UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Politècnica Superior d'Alcoi

Universitat Politècnica de València

Aplicación Web para la gestión de un salón de belleza

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Domínguez Victoriano

Director Proyecto: Javier Esparza Peidro

Marzo 2021

Resumen

Desarrollar una aplicación web para la gestión de un salón de belleza, utilizando el framework PHP de Symfony.

Mediante la aplicación web, se pretende llevar a cabo todo el control del salón de belleza, tanto a nivel financiero como a nivel organizativo y de producción.

Los empleados del salón de belleza podrán registrar nuevos clientes, dar citas a los clientes y tendrán acceso al historial de servicios de los clientes.

El jefe del salón de belleza además podrá dar de alta a nuevos trabajadores, tener acceso al apartado de servicios, de proveedores, al cobro de servicios y de productos y al apartado de estadísticas.

Palabras clave: symfony, php, framework, base de datos, apache.

Resum

Desenvolupar una aplicació web per a la gestió d'un saló de bellesa, utilitzant el framework PHP de Symfony.

Mitjançant l'aplicació web, es pretén dur a terme tot el control del saló de bellesa, tant a nivell financer com a nivell organitzatiu i de producció.

Els empleats del saló de bellesa podran enregistrar nous clients, donar cites als clients i tindran accés a l'historial de servicis dels clients.

El cap del saló de bellesa també podrà donar de alta a nous treballadors, tindre accés a l'apartat de servicis, de proveïdors, al cobrament de servicis i productes i al apartat d'estadístiques.

Paraules clau: symfony, php, framework, base de dades, apache.

Abstract

Develop a web application for the management of a beauty salon, using the PHP framework of Symfony.

Through the web application, it is intended to carry out all the control of the beauty salon, both financially and at the organizational and production level. You also want to have control of the stock of products, both those that are applied to customers and the products that are sold.

Beauty salon employees will be able to register new clients, give clients appointments, and have access to clients' service history.

The head of the beauty salon may also register new workers, have access to the services, suppliers, collection of services and products section and the statistics section.

Keywords: symfony, php, framework, database, apache.

Tabla de contenidos

1.- Introducción

A continuación, voy a explicar las motivaciones personales por las cuáles he elegido este proyecto.

1.1.- Motivación

Esta aplicación parte de la necesidad de facilitar la gestión del salón de belleza, ya que, hasta la fecha, todo el control se realiza de forma manual siendo muy tedioso y lento todo este proceso.

1.2.- Motivación personal

Las motivaciones que me llevan a realizar este proyecto son, que la aplicación se puede implantar en el salón de belleza, pudiendo ayudar a mi pareja y a su jefa a llevar un mejor control del salón, permitiendo automatizar y dinamizar la mayoría de las tareas que ahora se hacen a mano.

Personalmente me permite aprender y expandir mis conocimientos sobre el mundo de la programación web y sobre las bases de datos, muy demandado en estos tiempos ya que con este proyecto desarrollo mis conocimientos sobre PHP, HTML5, CSS, JS, MySQL, …

1.3.- Objetivos del proyecto

Los objetivos del proyecto son en primer lugar, aprender a desarrollar una aplicación web completa y usable en entornos de producción.

En segundo lugar, que la aplicación cumpla con los requisitos demandados por el cliente y que sea fácil de utilizar.

1.4.- Estructura de la Memoria

- En el apartado 2 voy a analizar el problema para ir desgranando los requisitos de la aplicación e ir explicando todo lo que se pretende hacer con la aplicación.
- En el apartado 3, una vez tenga claro de qué manera enfrentarme al problema, voy a realizar el diseño y la solución del problema.
- En el apartado 4 voy a realizar un tour por la aplicación explicando mediante capturas todos los apartados de la aplicación.
- En el apartado 5 voy a exponer las conclusiones y los trabajos futuros para así comentar las mejoras o ampliaciones que se pueden realizar en la aplicación.
- En el apartado 6 voy a hacer referencia a la bibliografía para exponer las fuentes de información que se han usado para la realización del proyecto.
- En el apartado 7 voy a explicar mediante un anexo las instalaciones y configuraciones previas para poder realizar el proyecto.

2.- Análisis del problema

El problema que se pretende resolver con esta aplicación es poder llevar a cabo la gestión del salón de belleza de forma integral, es decir, llevar el control financiero, organizativo y de producción.

Tras realizar varias reuniones con los dueños del salón de belleza, y después del proceso de análisis de los requisitos funcionales, estos son los requisitos funcionales que se han demandado desde el salón de belleza:

Los **requisitos funcionales** de la aplicación son los siguientes:

Clientes:

- Crear/leer/actualizar/borrar un cliente.
- Listar a todos los clientes.
- Buscar a un cliente por su nombre.

Trabajadores:

- Crear/leer/actualizar/borrar un trabajador.
- Listar a todos los trabajadores.
- Buscar a un trabajador por su nombre.

Productos:

- Crear/leer/actualizar/borrar un producto.
- Listar todos los productos.
- Buscar un producto por su nombre.
- Listar los productos con falta de stock**.**
- Listar los productos más demandados por un cliente.
- Listar los productos más demandados en un rango de fechas por los clientes.

Servicios:

- Crear/leer/actualizar/borrar un servicio.
- Listar todos los servicios.
- Buscar un servicio por su nombre.
- Listar los servicios más demandados por un cliente.
- Listar los servicios más demandados entre un rango de fechas por los clientes.
- Imprimir un listado con los servicios que se ofrecen con su precio actual.

Proveedores:

- Crear/leer/actualizar/borrar un proveedor.
- Listar todos los proveedores.
- Buscar a un proveedor por su nombre.
- Buscar a un proveedor por la marca a la que representa.

- Listar los productos comprados a un proveedor.
- Listar los productos comprados a todos los proveedores en un rango de fechas.
- Imprimir un listado con los proveedores junto a la marca y al teléfono de contacto.

Citas:

- Crear/leer/actualizar/borrar una cita de un cliente.
- Listar todas las citas de un día pudiendo seleccionar la fecha.
- Imprimir un listado de las citas que se tienen en la fecha actual.

Ficha técnica:

- Crear/leer/actualizar/borrar una entrada a la ficha técnica de un cliente.
- Listar la ficha técnica de un cliente buscando su nombre.

Cobro de Productos:

- Crear/leer/actualizar/borrar una entrada al cobro de productos a un cliente.
- Listar todos los cobros de productos.
- Listar a los morosos que no han pagado un producto.
- Listar la facturación diaria de los cobros de productos.
- Listar la facturación realizada en un rango de fechas de los cobros de productos.

Cobro de Servicios:

- Crear/leer/actualizar/borrar una entrada al cobro de servicios a un cliente.
- Listar todos los cobros de servicios.
- Listar a los morosos que no han pagado un servicio.
- Listar la facturación diaria de los cobros de servicios.
- Listar la facturación realizada en un rango de fechas de los cobros de servicios.

Gráficas:

- Mostrar mediante gráficos la evolución mes a mes de las ventas comparando el año actual con el año anterior de los productos y de los servicios.
- Listar los servicios más demandados por los clientes.
- Listar los productos más vendidos a los clientes.
- Imprimir una tabla con todas las ventas desglosado por meses tanto de productos como de servicios.

En cualquier aplicación que trabaje sobre una base de datos, es necesario que se puedan realizar las acciones básicas de persistencia de datos llamadas **CRUD** del acrónimo en inglés “Create, Read, Update and Delete”.

Para poder acceder a cualquier apartado de la aplicación, en primer lugar, el trabajador tiene que iniciar sesión con su nombre y su contraseña.

Según el tipo de trabajador, tendrá acceso a todo el panel o únicamente a unos pocos apartados para poder desarrollar su trabajo. Estos apartados se irán desarrollando a continuación.

2.1.- Clientes

Este apartado pretende aunar a todos los clientes del salón de belleza.

Todas estas acciones las puede realizar tanto un trabajador con el rol de usuario como un trabajador con el rol de administrador.

Crear nuevo cliente

Para la creación de un nuevo cliente, desde el panel principal de los clientes un empleado pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los datos del nuevo cliente, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de los clientes, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro del nuevo cliente contiene el nombre, el teléfono, la dirección, la población, el correo electrónico y la fecha de nacimiento, opcionalmente se pueden añadir unos comentarios. Automáticamente también se guarda el día en el que se ha registrado al nuevo cliente para futuras ofertas o descuentos.

Desde este panel también se puede **Volver a la lista.**

Leer cliente

Para ver los datos de un cliente, un empleado accede al menú de acciones en el panel principal de los clientes, al pinchar sobre la acción **Mostrar** aparece una nueva página con todos los datos del cliente de forma ordenada donde se pueden ver los datos de un cliente en concreto.

Desde este panel también se puede **Volver a la lista, Editar** el registro o **Borrar** el registro.

Actualizar cliente

Para poder modificar los datos de un cliente, un empleado accede al menú de acciones en el panel principal de los clientes, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del cliente de forma ordenada donde se pueden modificar los datos del cliente en concreto, una vez modificados al pinchar en el botón **Actualizar** se actualizan los datos de este cliente.

Se pueden modificar todos los datos del cliente menos la fecha en que se ha registrado.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar cliente

Para poder borrar un cliente, un empleado accede al menú de acciones en el panel principal de los clientes, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del cliente de forma ordenada donde se puede borrar el cliente pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar al cliente.

Listar clientes

En el panel principal de los clientes aparecerán listados todos los clientes registrados en la aplicación mostrando su nombre y apellidos, el teléfono, su dirección, su población, su email, su fecha de nacimiento, la fecha en que se registraron en la plataforma, un apartado de comentarios y las acciones de **Mostrar** y **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos clientes.

Si el listado de clientes fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de clientes accediendo a diferentes páginas creadas por este paginador.

El listado de los clientes aparecerá ordenado por fecha de creación.

Buscar clientes

En el panel principal de los clientes se habilitará un buscador para buscar a los clientes por su nombre, donde podremos buscar por su nombre o cualquiera de sus apellidos.

Al coincidir la búsqueda con el nombre de algún cliente, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada cliente de forma ordenada.

Si el listado de clientes buscados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de clientes buscados, accediendo a diferentes páginas creadas por este paginador.

2.2.- Trabajadores

El apartado de trabajadores pretende tener un listado de todos los trabajadores del salón de belleza.

Todas estas acciones únicamente las puede realizar un trabajador con el rol de administrador.

Crear nuevo trabajador

Para la creación de un nuevo trabajador, desde el panel principal de los trabajadores un empleado con privilegios de administrador pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los datos del nuevo trabajador, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de los trabajadores, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro del nuevo trabajador contiene el nombre, una contraseña, la selección del rol y opcionalmente se puede subir una imagen para identificar al trabajador. El rol implicará poder tener acceso o no a los diferentes paneles de la aplicación.

La contraseña de cada usuario tiene que guardarse encriptada en la base de datos, de forma que absolutamente nadie pueda tener acceso a la contraseña de forma no encriptada.

La imagen del trabajador se usará para identificar al trabajador registrado en la aplicación junto a su nombre.

Desde este panel también se puede **Volver a la lista.**

Leer trabajador

Para ver los datos de un trabajador, un empleado con rol de administrador accede al menú de acciones en el panel principal de los trabajadores, al pinchar sobre la acción **Mostrar** aparece una nueva página con el nombre del trabajador y si es administrador o no, en ningún caso se mostrará la contraseña de ningún empleado.

Desde este panel también se puede **Volver a la lista, Editar** el registro o **Borrar** el registro.

Actualizar trabajador

Para poder modificar los datos de un trabajador, un empleado con rol de administrador accede al menú de acciones en el panel principal de los trabajadores, al pinchar sobre la acción **Editar** aparece una nueva página con su nombre, un apartado para introducir una contraseña nueva, elegir el rol del trabajador y si se quiere subir una foto del trabajador, una vez modificados los datos, al pinchar en el botón **Actualizar** se actualizan los datos de este trabajador.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar trabajador

Para poder borrar un trabajador, un empleado con rol de administrador accede al menú de acciones en el panel principal de los trabajadores, al pinchar sobre la acción **Editar** aparece una nueva página con su nombre, un apartado para introducir una contraseña nueva, elegir el rol del trabajador y si se quiere subir una foto del trabajador, donde se puede borrar el

trabajador pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar al trabajador.

Listar trabajadores

En el panel principal de los trabajadores aparecerán listados todos los trabajadores registrados en la aplicación mostrando su nombre y apellidos, si tiene el rol de administrador o no y las acciones de **Mostrar** y **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos trabajadores.

Si el listado de trabajadores fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de clientes accediendo a diferentes páginas creadas por este paginador.

El listado de los trabajadores aparecerá ordenado por fecha de creación.

