, escalables y versátiles [18].

3.2.3 Flask

De nuevo, se trata framework para Python, que se caracteriza por ser más ligero y flexible que Django, lo cual resulta ideal para proyectos que requieren una mayor personalización sin la sobrecarga de funcionalidades predefinidas. Se facilita el desarrollo web bajo el patrón MVC (Modelo-Vista-Controlador) [19].

3.2.4 FastAPI

Se trata de un framework moderno para construir APIs con Python, que se basa en los estándares como OpenAPI y JSON schema. Destaca por su rendimiento y su soporte para la validación de datos de tipado estático. Nos permite definir la estructura de la aplicación y cómo todas sus partes se conectarán [20].

3.2.5 Comparación y elección

He optado por la utilización de FastAPI para el desarrollo de *back-end*, debido a diversos motivos. Principalmente porque se trata de uno de los frameworks más rápidos disponibles para Python, con el uso de Python asíncrono y Uvicorn como servidor. Esto hace que pueda construir y mantener la aplicación con agilidad. Es factible llevar a cabo múltiples tareas simultáneamente, dado que trabaja de manera asíncrona y agiliza el flujo de trabajo de desarrollo, lo que posibilita la implementación de aplicaciones robustas y escalables. Es muy fácil de usar, acelera el proceso de desarrollo, reduce la curva de aprendizaje, lo que me permite poner en marcha el proyecto de forma rápida y eficaz. Asimismo, cuenta con una documentación automática, que proporciona instrucciones precisas, y te da acceso a información detallada acerca de cómo usar la API sin necesidad de que escribir extensa documentación.

3.3 Base de datos

Se va a utilizar SQLAlchemy [21] para definir la base de datos, la cual es una biblioteca de SQL para Python que proporciona un ORM (Object-Relational Mapping) y nos brinda un poder completo y flexible de SQL. Nos permite interactuar con bases de datos relaciones de una manera más intuitiva y orientada a objetos.

Otras opciones similares para ORM son Django ORM, que está integrado con el framework Django; y Peewee que es más ligero, pero menos potente que SQLAlchemy.

Utilizaremos SQLAlchemy por su flexibilidad y su amplio soporte para diferentes bases de datos, así como por su capacidad de trabajar de forma eficaz con FastAPI.

En relación con el lenguaje de la base de datos, emplearemos MySQL [22], un sistema de gestión de base de datos relacional, de código abierto desarrollado por Oracle. Este sistema permite almacenar, organizar y recuperar datos de manera eficiente.

Alternativas a esta, son PostgreSQL, SQLite o MongoDB. PostgreSQL [23] es conocido por ser robusto y tener características avanzadas, SQLite [24] por ser ligero y flexible y MongoDB [25]se usa para bases de datos no relacionales ya que utiliza documentos flexibles en lugar de tablas y filas. He optado por usar MySQL por su rendimiento y mi familiaridad previa con esta tecnología, que me ayudará a facilitar el desarrollo y mantenimiento de la base de datos.

3.4 Otras herramientas empleadas

Para el desarrollo del proyecto, se utilizarán una serie de herramientas extras que me ayudarán a conseguir el objetivo esperado:

3.4.1 GitHub – GitHub Desktop

Los proyectos informáticos pueden resultar frágiles si no realizamos copias de seguridad para mantener un orden de la progresión en estos. Por ello, usamos GitHub, que es una plataforma de desarrollo colaborativo que permite alojar proyectos mediante el sistema de control de versiones Git. Facilita el seguimiento de cambios y la gestión de las versiones de código [26].

En este proyecto, se usará GitHub Desktop, que es la interfaz gráfica de usuario de GitHub, lo que hace más intuitivo su uso sin necesidad de usar la línea de comandos. De esta manera, almacenaremos el código fuente y se mantendrá un registro de las diferentes versiones del proyecto en un repositorio. Esto permitirá realizar copias de seguridad regulares, asegurando que el trabajo no se pierda y que siempre haya un historial de cambios accesible.

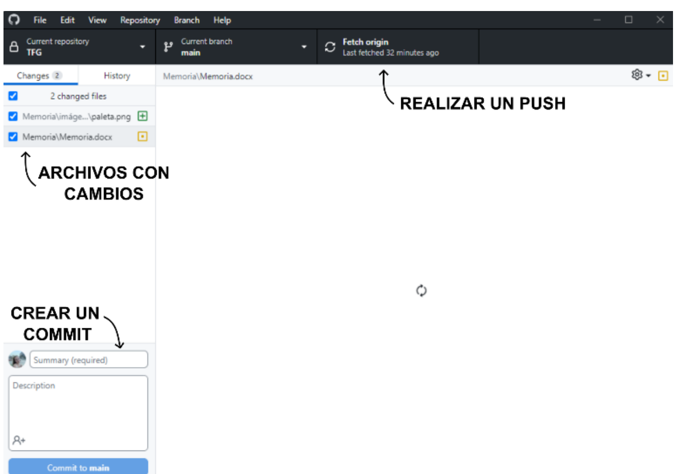


Figura 1. Captura de pantalla interfaz gráfica GitHub Desktop

En GitHub, cuando queremos guardar una versión (los cambios que hemos generado), tenemos que hacer un *commit*, dándole un título y una descripción (opcional). Después para que se guarde y se suba la versión al repositorio, hay que hacer un *push*. GitHub es perfecto para trabajar en equipo, porque puedes crear diferentes ramas para cada miembro del equipo, y los cambios que realice, los puedes descargar realizando lo que es llamado *pull*. En este proyecto no ha hecho falta el uso de ramas, ya que ha sido realizado por una única persona.

3.4.2 Canva

Como apasionada del diseño gráfico, esta es una herramienta muy versátil en línea que nos permite crear gráficos y presentaciones, o contenido visual de forma intuitiva y accesible [27].

En este proyecto, se ha utilizado para elaborar el boceto y prototipo de la aplicación web. Con una maqueta visual que ha servido como guía para el desarrollo, asegurando que el diseño de la interfaz de usuario fuese coherente y atractivo. En ella se incluyó la disposición de los elementos, y cómo debían actuar, proporcionando una visión clara del producto final antes de comenzar con la programación.

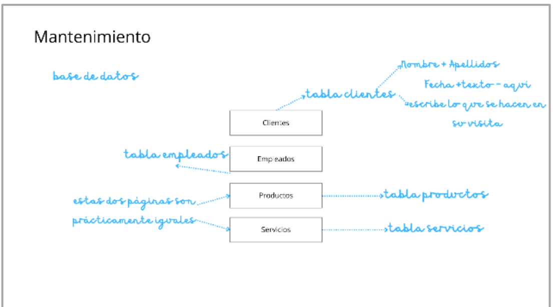


Figura 2. Boceto página "Mantenimiento"

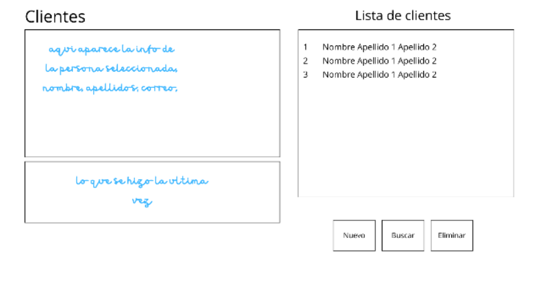


Figura 3. Boceto página "Clientes"

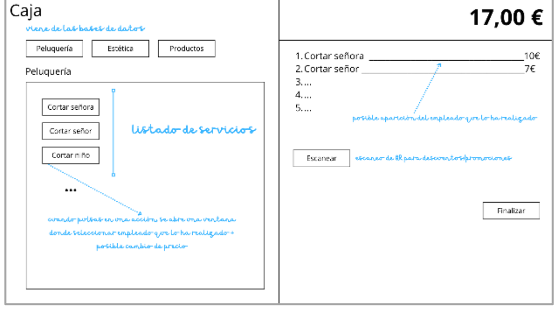


Figura 4. Boceto página "Caja"

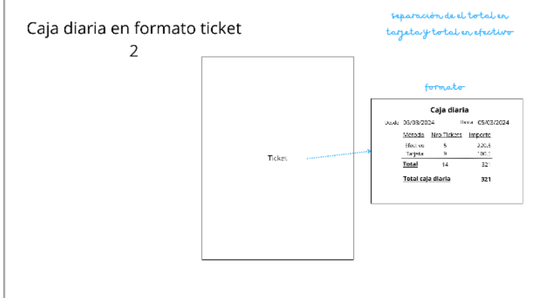


Figura 5. Boceto página "Caja formato ticket"

3.4.3 Docker

Esta es una plataforma que permite empaquetar aplicaciones y sus dependencias en contenedores, asegurando que se ejecuten de manera consistente en cualquier entorno. Divide los procesos y se ejecutan de manera independiente, es decir, diferentes procesos y aplicaciones se ejecutan por separado para que se pueda aprovechar mejor la infraestructura y conservar la seguridad al mismo tiempo [28].

En este proyecto se ha utilizado como contenedor para el servidor de la base de datos. Eso ha permitido aislar la base de datos del entorno de desarrollo, asegurando que las configuraciones y dependencias específicas no interfieran con otros componentes del sistema.

3.4.4 Heidi SQL

Se trata de una herramienta gratuita y de código abierto para la gestión y administración de bases de datos por medio de una interfaz gráfica de usuario. Soporta varios sistemas de gestión de datos [29].

En este proyecto se ha utilizado para visualizar y gestionar la base de datos MySQL del proyecto. Permitiendo inspeccionar las tablas, ejecutar consultas SQL y verificar la integridad de los datos. También ha facilitado la detección y corrección de errores en el código, asegurando que las interacciones de este con la base de datos funcionaran correctamente.

3.4.5 Jinja2

Es un motor de plantillas con todas las funciones para Python, es uno de los motores de plantillas más utilizados. Se inspira en el sistema de plantillas de Djanjo, pero está ampliado ofreciendo un conjunto de herramientas más potente [30].

En este proyecto se ha utilizado para conseguir la visualización requerida en las páginas HTML.

3.5 Funcionamiento

Una vez decididas las tecnologías que voy a emplear, concluimos en cómo debe ser la estructura general del proyecto en: dispondremos de un ordenador que será el servidor, en él estará:

- La base de datos almacenada en un Docker.
- El programa en Python con FastAPI (que tiene acceso a la base de datos), y funcionará como servidor web.
- Los archivos estáticos (HTML, CSS, JS, imágenes…), que serán servidos por el servidor web.

Para usar la aplicación, se dispondrá de un terminal (ordenador o Tablet) que abrirá un navegador y accederá a la web del servidor web en el ordenador previamente ha sido montado.

Capítulo 4. DISEÑO BASE DE DATOS Y CREACIÓN DEL DOCKER

4.1 Estructura y definición de base de datos necesaria

4.1.1 Definición base de datos

Para la base de datos se utilizará MySQL, como he mencionado anteriormente. Primeramente, necesitamos definir las diferentes tablas que necesitaremos, así como las relaciones y los datos que éstas deben tener, y después realizaremos un EER (Diagrama Entidad-Relación Mejorado) para visualizar las relaciones de esta y tenerlo de manera más esquematizada.

Esta base de datos está compuesta por 8 tablas:

La tabla **"Clientes"** está compuesta por 5 columnas. Su función es guardar los datos de los clientes registrados:

- id: entero autonumérico, es el identificador único de cada cliente.
- nombre: texto, almacena el nombre del cliente.
- apellidos: texto, almacena los apellidos del cliente.
- teléfono: texto, guarda el número de teléfono del cliente.
- email: texto, almacena la dirección de correo electrónico del cliente.

Además, esta tabla tiene una relación uno a muchos con la tabla "Citas", representando las citas asociadas a cada cliente.

La tabla **"Citas"** está compuesta por 5 columnas. Su función es gestionar las citas de los clientes:

- id: entero autonumérico, es el identificador único de cada cita.
- cliente_id: entero, es una clave foránea que referencia al cliente asociado a la cita.
- empleado_id: entero, es una clave foránea que referencia al empleado que atenderá la cita.
- fecha: fecha, indica el día de la cita.
- tratamiento: texto, describe el tratamiento que se realizará durante la cita.

Esta tabla establece relaciones con las tablas "Clientes" y "Empleados" a través de las claves foráneas cliente_id y empleado_id respectivamente.

La tabla **"Empleados"** está compuesta por 8 columnas. Su función es guardar los datos de los empleados:

- id: entero autonumérico, es el identificador único de cada empleado.
- nombre: texto, almacena el nombre del empleado.
- apellidos: texto, almacena los apellidos del empleado.
- estado: enum ('activo', 'inactivo'), indica el estado actual del empleado.
- username: texto, nombre de usuario único del empleado.
- hashed_pin: texto, PIN del empleado en formato hash.
- is_admin: enum ('admin', 'empleado'), indica el rol del empleado.

Esta tabla tiene relaciones con las tablas "Citas", "TicketProducto" y "TicketServicio", representando las citas, productos y servicios asociados a cada empleado.

La tabla **"Productos"** está compuesta por 3 columnas. Su función es guardar los datos de los productos disponibles:

- id: entero autonumérico, es el identificador único de cada producto.
- nombre: texto, almacena el nombre del producto.
- precio: decimal, almacena el precio del producto.
- Esta tabla tiene una relación uno a muchos con la tabla "TicketProducto", que gestiona los productos vendidos en cada ticket.

La tabla **"Servicios"** está compuesta por 4 columnas. Su función es guardar los datos de los servicios ofrecidos:

- id: entero autonumérico, es el identificador único de cada servicio.
- nombre: texto, almacena el nombre del servicio.
- precio: decimal, almacena el precio del servicio.
- tipo: enum ('estética', 'peluquería'), indica el tipo de servicio.

Esta tabla tiene una relación uno a muchos con la tabla "TicketServicio", que gestiona los servicios vendidos en cada ticket.

La tabla **"Tickets"** está compuesta por 5 columnas. Su función es gestionar las transacciones de venta:

- id: entero autonumérico, es el identificador único de cada ticket.
- fecha: fecha, indica el día de la transacción.
- importe_total: decimal, almacena el importe total de la transacción.
- pago: enum ('efectivo', 'tarjeta'), indica el método de pago.
- correo: texto, almacena la dirección de correo electrónico asociada al ticket.

Esta tabla tiene relaciones uno a muchos con las tablas "TicketProducto" y "TicketServicio", que gestionan los productos y servicios vendidos en cada ticket.