Buscar trabajadores

En el panel principal de los trabajadores se habilitará un buscador para buscar a los trabajadores por su nombre, donde podremos buscar por su nombre o cualquiera de sus apellidos.

Al coincidir la búsqueda con el nombre de algún trabajador, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada trabajador de forma ordenada.

Si el listado de trabajadores buscados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de trabajadores buscados, accediendo a diferentes páginas creadas por este paginador.

2.3.- Productos

En el apartado de productos tendremos todo lo referente a los productos que se venden en el salón de belleza.

Todas estas acciones únicamente las puede realizar un trabajador con el rol de administrador.

Crear nuevo producto

Para la creación de un nuevo producto, desde el panel principal de los productos un empleado con privilegios de administrador pincha en la acción **Nueva compra a proveedores**, se abre otra ventana con el formulario de registro para introducir los datos del nuevo producto, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de los productos,

si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro del nuevo producto contiene el nombre, el precio de venta al público, el precio de compra al proveedor, las unidades compradas al proveedor, la cantidad mínima de stock necesaria, la fecha de compra y el proveedor al que se le ha comprado el producto.

Automáticamente también se guarda el stock actual del producto, copiándose de las unidades compradas al proveedor, el stock del producto irá disminuyendo según se vayan realizando compras a este producto.

El stock mínimo se usará para mostrar unas alertas de falta de stock o en caso de no haber stock, no se permitirá comprar el producto.

Desde este panel también se puede **Volver a la lista**.

Leer producto

Para ver los datos de un producto, un empleado con rol de administrador accede al menú de acciones en el panel principal de los productos, al pinchar sobre la acción **Mostrar** aparece una nueva página con todos los datos de ese producto.

Desde este panel también se puede **Volver a la lista, Editar** el registro o **Borrar** el registro.

Actualizar producto

Para poder modificar los datos de un producto, un empleado con rol de administrador accede al menú de acciones en el panel principal de los productos, al pinchar sobre la acción **Editar** aparece una nueva página con los datos para ser modificados, una vez modificados los datos, al pinchar en el botón **Actualizar** se actualizan los datos de este producto.

En ningún caso se puede modificar el stock que se tiene actualmente del producto.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar producto

Para poder borrar un producto, un empleado con rol de administrador accede al menú de acciones en el panel principal de los productos, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del producto de forma ordenada donde se puede borrar el producto pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar al producto.

Listar productos

En el panel principal de los productos aparecerán listados todos los productos registrados en la aplicación mostrando el nombre, el precio de venta, el precio del proveedor,

las unidades compradas, el stock actual, el stock mínimo, la fecha de compra y las acciones de **Mostrar** y **Editar**.

Desde el panel principal también aparecerá la acción de **Nueva compra a Proveedores** para poder registrar nuevos productos.

Si el listado de productos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos accediendo a diferentes páginas creadas por este paginador.

El listado de productos aparecerá ordenado por fecha de compra a proveedores.

Buscar productos

En el panel principal de los productos se habilitará un buscador para buscar los productos por su nombre.

Al coincidir la búsqueda con el nombre de algún producto, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada producto de forma ordenada.

Si el listado de productos buscados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos buscados, accediendo a diferentes páginas creadas por este paginador.

Falta de stock de los productos

Según el tipo de producto, querremos tener una cantidad de stock u otro, para ello tenemos los campos stock actual y stock mínimo de cada producto.

Cuando se vaya vendiendo el producto, su stock irá descendiendo según su demanda, para saber cuándo tenemos poco stock y así poder pedir más al proveedor, tenemos que crear un apartado que denominaremos **Falta de stock**, en esta página aparecerán todos los productos cuyo stock actual sea igual o inferior al stock mínimo de cada producto. Sabiendo de esta forma que tenemos que pedir este producto al proveedor.

Desde esta página se puede volver a la página principal de productos mediante un enlace llamado **Volver al Índice**.

Productos demandados por un cliente

Puede resultar muy interesante tener un listado de todos los productos que ha comprado un **cliente** para hacerle ofertas o descuentos.

Para ello podemos crear un apartado llamado **Productos demandados por cliente** donde podemos crear un formulario desplegable con todos nuestros clientes, una vez seleccionado el cliente, nos aparecerán todos los productos comprados por un cliente con la cantidad comprada y la suma de precios, de esta forma podremos saber el volumen de compras de este cliente.

Si al buscar los productos comprados por un cliente, éste no ha comprado ningún producto, aparecerá un mensaje advirtiendo de que no se ha seleccionado ningún cliente o el cliente seleccionado no tiene registros.

Si el listado de productos comprados por un cliente fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de productos mediante un enlace llamado **Volver al Índice**.

Productos demandados entre dos fechas

También puede resultar muy interesante tener un listado con todos los productos que se han vendido en un rango de fechas, para saber todas las ventas que se han realizado durante un mes o año, o para saber si puede influir la venta de productos por un evento concreto y así reforzar el stock de los productos según las necesidades.

Para ello creamos una página llamada **Productos demandados entre fechas** donde podemos crear un formulario para seleccionar entre dos fechas.

Una vez seleccionadas las dos fechas, nos aparecerán todos los productos vendidos durante este rango de fechas con la cantidad vendida y la suma de precios, de esta forma podremos saber el volumen de ventas en el rango de fechas seleccionado.

Si el listado de productos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de productos mediante un enlace llamado **Volver al Índice**.

2.4.- Servicios

Mediante este apartado podemos ver todo lo referente a los servicios que ofrece el salón de belleza.

Todas estas acciones únicamente las puede realizar un trabajador con el rol de administrador.

Crear nuevo servicio

Para la creación de un nuevo servicio, desde el panel principal de los servicios un empleado con privilegios de administrador pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los datos del nuevo servicio, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de los productos, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro del nuevo servicio contiene el nombre del servicio, la descripción del servicio y el precio actual del servicio.

Desde este panel también se puede **Volver a la lista.**

Leer servicio

Para ver los datos de un servicio, un empleado con rol de administrador accede al menú de acciones en el panel principal de los servicios, al pinchar sobre la acción **Mostrar** aparece una nueva página con todos los datos de ese servicio.

Desde este panel también se puede **Volver a la lista, Editar** el registro o **Borrar** el registro.

Actualizar servicio

Para poder modificar los datos de un servicio, un empleado con rol de administrador accede al menú de acciones en el panel principal de los servicios, al pinchar sobre la acción **Editar** aparece una nueva página con los datos para ser modificados, una vez modificados los datos, al pinchar en el botón **Actualizar** se actualizan los datos de este servicio.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar servicio

Para poder borrar un servicio, un empleado con rol de administrador accede al menú de acciones en el panel principal de los servicios, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del servicio de forma ordenada donde se puede borrar el servicio pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar el servicio.

Listar servicios

En el panel principal de los servicios aparecerán listados todos los servicios registrados en la aplicación mostrando el nombre, la descripción, el precio y las acciones de **Mostrar** y **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos servicios.

Si el listado de servicios fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de servicios accediendo a diferentes páginas creadas por este paginador.

El listado de servicios aparecerá por orden de creación.

Buscar servicios

En el panel principal de los servicios se habilitará un buscador para buscar los servicios por su nombre.

Al coincidir la búsqueda con el nombre de algún servicio, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada servicio de forma ordenada.

Si el listado de servicios buscados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de servicios buscados, accediendo a diferentes páginas creadas por este paginador.

Servicios demandados por un cliente

Puede resultar muy interesante tener un listado de todos los servicios que ha demandado un **cliente** para hacerle ofertas o descuentos.

Para ello podemos crear un apartado llamado **Servicios demandados por cliente** donde podemos crear un formulario desplegable con todos nuestros clientes, una vez seleccionado el cliente, nos aparecerán todos los servicios demandados por un cliente con la cantidad de veces que se ha realizado un servicio y la suma del precio de cada servicio, de esta forma podremos saber el volumen de servicios demandados de este cliente.

Si al buscar los servicios demandados por un cliente, éste no ha demandado ningún servicio, aparecerá un mensaje advirtiendo de que no se ha seleccionado ningún cliente o el cliente seleccionado no tiene registros.

Si el listado de servicios demandados por un cliente fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de servicios, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de servicios mediante un enlace llamado **Volver al Índice**.

Servicios demandados entre dos fechas

También puede resultar muy interesante tener un listado con todos los servicios que se han demandado en un rango de fechas, para saber todos los servicios que se han realizado durante un mes, un año, o para saber si puede influir la demanda de servicios por un evento concreto.

Para ello creamos una página llamada **Servicios demandados entre fechas** donde podemos crear un formulario para seleccionar entre dos fechas.

Una vez seleccionadas las dos fechas, nos aparecerán todos los servicios demandados durante este rango de fechas con la cantidad de veces que se ha realizado este servicio y la suma de precios, de esta forma podremos saber el volumen de demanda en el rango de fechas seleccionado.

Si el listado de servicios fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de servicios, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de productos mediante un enlace llamado **Volver al Índice**.

Imprimir Servicios

Los clientes del salón de belleza tendrán a su disposición los servicios que se ofrecen junto a su precio colgados en el salón mediante un documento impreso en papel.

Para ello, en el panel principal de los servicios se habilitará un botón llamado **Imprimir Servicios** que al pulsarlo se descargará un documento en formato PDF para poder imprimir, donde aparecerá el nombre del servicio junto a su precio.

2.5.- Proveedores

Los proveedores son los que abastecen de productos al salón de belleza.

Todas estas acciones únicamente las puede realizar un trabajador con el rol de administrador.

Crear nuevo proveedor

Para la creación de un nuevo proveedor, desde el panel principal de los proveedores un empleado con privilegios de administrador pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los datos del nuevo proveedor, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de los proveedores, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro del nuevo proveedor contiene el nombre del proveedor, la marca que representa, un teléfono de contacto y un apartado para comentarios.

Desde este panel también se puede **Volver a la lista.**

Leer proveedor

Para ver los datos de un proveedor, un empleado con rol de administrador accede al menú de acciones en el panel principal de los proveedores, al pinchar sobre la acción **Mostrar** aparece una nueva página con todos los datos de ese proveedor.

Desde este panel también se puede **Volver a la lista, Editar** el registro o **Borrar** el registro.

Actualizar proveedor

Para poder modificar los datos de un proveedor, un empleado con rol de administrador accede al menú de acciones en el panel principal de los proveedores, al pinchar sobre la acción **Editar** aparece una nueva página con los datos para ser modificados, una vez modificados los datos, al pinchar en el botón **Actualizar** se actualizan los datos de este proveedor.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar proveedor

Para poder borrar un proveedor, un empleado con rol de administrador accede al menú de acciones en el panel principal de los proveedores, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del proveedor de forma ordenada donde se puede borrar el proveedor pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar al proveedor.

Listar proveedores

En el panel principal de los proveedores aparecerán listados todos los proveedores registrados en la aplicación mostrando el nombre, la marca que representan, un teléfono de contacto, un apartado de comentarios y las acciones de **Mostrar** y **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos proveedores.

Si el listado de proveedores fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de proveedores accediendo a diferentes páginas creadas por este paginador.

El listado de proveedores aparecerá por ordena de creación.

Buscar proveedores

En el panel principal de los proveedores se habilitarán dos buscadores, uno para buscar a los proveedores por su **nombre** y otro para buscar a los proveedores por la **marca** que representan.

Al coincidir la búsqueda con el **nombre** de algún proveedor, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada proveedor de forma ordenada.

De igual manera, al coincidir la búsqueda con la **marca** a la que representa algún proveedor, aparecerán todos sus datos, si no hubiese ninguna coincidencia de búsqueda, aparecerá un mensaje advirtiendo de que no se encontraron registros, si hubiese una coincidencia o varias, aparecerían los datos de cada proveedor de forma ordenada.

Si el listado de proveedores buscados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de proveedores buscados, accediendo a diferentes páginas creadas por este paginador.

Compra a proveedores

Puede resultar muy interesante tener un listado de todos los **productos** que se han comprado a un **proveedor** para saber, por ejemplo, si sale más rentable comprar a un proveedor u otro según los precios que nos hagan.

Para ello podemos crear un apartado llamado **Compra a proveedores** donde podemos crear un formulario desplegable con todos nuestros proveedores, una vez seleccionado el proveedor, nos aparecerán todos los productos comprados a este proveedor con la cantidad comprada, el montante de los productos comprados y la fecha de compra, de forma que podemos saber todas las compras que se le han realizado a este proveedor a lo largo del tiempo.

Si al buscar un proveedor, a este no se le ha comprado ningún producto, aparecerá un mensaje advirtiendo de que no se ha seleccionado ningún proveedor o este proveedor no tiene registros.

Si el listado de productos comprados a un proveedor fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de proveedores mediante un enlace llamado **Volver al Índice**.

Compra a proveedores entre dos fechas

También puede resultar muy interesante tener un listado con todos los productos que se han comprado a los diferentes proveedores en un rango de fechas, para saber todos los productos que se han comprado durante un mes, un año, o para saber si puede influir la compra de productos por un evento concreto.

Para ello creamos una página llamada **Compra a proveedores entre fechas** donde podemos crear un formulario para seleccionar entre dos fechas.

Una vez seleccionadas las dos fechas, nos aparecerán todos los productos comprados durante este rango de fechas a los proveedores con el nombre del producto, la cantidad de productos comprada, el montante del lote de productos, la fecha de compra y el nombre de proveedor.

Si el listado de productos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de productos, accediendo a diferentes páginas creadas por este paginador.

Desde esta página se puede volver a la página principal de productos mediante un enlace llamado **Volver al Índice**.

Imprimir Proveedores

Para agilizar la gestión del salón de belleza se creará un documento impreso en papel donde se podrá tener un listado de todos los proveedores con el nombre del proveedor, la marca que representa y un teléfono de contacto para llamarle o escribirle un mensaje y así poder realizar un pedido.

Para ello, en el panel principal de los proveedores se habilitará un botón llamado **Imprimir Proveedores** que al pulsarlo se descargará un documento en formato PDF para poder imprimir.

2.6.- Citas

Mediante esta herramienta se quiere organizar todas las citas que se van dando a los clientes para tener un control diario del trabajo que se realiza.

Todas estas acciones las puede realizar tanto un trabajador con el rol de usuario como un trabajador con el rol de administrador.

Crear nueva cita

Para la creación de una nueva cita, desde el panel principal de las citas un empleado pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los datos de la nueva cita, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de las citas, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan los datos.

El registro de la nueva cita contiene la fecha y la hora de la cita, el cliente que demanda la cita, el servicio que demanda y el empleado que tendrá asignado para realizarse el servicio.

Desde este panel también se puede **Volver a la lista.**

Actualizar cita

Para poder modificar los datos de una cita, un empleado selecciona el día de la cita y accede al menú de acciones en el panel principal de las citas, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos de la cita de forma ordenada donde se pueden modificar los datos de la cita en concreto, una vez modificados al pinchar en el botón **Actualizar** se actualizan los datos de esta cita.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar cita

Para poder borrar una cita, un empleado selecciona el día de la cita y accede al menú de acciones en el panel principal de las citas, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos de la cita de forma ordenada donde se puede borrar la cita pinchando

sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar la cita.

Listar citas

En el panel principal de las citas aparecerán listadas todas ellas registradas en la aplicación seleccionando la fecha que nos interese listar, mostrando la hora de la cita, el cliente que la demanda, el servicio, el empleado que tendrá asignado para realizar el servicio y la acción de **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevas citas.

Si el listado de citas fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de citas accediendo a diferentes páginas creadas por este paginador.

Las citas de cada día aparecerán ordenadas por la hora de forma ascendente.

Imprimir citas

Para agilizar la gestión del salón de belleza se creará un documento impreso en papel donde se podrá tener un listado de todas las citas de ese día mostrando la hora, el cliente que la demanda, el servicio y el empleado que tendrá asignado para realizar el servicio.

Para ello, en el panel principal se habilitará un botón llamado **Imprimir Citas** que al pulsarlo se descargará un documento en formato PDF para poder imprimir.

2.7.- Ficha Técnica

La ficha técnica pretende ser una herramienta para los trabajadores del salón de belleza donde poder consultar los trabajos anteriores que se le han realizado a un cliente para saber qué material se ha usado por si se tiene que modificar algún aspecto del proceso, como por ejemplo el tinte utilizado o cualquier otro producto con el fin de mejorar el servicio.

Todas estas acciones las puede realizar tanto un trabajador con el rol de usuario como un trabajador con el rol de administrador.

Crear nuevo registro en la ficha técnica del cliente

Para la creación de un nuevo registro en la ficha técnica de un cliente, desde el panel principal, un empleado pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los nuevos dato, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal de la ficha técnica, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan nuevos.

El nuevo registro de la ficha técnica contiene la fecha de realización del servicio, una descripción detallada del trabajo realizado, una foto del cliente, un desplegable para seleccionar al cliente y otro para seleccionar el servicio realizado.

Si el trabajador no ve necesario subir la foto o el cliente no está de acuerdo, no es necesario subirla, en su defecto aparecerá una foto con la silueta en negro de una persona.

Desde este panel también se puede **Volver a la lista.**

Actualizar registro en la ficha técnica del cliente

Para poder modificar el registro de la ficha técnica de un cliente, un empleado selecciona su nombre donde aparecen todos sus registros y accede al menú de acciones en el panel principal de la ficha técnica, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del registro de forma ordenada donde se pueden modificar los datos del registro en concreto, una vez modificados al pinchar en el botón **Actualizar** se refrescan los datos de este registro en la ficha técnica de este cliente.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar registros en la ficha técnica del cliente

Para poder borrar un registro de la ficha técnica, un empleado selecciona el nombre del cliente donde aparecen todos sus registros y accede al menú de acciones en el panel principal, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos de ese registro en concreto, de forma ordenada donde se puede eliminar de la ficha técnica del cliente pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar este registro de la ficha técnica del cliente.

Listar registros de la ficha técnica del cliente

Para poder listar todos los registros de la ficha técnica de un cliente, un empleado selecciona su nombre desplegando el formulario de todos ellos, cada registro contiene la fecha de realización del servicio, una descripción detallada del trabajo que se ha realizado, una foto del cliente, un desplegable para seleccionar su nombre y otro desplegable para seleccionar el servicio que se ha realizado.

Si el trabajador no ve necesario subir la foto o el cliente no está de acuerdo, no es necesario subirla, en su defecto aparecerá una imagen con la silueta en negro de una persona.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevas citas.

Si el listado de registros de la ficha técnica de un cliente fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de registros accediendo a diferentes páginas creadas por este paginador.

La ficha técnica de cada cliente aparece ordenada por fecha de forma ascendente.

2.8.- Cobro de Productos

El cobro de productos pretende ser una herramienta para los trabajadores del salón de belleza donde poder cobrar los productos que los clientes compran y así poder saber el volumen de ventas que se facturan en el salón.

Todas estas acciones las puede únicamente realizar un trabajador con el rol de administrador.

Crear nuevo cobro de productos

Para la creación de un nuevo cobro de productos, desde el panel principal, un empleado pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los nuevos datos, al pinchar sobre el botón **Registrar** si todos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal del cobro de productos, si se ha olvidado cumplimentar algún campo o no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan nuevos.

El nuevo cobro de productos contiene las unidades que se quiere comprar, el cliente que lo compra, el nombre, la fecha y hora de compra, la fecha de pago, si se ha cobrado, como se va a realizar el pago y un cuadro de texto para introducir un comentario si fuese necesario.

Se recoge el tipo de pago si es en efectivo o en tarjeta para así saber el dinero que se tiene en caja.

Si un producto se va a comprar y tiene su stock actual suficiente, el cobro de productos se realiza de forma correcta y se descuenta la cantidad vendida del stock de ese producto.

Si el stock actual es menor o igual al mínimo, aparece un mensaje de advertencia comunicando que se tiene que revisar el stock de ese producto, se realiza la compra y se descuenta la cantidad vendida.

Si no hay stock de un producto, aparece un mensaje de advertencia comunicando que no hay unidades y no se puede realizar la compra.

Se distingue la fecha de compra del producto de la fecha de pago porque puede que un cliente en ese momento no pague y lo haga en otro momento, de esta forma se puede saber cuándo ha comprado el producto y cuando ha pagado.

Por otro lado, también se quiere saber si se ha pagado el producto o no, esto se hace marcándolo en una casilla. Si no se ha pagado, este registro aparecerá en el listado de morosos. El apartado de comentarios puede servir para tener constancia de porqué no se ha pagado.

Cuando se registra el nuevo cobro, el precio de ese producto se coge del precio actual, de esta forma, aunque el precio del producto varíe en el tiempo, siempre se tendrá el valor al que se vendió el producto en ese momento. A la hora de sacar la facturación, el volumen total nos arrojará un valor correcto, si por el contario no guardásemos el precio actual de ese producto, no tendríamos la facturación real.

Para calcular el precio total de la venta si se han comprado varias unidades del mismo, se suma el precio actual por la cantidad comprada, este registro será esencial para realizar las sumas totales de ventas de los productos.

Desde este panel también se puede **Volver a la lista.**

Actualizar cobro de productos

Para poder modificar el registro del cobro de un producto, un empleado accede al menú de acciones en el panel principal, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del registro de forma ordenada donde se pueden modificar, una vez modificados al pinchar en el botón **Actualizar** se refrescan.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar cobro de productos

Para poder borrar un registro del cobro de productos, se accede al menú de acciones en el panel principal, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del registro de forma ordenada donde se puede eliminar pinchando sobre el botón **Borrar**, al pinchar el botón aparece un cuadro de diálogo preguntando si estamos seguros de querer borrarlo.

Listar cobro de productos

En el panel principal del cobro de productos aparecerán listados todos los cobros registrados en la aplicación, mostrando la fecha y la hora de compra, el precio por unidad, la cantidad comprada, el precio total, el cliente que lo ha comprado, el trabajador que ha vendido el producto, el nombre, si se ha pagado o no, la fecha de pago y la acción de **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos cobros de productos.

Si el listado de productos vendidos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de citas accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de productos aparecerán ordenadas por fecha y hora de forma ascendente.

Morosos

Para el control de los pagos de un producto, resulta muy útil saber los clientes que no han pagado un producto para no fiarles en más ocasiones, para ello se puede crear una página llamada **Morosos** donde aparecerá un listado de los productos que no se han pagado, para así poder reclamar el pago a un cliente, en este listado aparecerá la fecha y la hora de compra, el precio por unidad, la cantidad comprada, el precio total, el cliente que lo ha comprado, el trabajador que ha vendido el producto, el nombre, si se ha pagado o no, la fecha de pago, un apartado de comentarios y las acciones de **Mostrar** y **Editar.**

Si un cliente paga la factura que tiene pendiente, se puede acceder a este listado y en el menú de acciones se puede **Editar** este pago marcando que se ha sbonado el producto y modificando la fecha de pago, de esta forma el cliente desaparece de la lista de morosos.

Desde este panel también se puede **Volver al índice.**

Facturación diaria

Para saber la facturación que se ha realizado durante un día, se puede crear una página llamada **Facturación diaria**, mediante esta página se pretende saber cuál ha sido la facturación durante un día para así llevar a cabo el control de los beneficios de la venta de productos.

También resulta interesante poder ver la facturación que hay ese día en la caja para así saber si se puede hacer frente al pago de la factura a un proveedor que haya podido venir a traernos un pedido, si tenemos suficiente dinero en metálico en ese momento.

Automáticamente aparece la facturación del día actual, con el montante que se ha recaudado y también un listado de todos los registros que se han realizado ese día.

Si se quiere saber la facturación que se ha realizado un día concreto, únicamente tenemos que modificar la fecha del formulario a cualquier otro día, de esta forma aparecerán los registros del día que se ha seleccionado.

Si el listado de productos vendidos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de registros accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de productos aparecerán ordenadas por fecha y hora de forma ascendente.

Desde este panel también se puede **Volver al índice.**

Facturación entre dos fechas

Para saber la facturación que se ha realizado entre dos fechas, se puede crear una página llamada **Facturación entre fechas**, mediante esta página se pretende saber cuál ha sido la facturación durante un periodo de tiempo para así llevar a cabo el control de los beneficios de la venta de productos.

Esta página resulta muy interesante para saber la facturación de la venta de productos en un periodo de tiempo, por ejemplo, en navidad, durante las fiestas locales, en verano, …

Para poder ver el montante que se ha recaudado y el listado de registros, es necesario seleccionar la fecha de inicio y la fecha de fin.

Una vez seleccionadas las fechas, aparecerá la suma total recaudada y un listado con todos los registros.

Si el listado de productos vendidos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de registros accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de productos aparecerán ordenadas por fecha y hora de forma ascendente.

Desde este panel también se puede **Volver al índice.**

2.9.- Cobro de Servicios

El cobro de servicios pretende ser una herramienta para los trabajadores del salón de belleza donde poder cobrar los servicios que se realizan a los clientes y así poder saber el volumen de servicios que se facturan en el salón de belleza.

Todas estas acciones las puede únicamente realizar un trabajador con el rol de administrador.

Crear nuevo cobro de servicios

Para la creación de un nuevo cobro de servicios, desde el panel principal, un empleado pincha en la acción **Crear nuevo**, se abre otra ventana con el formulario de registro para introducir los nuevos datos, al pinchar sobre el botón **Registrar** si todos los datos están debidamente cumplimentados se guarda el registro en la BBDD y se vuelve automáticamente al panel principal, si se ha olvidado cumplimentar algún campo o los datos no son correctos, se arroja un mensaje de advertencia para que se corrijan o se añadan nuevos.