La tabla **"TicketProducto"** está compuesta por 7 columnas. Su función es gestionar los productos vendidos en cada ticket:

- id: entero autonumérico, es el identificador único de cada registro.
- ticket_id: entero, clave foránea que referencia al ticket.
- producto_id: entero, clave foránea que referencia al producto.
- empleado_id: entero, clave foránea que referencia al empleado.
- cantidad: entero, cantidad de productos vendidos.
- precio: decimal, precio unitario del producto.
- total: decimal, precio total (cantidad * precio).

Esta tabla tiene relaciones con las tablas "Tickets", "Productos" y "Empleados".

La tabla **"TicketServicio"** está compuesta por 6 columnas. Su función es gestionar los servicios vendidos en cada ticket:

- id: entero autonumérico, es el identificador único de cada registro.
- ticket_id: entero, clave foránea que referencia al ticket.
- servicio_id: entero, clave foránea que referencia al servicio.
- empleado_id: entero, clave foránea que referencia al empleado.
- precio: decimal, precio unitario del servicio.

Esta tabla tiene relaciones con las tablas "Tickets", "Servicios" y "Empleados".

4.1.2 Esquema base de datos

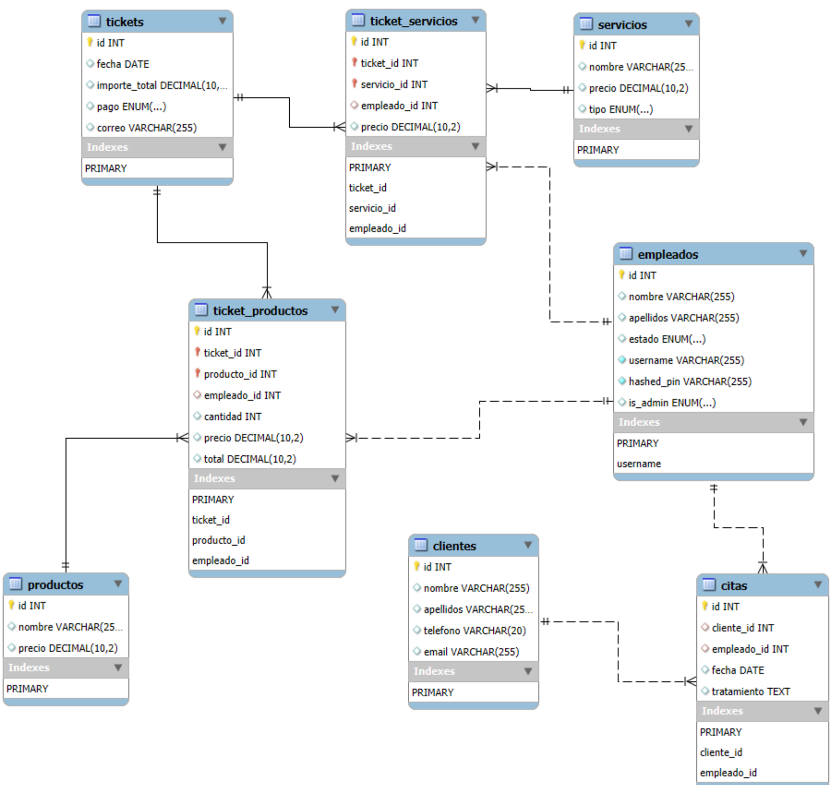


Figura 6. Esquema base de datos

4.2 Creación del Docker para almacenar la base de datos

La base de datos se debe encontrar almacenada en un Docker, para ello, tenemos que crear un contenedor en Docker, utilizando la imagen de MySQL desde Docker Hub. Este va a contener todos los archivos que necesito para ejecutar una instancia de MySQL. Los siguientes *code snippet*, son código empleado para realizar el proyecto, y el diseño de estas figuras se ha realizado con Carbon [31].

comandos Docker

```
docker pull mysql
docker run --name MySQL_Cont -e MYSQL_USER=Raquel -e MYSQL_PASSWORD=********* -e MYSQL_ROOT_PASSWORD=*********
-d -p 3306:3306 mysql:latest
```

Figura 7. Code snippet comandos empleados para creación del contenedor en Docker

`docker pull mysql` – este comando descarga la imagen oficial de MySQL desde Docker Hub en mi máquina local.

El segundo comando, crea y ejecuta un contenedor basado en la imagen MySQL, con los siguientes detalles:

- `--name MySQL_Cont:` le asigna el nombre MySQL_Cont al contenedor.
- `-e MYSQL_USER = Raquel:` establece el usuario de MYSQL como Raquel.
- `-e MYSQL_PASSWORD=********` establece la contraseña del usuario Raquel como ******** (se sustituye * por la contraseña que uno mismo elija).
- `-e MYSQL_ROOT_PASSWORD= *********` establece la contraseña del usuario root como ********* (se sustituye * por la contraseña que se elija).
- `-d:` ejecuta el contenedor en segundo plano, modo *detached*. Esto permite que el terminal quede libre para otros comandos o tareas y el contenedor sigue ejecutándose incluso si se cierra el terminal.
- `-p 3306:3306`: mapea el puerto 3306 del contenedor al puerto 3306 de mi máquina local.

Una vez completado, tenemos creado nuestro contenedor para almacenar la base de datos.

En este punto aún no crearemos las tablas, ya que eso lo realizaremos en la parte *back-end* con ayuda de Python.

Para visualizar nuestra base de datos y acceder a ella, se usa HeidiSQL [29], rellenando los campos con los datos que hemos definido en nuestro comando de Docker.

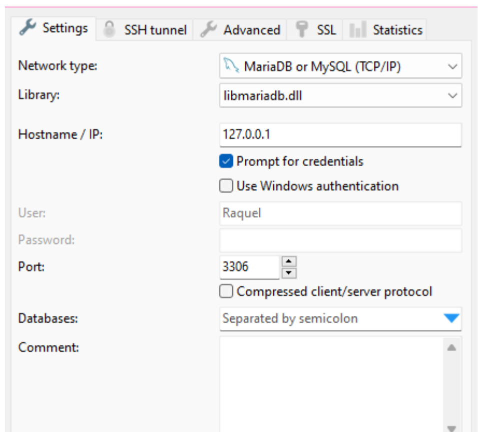


Figura 8. Datos introducidos en HeidiSQL.

Capítulo 5. DISEÑO Y DESARROLLO DEL FRONT-END

En el siguiente capítulo, nos centraremos en el desarrollo *front-end* de la aplicación al completo, desde su diseño hasta la implementación de código.

5.1 Diseño de la interfaz

5.1.1 Paleta de colores y tipografía

Deseamos hallar una interfaz sencilla, simple e intuitiva, con el propósito de decidir qué gama de colores utilizar, he tenido como referencia el color marrón, ya que, para mí, se encuentra en un equilibrio entre colores muy básicos como blancos y negros, y colores más animados y llamativos, que “ensuciarían” la interfaz y no tendría una armonía. Con esto como referencia, se propone el uso del tono marrón en una variedad de colores pastel:

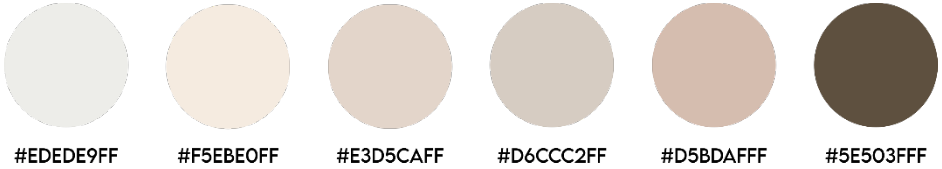


Figura 9. Paleta de colores empleada

En cuanto a la tipografía, he querido implementar una fuente vistosa y sencilla de leer, por ello he elegido *Figtree*, de la familia *sans-serif* [32]. Es una fuente creada para usar en web y aplicaciones móviles, lo que la hace ideal para nuestro propósito.

Light 300	Regular 400	Medium 500	SemiBold 600	Bold 700	ExtraBold 800	Black 900
Inicio	Inicio	Inicio	Inicio	Inicio	Inicio	Inicio

Figura 10. Tipografía Figtree de Google Fonts.

En el proyecto, la fuente se está cargando y utilizando de manera embebida a través de Google Fonts. Esto significa que la fuente se está integrando directamente en el documento HTML mediante enlaces a recursos externos:

fuente embebida HTML

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?
family=Figtree:ital,wght@0,300..900;1,300..900&display=swap"
rel="stylesheet">
```

Figura 11. Code snippet fuente embebida en HTML

definición en CSS