El nuevo cobro de servicios contiene el cliente al que se le ha realizado el servicio, el nombre, la fecha y hora de realización del servicio, la fecha de pago, si se ha pagado o no, un desplegable para elegir el tipo de pago y un cuadro de texto para introducir un comentario si fuese necesario.

Se distingue la fecha de realización del servicio de la fecha de pago porque puede que un cliente en ese momento no abone el servicio y venga en otro momento, de esta forma se puede saber cuándo se le ha realizado el servicio y cuando lo ha pagado.

Se recoge el tipo de pago si es en efectivo o en tarjeta para así saber el dinero que se tiene en caja.

Por otro lado, también se quiere saber si se ha pagado el servicio o no, esto se hace marcando una casilla. Si no se ha pagado el servicio, este registro aparecerá en el listado de morosos. El apartado de comentarios puede servir para tener constancia de porqué no se ha pagado.

Cuando se registra el nuevo cobro del servicio, ese precio se coge del que tiene ese servicio en ese momento, de esta forma, aunque el precio del servicio varíe en el tiempo, siempre se tendrá el valor al que se cobró ese servicio en ese momento. A la hora de sacar la facturación, el volumen total nos arrojará un valor correcto, si por el contario no guardásemos el precio actual de ese servicio, no tendríamos la facturación real.

Desde este panel también se puede **Volver a la lista.**

Actualizar cobro de servicios

Para poder modificar el registro del cobro de un servicio, un empleado accede al menú de acciones en el panel principal, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del registro de forma ordenada donde se pueden modificar, una vez cambiados al pinchar en el botón **Actualizar** se refrescan los datos de este registro del cobro de un servicio.

Desde este panel también se puede **Volver a la lista** o **Borrar** el registro.

Borrar cobro de servicios

Para poder borrar un registro del cobro de servicios, accede al menú de acciones en el panel principal, al pinchar sobre la acción **Editar** aparece una nueva página con todos los datos del registro de forma ordenada donde se puede eliminar el registro del cobro de un servicio pinchando sobre el botón **Borrar**, al pinchar aparece un cuadro de diálogo preguntando si estamos seguros de querer borrar el pago.

Listar cobros de servicios

En el panel principal del cobro de servicios aparecerán listados todos los cobros registrados en la aplicación, mostrando la fecha y la hora en que se ha cobrado, el precio, el cliente, el trabajador que lo ha realizado, el nombre del servicio, si se ha pagado o no, la fecha de cobro y la acción de **Editar**.

Desde el panel principal también aparecerá la acción de **Crear nuevo** para poder registrar nuevos.

Si el listado de productos vendidos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de citas accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de productos aparecerán ordenadas por fecha y hora de forma ascendente.

Morosos

Para el control de los pagos de un servicio, resulta muy útil saber los clientes que no han pagado para no fiarles en más ocasiones, para ello se puede crear una página llamada **Morosos** donde aparecerá un listado de los servicios que no se han abonado, para así poder reclamar el pago a un cliente, en este listado aparecerá la fecha y la hora de realización del servicio, el precio, el cliente, el trabajador que ha realizado el servicio, si se ha pagado o no, la fecha, un apartado de comentarios y las acciones de **Mostrar** y **Editar.**

Si un cliente paga la factura que tiene pendiente, se puede acceder a este listado y en el menú de acciones se puede **Editar** marcando que se ha abonado el servicio, modificando la fecha y el tipo de pago realizado, de esta forma el cliente desaparece de la lista de morosos.

Desde este panel también se puede **Volver al índice.**

Facturación diaria

Para saber la facturación que se ha realizado durante un día, se puede crear una página llamada **Facturación diaria**, mediante esta página se pretende saber cuál ha sido la facturación durante un día para así llevar a cabo el control de los beneficios de la realización de los diferentes servicios.

También resulta interesante poder ver la facturación que hay ese día en la caja para así saber si se puede hacer frente al pago de la factura a un proveedor que haya podido venir a traernos un pedido, si tenemos suficiente dinero en metálico en ese momento.

Automáticamente aparece la facturación del día actual, con el montante que se ha recaudado y también un listado de todos los registros que se han realizado ese día.

Si se quiere saber la facturación que se ha realizado un día concreto, únicamente tenemos que modificar la fecha del formulario a cualquier otro día, de esta forma aparecerán los registros del día que se ha seleccionado.

Si el listado de servicios realizados fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de registros accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de servicios aparecerán ordenadas por fecha y hora de forma ascendente.

Desde este panel también se puede **Volver al índice.**

Facturación entre dos fechas

Para saber la facturación que se ha realizado entre dos fechas, se puede crear una página llamada **Facturación entre fechas**, mediante esta página se pretende saber cuál ha sido la facturación durante un periodo de tiempo para así llevar a cabo el control de los beneficios de la realización de servicios.

Esta página resulta muy interesante para saber la facturación que se ha realizado en un periodo de tiempo, por ejemplo, en navidad, durante las fiestas locales, en verano, …

Para poder ver el montante que se ha recaudado y el listado de registros, es necesario seleccionar la fecha de inicio y la fecha de fin.

Una vez seleccionadas las fechas, aparecerá la suma total recaudada y un listado con todos los registros.

Si el listado de productos vendidos fuese superior al tamaño de la página, se habilitará un paginador para poder acceder a todo el listado de registros accediendo a diferentes páginas creadas por este paginador.

Los registros del cobro de servicios aparecerán ordenadas por fecha y hora de forma ascendente.

Desde este panel también se puede **Volver al índice.**

2.10.- Gráficas

Esta página pretende dar una visión global mediante gráficas lineales y de barras de cómo está funcionando el negocio, pudiendo ver las ganancias de forma desglosada con una comparativa mes a mes entre el año anterior y el año actual.

Ganancias por Servicios

Resulta muy interesante poder saber cómo está funcionando el negocio con respecto del año anterior para así prever como va a ir evolucionando. Para ello se creará una gráfica de líneas donde se mostrarán mes a mes las ganancias de los servicios que se vayan realizando.

También aparecerán las ganancias totales del año anterior y del actual, junto al porcentaje de beneficio del año actual respecto del anterior para así saber si el año actual está funcionando mejor que el anterior.

Ganancias por Productos

Igual que para la ganancia de servicios, para la ganancia de productos se va a realizar una gráfica de barras para saber cómo está funcionando mes a mes la venta de productos este año respecto del anterior en cuanto a la venta de productos.

También aparecerán las ganancias totales del año anterior y de este, junto al porcentaje de beneficio del año actual respecto del anterior para así saber si el año actual está funcionando mejor que el anterior en cuanto a la venta de productos.

Servicios más demandados

Puede resultar también interesante saber cuáles son los servicios más demandados durante el año actual, para así poder ajustar el precio del servicio.

Productos más vendidos

De igual modo con los productos, saber los que han sido más comprados por los clientes para así poder pedir un mejor precio al proveedor y tener mayor margen.

Informe de Servicios

Puede resultar muy interesante poder tener un documento para poder imprimir donde aparezcan reflejadas las ganancias de los servicios mes a mes del año actual y del anterior, el porcentaje de beneficios del año actual respecto del anterior y las ganancias totales del actual y del anterior.

Para ello, se puede crear una tabla donde aparezcan desglosados mes a mes estos beneficios y así tener una comparativa de las ganancias que se van teniendo cada mes pudiendo compararlos con el mes del año anterior.

Este informe está actualizado en la fecha en la que se genera, pudiendo crearlo anualmente para así tener un documento donde poder tener reflejadas las ganancias de cada año.

Informe de Productos

De igual forma, también se puede generar un documento para poder imprimir donde aparezcan reflejadas las ganancias de la venta de productos mes a mes del año actual y del anterior, el porcentaje de beneficios y ganancias totales del año actual respecto del anterior.

Para ello, se puede crear una tabla donde aparezcan desglosados mes a mes estos beneficios y así tener una comparativa de las ganancias que se van teniendo cada mes pudiendo compararlos con el mes del año anterior.

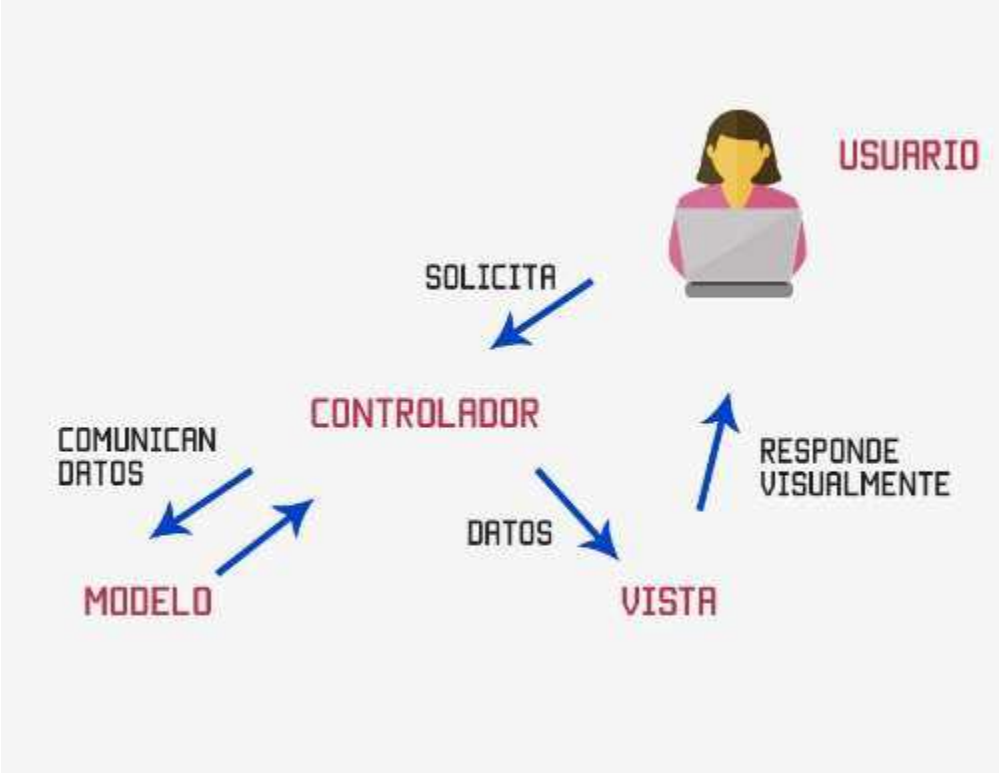
Este informe está actualizado en la fecha en la que se genera, pudiendo crearlo anualmente para así tener un documento donde poder tener reflejadas las ganancias de cada año.

3.- Diseño o solución del problema

En este apartado se va a describir cómo se ha abordado el diseño de la aplicación.

En primer lugar, se describe la arquitectura general del sistema y en los subapartados se detallan cada uno de los componentes.

La aplicación está diseñada siguiendo el estilo de arquitectura **Modelo Vista Controlador** que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.



3.1.-Arquitectura de la aplicación

Se va a trabajar con una arquitectura **cliente-servidor**, el cliente es el navegador web y el servidor es un servidor web. La idea es desarrollar una aplicación web donde el contenido se genere de forma dinámica desde el servidor.

Para la realización de este proyecto he decidido usar el framework **PHP** **Symfony** porque su curva de aprendizaje es muy rápida, además para resolver algunos problemas comunes como puede ser el login del usuario, la seguridad de credenciales si tenemos diferentes tipos de roles, formularios, persistencia de datos, internacionalización etc.

Symfony resuelve estos repetitivos temas de una forma fácil evitándonos tiempo en el desarrollo. Nos ofrece una serie de herramientas que nos permiten, centrarnos en la lógica de nuestra aplicación y no en estos asuntos que ya están solucionados. No hay que reinventar nada que ya esté inventado.

3.1.1.- Modelo

Esta capa es la encargada de proporcionar la persistencia de la información la arquitectura MVC. El framework Symfony nos ofrece la posibilidad de crear las **Entidades** de forma asistida por el framework o de forma totalmente manual.

Estas entidades se traducen en las tablas de la base de datos, donde se puede crear cada tipo de dato y posteriormente se pueden generar las diferentes dependencias de las tablas, claves primarias, claves ajenas, …

Para realizar todo el diseño de la aplicación se ha creado una base de datos en lenguaje **MySQL** para persistir todos los datos de la aplicación.

En nuestra aplicación las clases contenidas en esta capa son:

- **Client**: En esta clase guardamos los datos de los clientes pertenecientes a salón de belleza.
- **User**: En esta clase guardamos los datos de los trabajadores pertenecientes al salón de belleza.
- **Product**: En esta clase guardamos los datos de los productos que se venden en el salón de belleza.
- **Service**: En esta clase guardamos los datos de los servicios que se realizan en el salón de belleza.
- **Provider**: En esta clase guardamos los datos de los proveedores que nos abastecen en el salón de belleza.
- **Schedule**: En esta clase guardamos los datos de las citas que se dan a los clientes del salón de belleza.
- **Technical_sheet:** En esta clase guardamos los datos de la ficha técnica de cada cliente del salón de belleza.
- **Payment_product:** En esta clase guardamos los datos de los cobros que se realizan por la venta de productos en el salón de belleza.
- **Payment_service:** En esta clase guardamos los datos de los cobros que se realizan por los servicios que se realizan en el salón de belleza.

Para la creación del modelo se han usado las siguientes tecnologías:

Doctrine es un mapeador objeto-relacional (ORM) escrito en PHP que proporciona una capa de persistencia para objetos PHP. Es una capa de abstracción que se sitúa justo encima de un sistema de gestión de bases de datos (SGBD).

MySQL es un sistema de gestión de bases de datos relacional desarrollado bajo licencia dual: Licencia pública general/Licencia comercial por Oracle Corporation y está considerada como la base de datos de código abierto más popular del mundo, y una de las más populares en general junto a Oracle y Microsoft SQL Server, sobre todo para entornos de desarrollo web.

3.1.2.- Vista

La vista, también conocida como presentación, es la interfaz que utiliza el usuario para interactuar con el sistema. Es decir, es la parte visual de la aplicación donde se muestra los menús y las diferentes opciones para que el usuario pueda utilizar la aplicación.

El motor de plantillas TWIG forma parte del framework Symfony por defecto, con ella se define el aspecto que queremos presentar. Además, es conveniente que la aplicación sea responsive, para que se adapte a todo tipo de pantallas, para permitir la correcta visualización en terminales móviles, tabletas …

Para las diferentes vistas de la aplicación se han usado estas tecnologías:

Twig es un motor de plantilla para el lenguaje de programación PHP. Su sintaxis origina de Jinja y las plantillas Django. Es un producto de código abierto autorizado bajo Licencia BSD y mantenido por Fabien Potencier. La versión inicial estuvo creada por Armin Ronacher.

HTML5 (*HyperText Markup Language*, versión 5) es la quinta revisión importante del lenguaje básico de la World Wide Web, HTML. HTML5 especifica dos variantes de sintaxis para HTML: una «clásica», HTML (`text/html`), conocida como *HTML5*, y una variante XHTML conocida como sintaxis *XHTML 5* que deberá servirse con sintaxis XML (`application/xhtml+xml`).

CSS (siglas en inglés de **C**ascading **S**tyle **S**heets), en español «Hojas de estilo en cascada», es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Es muy usado para establecer el diseño visual de los documentos web, e interfaces de usuario escritas en HTML o XHTML. Te puede ayudar a crear tu propio sitio web. Junto con HTML y JavaScript, CSS es una tecnología usada por muchos sitios web para crear páginas visualmente atractivas, interfaces de usuario para aplicaciones web y GUIs para muchas aplicaciones móviles.

Bootstrap es una biblioteca multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como extensiones de JavaScript adicionales. A diferencia de muchos frameworks web, solo se ocupa del desarrollo front-end.

JavaScript (abreviado comúnmente **JS**) es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en su forma del lado del cliente (*client-side*), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas y JavaScript del lado del servidor (*Server-side JavaScript* o *SSJS*).

3.1.3.- Controlador

El controlador es el encargado de atender los eventos producidos en el sistema, realiza peticiones a la capa modelo para consultar datos y pasando la información necesaria para que la capa de presentación pueda presentarlos. En Symfony los controladores contienen el código PHP necesario para realizar las tareas necesarias de esta capa. Por otro lado, Symfony funciona con routing, donde definimos las direcciones URL que atenderá nuestra aplicación.

Para realizar todo el control de la aplicación se ha usado esta tecnología:

PHP, acrónimo recursivo en inglés de **PHP: Hypertext Preprocessor** (preprocesador de hipertexto), es un lenguaje de programación de propósito general de código del lado del servidor originalmente diseñado para el preprocesado de texto plano en UTF-8. Posteriormente se aplicó al desarrollo web de contenido dinámico, dando un paso evolutivo en el concepto de aplicación en línea, por su carácter de servicio.

3.2.- Creación del Proyecto

Me voy a centrar directamente en explicar la creación del modelo, obviando la creación del proyecto y la puesta a punto del entorno de desarrollo (explicado en el anexo).

3.2.1.- Capa Modelo

Para la creación de cada una de nuestras entidades se puede realizar de forma totalmente manual o de forma asistida.

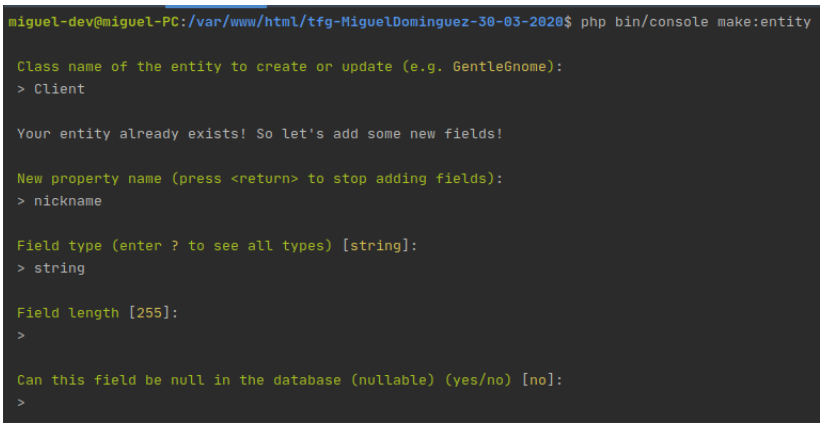
Mediante el asistente

```
php bin/console make:entity
```

Este asistente en primer lugar nos pregunta por el nombre de la entidad y posteriormente por sus campos y sus tipos. Utilizando este método, tenemos la ventaja de no tener que preocuparnos por los getter y setter, dado que el propio gestor las crea y añade.

Para ver los diferentes tipos de campos que acepta **Doctrine**, podemos acceder a esta web:

https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/basic-mapping.html



Creación manual de las entidades

La otra opción que tenemos para la generación de las diferentes entidades pasa por crearlas directamente. Es una opción más "peligrosa" ya que debemos recordar implementar todos los métodos para acceder o modificar los atributos de la clase.

Una vez creadas todas las entidades con sus respectivos campos, tenemos que crear las diferentes relaciones o asociaciones de cada entidad.

La relación más típica entre cada entidad es la relación conocida como **Muchos a uno/uno a muchos**. De igual forma que con la creación de las entidades, Symfony posee un asistente para la creación de dichas relaciones, pero me parece más sencillo en este caso crear las relaciones de forma manual mediante código.

Por ejemplo, tenemos la entidad **Client** y la entidad **TechnicalSheet** que hacen referencia al cliente y a su ficha técnica por tanto la relación entre las dos entidades sería que un cliente tiene o posee muchos registros en su ficha técnica y el registro en la ficha técnica pertenece a un cliente.

Esto expresado en código PHP en la parte de la entidad **Client**:

```
/**
*@ORM\OneToMany(targetEntity="App\Entity\TechnicalSheet", mappedBy="client")
*/
private $technicalSheet;
```

Esto expresado en código PHP en la parte de la entidad **TechnicalSheet**:

```
/**
*@ORM\ManyToOne(targetEntity="App\Entity\Client", inversedBy="technicalSheet")
*/
private $client;
```

La otra relación posible es **Muchos a muchos**, esta relación permite la asociación de muchos objetos X con muchos objetos Y. En nuestro caso, no tenemos dicha relación.

Una vez tengamos creadas todas las relaciones, tenemos que actualizar la base de datos para poder ver todos los cambios que hemos realizado mediante el comando:

```
php bin/console doctrine:schema:update --force
```

La única entidad especial es la entidad **user**, con ella vamos a realizar el acceso a la plataforma de los diferentes trabajadores.

Para poder tener seguridad en nuestra aplicación, Symfony posee un bundle llamado **security**-**bundle**, con este bundle podemos manejar toda la seguridad de la aplicación pudiendo crear el acceso registrado mediante usuario y contraseña, el login y el logout entre otros. Para ello la entidad **user** hará referencia a los trabajadores de la aplicación, usándolos para el acceso

a la aplicación, como curiosidad el bundle permite crear diferentes roles en función del acceso que queramos dar a estos trabajadores, teniendo la posibilidad de otorgar a los trabajadores el **ROL_USER** o el **ROL_ADMIN** entre otros**,** con ello podemos restringir el acceso a la plataforma según el tipo de usuario que intente acceder.

Para ver toda la documentación referente a este bundle podemos acceder a su web:

https://symfony.com/doc/current/security.html

Las diferentes entidades creadas están alojadas en src/Entity y son:

Client

Entidad **Client**, esta entidad hace referencia a los clientes del salón de belleza, en ella se guardan los datos de los clientes, por ejemplo, su nombre, teléfono de contacto, localización, …

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string", length=255)
 */
private $clientName;

/**
 * @ORM\Column(type="integer", nullable=true)
 */
private $phone;

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $street;

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $location;
```

```php
/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $mail;

/**
 * @ORM\Column(type="date", nullable=true)
 */
private $dob;

/**
 * @ORM\Column(type="date", nullable=true)
 */
private $antiquity;

/**
 * @ORM\Column(type="string", length=1000, nullable=true)
 */
private $comments;
```

La entidad **Client** está relacionada con las entidades **Schedule**, **TechnicalSheet**, **PaymentService** y **PaymentProduct**.

```php
/**
 * @ORM\OneToMany(targetEntity="App\Entity\Schedule", mappedBy="client")
 */
private $schedule;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\TechnicalSheet", mappedBy="client")
 */
private $technicalSheet;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentService", mappedBy="client")
 */
private $paymentService;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentProduct", mappedBy="client")
 */
private $paymentProduct;
```

La relación entre **Client** y **Schedule** se hace para saber qué cliente ha pedido cita en el salón de belleza.

La relación entre **Client** y **TechnicalSheet** se hace para poder registrar los servicios que se le han realizado a un cliente en su ficha técnica para así poder tener un historial de servicios realizados.

La relación entre **Client** y **PaymentProduct** se hace para saber qué cliente ha comprado un determinado producto.

La relación entre **Client** y **PaymentService** se hace para saber qué cliente ha demandado un determinado servicio.

User

Entidad **User**, esta entidad es especial ya que mediante ella se realiza el control de acceso a la aplicación y hace referencia a los trabajadores del salón de belleza, en ella se guardan los datos de los trabajadores, por ejemplo, el nombre y la contraseña de acceso debidamente encriptada, el rol de acceso a la aplicación, si es o no administrador y una foto.

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string", length=180, unique=true)
 */
private $username;

/**
 * @ORM\Column(type="json")
 */
private $roles = [];
```

```php
/**
 * @var string The hashed password
 * @ORM\Column(type="string")
 */
private $password;

/**
 * @ORM\Column(type="boolean", nullable=true)
 */
private $isAdmin;

/**
 * @ORM\Column(type="string", nullable=true)
 */
private $photo;
```

La entidad **User** está relacionada con las entidades **Schedule**, **TechnicalSheet**, **PaymentService** y **PaymentProduct.**

```php
/**
 * @ORM\OneToMany(targetEntity="App\Entity\TechnicalSheet", mappedBy="user")
 */
private $technicalSheet;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentService", mappedBy="user")
 */
private $paymentService;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentProduct", mappedBy="user")
 */
private $paymentProduct;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\Schedule", mappedBy="user")
 */
private $schedule;
```

La relación entre **User** y **TechnicalSheet** se hace para saber qué trabajador ha realizado el servicio y así poder anotarlo en la ficha técnica para tener un historial de servicios realizados.

La relación entre **User** y **PaymentService** se hace para saber qué trabajador ha cobrado el servicio que ha pagado un cliente.

La relación entre **User** y **PaymentProduct** se hace para saber qué trabajador ha cobrado los productos que se han vendido a un cliente.

La relación entre **User** y **Schedule** se hace para saber qué trabajador se va a encargar de realizar el servicio al cliente.

Service

Entidad **Service**, esta entidad hace referencia a los servicios que se ofrecen en el salón de belleza, en ella se guardan los datos de los servicios, por ejemplo, su nombre, su descripción y el precio actual.