```
body {
    font-family: 'Figtree', sans-serif;
}
```

Figura 12. Code snippet definición de fuente en CSS

5.1.2 Logotipo

Una vez determinados los colores a utilizar en la aplicación web, he creado un logotipo sencillo con ayuda de Canva:

Figura 13. Logotipo aplicación

Mi idea era crear un logotipo muy simple, por ello el nombre: "TPV Peluquería y Estética" y he añadido "By Raquel Parcet", a modo de firma de autora. He usado una flor como símbolo de belleza, puesto que este programa está dirigido a los salones de belleza. Además de logotipo, también lo usaré como *favicon* (logotipo que se usa en las pestañas de las páginas webs).

5.1.3 Prototipo

En primer lugar, la página principal al acceder al programa ha de ser un formulario para iniciar sesión en la plataforma, de manera que, introduciendo el usuario y contraseña adecuados, conseguiremos entrar en la aplicación.

Figura 14. Formulario página principal, inicio de sesión

Una vez iniciado sesión correctamente, entraremos en la página principal con las 3 funcionalidades primarias: bonos, mantenimiento y caja, a partir de esta página, tendremos en todas ellas un botón para poder cerrar sesión.

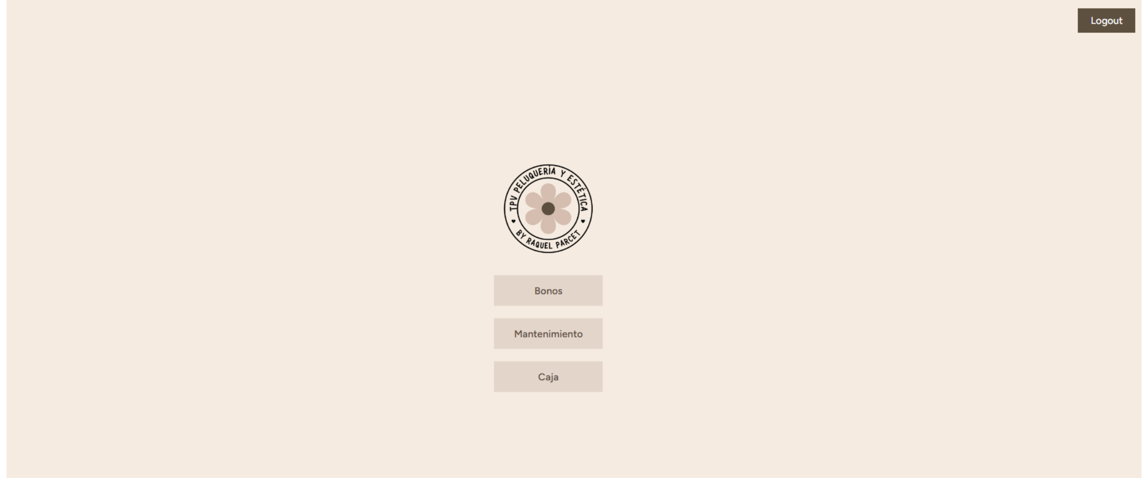


Figura 15. Página principal

5.1.3.1 Bonos

El acceso a bonos estará restringido a solo usuarios con la condición "administrador", ya que en esta página podremos hacer dos cosas: obtener el monto total de dinero realizado en un periodo de tiempo seleccionado, u obtener el monto total de dinero realizado por cada empleado en un periodo de tiempo seleccionado. De cada uno de ellos, podremos descargarnos un PDF de la información generada.

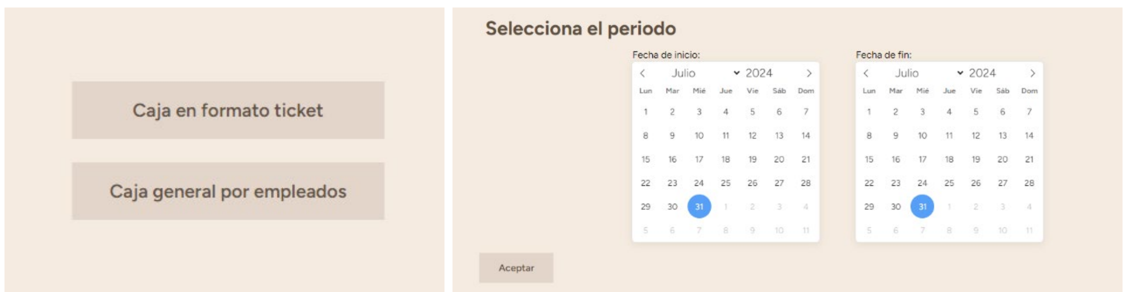


Figura 16. Botones página bonos **Figura 17. Calendarios para seleccionar periodo**

Las dos funciones de la página bonos tienen la misma estructura, primero seleccionamos un periodo de tiempo en el cual queremos ver los totales, y una vez pulsemos en aceptar, cada función hará su respectivo trabajo.

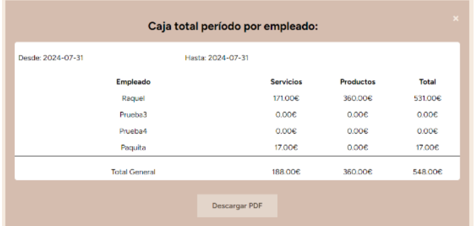
Caja total período:

Desde: 2024-07-31 Hasta: 2024-07-31

Método	Nro Tickets	Importe
Efectivo	3	162.5€
Tarjeta	1	70.5€
Total	4	233€

Descargar PDF

Figura 18. Ticket de caja total en un periodo de tiempo establecido



Caja total período por empleado:

Desde: 2024-07-31 Hasta: 2024-07-31

Empleado	Servicios	Productos	Total
Raquel	171.00€	360.00€	531.00€
Prueba3	0.00€	0.00€	0.00€
Prueba4	0.00€	0.00€	0.00€
Paquita	17.00€	0.00€	17.00€
Total General	188.00€	360.00€	548.00€

Descargar PDF

Figura 19. Ticket de caja total por empleados en un periodo de tiempo

En ambos tipos, podremos descargar en formato PDF el ticket.

Desde: 2024-07-31Hasta: 2024-07-31		
Método	Nro Tickets	Importe
Efectivo	3	162.5€
Tarjeta	1	70.5€
Total	4	233€

Figura 20. Captura de cómo se ve el PDF que descargamos del ticket

5.1.3.2 Mantenimiento

En esta sección contaremos con cuatro páginas nuevas: clientes, empleados, productos y servicios, todas ellas tienen que ver con las tablas que habrá que crear en la base de datos. En ellas podemos crear o editar cada tipo de dato.

Algo común a todas las páginas, será un “modal” con un formulario, tanto para editar como para crear un nuevo cliente/empleado/producto/servicio, cada uno de ellos con sus respectivos datos.

Figura 21. Botones página mantenimiento

Figura 22. Estilo formulario/modal común para todas las páginas

Cada página tiene su estilo y sus funcionalidades. En primer lugar, la página cliente nos permite visualizar los clientes presentes en la base de datos a modo de lista, en ella se puede hacer una búsqueda para encontrar a un cliente en concreto también se pueden visualizar los datos, así como editarlos.

Figura 23. Página principal de clientes

Figura 24. Ejemplo de visualización de datos.

La página de empleados visualizará los empleados registrados, así como si están en modo activo o inactivo. Así como poder crear un nuevo empleado a través del botón correspondiente o editar uno existente si pulsamos en su recuadro asignado.

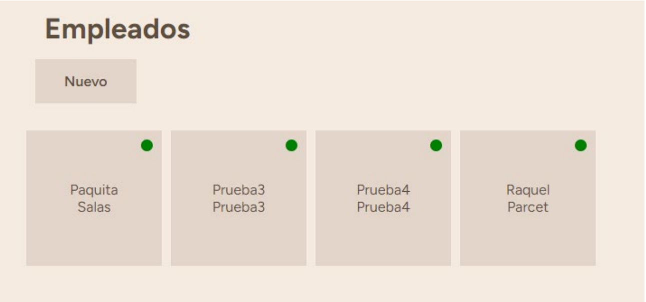


Figura 25. Vista página empleados con empleados activos

Figura 26. Vista página empleados con un empleado inactivo

La página productos y la página servicios, funcionan prácticamente de la misma manera, y la estética es común en ambas, la única diferencia es que servicios estará filtrada por tipos (estética y peluquería). Como sus páginas hermanas, también se podrá crear nuevos productos o servicios, así como editar los ya creados.

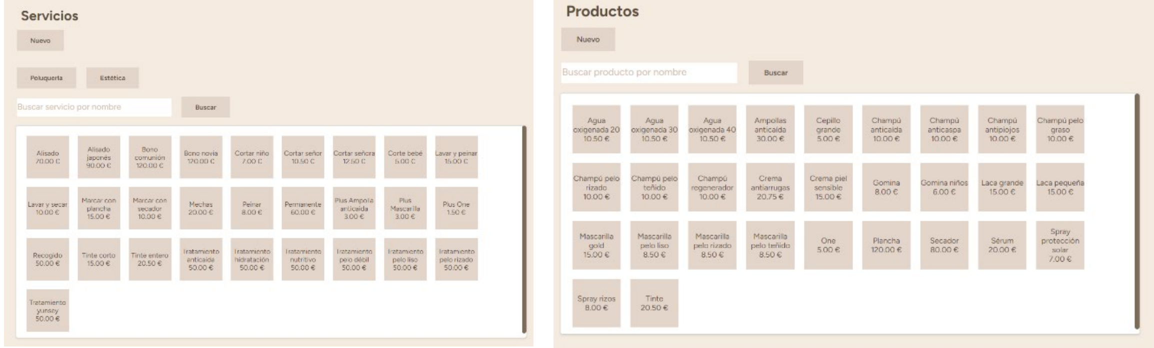


Figura 27. Página servicios

Figura 28. Página productos

5.1.3.3 Caja

Esta página está destinada para introducir los tickets realizados en el día en el sistema. Se podrá seleccionar el servicio/producto, en el marcaremos qué empleado realiza el servicio o la venta y se añadirá al ticket en forma de lista. Hará una suma automática de lo que vamos generando, tendremos la opción de eliminar del ticket un elemento.

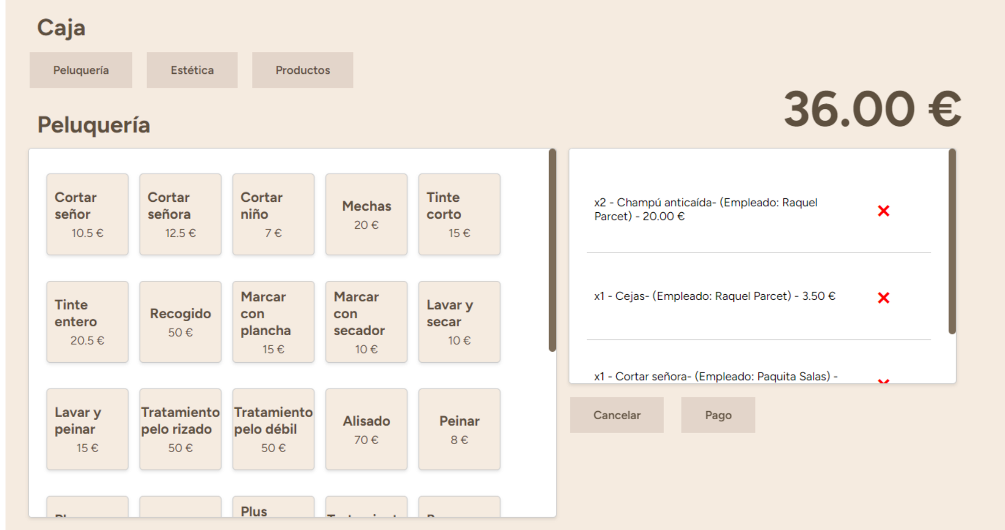


Figura 29. Página caja

Esta página cuenta con dos tipos de formularios/modales. El primero se activa cuando seleccionamos un ítem de producto o servicio. En él se deberá marcar el empleado que va asignado con ese servicio o venta, y también una posible modificación del precio. Al aceptar, se añadirá a nuestra lista de ticket.

El otro formulario, se activa cuando queremos finalizar un ticket, se accede pulsando el botón pago, donde se seleccionará si el pago se realiza en efectivo o tarjeta, y, de manera opcional, la posibilidad de asignar un correo electrónico al ticket, para enviarle por correo una copia del ticket con los servicios y productos adquiridos con sus respectivos precios.

Figura 30. Formulario para seleccionar datos asignados al servicio/producto

Figura 31. Formulario para seleccionar método de pago y enviar correo.

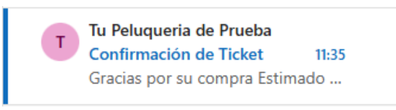


Figura 32. Email con ticket recibido

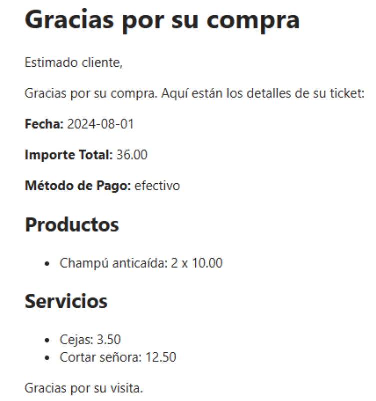


Figura 33. Contenido del email de ticket.

5.1.4 Flujo de navegación

Es fundamental considerar cómo explorar la plataforma y los puntos de acceso a cada vista. La arquitectura del sistema facilita un acceso eficiente. Se implementarán mecanismos de seguridad para asegurar que solo los usuarios autorizados accedan en áreas específicas.

Las diferentes conexiones que componen este sistema se explican en el siguiente diagrama, que ilustra las relaciones entre los componentes clave. Con él facilitamos la compresión de la estructura y rutas de acceso:

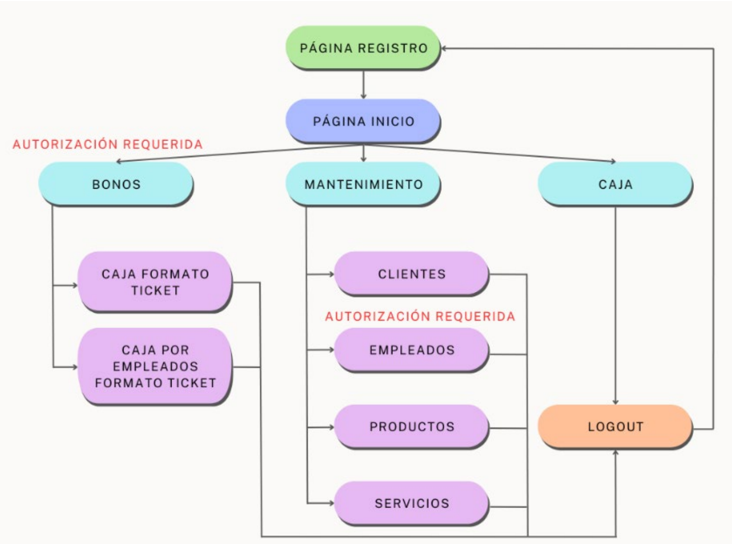


Figura 34. Diagrama de rutas

5.2 Desarrollo del *front-end*

En este proyecto, el *front-end* consiste en los documentos HTML, CSS y JavaScript.

Están divididos en dos carpetas:

- ***Static***: donde se encuentran todos los archivos estáticos, como las imágenes, los archivos JavaScript y las hojas de estilos.
- ***Templates***: donde se encuentran las plantillas HTML.

Los archivos JavaScript son una parte fundamental del código de este proyecto, ya que es donde se definen las funciones de cómo deben funcionar nuestras plantillas y la aplicación junto con el *back-end*.

Para explicarlo de manera más organizada, he dividido los distintos *templates* en cuatro grupos; en cada grupo, las *templates* son parecidos.

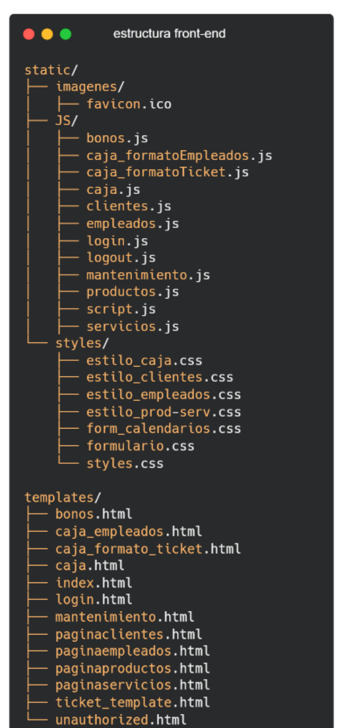


Figura 35. Estructura de carpetas *front-end*

5.2.1 Header, botón atrás, logotipo y botón logout.

Antes de presentar cada grupo, procederé a exponer las características comunes de todas las *templates*, tales como el *header*.

Explicado de una manera sencilla, una *template* html se compone de dos partes, la cabecera (conocida como *head*) y el cuerpo (conocido como *body*). En la cabecera encontramos metadatos y links con los que damos la información necesaria para que el navegador entienda como usar la página. En el cuerpo escribimos todo el contenido "en plano" de la página, al cual después daremos forma con ayuda de las hojas de estilo.

Aunque es verdad que cada *template* tendrá una hoja de archivos específica y un título de página distinto, la estructura es uniforme y similar en todos los *templates*, ya que en este *header* del HTML hemos definido la fuente que se utilizará de manera embebida, así como otros datos cruciales.

header común

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="{{ url_for('static', path='/styles/styles.css') }}" rel="stylesheet">
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Figtree:ital,wght@0,300..900;1,300..900&display=swap" rel="stylesheet">
  <link rel="icon" href="{{ url_for('static', path='/imagenes/favicon.ico') }}" type="image/x-icon">
  <title>Inicio Sesión</title>
</head>
```

Figura 36. Ejemplo de header común en las *templates*

Esta sección de código HTML es común en todas las *templates*. A continuación, detallaré lo que hace cada línea:

- `<meta charset="UTF-8">`: establece el conjunto de caracteres para el documento como UTF-8, lo que nos asegura que se manejan correctamente caracteres especiales y acentos en diferentes idiomas.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: le indica al navegador como debe comportarse con respecto al tamaño de la pantalla, establecemos que el ancho de la página debe ser igual al ancho del dispositivo, y aseguramos que el nivel de zoom inicial al cargar la página sea del 100%. Punto importante para que las páginas sean responsive y se vean correctamente en diferentes pantallas.
- `<link href="{{url_for ('static', path='/styles/styles.css')}} “rel="stylesheet">`: es la línea donde enlazamos la hoja de estilos que tiene que usar la página que estamos creando. En este caso está definido como una función de Jinja2. La hoja de estilos “styles.css” es común en todas mis páginas, aunque después, cada página tiene añadida otra hoja de estilos, con las características necesarias adaptadas a su contenido.
- `<link rel="preconnect" href="https://fonts.googleapis.com">` y `<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>`: optimizan la carga de las fuentes web alojadas en Google Fonts.
- `<link href="https://fonts.googleapis.com/css2?family=Figtree:ital,wght@0,300..900;1,300..900&display=swap" rel="stylesheet">`: carga la fuente que vamos a utilizar en todo el proyecto.

- `<link rel="icon" href="{{url_for ('static', path='/imagenes/favicon.ico')}}` "`type="image/x-icon">`: cargamos y especificamos el favicon de la página, esto es el logotipo que aparece en la pestaña de la página web.
- `<title>Inicio Sesión</title>`: en esta línea definimos el título de la página (que aparecerá en la pestaña), por supuesto, cada página tiene su título correspondiente.

Otro código que es común en todas las páginas, a excepción de index.html y login.html, es el botón para navegar hacia atrás en las páginas (implementado por si se pusiera en pantalla completa la aplicación, poder retroceder fácilmente), así como el logotipo de la aplicación. Este código se encuentra en el *body* del html, es decir, en el cuerpo.

botón atrás, logotipo y botón logout

```html
<body>
    <div class="back-button-container">
        <img src="{{ url_for('static', path='/imagenes/favicon.ico') }}" alt="logo" class="logo">
        <div class="div-atras-cont">
            <button onclick="goBack()" class="back-button">Atrás</button>
        </div>
    </div><!--back-button-container-->
    <button id="logout-button">Logout</button>
</body>
```

Figura 37. Code snippet de botón atrás y logotipo

En este código, añadimos el logotipo a la página, que se encuentra en la carpeta de archivos estáticos, concretamente en la carpeta de imágenes. Además, se añade el botón de navegación a la página anterior, que, como podemos observar, le hemos añadido una función mediante el método *onclick()*, cuya función *goBack()* consiste en navegar a la página anterior con ayuda del historial, se encuentra en el archivo JavaScript asociado a la *template* en la que estemos trabajando.