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string", length=255)
 */
private $serviceName;

/**
 * @ORM\Column(type="string", length=1000, nullable=true)
 */
private $description;

/**
 * @ORM\Column(type="decimal", precision=5, scale=2)
 */
private $price;
```

La entidad **Service** está relacionada con las entidades **Schedule**, **TechnicalSheet** y **PaymentService.**

```php
/**
 * @ORM\OneToMany(targetEntity="App\Entity\Schedule", mappedBy="service")
 */
private $schedule;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\TechnicalSheet", mappedBy="service")
 */
private $technicalSheet;

/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentService", mappedBy="service")
 */
private $paymentService;
```

La relación entre **Service** y **Schedule** se hace para saber qué servicio ha pedido el cliente a la hora de pedir cita en el salón de belleza.

La relación entre **Service** y **TechnicalSheet** se hace para saber qué servicio se le ha realizado al cliente y así poder anotarlo en la ficha técnica para tener un historial de servicios realizados.

La relación entre **Service** y **PaymentService** se hace para saber qué servicio se le ha cobrado a un cliente a la hora de pagar.

Provider

Entidad **Provider**, esta entidad hace referencia a los proveedores que suministran los productos al salón de belleza, en ella se guardan los datos de los proveedores, por ejemplo, su nombre, su teléfono de contacto y la marca que representan.

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string", length=255)
 */
private $providerName;

/**
 * @ORM\Column(type="string", length=255)
 */
private $brand;

/**
 * @ORM\Column(type="integer")
 */
private $phone;

/**
 * @ORM\Column(type="string", length=1000, nullable=true)
 */
private $comments;
```

La entidad **Provider** está relacionada con la entidad **Product.**

```php
/**
 * @ORM\OneToMany(targetEntity="App\Entity\Product", mappedBy="provider")
 */
private $product;
```

La relación entre **Provider** y **Product** se hace para saber qué proveedor nos ha vendido un determinado producto.

Product

Entidad **Product**, esta entidad hace referencia a los productos que se venden en el salón de belleza, en ella se guardan los datos de los productos, por ejemplo, su nombre, su precio de venta, el precio del proveedor, el stock actual, la cantidad comprada al proveedor, la fecha de compra al proveedor, …

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string", length=255)
 */
private $productName;

/**
 * @ORM\Column(type="decimal", precision=5, scale=2)
 */
private $salePrice;

/**
 * @ORM\Column(type="decimal", precision=5, scale=2)
 */
private $purchasePrice;
```

```php
/**
 * @ORM\Column(type="decimal", precision=10, scale=2)
 */
private $totalPurchase;
/**
 * @ORM\Column(type="integer")
 */
private $amount;

/**
 * @ORM\Column(type="integer")
 */
private $currentAmount;

/**
 * @ORM\Column(type="integer")
 */
private $minAmount;

/**
 * @ORM\Column(type="date")
 */
private $datePurchase;
```

Para trabajar con precios es necesario usar decimales, en el caso del precio de venta o de compra, los campos son capaces de guardar un precio por valor de hasta 999,99€, he contemplado que no va a haber un precio mayor del fijado, en cambio en el campo de compra total he contemplado que el precio total de una compra pueda ascender hasta a 99.999.999,99.

Este campo está un poco sobredimensionado, pero he preferido hacerlo así para no tener errores.

La entidad **Product** está relacionada con las entidades **PaymentProduct y Provider.**

```php
/**
 * @ORM\OneToMany(targetEntity="App\Entity\PaymentProduct", mappedBy="product")
 */
private $paymentProduct;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Provider", inversedBy="product")
 */
private $provider;
```

La relación entre **Product** y **PaymentProduct** se hace para saber qué productos se han vendido en el salón de belleza.

La relación entre **Provider** y **Product** se hace para saber qué proveedor nos ha vendido un determinado producto.

Schedule

Entidad **Schedule**, esta entidad hace referencia al horario o a las citas que tiene un cliente en el salón de belleza, en ella se guardan los datos de una cita, por ejemplo, el cliente, el servicio, el día y la hora de la cita y el trabajador que va a atender al cliente.

```
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="time")
     */
    private $hour;

    /**
     * @ORM\Column(type="date")
     */
    private $date;
```

La entidad **Schedule** está relacionada con las entidades **Client**, **Service** y **User.**

```
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Client", inversedBy="schedule")
     */
    private $client;

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Service", inversedBy="schedule")
     */
    private $service;

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="schedule")
     */
    private $user;
```

La relación entre **Client** y **Schedule** se hace para saber qué cliente ha pedido cita en el salón de belleza.

La relación entre **Service** y **Schedule** se hace para saber qué servicio ha pedido el cliente a la hora de pedir cita en el salón de belleza.

La relación entre **User** y **Schedule** se hace para saber qué trabajador se va a encargar de realizar el servicio al cliente.

TechnicalSheet

Entidad **TechnicalSheet**, esta entidad hace referencia a la ficha técnica o al historial de los clientes del salón de belleza, en ella se guardan los datos referentes a como se ha realizado un servicio, por ejemplo, el cliente, el servicio, el día y la hora de la cita y el trabajador, una descripción técnica del trabajo realizado y una foto para plasmas el trabajo realizado.

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="date")
 */
private $date;

/**
 * @ORM\Column(type="string", length=1000, nullable=false)
 */
private $description;

/**
 * @ORM\Column(type="string", nullable=true)
 */
private $photo;
```

La entidad **TechnicalSheet** está relacionada con las entidades **Client**, **User** y **Service.**

```php
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Client", inversedBy="technicalSheet")
 */
private $client;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="technicalSheet")
 */
private $user;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Service", inversedBy="technicalSheet")
 */
private $service;
```

La relación entre **Client** y **TechnicalSheet** se hace para poder registrar los servicios en la ficha técnica que se le han realizado a un cliente para así poder tener un historial de servicios realizados.

La relación entre **User** y **TechnicalSheet** se hace para saber qué trabajador ha realizado el servicio y así poder anotarlo en la ficha técnica para tener un historial de servicios realizados.

La relación entre **Service** y **TechnicalSheet** se hace para saber qué servicio se le ha realizado al cliente y así poder anotarlo en la ficha técnica para tener un historial de servicios realizados.

PaymentProduct

Entidad **PaymentProduct**, esta entidad hace referencia a los productos que se han vendido en el salón de belleza, en ella se guardan los datos referentes a la venta de productos, por ejemplo, el producto, el cliente, el trabajador, el día y la hora de la venta, el precio, la cantidad, el precio total, la forma de pago, …

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="time")
 */
private $hour;

/**
 * @ORM\Column(type="date")
 */
private $dateProduct;

/**
 * @ORM\Column(type="decimal", precision=5, scale=2)
 */
private $price;

/**
 * @ORM\Column(type="integer")
 */
private $quantity;
```

```php
/**
 * @ORM\Column(type="decimal", precision=10, scale=2)
 */
private $sumPrice;

/**
 * @ORM\Column(type="boolean", nullable=true)
 */
private $paid;

/**
 * @ORM\Column(type="date")
 */
private $datePaid;

/**
 * @ORM\Column(type="string", length=255)
 */
private $paymentType;

/**
 * @ORM\Column(type="string", length=1000, nullable=true)
 */
private $comments;
```

La entidad **PaymentProduct** está relacionada con las entidades **Product**, **Client** y **User.**

```php
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Product", inversedBy="paymentProduct")
 */
private $product;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Client", inversedBy="paymentProduct")
 */
private $client;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="paymentProduct")
 */
private $user;
```

La relación entre **Product** y **PaymentProduct** se hace para saber qué productos se han vendido en el salón de belleza.

La relación entre **Client** y **PaymentProduct** se hace para saber qué cliente ha comprado un producto.

La relación entre **User** y **PaymentProduct** se hace para saber qué trabajador ha cobrado los productos que se han vendido a un cliente.

PaymentService

Entidad **PaymentService**, esta entidad hace referencia a los servicios que se han realizado en el salón de belleza, en ella se guardan los datos referentes al cobro de los servicios, por ejemplo, el servicio, el cliente, el trabajador, el día y la hora de la realización del servicio, el precio, la forma de pago, …

```php
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="time")
 */
private $hour;

/**
 * @ORM\Column(type="date")
 */
private $dateService;

/**
 * @ORM\Column(type="date")
 */
private $datePaid;
```

```php
/**
 * @ORM\Column(type="decimal", precision=5, scale=2)
 */
private $price;

/**
 * @ORM\Column(type="boolean")
 */
private $paid;

/**
 * @ORM\Column(type="string", length=255)
 */
private $paymentType;

/**
 * @ORM\Column(type="string", length=1000, nullable=true)
 */
private $comments;
```

Le entidad **PaymentService** está relacionada con las entidades **Client**, **User** y **Service**.

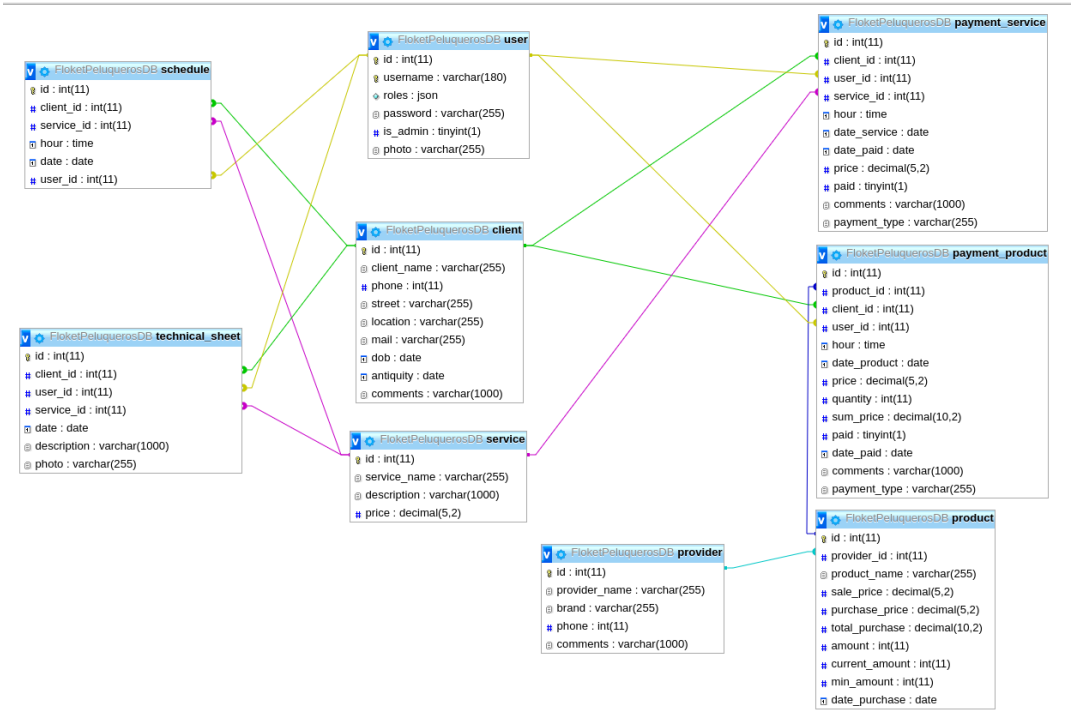
```php
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Client", inversedBy="paymentService")
 */
private $client;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="paymentService")
 */
private $user;

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Service", inversedBy="paymentService")
 */
private $service;
```

La relación entre **Client** y **PaymentService** se hace para saber qué cliente ha demandado un servicio.

La relación entre **User** y **PaymentService** se hace para saber qué trabajador ha cobrado el servicio que ha pagado un cliente.

La relación entre **Service** y **PaymentService** se hace para saber qué servicio se le ha cobrado a un cliente a la hora de pagar.

Desde la aplicación **phpMyAdmin** podemos ver el esquema de la base de datos que hemos creado, pudiendo observar todas las relaciones creadas.



Para la realización de las **consultas** a la base de datos tenemos dos opciones:

- Doctrine queries (DQL)
- Doctrine Query Builder.

Personalmente me he decantado por usar Doctrine queries porque me parece mucho más cercano a las consultas SQL de toda la vida y se pueden realizar consultas más complejas.

La diferencia más grande es que se tiene que pensar en términos de **objetos** en lugar de **filas** en una base de datos, pero por lo demás es muy parecido.

Consultar objetos con DQL

Aquí podemos ver una consulta realizada para saber la facturación diaria, pasándole como parámetro una fecha.

```
public function findPaymentByDay($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentProduct.id, paymentProduct.hour, paymentProduct.dateProduct, paymentProduct.price,
           paymentProduct.quantity, paymentProduct.sumPrice, paymentProduct.paid, paymentProduct.datePaid,
           paymentProduct.comments, client.clientName, user.username, product.productName
    FROM App:PaymentProduct paymentProduct, App:Client client, App:User user, App:Product product
    WHERE paymentProduct.client = client.id
    AND paymentProduct.user = user.id
    AND paymentProduct.product = product.id
    AND paymentProduct.dateProduct = :date
    ORDER BY paymentProduct.dateProduct DESC
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Product objects
    return $query->getResult();
}
```

Esta consulta es algo más simple, nos devuelve todos los datos de los clientes.