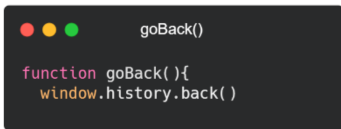


Figura 38. Code snippet función goBack()

También tenemos un botón para poder cerrar sesión en cada página, este funciona gracias al archivo logout.js.

logout.js

```javascript
document.addEventListener('DOMContentLoaded', () => {
    const logoutButton = document.getElementById('logout-button');

    if (logoutButton) {
        logoutButton.addEventListener('click', () => {
            // Realiza la solicitud para cerrar sesión
            fetch('/logout', {
                method: 'POST',
                credentials: 'include',
            })
            .then(response => {
                if (!response.ok) {
                    throw new Error('Error al cerrar sesión');
                }
                // Redirige a la página de login
                window.location.href = '/login';
            })
            .catch(error => {
                console.error('Error:', error);
            });
        });
    }
})
```

Figura 39. Code snippet del archivo logout.js

El funcionamiento de esta función es la siguiente, una vez se ha cargado todo el contenido de la página, se selecciona el elemento que hemos definido con identificador (id) *logout-button*, de manera que si existe, le añadimos un evento para que cuando pulsemos en él, nos haga una solicitud POST de la ruta `/logout` al servidor (en la parte de

back-end explicaremos en profundidad en qué consiste esto), si nos diera error o la respuesta no fuese ok, nos lanzaría un error el servidor pero si la solicitud es satisfactoria, cerraremos sesión y nos redirigirá a la página de inicio de sesión.

5.2.2 Grupo 1 – Páginas de navegación

Este grupo está compuesto por las *templates* **bonos.html**, **index.html** y **mantenimiento.html**. La particularidad que tienen estas, es que se tratan de páginas simples con botones para navegar por las diferentes funcionalidades de la aplicación. La estructura es común en las tres:

estructura grupo 1

```
<div class="button-container">

  <button id="clientes-button">Clientes</button>
  <button id="empleados-button">Empleados</button>
  <button id="productos-button">Productos</button>
  <button id="servicios-button">Servicios</button>

</div>

<script src="static/JS/mantenimiento.js"></script>
<script src="static/JS/logout.js"></script>
```

Figura 40. Code snippet de la estructura común en las páginas del grupo 1.

En estas páginas, contamos con un contenedor al que le hemos dado una clase, *button-container*. Esto lo hacemos así, porque en todas las páginas queremos aplicar los mismos estilos de contenedor, y eso nos lo permite hacer el atributo “clase”, ya que “id” es único, no se puede repetir. A cada botón le damos un identificador, porque estos, en su archivo JS, tendrán asociado una función a realizar, en este caso, navegar a la correspondiente página. Es importante no olvidarse de añadir los scripts, que es la manera en la que enlazamos la *template* con su correspondiente archivo JS.

ejemplo css 1

```
.button-container {
    text-align: center;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    padding: 20px;
    display: flex;
    flex-direction: column;
}
```

Figura 41. Code snippet ejemplo de CSS 1

ejemplo css 2

```
button, .back-button {
    font-family: "Figtree", sans-serif;
    background-color: #e3d5ca;
    color: #5e503f;
    padding: 15px 32px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    margin: 10px;
    cursor: pointer;
    border: none;
    font-weight: 600;
    transition: background-color 0.3s ease, color 0.3s ease;
}

button:hover, .back-button:hover {
    background-color: #d5bdaf;
}
```

Figura 42. Code snippet ejemplo CSS 2

Figura 43. Ejemplo de visualización con CSS aplicado

En la Figura 41 y 42, podemos ver un ejemplo de cómo se asignan estilos a los diferentes elementos de la página, y en la Figura 43, un ejemplo de la visualización que conseguimos.

Un comando que destacar es `display: flex` que nos permite convertir un contenedor en *flexbox*, esto hace que podamos distribuir los elementos en filas o en columnas. Es un comando al que se recurrirá en toda la aplicación para conseguir la visualización requerida.

En el *code snippet* de la Figura 41, se define un código de estilo que será común en todas las páginas, ya que definimos cómo se verán los botones, en cuanto a fuente, color de fondo, color de letra, relleno, tamaño de fuente entre otros. Además de añadir un efecto `:hover`, esto se refiere al efecto que realiza un elemento cuando pasamos el cursor por encima de estos.

En cuanto al código JavaScript que utiliza este grupo de *templates*, los tres tienen una estructura y funcionamiento similar. Cuando definimos los botones en HTML, es importante asignarles un identificador único, porque es la manera que tenemos de poder acceder a ellos en el JavaScript.

Funciona de la siguiente manera: declaramos unas constantes que harán referencia a los botones definidos en HTML, accedemos a ellos a través del identificador. Después, le añadimos un *eventListener "click",* es decir, le añadimos un "escuchador" que va a esperar a que el usuario haga click, para realizar la función que le asignemos, en este caso, navegar a la correspondiente página.

```html
<button id="bonos-button">Bonos</button>
<button id="mantenimiento-button">Mantenimiento</button>
<button id="caja-button">Caja</button>
```

Figura 44. Ejemplo de HTML con "id" asignados a los botones.

```javascript
//obtener los botones

const bonosButton = document.getElementById('bonos-button');
const mantenimientoButton = document.getElementById('mantenimiento-button');
const cajaButton = document.getElementById('caja-button');

//añadir event listeners a los botones

bonosButton.addEventListener('click', () => {
  location.href = '/bonos';
});

mantenimientoButton.addEventListener('click', () => {
  location.href = '/mantenimiento';
});

cajaButton.addEventListener('click', () => {
  location.href = '/caja';
})
```

Figura 45. Ejemplo de JS, cómo se accede a los botones con su "id" y agregar el evento

Estas rutas a las que navegamos con `location.href`, son rutas que estarán descritas en el *back-end* de nuestra aplicación, que más adelante será explicado.

5.2.3 Grupo 2 – Páginas para interactuar con la base de datos

Este grupo está compuesto por las *templates* **paginaclientes.html**, **paginaempleados.html**, **paginaproductos.html**, **paginaservicios.html** y **caja.html**. Estas plantillas tienen en común que sirven para manipular y acceder a la base de datos, pudiendo visualizar los datos que tenemos presentes en ella, así como editar, crear o eliminar datos. La distribución y visualización, depende de cada página, por ejemplo, productos y servicios, son similares, pero las tres restantes cada una tiene sus peculiaridades.

Voy a tomar como ejemplo **paginaservicios.html**, por lo que hablaremos continuamente sobre servicios, pero se aplica de la misma manera para empleados, clientes y productos.

Para visualizar en nuestra página los servicios que se encuentran definidos en la base de datos, se ha usado Jinja2. He creado un pequeño esquema para que quede más claro cómo funciona:

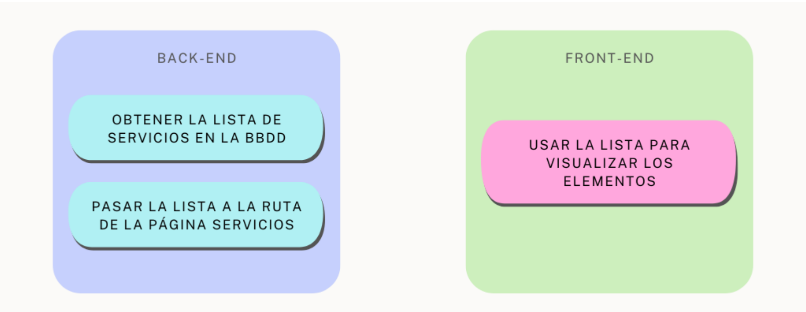


Figura 46. Esquema funcionamiento visualización de Servicios en la BBDD.

Cómo se obtiene en el *back-end* la lista, lo veremos en su correspondiente apartado. En el HTML, debemos agregar código específico de plantilla Jinja2 para visualizarlo. Supongamos que nuestra lista o colección se llama servicios:

```
<div id="servicio-cont">
  {% for servicio in servicios %}
  <div class="servicio" data-id="{{ servicio.id }}">
    <div class="nombre">{{ servicio.nombre }}</div>
    <div class="precio">{{ servicio.precio }} €</div>
  </div>
   {% endfor %}
</div> <!--servicio-cont-->
```

Figura 47. Code snippet código Jinja2 para visualizar datos de la BBDD.

Este fragmento de código (Figura 47) genera dinámicamente una lista de servicios en HTML. Utiliza un bucle de plantilla Jinja2 para iterar sobre una colección de servicios (servicios) y crea un `div` para cada servicio, mostrando su nombre y precio.

Cada `div` de servicio tiene un atributo `data-id` que contiene el `id` del servicio correspondiente. Un `div` sirve para crear una sección/contenedor.

```
modal

<!-- Ventana emergente NUEVO SERVICIO-->
<div id="modal-nuevoServicio" class="modal">
  <div class="modal-content">
    <span class="close" id="closeModalNuevoServicio">&times;</span>
    <h2>Crear nuevo servicio</h2>

    <!-- Formulario para crear un nuevo servicio -->
    <form id="servicio-form">
      <span>
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre">
      </span>

      <span>
        <label for="precio">Precio:</label>
        <input type="text" id="precio" name="precio">
      </span>

      <span>
          <label for="tipo">Tipo:</label>
          <select id="tipo" name="tipo">
              <option value="peluqueria">Peluquería</option>
              <option value="estetica">Estética</option>
          </select>
      </span>

      <span>
        <button id="btn-newServicio-guardar" type="button">Guardar</button>
      </span>

    </form>
  </div> <!-- modal-content-->
</div> <!--modal-->
```

Figura 48. Code-snippet ejemplo de modal para crear un nuevo servicio

Todas las páginas tienen en común los modales y formularios, definidos para crear y editar una entrada en la base de datos. El modal lo que hace es estar oculto en la página, hasta que realicemos la acción necesaria para activarlo y visualizarlo (por ejemplo, pulsar un botón). A todos los modales les definimos la misma clase, para poder darles el mismo formato a través de las hojas de estilos. Como anteriormente, es importante definir identificadores en los elementos, para poder darles las funcionalidades requeridas con ayuda de JavaScript.

Vamos a tomar como ejemplo "Servicios", pero la explicación se aplica de la misma manera para clientes, productos y empleados. Cuando se trata de un modal para editar un servicio que ya existe, es necesario que, en los elementos del formulario, aparezcan los datos actuales de ese servicio. Esto se lleva a cabo con ayuda de JavaScript, y una función que denominamos *abrirModalEditar(servicioId)*.

```
funcion abrirModalEditar(id)

function abrirModalEditar(servicioId){
    const servicioSeleccionado = document.querySelector(`.servicio[data-id="${servicioId}"]`);
    if (servicioSeleccionado) {
        const nombre = servicioSeleccionado.querySelector('.nombre').textContent;
        const precio = servicioSeleccionado.querySelector('.precio').textContent.replace(' €', '');
        const tipo = document.body.getAttribute('data-tipo');

        editarServicioNombre.value = nombre;
        editarServicioPrecio.value = precio;
        editarServicioTipo.value = tipo;

        modalEditar.style.display = "block";
    }
```

Figura 49. Code snippet función abrirModalEditar(id)

La función `abrirModalEditar` facilita la edición de un servicio específico al abrir un modal pre-rellenado con los datos del servicio seleccionado. Estos son los diferentes pasos que sigue la función:

1. Recibe un parámetro `servicioId`, que es el identificador del servicio que se desea editar.
2. Se utiliza `document.querySelector` para seleccionar el elemento del DOM (Document Object Model), que representa el servicio con el `data-id` igual al `servicioId` proporcionado de esta manera, `servicioSeleccionado` contendrá el elemento del servicio si existe, o `null` si no se encuentra.
3. Se verifica si `servicioSeleccionado` no es `null`, es decir, si el servicio con el `servicioId` proporcionado existe en el DOM.

4. Se extrae el nombre del servicio, utilizando `querySelector` para seleccionar el elemento con la clase `nombre` dentro del `servicioSeleccionado`, y se obtiene su contenido de texto (`textContent`). Similarmente, se extrae el precio del servicio seleccionado. Se utiliza `replace` para eliminar el símbolo de euro (€) del texto.
5. Se obtiene el atributo `data-tipo` del elemento `body` del documento, que representa el tipo de servicio (esto es algo exclusivo de la página servicios, ya que queremos que estén filtrados por su columna "tipo").
6. Se asignan los valores extraídos (`nombre, precio, tipo`) a los campos correspondientes del formulario de edición (`editarServicioNombre, editarServicioPrecio, editarServicioTipo`).
7. Por último, se cambia el estilo del modal de edición (`modalEditar`) para que se muestre en pantalla, estableciendo su propiedad `display` a `"block"`.

Para cerrar los modales, se crea una función que cambie el display del modal a "`none`", y se le asigna al símbolo de cerrar, (`×`) un `eventListener` para que cuando se pulse en él, se cierren los modales.

Char	Entity	Dec	Hex	Description
×	×	×	×	multiplication sign

Figura 50. Datos del símbolo para cerrar

Servicios, productos, empleados y caja tienen una función `ordenarAlfabeticamente()` que se encarga de ordenar los elementos en la interfaz de usuario en orden alfabético basado en el nombre. Para ello, selecciona todos los elementos del DOM que tienen la clase específica (por ejemplo, servicio) y se convierten en un array utilizando el comando `Array.from`.

```
ordenar alfabéticamente

// Ordenar servicios alfabéticamente
function ordenarServiciosAlfabeticamente() {
    const serviciosA = Array.from(document.querySelectorAll(".servicio"));

    serviciosA.sort((a, b) => {
        const nombreA = a.querySelector(".nombre").textContent.toLowerCase();
        const nombreB = b.querySelector(".nombre").textContent.toLowerCase();
        return nombreA.localeCompare(nombreB);
    });

    serviciosA.forEach(servicio => {
        serviciosCont.appendChild(servicio);  // Reagregar servicio ordenado al DOM
    });
}
```

Figura 51. Code snippet función para ordenar alfabéticamente

Se ordenan los elementos del array utilizando la función `sort`. Para cada par de elementos `a` y `b`, se obtiene el texto del nombre del servicio (`.nombre`) y se convierte a minúsculas para asegurar una comparación insensible a mayúsculas/minúsculas, después se utiliza `localeCompare` para comparar los nombres y determinar el orden alfabético. Por último, se itera sobre el array ordenado y se re-agregan los elementos al contenedor `serviciosCont`, esto actualiza el DOM para reflejar el nuevo orden de los servicios.

En estas páginas también se ha implementado una barra de búsqueda, para encontrar un elemento de la base de datos en concreto, y a su vez, una función para autocompletar, es decir, que aparezcan los elementos que vayan coincidiendo con lo que estamos buscando. Esto se ha hecho con ayuda de dos funciones, `buscarServicio()` e `inicializarAutocompletado()`.

La función `buscarServicio` se encarga de filtrar y mostrar los servicios que coinciden con el término de búsqueda ingresado por el usuario. Se obtiene el valor del campo de entrada de búsqueda (`buscarInput`) y se convierte a minúsculas para realizar una comparación insensible a mayúsculas/minúsculas, se seleccionan todos los elementos del DOM que tienen la clase `servicio`. Para cada servicio, se obtiene el texto del nombre (`.nombre`) y se convierte a minúsculas. Se verifica si el nombre del servicio incluye el término de búsqueda (`searchTerm`), si coincide, se remueve la clase `hidden` para mostrar el servicio, si no coincide, se agrega la clase `hidden` para ocultar el servicio.

La función `inicializarAutocompletado` configura el evento necesario para que la búsqueda se realice en tiempo real a medida que el usuario escribe en el campo de búsqueda. Es decir, cada vez que el usuario escribe en el campo de búsqueda, se llama a la función `buscarServicio` para actualizar la lista de servicios mostrados en tiempo real.

```
buscarServicio()

//buscar servicio
function buscarServicio(){
    const searchTerm = buscarInput.value.toLowerCase();
    const servicioItems = document.querySelectorAll(".servicio");

    servicioItems.forEach((servicio) => {
        const nombreCompleto = servicio.querySelector(".nombre").textContent.toLowerCase();

        if (nombreCompleto.includes(searchTerm)) {
            servicio.classList.remove("hidden");  // Mostrar el servicio si coincide con la búsqueda
        } else {
            servicio.classList.add("hidden");   // Ocultar el servicio si no coincide
        }
    });
}
```

Figura 52. Code snippet función buscar

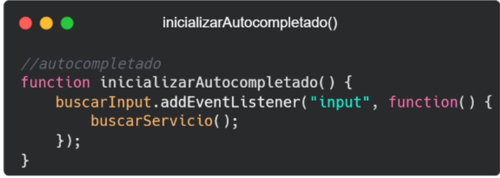


Figura 53. Code snippet función autocompletado

5.2.3.1 Funciones para crear, editar o eliminar

Estas funcionalidades se realizan a través de los modales mencionados anteriormente, pero la acción está asignada a los botones que tienen los modales con *eventListener* cuando se haga *click* en ellos. De esta manera, se han definido tres funciones en JavaScript: `enviarNuevoServicio()`, `guardarCambiosServicio()`, `eliminarServicio()`.

La función `enviarNuevoServicio` se encarga de enviar los datos de un nuevo servicio al servidor mediante una solicitud HTTP POST y actualizar la interfaz de usuario con el nuevo servicio creado. Para ello, se obtienen los datos introducidos en el formulario, esos datos se convierten a un objeto JavaScript, para después pasarlo a una cadena JSON (que es la manera en la que nuestro *back-end* entiende los datos). Después se envía la solicitud HTTP POST, y si tiene éxito cerramos el formulario y actualizamos los servicios para que aparezca en nuestra interfaz.

La función `guardarCambiosServicio` se encarga de actualizar los datos de un servicio existente en el servidor y reflejar los cambios en la interfaz de usuario. Se obtienen los valores actuales de los campos de entrada del formulario, y se crea un objeto de datos actualizados que convertimos a una cadena JSON. Se envía una solicitud HTTP PUT al servidor y si la respuesta es exitosa se actualiza la interfaz de usuario con los nuevos datos.

La función `eliminarServicio` se encarga de eliminar un servicio existente del servidor y actualizar la interfaz de usuario para reflejar esta eliminación. Se lleva a cabo mediante una solicitud HTTP DELETE al servidor.

5.2.4 Grupo 3 – Visualización de datos en un periodo de tiempo

Dentro de este grupo se encuentran las *templates*, **caja_formato_ticket.html** y **caja_empleados.html**. Estas se tratan de páginas webs que permite a los usuarios seleccionar un periodo de tiempo y generar un reporte en formato de ticket. A continuación, se detallan las particularidades técnicas.

Estás páginas usan dos librerías de JavaScript, flatpickr para la selección de fechas y jsPDF y jsPDF-autoTable para la generación de documentos PDF directamente desde el navegador. Contienen un formulario que permite a los usuarios seleccionar un rango de fechas y enviar esta información para generar un reporte y un modal que proporciona una interfaz para mostrar los resultados y permite la descarga del reporte en formato PDF.

Las funciones JavaScript de ambas *templates* son similares, son cinco funciones principales, para mostrar el modal, cerrar el modal, manejar el envío del formulario, realizar una solicitud fetch al servidor para obtener los datos y una función para mostrar los resultados en la interfaz de usuario. Las funciones para abrir y cerrar el modal funcionan de la misma manera como hemos mencionado anteriormente, por lo que solo entraremos en detalle en las nuevas funciones, `handleFormSubmit(event), fetchData() y displayResults()`.

La función `handleFormSubmit` se encarga de interceptar el evento de envío de un formulario, prevenir la recarga de la página y extraer las fechas seleccionadas por el usuario para luego procesarlas mediante otra función (`fetchData`). Esta función es un manejador de eventos que se asocia con el evento de envío del formulario (`submit`). Al interceptar este evento, la función puede realizar operaciones adicionales antes de permitir que el formulario se envíe.

La función `fetchData` toma dos parámetros, `startDate` y `endDate`, que representan un rango de fechas. Utiliza estos parámetros para construir una URL de solicitud y luego realiza una solicitud fetch para obtener datos de un *endpoint* específico. Los datos obtenidos se procesan y se muestran en la interfaz de usuario. Esta función utiliza promesas (`then` y `catch`) para manejar la solicitud HTTP de manera asíncrona. Esto permite que la aplicación continúe respondiendo a las interacciones del usuario mientras se espera la respuesta del servidor.

```
// Función para realizar la solicitud al servidor y obtener los datos
function fetchData(startDate, endDate) {
    fetch(`/api/caja/tickets?start_date=${startDate}&end_date=${endDate}`)
        .then(response => response.json())
        .then(data => {
            displayResults(data); //mostrar los resultados
            showModal();
        })
        .catch(error => console.error('Error:', error));
}
```

Figura 54. Code snippet función fetchData

La función `displayResults` toma un objeto data y muestra los resultados en una página web. Limpia los resultados anteriores, crea y estructura una tabla con la información de fechas, métodos de pago (efectivo y tarjeta), y totales, y luego muestra esta tabla en un contenedor de resultados en la página.

Para hacer uso de flatpickr, es importante que definamos inputs con id para nuestro calendario de inicio de periodo y fin de periodo. Porque en JavaScript, definiremos como se deben de visualizar estos calendarios con la librería una vez se haya cargador el DOM.

```html
<div class="calendar-item">
  <label for="start-date" class="labelItem">Fecha de inicio:</label>
  <input type="hidden" id="start-date" name="start_date" required>
  <div id="start-calendar"></div>
</div> <!--calendar-item-->

<div class="calendar-item">
  <label for="end-date" class="labelItem">Fecha de fin:</label>
  <input type="hidden" id="end-date" name="end_date" required>
  <div id="end-calendar"></div>
</div> <!--calendar-item--
```

Figura 55. Code snippet calendarios definidos en HTML

```javascript
// Inicializar el calendario de inicio
flatpickr("#start-calendar", {
    inline: true,
    dateFormat: "Y-m-d",
    locale: "es",
    onChange: function(selectedDates, dateStr, instance) {
        startDateInput.value = dateStr;

    }
});

// Inicializar el calendario de fin
flatpickr("#end-calendar", {
    inline: true,
    dateFormat: "Y-m-d",
    locale: "es",
    onChange: function(selectedDates, dateStr, instance) {
        endDateInput.value = dateStr;

    }
});
```

Figura 56. Code snippet definición calendarios con librería flatpickr en JavaScript

5.2.5 Grupo 4 – Páginas independientes

Este grupo lo forman las *templates* **login.html**, **unauthorized.html** y **ticket_template.html**. Cada una de ellas sirve para una función diferente, es por ello por lo que son independientes a las demás.

La *template* **login.html**, se trata de la página de registro, la primera página que se nos abre al acceder a nuestra aplicación web, en ella tenemos un formulario para introducir usuario y contraseña. Tiene asociado un JavaScript denominado **login.js**, en el hemos definido una función que se encarga de manejar el proceso de autenticación del usuario. Captura las credenciales del usuario y realiza una solicitud HTTP POST para obtener un token de autenticación y manejar la respuesta del servidor, almacenando el token y redirigiendo al usuario si la autenticación es exitosa o mostrando un mensaje de error si falla. Este token nos servirá para darle permisos a los usuarios para entrar en páginas específicas, cómo se restringe lo explicaremos en la parte de *back-end*.

La *template* **unauthorized.html** es simplemente una página que contiene un mensaje que se muestra a los usuarios cuando intentan acceder a una página en la que no tienen autorización, de esta manera, esta página les avisa que no disponen la autorización requerida.

Por último, la página **ticket_template.html**, es una plantilla de email que pueden recibir los clientes por email, si así lo desean, una vez vamos a finalizar un ticket en **caja.html**. Esta funcionalidad de enviar el correo la explicaremos en su correspondiente apartado, *back-end*.

5.2.6 Conexión *front-end* y *back-end*

Como se ha explicado, el *front-end* hace peticiones al servidor para obtener información de la base datos, o bien, editar y modificar datos.

La conexión entre el *front-end* y el *back-end* en este proyecto se realiza mediante solicitudes HTTP utilizando la función fetch. Este enfoque permite una comunicación eficiente y asíncrona entre el cliente y el servidor. El manejo de eventos, la programación asíncrona y el manejo de errores son componentes clave que aseguran una experiencia de usuario fluida y robusta.

Capítulo 6. LÓGICA Y DESARROLLO DEL BACK-END

6.1 Estructura del proyecto

En esta sección se detalla la organización y estructura del proyecto de desarrollo del *back-end* de nuestra aplicación. Una buena estructura de archivos y módulos es crucial para mantener el código limpio, modular y fácil de mantener.

Este proyecto se ha organizado en una serie de archivos Python, cada uno con responsabilidades específicas, lo que facilita tanto el desarrollo como la futura ampliación y mantenimiento del sistema.

Los archivos se encuentran almacenados en la carpeta app/, y cada uno de ellos tiene una funcionalidad asignada:

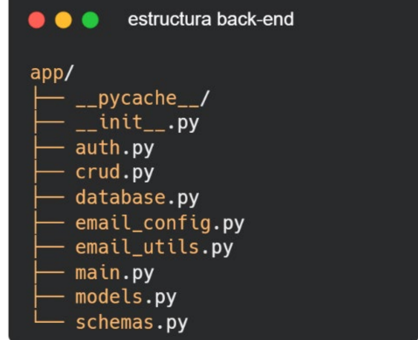


Figura 57. Estructura archivos *back-end*

- **__pycache__/** , es el directorio de caché para archivos compilados de Python.
- **__init__.py** permite que el directorio app sea tratado como un paquete de Python.
- **auth.py** está destinado para manejar la autenticación y autorización de usuarios.
- **crud.py** contiene las funciones CRUD (Crear, Leer, Actualizar y Borrar).
- **database.py** configura y gestiona la conexión a la base de datos.
- **email_config.py** configura el servicio de correo electrónico.
- **email_utils.py** proporciona utilidades para el manejo de correos electrónicos.
- **main.py** es el punto de entrada principal de la aplicación.
- **models.py** define los modelos de la base de datos (nuestras tablas en la base de datos).
- **schemas.py** define los esquemas de datos para la API.

6.2 Configuración del proyecto

6.2.1 Modelos y esquemas

El archivo **models.py** de este proyecto define los modelos de datos utilizando SQLAlchemy. Estos modelos representan las tablas de la base de datos y las relaciones entre ellas. SQLAlchemy permite interactuar con la base de datos de una manera más intuitiva mediante el uso de clases y objetos, lo que facilita el manejo y la manipulación de los datos [33].

Cada clase en este archivo representa una tabla en la base de datos:

- **Clase Cliente**: representa la tabla **clientes**.
  - Almacena información de clientes como nombre, apellidos, teléfono y email.

    - Define una relación uno-a-muchos con las citas (Cita), permitiendo asociar múltiples citas a un cliente.
- **Clase Cita**: representa la tabla **citas**.
    - Almacena información sobre las citas, incluyendo el cliente asociado, el empleado que realiza la cita, la fecha y el tratamiento.
    - Define relaciones muchos-a-uno con Cliente y Empleado.
- **Clase Empleado**: representa la tabla **empleados**.
    - Almacena información de empleados como nombre, apellidos, estado, nombre de usuario, PIN cifrado y rol (admin o empleado).
    - Define relaciones uno-a-muchos con Cita, TicketProducto y TicketServicio.
    - Se ha hecho uso de enums, para limitar los valores de los campos, por ejemplo, en el estado de un empleado se puede elegir entre activo o inactivo y en los roles, admin o empleado, así hacemos que los datos almacenados sean consistentes y válidos.
- **Clase Producto**: representa la tabla **productos**.
    - Almacena información de productos como nombre y precio.
    - Define una relación uno-a-muchos con TicketProducto.
- **Clase Servicio**: representa la tabla servicios.
    - Almacena información de servicios como nombre, precio y tipo (estética o peluquería).
    - Define una relación uno-a-muchos con TicketServicio.
- **Clase Ticket**: representa la tabla **tickets**.
    - Almacena información de los tickets, incluyendo la fecha, el importe total, el método de pago y el correo electrónico del cliente.
    - Define relaciones uno-a-muchos con TicketProducto y TicketServicio, con una opción de eliminación en cascada.
    - En esta clase se define un método `recalcular_importe_total` para calcular el importe total sumando los precios de los productos y servicios asociados al ticket.
- **Clase TicketProducto**: representa la tabla intermedia **ticket_productos**.
    - Almacena la relación entre Ticket y Producto, incluyendo la cantidad de productos, el precio y el total.
    - Define relaciones muchos-a-uno con Ticket, Producto y Empleado.
- **Clase TicketServicio**: representa la tabla intermedia **ticket_servicios**.
    - Almacena la relación entre Ticket y Servicio, incluyendo el precio del servicio.
    - Define relaciones muchos-a-uno con Ticket, Servicio y Empleado.