```
public function findAllCli()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT client.id, client.clientName, client.phone, client.street,
        client.location, client.mail, client.dob, client.antiquity, client.comments
        FROM App:Client client
        ')->getResult();
}
```

Consultar objetos con Doctrine Query Builder

En lugar de escribir un string DQL, se puede emplear el **objeto QueryBuilder** para que lo construya.

El objeto **QueryBuilder** contiene cada método necesario para construir la consulta. Llamando al método *getQuery()* el query builder devuelve un objeto normal Query, que puede usarse para obtener el resultado del query.

```
/**
 * @return Client[] Returns an array of Client objects
 */
public function findAllClients()
{
    return $this->createQueryBuilder( alias: 'u')
        ->orderBy( sort: 'u.clientName',  order: 'ASC')
        ->getQuery()
        ->execute()
        ;
}
```

Esta consulta tiene el mismo resultado que la anterior creada con DQL, devuelve todo el objeto clientes, sin tener que especificar qué campos te interesan.

Puede resultar más cómodo si te interesan todos los datos de un objeto, pero si solamente te interesan unos campos determinados o quieres pasar variables desde el controlador o agrupar mediante un GROUP BY o realizar un WHERE por año o mes no es factible este método.

Únicamente tenemos una consulta para actualizar el stock de los productos donde se le pasa como parámetros el identificador del producto y la cantidad que se va a comprar para poder actualizar el stock de ese producto; el resto de las consultas son para obtener datos.

```php
public function updateStock($amountDiscount, $productId)
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    UPDATE App:Product product SET product.currentAmount = product.currentAmount - :amountDiscount
    WHERE product.id = :productId
    '
    )->setParameters(array('amountDiscount' => $amountDiscount,
        'productId' => $productId));

    // returns an array of Schedule objects
    return $query->execute();
}
```

Para crear todas las consultas necesarias en nuestra aplicación tenemos la carpeta llamada **Repository** y dentro de esta carpeta tenemos un archivo por cada entidad, estos archivos son:

- ClientRepository.php → 2 consultas hechas
- UserRepository.php → 2 consultas hechas
- ServiceRepository.php → 4 consultas hechas
- ProviderRepository.php → 5 consultas hechas
- TechnicalSheetRepository.php → 1 consulta hecha
- ScheduleRepository.php → 1 consulta hecha
- ProductRepository.php → 5 consultas hechas
- PaymentProductRepository.php → 14 consultas hechas
- PaymentServiceRepository.php → 11 consultas hechas

Un total de 45 sentencias creadas, 44 de ellas para consultar datos, únicamente una sentencia para actualizar el stock de los productos.

A continuación, voy a ir explicando todas las sentencias que he creado.

En primer lugar, explicaré el flujo de como un usuario interactúa con la aplicación y como trabaja internamente para poder mostrar los datos, el concepto de muchas sentencias es el mismo, por tanto, explicaré de forma más pormenorizada la primera vez que aparezca la sentencia, pero en el mismo tipo de sentencias únicamente las comentaré para que la explicación no se haga tan tediosa.

Client (Clientes)

- **findAllCli** (Buscar todos los clientes)
- **findClientByName** (Buscar un cliente por su nombre)

En el apartado de clientes tenemos un buscador, de forma automática si no se busca ningún cliente, aparecen todos los clientes registrados, pero si introducimos el nombre de un cliente, se realiza una búsqueda.

Para discernir si se tiene que hacer una búsqueda por el nombre introducido o se tienen que mostrar todos los datos, tenemos al controlador, este controlador se encarga de la lógica de la aplicación. Según si en la casilla de búsqueda tenemos un nombre introducido o no, le manda al repositorio que ejecute una consulta u otra.

En este caso va a ejecutar la sentencia **findClientByName** pasándome como parámetro el nombre introducido en la casilla de búsqueda.

Al ejecutar la sentencia, se obtienen una serie de valores que se devuelven al controlador, el controlador le pasa esos datos a la plantilla donde esta se encarga de presentar de forma gráfica los datos.

```
$clientname = $request->get( key: 'query');

if (!empty($clientname)) {
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Client::class)->findClientByName($clientname);
} else {
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Client::class)->findAllCli();
}
```

```php
public function findAllCli()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT client.id, client.clientName, client.phone, client.street,
        client.location, client.mail, client.dob, client.antiquity, client.comments
        FROM App:Client client
        ')->getResult();
}

public function findClientByName($clientname): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT client.id, client.clientName, client.phone, client.street,
        client.location, client.mail, client.dob, client.antiquity, client.comments
        FROM App:Client client
    WHERE client.clientName LIKE :clientname
    '
    )->setParameter( key: 'clientname',  value: '%'.$clientname.'%');

    // returns an array of Product objects
    return $query->getResult();
}
```

La primera consulta selecciona todos los datos de los clientes y obtiene el resultado, la segunda consulta es algo más compleja pero mucho más potente ya que obtiene datos de forma dinámica. Previamente desde el controlador se pasa como parámetro el texto introducido en la búsqueda y se añade a la sentencia para realizar la búsqueda de los datos. El nombre buscado se pasa entre porcentajes para que la búsqueda no sea estricta y aparezcan nombres que contengan el texto introducido.

```twig
{% for client in pagination %}
    <tr {% if loop.index is odd %}class="color"{% endif %}>
        <td>{{ client.clientName }}</td>
        <td>{{ client.phone }}</td>
        <td>{{ client.street }}</td>
        <td>{{ client.location }}</td>
        <td>{{ client.mail }}</td>
        <td>{{ client.dob ? client.dob|date('d-m-Y') : '' }}</td>
        <td>{{ client.antiquity ? client.antiquity|date('d-m-Y') : '' }}</td>
        <td>{{ client.comments }}</td>
        <td>
            <a href="{{ path('client_show', {'id': client.id}) }}">Mostrar</a>
            <a href="{{ path('client_edit', {'id': client.id}) }}">Editar</a>
        </td>
    </tr>
{% else %}
    <tr>
        <td colspan="10">No se encontraron registros</td>
    </tr>
{% endfor %}
```

User (Trabajadores)

- **findAllUser** (Buscar todos los trabajadores)
- **findUserByName** (Buscar un trabajador por su nombre)

En el caso de las sentencias creadas en el repositorio de trabajadores son exactamente las mismas que para los usuarios, únicamente voy a mostrar las 2 sentencias creadas.

FindAllUser se usa para mostrar a todos los trabajadores, **findUserByName** se usa para mostrar todos los datos de los trabajadores cuyo nombre coincida con el introducido en la casilla de búsqueda.

```php
public function findAllUser()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT user.id, user.username, user.isAdmin
        FROM App:User user
        ')->getResult();
}

public function findUserByName($username): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT user.id, user.username, user.isAdmin
    FROM App:User user
    WHERE user.username LIKE :username
    '
    )->setParameter( key: 'username', value: '%'.$username.'%');

    // returns an array of Product objects
    return $query->getResult();
}
```

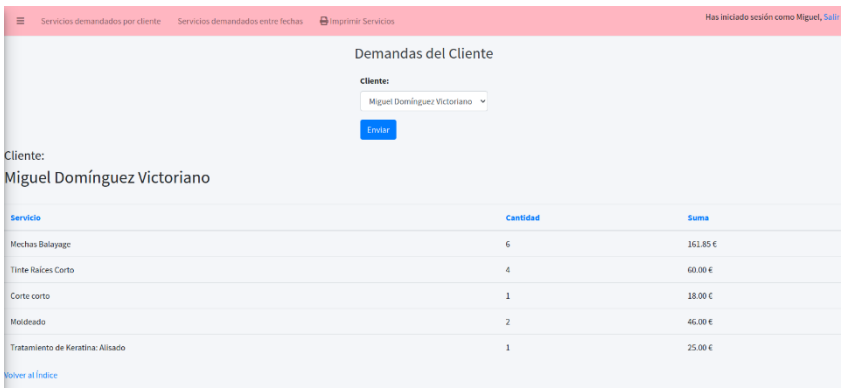
Service (Servicios)

- **findAllServ** (Buscar todos los servicios)
- **findServiceByName** (Buscar un servicio por su nombre)
- **findDemandForServicesByAClient** (Buscar servicios demandados por un cliente)
- **findDemandForServicesBetweenDates** (Buscar servicios demandados entre fechas)

En el apartado de servicios, igual que en los de clientes y trabajadores, tenemos las consultas **findAllServ** para consultar todos los servicios que se prestan en el salón de belleza y la consulta **findServiceByName** para buscar los servicios que coincidan con el nombre del servicio que se busca.

```php
public function findAllServ()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT service.id, service.serviceName, service.description, service.price
        FROM App:Service service
        ')->getResult();
}

public function findServiceByName($servicename): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT service.id, service.serviceName, service.description, service.price
        FROM App:Service service
     WHERE service.serviceName LIKE :servicename
    '
    )->setParameter( key: 'servicename', value: '%' . $servicename . '%');

    // returns an array of Product objects
    return $query->getResult();
}
```

Por otro lado, tenemos también un apartado para consultar todos los servicios que se ha realizado una persona, con el mismo procedimiento explicado anteriormente podemos observar el flujo de datos desde que se selecciona el nombre de un cliente hasta que aparecen sus datos.

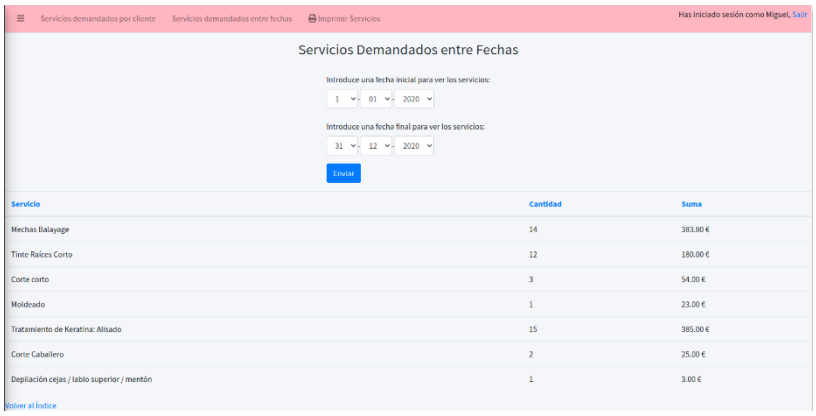



```php
public function findDemandForServicesByAClient($clientId): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price) as serviceSum, COUNT(paymentService.service) as serviceCount, service.serviceName, client.clientName
    FROM App:PaymentService paymentService, App:Service service, App:Client client
    WHERE paymentService.service = service.id
    AND paymentService.client = client.id
    AND paymentService.client = :clientId
    GROUP BY paymentService.service
    '
    )->setParameter( key: 'clientId', $clientId);
    return $query->getResult();
}
```

Esta sentencia involucra a 3 entidades; a la entidad **PaymentService** para saber de cada servicio que ha demandado un cliente las veces que lo ha solicitado y el precio total gastado en cada servicio, la entidad **Service** para saber el nombre del servicio y la entidad **Client** para saber qué cliente ha demandado el servicio, agrupando los datos por el nombre del servicio para que aparezcan de forma correcta.

```php
public function serviceClient(PaginatorInterface $paginator, Request $request): Response
{
    $serviceClient = new ServiceDemandClient();
    $form = $this->createForm( type: ServiceDemandClientType::class, $serviceClient);
    $form->handleRequest($request);

    $clientId = $serviceClient->getClient();
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Service::class)->findDemandForServicesByAClient($clientId);

    if ($query == null) {
        $cliName = Service::REGISTROS;
    } else {
        $cliName = $query[0]["clientName"];
    }
    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 1000 /*limit per page*/
    );
```

También tenemos otra sección para saber los servicios demandado entre un rango de fecha que se puede seleccionar.




```php
public function findDemandForServicesBetweenDates($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price) as serviceSum, COUNT(paymentService.service) as serviceCount, service.serviceName
    FROM App:PaymentService paymentService, App:Service service
    WHERE paymentService.service = service.id
    AND paymentService.dateService BETWEEN :initDate AND :endDate
    GROUP BY paymentService.service
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}
```

Esta sentencia involucra a 2 entidades; a la entidad **PaymentService** para saber la cantidad de veces que se ha demandado un servicio y el beneficio obtenido por cada servicio, la entidad **Service** para saber el nombre del servicio, agrupando los datos por el nombre del servicio para que aparezcan de forma correcta, a esta sentencia se le pasan 2 parámetros desde el controlador que son la fecha de inicio de la búsqueda y la fecha final, estas dos fechas determina el rango de fechas a buscar.