Las relaciones entre tablas se han realizado con el uso de `relationship`, para definir como las tablas se relacionan entre sí, como la relación uno-a-muchos entre cliente y cita o la relación muchos-a-uno entre cita y empleado.

Las relaciones entre Ticket, TicketProducto y TicketServicio tienen incluida la opción de eliminación en cascada `cascade="all, delete-orphan"`, esto garantiza que cuando se elimina un ticket, también se eliminan sus productos y servicios asociados.

```
clase cliente

class Cliente(Base):
    __tablename__ = "clientes"

    id = Column(Integer, primary_key=True, autoincrement=True)
    nombre = Column(String(255))
    apellidos = Column(String(255))
    telefono = Column(String(20))
    email = Column(String(255))

    # Define la relación con las citas asociadas a este cliente
    citas = relationship("Cita", back_populates="cliente")
```

Figura 58. Code snippet clase cliente

```
clase cita

class Cita(Base):
    __tablename__ = "citas"

    id = Column(Integer, primary_key=True, autoincrement=True)
    cliente_id = Column(Integer, ForeignKey("clientes.id"))
    empleado_id = Column(Integer, ForeignKey("empleados.id"))
    fecha = Column(Date)
    tratamiento = Column(Text)

    cliente = relationship("Cliente", back_populates="citas")
    empleado = relationship("Empleado", back_populates="citas")
```

Figura 59. Code snippet clase cita

El archivo **schemas.py** utiliza Pydantic, una herramienta de validación de datos en Python basada en tipos de datos [20]. Los modelos definidos en el código se utilizan para validar y estructurar datos en la aplicación.

- **Modelos de Usuario y Autenticación**:
  - `UserCreate`, `UserLogin` y `LoginData` definen los datos necesarios para la creación de nuevos usuarios y la autenticación de usuarios existentes.
- **Modelos de Cliente**:
  - `ClienteBase`, `ClienteCreate`, `ClienteUpdate` y `Cliente` se utilizan para manejar la información de clientes. `ClienteBase` define los campos básicos, mientras que `ClienteCreate` y `ClienteUpdate` permiten la creación y actualización de clientes. `Cliente` incluye un identificador único y configura los modelos para ser compatibles con ORMs (Object-Relational Mappers) gracias a orm_mode.
- **Modelos de Citas**:
  - `CitaBase`, `CitaCreate`, y `Cita` gestionan la información de las citas, incluyendo el cliente, el empleado, la fecha y el tratamiento. Similar a los modelos anteriores, orm_mode facilita la integración con ORMs.

```
schema cita

#------------CITA------------
class CitaBase(BaseModel):
    cliente_id: int
    empleado_id: int
    fecha: date
    tratamiento: str

class CitaCreate(CitaBase):
    pass


class Cita(CitaBase):
    id: int

    class Config:
        orm_mode = True
```

Figura 60. Code snippet ejemplo schema Cita

- **Modelos de Empleados**:
  - `EmpleadoBase`, `EmpleadoCreate`, `EmpleadoUpdate`, y `Empleado` definen y validan la información de los empleados, incluyendo su estado y si son administradores.
- **Modelos de Productos y Servicios**:
  - `ProductoBase`, `ProductoCreate`, `ProductoUpdate`, `Producto`, `ServicioBase`, `ServicioCreate`, `ServicioUpdate`, y `Servicio` gestionan los productos y servicios disponibles. Estos modelos definen campos como el nombre, precio y tipo para ambos, productos y servicios.
- **Modelos de Tickets**:
  - `TicketBase`, `TicketCreate`, `TicketUpdate`, `Ticket`, `TicketProductoBase`, `TicketProductoCreate`, `TicketProducto`, `TicketServicioBase`, `TicketServicioCreate`, `TicketServicio` definen y validan la información de los tickets. Estos incluyen detalles sobre la fecha, el importe total, y los productos y servicios asociados a cada ticket.

Todos los modelos heredan de BaseModel, lo que permite la validación automática de los datos según los tipos de los atributos definidos, esto garantiza que los datos cumplen con las especificaciones antes de ser procesados o almacenados.

La opción `orm_mode` en la configuración de los modelos, permite que los modelos Pydantic sean compatibles con ORMs como es SQLAlchemy. Esto significa que los modelos pueden ser utilizados directamente para la conversión entre datos de la base de datos y estructuras de datos en la aplicación.

ORM significa Object-Relational Mapping (Mapeo Objeto-Relacional). Es una técnica en programación que permite interactuar con bases de datos relacionales utilizando objetos en lugar de escribir directamente consultas SQL. Los ORMs son bibliotecas o frameworks que proporcionan una capa de abstracción sobre la base de datos, facilitando la interacción con ella mediante objetos y clases [34], en nuestro caso, nuestra ORM es SQLAlchemy.

Pydantic no solo valida los datos, sino que también los convierte a los tipos de datos correctos según las definiciones en los modelos.

6.2.2 Conexión a la base de datos

El archivo **database.py** configura la conexión a la base de datos utilizando SQLAlchemy. Para ello se ha importado `create_engine`, función de SQLAlchemy que crea una instancia `Engine`, la cual se usa para interactuar con la base de datos, `declarative_base`, función que devuelve una clase base para declarar modelos ORM y `sessionmaker`, función que crea una clase de fábrica para sesiones de base de datos.

Después creamos la cadena de conexión específica para la base de datos, en este caso MySQL, con el usuario, contraseña, dirección del servidor y el nombre de la base de datos. Esto lo definimos anteriormente cuando creamos el Docker para almacenar la base de datos.

A continuación, se ha creado una instancia de `Engine` que se conecta a la base de datos especificada en la cadena creada anteriormente, con esto podremos ejecutar consultas y transacciones en la base de datos. Con `sessionmaker` se crea una fábrica de sesiones, para poder realizar las acciones en la base de datos y, por último, se crea una base a partir de la cual se pueden definir los modelos ORM, esta clase contiene metadatos sobre las tablas y mapeos de la base de datos.

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

#DATABASE URL for SQLAlchemy
SQLALCHEMY_DATABASE_URL = "mysql://usuario:********@127.0.0.1/nombre_base_de_datos"

#SQLAlchemy engine
engine = create_engine(
    SQLALCHEMY_DATABASE_URL
)

#SessionLocal class
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

#Base class
Base = declarative_base()
```

Figura 61. Code snippet database.py

6.2.3 Autenticación y autorización

El archivo **auth.py** está destinado para contener funciones para la autenticación de usuarios en nuestra aplicación web. Para ello, se ha importado `passlib.context.CryptContext`, que es una herramienta de `passlib` [35] para manejar el *hashing* y verificación de contraseñas.

Hashing es el proceso de transformar una entrada (en este caso, una contraseña en texto plano), en una cadena de caracteres de longitud fija, que parece aleatoria. Este proceso es irreversible, lo que significa que no se puede convertir el hash de vuelta a la contraseña original. El hashing se utiliza para almacenar contraseñas de manera segura en bases de datos, lo que lo hace perfecto para nuestro propósito. Hay diferentes algoritmos de hashing, como SHA-256, MD4 y bcrypt.

En lo que a contraseñas respecta, bcrypt es preferido debido a su resistencia a ataques de fuerza bruta y su capacidad para ser configurado para ser más lento, lo que dificulta aún más los ataques. Además, también maneja el *salting*, esto es agregar una cadena aleatoria y única de caracteres a la contraseña original antes de que se haga el hash. Esto aborda el problema de las contraseñas duplicadas, ya que, si incluso dos usuarios tienen contraseñas idénticas, sus versiones hash van a ser diferentes gracias al *salting* [36].

Figura 62. Esquema traducido de [35].

También se han incluido en este archivo `sqalchemy.orm.Session` que es un módulo de SQLAlchemy para manejar sesiones de la base de datos y el modelo `Empleado` de la base de datos. Después, creamos el contexto de la contraseña con el comando `pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")`, configurando el contexto de hashing de contraseñas con el algoritmo bcrypt.

A continuación, se han definido tres funciones, `verify_password`, `get_password_hash` y `authenticate_empleado`: para verificar si una contraseña en texto plano introducida por el usuario coincide con su hash almacenado, generar un hash seguro para una contraseña en texto plano y para autenticar a un empleado verificando su nombre de usuario y contraseña, respectivamente.

6.2.4 Envío de correos electrónicos

Los archivos **email_config.py** y **email_utils.py** configuran y envían correos electrónicos utilizando el servidor SMPT de Gmail y plantillas HTML.

El archivo **email_config.py** contiene la configuración necesaria para conectar con el servidor SMPT de Gmail, se definen los parámetros esenciales para el envío de correos electrónicos, como el servidor, el puerto y las credenciales de la cuenta de correo y la información del remitente. Está pensado para que la peluquería que utiliza la aplicación web tenga un correo de Gmail con el cual envíe por correo a los clientes el ticket de su visita.

```python
#email_config.py

#configuracion del servidor SMTP de Gmail
SMTP_SERVER = 'smtp.gmail.com'
SMTP_PORT = 587 #Puerto para SMTP con TLS

#credenciales de la cuenta de correo de gmail
SMTP_USER = 'dirección de correo remitente'
SMTP_PASSWORD = 'contraseña generada'

#nombre y direccion del remitente
FROM_NAME = 'Tu Peluqueria de Prueba'
FROM_EMAIL = SMTP_USER
```

Figura 63. Code snippet email_config.py.

Para que funcione correctamente hay que seguir los siguientes pasos para habilitar que la cuenta de correo mande correos a través de esta aplicación: el uso de contraseñas de aplicaciones, esta es la opción más segura, especialmente si se tiene habilitada la autenticación en dos pasos en la cuenta de Google. Hay que hacer lo siguiente:

1. Habilitar la autenticación en dos pasos
2. Generar una contraseña de aplicación:
    a. En la sección de Seguridad, después de habilitar la verificación en dos pasos, hay una opción que es Contraseñas de aplicación, hay que seleccionarla.
    b. Seleccionar el tipo de aplicación, y darle los datos que nosotros consideremos para poder identificar para qué es la contraseña que vamos a crear.
    c. Pulsamos en generar y copiamos esa contraseña.

Esa contraseña copiada es la que tenemos que incluir en nuestro código en email_config.py como SMTP_PASSWORD.

El archivo **email_utils.py** contiene una función para enviar correos electrónicos utilizando el protocolo SMTP (Simple Mail Transfer Protocol), utiliza la biblioteca estándar de Python `smtplib`, la clase `MIMEMultipart` para crear un mensaje de correo electrónico con múltiples partes, `MIMEText` para crear una parte del mensaje de correo electrónico que contiene texto, la función `formataddr` para formatear la dirección de correo electrónico con un nombre, `jinja2` para renderizar la plantilla HTML a seguir para formar el correo y por último el archivo **email_config.py** que tiene la configuración del servidor SMTP.

La función se denomina `enviar_correo` y recibe como valores: la dirección de correo electrónico del destinatario, el asunto, el nombre del archivo de plantilla de HTML y el diccionario de datos que se pasarán a la plantilla para renderizar el contenido del correo.

El remitente y la contraseña las obtenemos del archivo email_config.py. Con ayuda de `MIMEMultipart` creamos un mensaje de correo con múltiples partes, como el remitente, el destinatario y el asunto del correo electrónico. Después creamos la conexión con el servidor SMTP especificado e iniciamos la conexión utilizando TLS (Transport Layer Security), protocolo criptográfico diseñado para proporcionar comunicaciones seguras a través de una red informática; se autentica el remitente con su dirección y contraseña y se envía el mensaje de correo.

```
try:
        with smtplib.SMTP(email_config.SMTP_SERVER, email_config.SMTP_PORT) as server:
            server.starttls()
            server.login(remitente, contraseña)
            server.send_message(msg)
        print("Correo enviado exitosamente")
    except Exception as e:
        print(f"Error al enviar correo: {e}")
```

Figura 64. Code snippet conexión con el servidor SMTP

6.2.5 Funciones CRUD

El archivo **crud.py** proporciona una serie de funciones para realizar operaciones CRUD (Crear Leer, Actualizar y Eliminar). Estas funciones están organizadas en diferentes secciones: usuarios, clientes, citas, empleados, productos, servicios, caja y bonos.

Por cada función que queramos hacer en la base de datos, tendremos que crear una función en crud para manejarlo. De esta manera tendremos las siguientes funciones:

- **Función para recuperar un cliente/empleado/producto/servicio/ticket/productos de un ticket/servicios de un ticket por su ID**.

```
def get_cliente(db: Session, cliente_id: int):
    return db.query(models.Cliente).filter(models.Cliente.id == cliente_id).first()
```

Figura 65. Code snippet ejemplo función para recuperar un cliente por su ID

Esta función recupera un cliente específico de la base de datos utilizando su `cliente_id`. La función toma como parámetros una sesión de la base de datos (`db`) y el identificador único del cliente (`cliente_id`). Utiliza una consulta SQLAlchemy para filtrar los registros de la tabla `Cliente` y devuelve el primer resultado que coincide con el `cliente_id` proporcionado.

- **Función para recuperar la lista de clientes/empleados/productos/servicios**.

```
def get_clientes(db: Session, skip: int = 0, limit: int = 100):
    clientes = db.query(models.Cliente).offset(skip).limit(limit).all()
    return clientes
```

Figura 66. Code snippet ejemplo función para recuperar lista clientes

Permite recuperar una lista de clientes de la base de datos con soporte para paginación. Recibe como parámetros una sesión de la base de datos (`db`), un valor de desplazamiento (`skip`) y un límite de registros (`limit`). Utiliza estos parámetros para omitir un número específico de registros y limitar la cantidad de resultados devueltos.

En empleados, hay una función denominada `get_empleados_activos`, que funciona de la misma manera, pero está filtrado para obtener en la lista solo los empleados que tengan su estado definido como "activo".

- **Función para crear un nuevo cliente/empleado/producto/servicio/ticket.**

```
def create_cliente(db: Session, cliente: schemas.ClienteCreate):
    db_cliente = models.Cliente(**cliente.model_dump())
    db.add(db_cliente)
    db.commit()
    db.refresh(db_cliente)
    return db_cliente
```

Figura 67. Code snippet ejemplo función crear nuevo cliente.

Se encarga de crear un nuevo cliente en la base de datos. Recibe una sesión de la base de datos (`db`) y un objeto `ClienteCreate` que contiene los datos del nuevo cliente. La función crea una instancia de `Cliente` utilizando los datos proporcionados, la añade a la sesión de la base de datos, realiza un commit para guardar los cambios y refresca la instancia para obtener los datos actualizados. Finalmente, devuelve el objeto `Cliente` recién creado.

- **Función para actualizar un cliente/empleado/producto/servicio.**

```
def update_cliente(db: Session, cliente: models.Cliente, cliente_update: schemas.ClienteUpdate):
    for var, value in vars(cliente_update).items():
        setattr(cliente, var, value)
    db.commit()
    db.refresh(cliente)
    return cliente
```

Figura 68. Code snippet ejemplo función para actualizar un cliente

Permite actualizar los datos de un cliente existente en la base de datos. Recibe como parámetros una sesión de la base de datos (`db`), un objeto `Cliente` existente (`cliente`) y un objeto `ClienteUpdate` que contiene los nuevos datos del cliente. La función itera sobre los atributos del objeto `ClienteUpdate` y actualiza los valores correspondientes en el objeto `Cliente`. Luego, realiza un commit para guardar los cambios y refresca la instancia para obtener los datos actualizados. Finalmente, devuelve el objeto `Cliente` actualizado.

- **Función para eliminar un producto/servicio/cliente**

```
def delete_producto(db: Session, producto: models.Producto):
    db.delete(producto)
    db.commit()
    return producto return go(f, seed, [])
}
```

Figura 69. Code snippet ejemplo función para eliminar un producto

Tiene como objetivo eliminar un producto de la base de datos utilizando SQLAlchemy. Recibe dos parámetros: una sesión de base de datos (`db`) y una instancia del modelo Producto (`producto`) que se desea eliminar.

- **Función para buscar un cliente/empleado/producto/servicio por su nombre o apellidos.**

```
def get_cliente_by_nombre_apellidos(db: Session, nombre: str, apellidos: str):
    return db.query(models.Cliente).filter(models.Cliente.nombre == nombre, models.Cliente.apellidos == apellidos).first()
```

Figura 70. Code snippet ejemplo función para obtener un cliente por nombre y apellidos

Esta función recupera un cliente específico de la base de datos utilizando su nombre y apellidos. Recibe como parámetros una sesión de la base de datos (`db`), el nombre (`nombre`) y los apellidos (`apellidos`) del cliente. Utiliza una consulta SQLAlchemy para filtrar los registros de la tabla `Cliente` que coinciden con el nombre y apellidos proporcionados, y devuelve el primer resultado encontrado.

- **Función para añadir producto/servicio al ticket.**

```
def add_producto_to_ticket(db: Session, ticket_id: int, producto_id: int, empleado_id: int, precio: float, cantidad: int):
    try:
        ticket_producto = models.TicketProducto(
            ticket_id=ticket_id,
            producto_id=producto_id,
            empleado_id=empleado_id,
            cantidad=cantidad,
            precio=precio
        )
        db.add(ticket_producto)
        db.commit()
        db.refresh(ticket_producto)

        # Recalcular el importe_total del ticket
        ticket = db.query(models.Ticket).filter(models.Ticket.id == ticket_id).first()
        if ticket:
            total_productos = db.query(func.sum(models.TicketProducto.precio * 
models.TicketProducto.cantidad)).filter(models.TicketProducto.ticket_id == ticket_id).scalar() or 0
            total_servicios = db.query(func.sum(models.TicketServicio.precio)).filter(models.TicketServicio.ticket_id == ticket_id).scalar() or 0
            ticket.importe_total = total_productos + total_servicios
            db.commit()

        return ticket_producto
    except Exception as e:
        db.rollback()
        raise e
```

Figura 71. Code snippet función ejemplo añadir producto a ticket

Tiene como objetivo agregar un producto a un ticket en la base de datos. Recibe como parámetros una sesión de base de datos (`db`), el identificador del ticket (`ticket_id`), el identificador del producto (`producto_id`), el identificador del empleado (`empleado_id`), el precio del producto (`precio`) y la cantidad del producto (`cantidad`).

Dentro de la función, se crea una instancia del modelo `TicketProducto` con los datos proporcionados y se añade a la sesión de la base de datos. Posteriormente, se confirma la transacción y se actualiza la instancia recién añadida.

Luego, se recalcula el importe total del ticket asociado. Para ello, se consulta el ticket correspondiente y se suman los precios de los productos y servicios asociados al ticket. El importe total del ticket se actualiza con esta suma y se confirma nuevamente la transacción.

- **Función para finalizar un ticket.**

  `finalizar_ticket` recibe como parámetros una sesión de base de datos (`db`), el identificador del ticket (`ticket_id`), el método de pago (`metodo_pago`) y opcionalmente un correo electrónico (`correo`).

  Inicialmente, la función recupera el ticket correspondiente al identificador proporcionado. Luego, valida que el método de pago sea uno de los permitidos, específicamente "efectivo" o "tarjeta". Si el método de pago no es válido, se lanza una excepción HTTP con un código de estado 400.

  A continuación, se actualizan los campos del ticket con el método de pago y, si se proporciona, el correo electrónico. Los cambios se confirman en la base de datos y se refresca la instancia del ticket para asegurar que los cambios se reflejen correctamente.

  La función procede a obtener los productos y servicios asociados al ticket, incluyendo detalles completos mediante consultas que unen las tablas correspondientes. Con esta información, se construye una respuesta que incluye el identificador del ticket, la fecha, el importe total, el método de pago, el correo electrónico, y listas detalladas de los productos y servicios.

  Si se proporciona un correo electrónico, se prepara un contexto con la información del ticket y se envía un correo de confirmación utilizando una plantilla HTML específica.

  Finalmente, la función retorna la respuesta construida con todos los detalles del ticket finalizado.

- **Función para obtener cajas en periodo de tiempo.**

  De este tipo de función contamos con dos, una para obtener la caja hecha por los empleados en un periodo de tiempo y la caja general en un periodo de tiempo. Ambas funcionan de la misma manera:

  Recibe como parámetros una sesión de base de datos (`db`), una fecha de inicio (`start_date`) y una fecha de fin (`end_date`).

La función realiza una consulta a la base de datos para obtener varios agregados sobre los tickets emitidos en el periodo especificado. Estos agregados incluyen el número total de tickets, el número de tickets pagados en efectivo, el número de tickets pagados con tarjeta, el importe total de la caja, el importe total pagado en efectivo y el importe total pagado con tarjeta. La consulta utiliza funciones de agregación y condiciones para calcular estos valores.

Una vez ejecutada la consulta, si se obtienen resultados, la función retorna un diccionario con las fechas de inicio y fin del periodo, así como los valores calculados. En caso contrario, retorna `None`.

6.3 Implementación de *endpoints*

En el desarrollo de aplicaciones web y API, un *endpoint* es una dirección URL que representa una ruta en el servidor donde los clientes pueden enviar solicitudes. Los *endpoints* son puntos de acceso que permiten a los clientes interactuar con el servidor para realizar operaciones como crear, leer, actualizar y eliminar recursos.

En este proyecto, hemos utilizado 4 tipos de métodos HTTP:

- **GET**: para recuperar datos del servidor.
- **POST**: enviar datos al servidor para crear un nuevo recurso.
- **PUT**: actualizar un recurso existente en el servidor.
- **DELETE**: eliminar un recurso del servidor.

Para la comunicación entre cliente y servidor, se ha utilizado el formato JSON, que es un formato de datos ligero y fácil de leer y escribir tanto para humanos como para máquinas. Los *endpoints* envían y reciben datos en formato JSON, lo que permite una interacción eficiente y estructurada con la API. Los endpoints en nuestro proyecto se han creado en **main.py** el archivo principal de la API.

6.3.1 Configuración de la API

```
models.Base.metadata.create_all(bind=engine)

app = FastAPI()

app.add_middleware(SessionMiddleware, secret_key="!secret")

app.mount("/static", StaticFiles(directory="./static"), name="static")

templates = Jinja2Templates(directory="templates")

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Figura 72. Code snippet comandos para configurar la API

Primero se han creado las tablas en la base de datos, con la línea `models.Base.metadata.create_all (bind=engine)`, encargada de crear todas las tablas definidas en los modelos (**models.py**). Después, se inicializa la aplicación FastAPI, con la instancia `app = FastAPI()`, que crea una nueva aplicación. A continuación, añadimos un middleware de sesión a la aplicación, para gestionar sesiones de usuario, utilizando una clave secreta para asegurar la integridad y confidencialidad de los datos de sesión.

También configuramos la aplicación para servir los archivos estáticos correctamente, que estarán accesibles desde la ruta `/static` y configuramos el motor de plantillas de Jinja2, pudiendo encontrarlas en el directorio `/template`. Por último, creamos la dependencia de la base de datos con `get_db()`, que proporciona una sesión de base de datos para las rutas de la aplicación, utiliza un generador para abrir una sesión con `SessionLocal()`, y asegura que la sesión se cierre correctamente después de su uso, independientemente de si la operación fue exitosa o no.

Cuando queramos poner en marcha nuestra API, tendremos que escribir en el terminal de nuestro proyecto: `uvicorn app.main:app –reload.`

6.3.2 *Endpoints* creados

Páginas estáticas

- **GET /:** Muestra la página de inicio de sesión.
- **GET /index**: Muestra la página principal después de iniciar sesión.
- **GET /bonos**: Muestra la página de bonos, accesible solo para administradores.
- **GET /caja**: Muestra la página de caja con la lista de empleados y empleados activos.
- **GET /mantenimiento**: Muestra la página de mantenimiento.
- **GET /paginaclientes**: Muestra la página de clientes con la lista de clientes y empleados.
- **GET /paginaempleados**: Muestra la página de empleados, accesible solo para administradores.
- **GET /paginaproductos**: Muestra la página de productos con la lista de productos.
- **GET /paginaservicios**: Muestra la página de servicios filtrados por tipo.

- **GET /caja_formatoTicket**: Muestra el formato del ticket de caja.
- **GET /caja_empleados**: Muestra la página de empleados en la sección de caja.

Clientes

- **POST /paginaclientes**: Crea un nuevo cliente.
- **GET /clientes/{cliente_id}**: Obtiene información de un cliente específico por ID.
- **PUT /clientes/{cliente_id}**: Actualiza la información de un cliente específico.
- **DELETE /clientes/{cliente_id}**: Elimina un cliente específico.
- **GET /clientes/{cliente_id}/citas/**: Obtiene las citas de un cliente específico.
- **POST /clientes/{cliente_id}/citas/**: Crea una nueva cita para un cliente específico.

Empleados

- **POST /paginaempleados**: Crea un nuevo empleado.
- **GET /empleados/{empleado_id}**: Obtiene información de un empleado específico por ID.
- **PUT /empleados/{empleado_id}**: Actualiza la información de un empleado específico.
- **GET /empleados/{empleado_id}**: Obtiene la información del empleado que atendió una cita específica.

Productos

- **POST /paginaproductos**: Crea un nuevo producto.
- **GET /productos/{producto_id}**: Obtiene información de un producto específico por ID.
- **PUT /productos/{producto_id}**: Actualiza la información de un producto específico.
- **DELETE /productos/{producto_id}**: Elimina un producto específico.

Servicios

- **POST /paginaservicios**: Crea un nuevo servicio.
- **GET /servicios/{servicio_id}**: Obtiene información de un servicio específico por ID.
- **PUT /servicios/{servicio_id}**: Actualiza la información de un servicio específico.
- **DELETE /servicios/{servicio_id}**: Elimina un servicio específico.

Caja

- **GET /api/productos**: Obtiene la lista de productos.
- **GET /api/servicios/{tipo}**: Obtiene la lista de servicios filtrados por tipo.
- **POST /api/tickets/add-producto**: Añade un producto a un ticket.
- **POST /api/tickets/add-servicio**: Añade un servicio a un ticket.
- **POST /api/tickets/create**: Crea un nuevo ticket.
- **GET /api/tickets/{ticket_id}**: Obtiene información de un ticket específico.

- **POST /api/tickets/{ticket_id}/finalizar**: Finaliza un ticket.
- **POST /api/tickets/{ticket_id}/cancelar**: Cancela un ticket.
- **POST /api/tickets/{ticket_id}/items/{item_id}/eliminar**: Elimina un ítem de un ticket.

Bonos

- **GET /api/caja/tickets**: Obtiene información de caja en un periodo específico.
- **GET /api/caja/ticketsempleados**: Obtiene información de caja por empleados en un periodo específico.

```python
@app.post("/paginaclientes", response_model=schemas.Cliente)
async def create_cliente(cliente_create: schemas.ClienteCreate, db: Session = Depends(get_db)):
    # Verificar si el cliente ya existe en la base de datos por su nombre y apellidos
    db_cliente = crud.get_cliente_by_nombre_apellidos(db, nombre=cliente_create.nombre,
apellidos=cliente_create.apellidos)
    if db_cliente:
        # Si el cliente ya existe, devolver un error
        raise HTTPException(status_code=400, detail="Cliente ya existente")
    else:
        # Si el cliente no existe, crear un nuevo cliente en la base de datos
        new_cliente = crud.create_cliente(db=db, cliente=cliente_create)
        return new_cliente  # Devuelve el nuevo cliente como JSON
```

Figura 73. Code snippet ejemplo de endpoint

Cada *endpoint* utiliza la correspondiente función creada en **crud.py** destinada para la funcionalidad y resultado que buscamos.

Los endpoints relacionados con la autenticación los vamos a explicar con más profundidad en el siguiente apartado.

6.3.2.1 *Endpoints* autenticación

La autenticación en esta aplicación es un proceso muy importante para garantizar que solo los usuarios autorizados puedan acceder a ciertos recursos y funcionalidades.

@app.post("/login") - *endpoint* de inicio de sesión

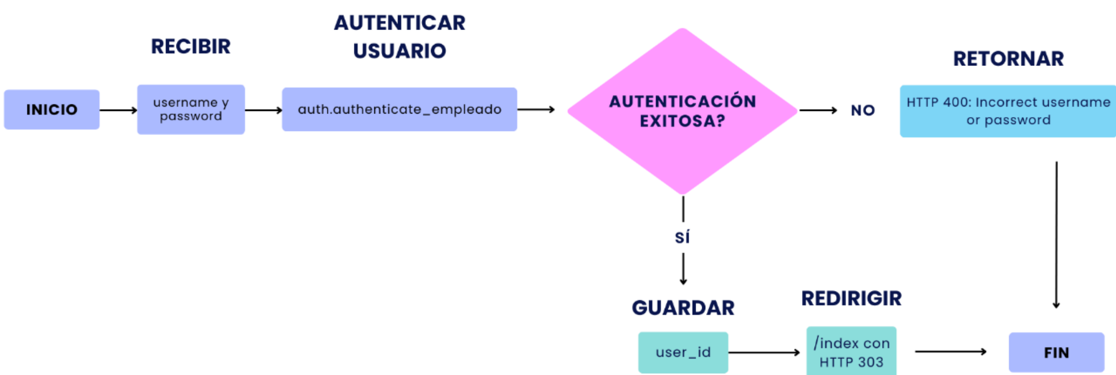


Figura 74. Diagrama de flujo endpoint POST /login

El servidor recibe una solicitud `POST` al *endpoint* `/login` con los datos del formulario (nombre de usuario y contraseña), se obtiene una sesión de la base de datos usando `Depends(get_db)` y después se llama a `auth.authenticate_empleado(db, username, password)`, para buscar en la base de datos un empleado que coincida con el nombre de usuario y la contraseña proporcionados. Si el empleado no es encontrado, se lanza una excepción con el código 400, y si es encontrado se guarda su ID en la sesión del usuario `request.session['user_id']`, por último, se redirige al usuario a la página `/index`.

@app.post("/logout") – *endpoint* para cerrar sesión

Figura 75. Diagrama de flujo endpoint POST logout

El servidor recibe una solicitud `POST` al *endpoint* `/logout`, se elimina la información del usuario de la sesión utilizando `request.session.pop('user_id', None)` y el usuario es redirigido a la página `/login`.

@app.get("/login", response_class = HTMLResponse) – endpoint mostrar página inicio de sesión.

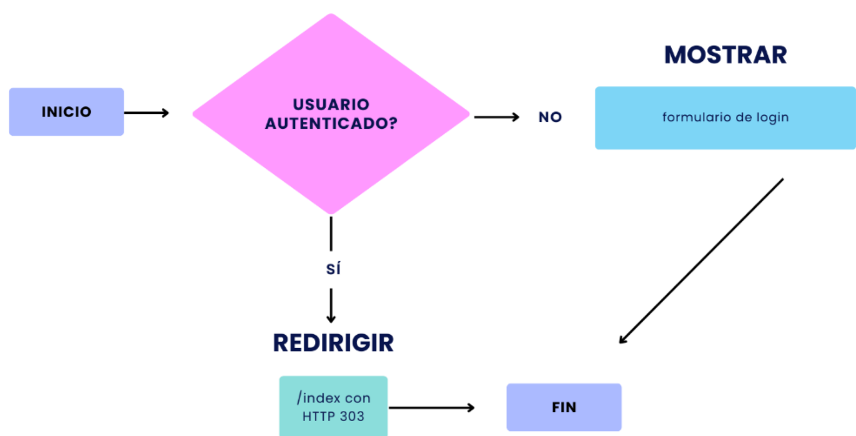


Figura 76. Diagrama de flujo endpoint GET login

El servidor recibe una solicitud `GET` al *endpoint* `/login` y se renderiza la plantilla **login.html**, mostrándose en el navegador.

6.3.2.2 *Endpoints* páginas protegidas

En las páginas que necesitan una autorización para entrar, es importante verificar si el usuario está presente en la sesión, si no lo está, se le redirige a la página de inicio sesión `/login`, si lo está, consultaremos en la base de datos si el usuario es un empleado y si tiene permisos de administrador, si no es administrador, se redirige al usuario a la página de acceso no autorizado `/unauthorized`, y si son exitosas las verificaciones, se mostrará la página en la que se quiere acceder. Las páginas que en nuestra aplicación están restringidas, son **bonos.html** y **paginaempleados.html**

```
@app.get("/bonos", response_class=HTMLResponse)
async def serve_bonos_page(request: Request, db: Session = Depends(get_db)):
    user_id = request.session.get('user_id')
    if not user_id:
        return RedirectResponse(url="/login", status_code=status.HTTP_302_FOUND)
    empleado = db.query(models.Empleado).filter(models.Empleado.id == user_id).first()
    if not empleado or empleado.is_admin != 'admin':
        return RedirectResponse(url="/unauthorized", status_code=status.HTTP_302_FOUND)
    return templates.TemplateResponse("bonos.html", {"request": request})
```

Figura 77. Code snippet ejemplo de endpoint protegido

Capítulo 7. CONCLUSIONES Y LINEAS FUTURAS

7.1 Conclusiones

Este proyecto se ha enfocado en la creación y desarrollo de una aplicación web con el propósito de gestionar diversos aspectos de una peluquería. Los objetivos establecidos en el primer capítulo de la presente memoria han sido alcanzados a medida que se ha desarrollado el trabajo, lo que ha permitido la creación de un sistema vistoso y funcional que satisface los requisitos requeridos.

El desarrollo de esta aplicación web ha sido una experiencia enriquecedora que me ha permitido aplicar y expandir conocimientos en diversas áreas del desarrollo de software, incluyendo tanto la programación de *front-end* como de *back-end*, la gestión de bases de datos, la seguridad informática y el diseño de interfaces de usuario. Los resultados obtenidos demuestran la viabilidad y eficacia de FastAPI como framework para el desarrollo de aplicaciones web modernas y escalables.

Este proyecto ha hecho que tenga motivación por aplicar los conocimientos aprendidos, e investigar y aprender de manera autodidacta recursos nuevos que aplicar en mi trabajo. Además, quiero destacar que este proyecto lo comencé durante mi estancia Erasmus en la University of Zagreb en Croacia, de manera que el simple hecho de tener este trabajo en mente me hizo buscar asignaturas que me fueran a resultar útiles de cara a obtener conocimientos aplicables en la aplicación web, por lo que ha sido toda una motivación para mi completar este proyecto.

La aplicación desarrollada no solo cumple con los requisitos iniciales, sino que también establece una base sólida para futuras expansiones y mejoras, asegurando su relevancia y utilidad a largo plazo. Este proyecto subraya la importancia de la planificación meticulosa, la implementación cuidadosa y la revisión constante para el éxito en el desarrollo de software.

Como futura desarrolladora de webs, resulta muy satisfactorio crear una aplicación que resultará útil y mejorará el negocio de alguien tan importante como lo es mi madre.

7.2 Líneas futuras

A pesar de los significativos logros alcanzados, siempre existe un potencial para la mejora continua. A continuación, se presentan algunas áreas clave que podrían beneficiarse de futuras optimizaciones:

- **Adaptación a dispositivos móviles**: Implementar una versión responsive de la aplicación que sea completamente funcional en tablets y teléfonos inteligentes. Esta adaptación permitirá que la peluquería pueda utilizar terminales móviles en lugar de ordenadores de escritorio o portátiles, facilitando así el acceso a la aplicación desde cualquier ubicación,

siempre y cuando el servidor se encuentre operativo. Esta funcionalidad no solo optimizaría la flexibilidad operativa, sino que también mejoraría la accesibilidad y la comodidad para los usuarios en movimiento.

- **Incorporación de nuevas funcionalidades**: Evaluar e integrar nuevas funcionalidades basadas en los requisitos emergentes de los usuarios, que incluyen tanto a los administradores como a los empleados de la peluquería. Esto puede implicar la adición de características adicionales o la mejora de las existentes para satisfacer mejor las necesidades y expectativas de los usuarios.

  Una de las funcionalidades a incorporar, es cambiar "eliminar" en productos y servicios, por "desactivar", de una manera parecida a cómo se realiza en empleados. Ya que, a la hora de eliminar un producto o servicio, si este está asociado a un ticket pasado, puede darnos error y no obtener los datos de manera correcta.

  Una continua retroalimentación y análisis de las necesidades operativas permitirán adaptar la aplicación de manera más efectiva a los procesos específicos de la peluquería.
- **Soporte multilenguaje**: Expandir la aplicación para que sea compatible con múltiples idiomas, facilitando su uso en entornos multiculturales o en ubicaciones con un idioma diferente al español. Esto no solo mejoraría la accesibilidad, sino que también podría atraer a una base de usuarios más amplia.

Estas iniciativas contribuirán significativamente a la evolución de la aplicación, garantizando que continúe satisfaciendo las necesidades cambiantes y proporcionando un valor duradero a los usuarios.

Capítulo 8. REFERENCIAS

[1] «Marco Estratégico en política de PYME 2030,» [En línea]. Available: https://industria.gob.es/es-es/Servicios/MarcoEstrategicoPYME/Marco%20Estrat%C3%A9gico%20PYME.pdf.

[2] «Digitalisation in Europe – 2023 edition,» [En línea]. Available: https://ec.europa.eu/eurostat/web/interactive-publications/digitalisation-2023?etrans=es.

[3] «España Digital 2026,» [En línea]. Available: https://portal.mineco.gob.es/en-us/ministerio/estrategias/Pages/00_Espana_Digital.aspx.

[4] «Software TPV para gestión de Peluquerías,» [En línea]. Available: https://www.solvermedia.com/software-tpv-para-gestion-de-peluquerias.html.

[5] «Software de Gestión para Peluquerías,» [En línea]. Available: https://www.shortcuts.es/soluciones/software-peluqueria/.

[6] «Las mejores metodologías para un correcto desarrollo de software,» [En línea]. Available: https://www.occamagenciadigital.com/blog/las-mejores-metodologias-para-un-correcto-desarrollo-de-software.

[7] «Metodologías de desarrollo de software: ¿qué son?,» [En línea]. Available: https://www.santanderopenacademy.com/es/blog/metodologias-desarrollo-software.html .

[8] «HTML Introduction,» [En línea]. Available: https://www.w3schools.com/html/html_intro.asp .

[9] «CSS Tutorial,» [En línea]. Available: https://www.w3schools.com/css/default.asp .

[10] «Tecnología para desarrolladores web : JavaScript,» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript .

[11] «Las 40 mejores bibliotecas y frameworks de JavaScript para 2024,» [En línea]. Available: https://kinsta.com/es/blog/bibliotecas-javascript/ .

[12] «React,» [En línea]. Available: https://es.react.dev/ .

[13] «Introduction,» [En línea]. Available: https://flatpickr.js.org/ .

[14] «Crear PDF con JavaScript y jsPDF,» [En línea]. Available: https://libreriasjs.com/libreria-javascript-crear-pdf-jspdf/ .

[15] "What is Angular?," [Online]. Available: https://angular.dev/overview .

[16] «Bootstrap: guía para principiantes de qué es, por qué y cómo usarlo,» [En línea]. Available: https://rockcontent.com/es/blog/bootstrap/ .

[17] «Qué es Node.js y Por qué Debes Usarlo,» [En línea]. Available: https://imaginaformacion.com/tutoriales/que-es-nodejs-y-por-que-debes-usarlo .

[18] «Introducción a Django,» [En línea]. Available: https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Introduction .

[19] «Qué es Flask,» [En línea]. Available: https://openwebinars.net/blog/que-es-flask/#:~:text=Te%20contamos%20qu%C3%A9%20es%20Flask,con%20pocas%20lineas%20de%20c%C3%B3digo.&text=En%20la%20actualidad%20existen%20muchas,sencilla%20aplicaciones%20web%20con%20Python. .

[20] «FastAPI: La herramienta definitiva para el desarrollo web,» [En línea]. Available: https://opensistemas.com/fastapi-la-herramienta-para-el-desarrollo-web/ .

[21] "The Python SQL Toolkit and Object Relational Mapper," [Online]. Available: https://www.sqlalchemy.org/ .

[22] «¿Qué es MySQL? Explicación y características,» [En línea]. Available: https://www.arsys.es/blog/mysql#:~:text=Qu%C3%A9%20es%20MySQL%3F-,MySQL%20es%20un%20sistema%20de%20gesti%C3%B3n%20de%20bases%20de%20datos,recuperar%20datos%20de%20manera%20eficiente. .

[23] «Descubre PostgreSQL: qué es, cómo funciona y ventajas,» [En línea]. Available: https://platzi.com/blog/que-es-postgresql/ .

[24] «sqlite3 — DB-API 2.0 interfaz para bases de datos SQLite,» [En línea]. Available: https://docs.python.org/es/3.8/library/sqlite3.html#:~:text=SQLite%20es%20una%20biblioteca%20de,usar%20SQLite%20para%20almacenamiento%20interno. .

[25] «¿Qué es MongoDB?,» [En línea]. Available: https://www.ibm.com/es-es/topics/mongodb .

[26] «Qué es GitHub y cómo empezar a usarlo,» [En línea]. Available: https://www.hostinger.es/tutoriales/que-es-github .

[27] «Qué es Canva y cómo usarlo para crear diseños profesionales,» [En línea]. Available: https://thepower.education/blog/que-es-canva-y-como-usarlo-para-crear-disenos-profesionales .

[28] «¿Qué es Docker y cómo funciona?,» [En línea]. Available: https://www.redhat.com/es/topics/containers/what-is-docker .

[29] «HeidiSQL: ¿qué es y para qué sirve?,» [En línea]. Available: https://www.arsys.es/blog/heidisql-que-es-y-para-que-sirve

[30] «Jinja,» [En línea]. Available: https://palletsprojects.com/p/jinja/ .

[31] «carbon,» [En línea]. Available: https://carbon.now.sh/ .

[32] «Figtree,» [En línea]. Available: https://fonts.google.com/specimen/Figtree/about?preview.text=Inicio .

[33] «SQL (Relational) Databases,» [En línea]. Available: https://fastapi.tiangolo.com/tutorial/sql-databases/ .

[34] «¿Qué es un ORM?,» [En línea]. Available: https://www.dreams.es/transformacion-digital/desarrolladores-paginas-web/que-es-un-orm .

[35] «passlib 1.7.4,» [En línea]. Available: https://pypi.org/project/passlib/ .

[36] «¿Qué es el hash de contraseñas?,» [En línea]. Available: https://www.dashlane.com/es/blog/que-es-el-hash-de-contrasenas .