```php
public function serviceBetween(PaginatorInterface $paginator, Request $request)
{
    $serviceBetweenDates = new ServiceDemandBetweenDates();
    $form = $this->createForm( type: ServiceDemandBetweenDatesType::class, $serviceBetweenDates);
    $form->handleRequest($request);

    $initDate = $serviceBetweenDates->getStartDay();
    $endDate = $serviceBetweenDates->getEndDay();

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Service::class)->findDemandForServicesBetweenDates($initDate, $endDate);

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
         limit: 1000 /*limit per page*/
    );

    return $this->render( view: 'service/serviceBetween.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
    ]);
}
```

Product (Productos)

- **findAllProd** (Buscar todos los productos)
- **findProductByName** (Buscar un producto por su nombre)
- **findAllStock** (Buscar productos con falta de stock)
- **findDemandForProductsByAClient** (Buscar productos demandados por un cliente)
- **findDemandForProductsBetweenDates (**Buscar productos demandados entre fechas**)**

En el apartado de productos, igual que en los de clientes, trabajadores y servicios, tenemos las consultas **findAllProd** para consultar todos los productos que se venden en el salón de belleza y la consulta **findProductByName** para buscar los productos que coincidan con el nombre del producto que se busca.

```php
public function findAllProds()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT product.id, product.productName, product.salePrice, product.purchasePrice, product.amount, product.currentAmount,
        product.minAmount, product.datePurchase
        FROM App:Product product
        WHERE product.currentAmount != 0
        ')->getResult();
}

public function findProductByName($productname): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT product.id, product.productName, product.salePrice, product.purchasePrice, product.amount, product.currentAmount,
        product.minAmount, product.datePurchase
    FROM App:Product product
    WHERE product.productName LIKE :productname
    '
    )->setParameter( key: 'productname', value: '%' . $productname . '%');

    // returns an array of Product objects
    return $query->getResult();
}
```

También tenemos un apartado para saber qué productos tenemos con falta de stock por si fuese necesario pedir más al proveedor de productos.


```php
public function findAllStock()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT product.id, product.productName, product.salePrice, product.purchasePrice, product.amount, product.currentAmount,
        product.minAmount, product.datePurchase
        FROM App:Product product
        WHERE product.currentAmount <= product.minAmount
        ')->getResult();
}
```

Esta sentencia únicamente involucra a la entidad **Product,** para saber si un producto tiene falta de stock se compara el stock mínimo con el stock actual del producto, si el stock actual es menor o igual al stock mínimo se muestra en este listado para saber que se tiene que realizar el pedido al proveedor.

```php
/**
 * @Route("/stock", name="product_stock", methods={"GET"})
 */
public function stock(PaginatorInterface $paginator, Request $request): Response
{
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Product::class)->findAllStock();

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );

    return $this->render( view: 'product/stock.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

Por otro lado, tenemos también un apartado para consultar todos los productos que se ha comprado una persona y los productos demandados entre un intervalo de fechas, estos procedimientos son exactamente los mismos que para el apartado de servicios, por tanto, únicamente voy a explicar las sentencias creadas.

.

```php
public function findDemandForProductsByAClient($clientId): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice) as productSum, SUM(paymentProduct.quantity) as productCount, product.productName, client.clientName
    FROM App:PaymentProduct paymentProduct, App:Product product, App:Client client
    WHERE paymentProduct.product = product.id
    AND paymentProduct.client = client.id
    AND paymentProduct.client = :clientId
    GROUP BY product.productName
    '
    )->setParameter( key: 'clientId', $clientId);
    return $query->getResult();
}

public function findDemandForProductsBetweenDates($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice) as productSum, SUM(paymentProduct.quantity) as productCount, product.productName
    FROM App:PaymentProduct paymentProduct, App:Product product
    WHERE paymentProduct.product = product.id
    AND paymentProduct.dateProduct BETWEEN :initDate AND :endDate
    GROUP BY product.productName
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}
```

En la sentencia **findDemandForProductsByAClient** están involucradas las entidades **PaymentProduct**, **Product** y **Client;** la entidad **PaymentProduct** para saber la cantidad de producto que se ha comprado y la suma, la entidad **Product** para saber el nombre del producto comprado y la entidad Client para saber qué cliente ha comprado el producto, agrupando los datos por en nombre del producto.

Por otro lado, tenemos la sentencia **findDemandForProductsBetweenDates** que involucra a las entidades **PaymentProduct** y **Product**; la entidad **PaymentProduct** para saber la suma de beneficios y la cantidad vendida de cada producto, la entidad **Product** para saber el nombre del producto vendido, a esta sentencia se le pasan 2 parámetros desde el controlador que son la fecha de inicio de la búsqueda y la fecha final, estas dos fechas determinan el rango de fechas a buscar.

Provider (Proveedores)

- **findAllProvid** (Buscar todos los proveedores)
- **findProviderByName** (Buscar un proveedor por su nombre)
- **findProviderByBrand** (Buscar un proveedor por su marca)
- **findProductPurchasedFromProviders** (Buscar productos comprados a un proveedor)
- **findProductPurchasedBetweenDates** (Buscar productos comprados entre fechas)

En el apartado de proveedores, igual que en los de clientes, trabajadores, servicios …, tenemos las consultas **findAllProvid**, para consultar todos los proveedores que nos abastecen de productos, en el caso del apartado de proveedores tenemos la posibilidad de buscar por el nombre del proveedor o por la marca que representa, por tanto, tenemos las sentencias **findProviderByName** y **findProviderByBrand**.

Muestro la parte del controlador que hace que se ejecute una sentencia u otra según si se busca por nombre, marca o se quieren ver todos los datos.

```php
/**
 * @Route("/", name="provider_index", methods={"GET"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $providername = $request->get( key: 'query');
    $brand = $request->get( key: 'query2');

    if (!empty($providername)) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Provider::class)->findProviderByName($providername);
    } elseif  (!empty($brand))
    {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Provider::class)->findProviderByBrand($brand);
    } else {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Provider::class)->findAllProvid();
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );
    return $this->render( view: 'provider/index.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

```php
public function findAllProvid()
{
    return $this->getEntityManager()
        ->createQuery('
        SELECT provider.id, provider.providerName, provider.brand, provider.phone, provider.comments
        FROM App:Provider provider
        ')->getResult();
}

public function findProviderByName($providername): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT provider.id, provider.providerName, provider.brand, provider.phone, provider.comments
        FROM App:Provider provider
     WHERE provider.providerName LIKE :providername
    '
    )->setParameter( key: 'providername', value: '%' . $providername . '%');

    // returns an array of Product objects
    return $query->getResult();
}
 public function findProviderByBrand($brand): array
 {
     $entityManager = $this->getEntityManager();

     $query = $entityManager->createQuery('
     SELECT provider.id, provider.providerName, provider.brand, provider.phone, provider.comments
         FROM App:Provider provider
      WHERE provider.brand LIKE :brand
     '
     )->setParameter( key: 'brand', value: '%' . $brand . '%');

     // returns an array of Product objects
     return $query->getResult();
 }
```

Como hemos visto, estas sentencias son excluyentes a la hora de su ejecución, podemos observar que son prácticamente iguales, lo único diferentes es la clausura WHERE para indicar el nombre del proveedor o la marca del proveedor.

En las tres sentencias únicamente tenemos involucrada a la entidad **Provider**, en la sentencia **findAllProvider** se muestran todos los proveedores; en la sentencia **findProviderByName** se muestran todos los proveedores que coincidan con el nombre que se introduce en la casilla de búsqueda correspondiente y en la sentencia **findProviderByBrand** se muestran todos los proveedores que coincidan con la marca que se introduce en la casilla de búsqueda correspondiente.

Por otro lado, tenemos las consultas **findProductPurchasedFromProviders** y **findProductPurchasedBetweenDates,** estas dos consultas son muy similares a las vistas anteriormente, voy a explicar únicamente las sentencias.

```php
public function findProductPurchasedFromProviders($providerId): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT  product.productName, product.amount, product.totalPurchase, product.datePurchase, provider.providerName
    FROM App:Product product, App:Provider provider
    WHERE product.provider = provider.id
    AND product.provider = :providerId
    ORDER BY product.datePurchase DESC
    '
    )->setParameter( key: 'providerId', $providerId);

    // returns an array of Product objects
    return $query->getResult();
}

public function findProductPurchasedBetweenDates($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT  product.productName, product.amount, product.totalPurchase, product.datePurchase, provider.providerName
    FROM App:Product product, App:Provider provider
    WHERE product.provider = provider.id
    AND product.datePurchase BETWEEN :initDate AND :endDate
    ORDER BY product.datePurchase DESC
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}
```

En la sentencia **findProductPurchasedFromProviders** están involucradas las entidades **Product** y **Provider,** se muestra el nombre, el precio, la cantidad y la fecha de compra del producto junto al nombre del proveedor, a esta sentencia se le pasa el parámetro **providerId** desde el controlador para mostrar únicamente los productos vendidos por un proveedor en concreto ordenado por la fecha de compra de los productos de forma descendente.

En la sentencia **findProductPurchasedBetweenDates** están involucradas las entidades **Product** y **Provider,** se muestra el nombre, el precio, la cantidad y la fecha de compra del producto junto al nombre del proveedor, a esta sentencia se le pasan los parámetros **initDate** y **endDate** desde el controlador para mostrar únicamente los productos vendidos por los diferentes proveedores en el rango de fechas seleccionado, ordenado por la fecha de compra de los productos de forma descendente.

TechnicalSheet (Ficha Técnica)

- **findTechnicalSheetByClientId** (Buscar la ficha técnica de un cliente)

En el apartado de la ficha técnica únicamente tenemos una sentencia creada, esta sentencia sirve para mostrar el historial de servicios que se ha realizado un cliente. La sentencia creada es muy similar a otras ya explicadas anteriormente, por tanto, voy a explicar directamente como se ha creado la sentencia.

```
public function findTechnicalSheetByClientId($clientId): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT  technicalSheet.id, technicalSheet.date, technicalSheet.description, technicalSheet.photo,
            client.clientName, service.serviceName, client.id as clientId
    FROM App:TechnicalSheet technicalSheet, App:Client client, App:Service service
    WHERE technicalSheet.client = client.id
    AND technicalSheet.client = :clientId
    AND technicalSheet.service = service.id
    ORDER BY technicalSheet.date DESC
    '
    )->setParameter( key: 'clientId', $clientId);
    return $query->getResult();
}
```

En esta sentencia intervienen las entidades **TechnicalSheet**, **Client** y **Service**, se muestra la fecha, la descripción, una foto, el nombre del cliente y el nombre del servicio, a esta sentencia se le pasa el parámetro **clientId** desde el controlador para mostrar únicamente los datos del cliente seleccionado, ordenado por la fecha de realización del servicio de forma descendente.

Schedule (Citas)

- **findScheduleByDate** (Buscar citas por fecha)

En el apartado de citas únicamente tenemos una sentencia creada, esta sentencia sirve para ver las citas que han pedido los clientes para el día que se selecciona. La sentencia creada es muy similar a otras ya explicadas anteriormente, por tanto, voy a explicar directamente como se ha creado la sentencia.

```
public function findScheduleByDate($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
        SELECT schedule.id, schedule.hour, schedule.date, client.clientName, service.serviceName, user.username
        FROM App:Schedule schedule, App:Client client, App:Service service, App:User user
        WHERE schedule.date = :date
        AND schedule.client = client.id
        AND schedule.service = service.id
        AND schedule.user = user.id
        ORDER BY schedule.hour ASC
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Schedule objects
    return $query->getResult();
}
```

En esta sentencia intervienen las entidades **Schedule**, **Client**, **Service y User**, se muestra la hora, la fecha, el nombre del cliente, el nombre del servicio, y el trabajador que va a realizar el servicio, a esta sentencia se le pasa el parámetro **date** desde el controlador para mostrar únicamente los datos de la fecha seleccionada, ordenado por la hora de las citas de forma ascendente.

PaymentProduct (Pago de Productos)

- **findAllPaymentProduct** (Buscar pago de productos)
- **findAllPaymentProductNotPaid** (Buscar pagos de productos no cobrados)
- **findPaymentByDay** (Buscar pagos de un día)
- **sumPaymentByDay** (Suma de los pagos de un día)
- **sumPaymentByDayInCash** (Suma de los pagos de un día en efectivo)
- **findPaymentBetweenDays** (Buscar pagos realizados entre un rango de días)
- **sumPaymentBetweenDays** (Suma de los pagos realizados entre un rango de días)
- **findProfitByMonths** (Buscar los beneficios de cada mes)
- **findProfitByYears** (Buscar los beneficios de un año)
- **mostDemandedProductsPerYear** (Productos más demandados del año)
- **findPriceById** (Buscar precio por su identificador)
- **findAmountById** (Buscar cantidad por identificador del producto)
- **findMinAmountById** (Buscar cantidad mínima por identificador del producto)
- **updateStock** (Actualizar stock de los productos)

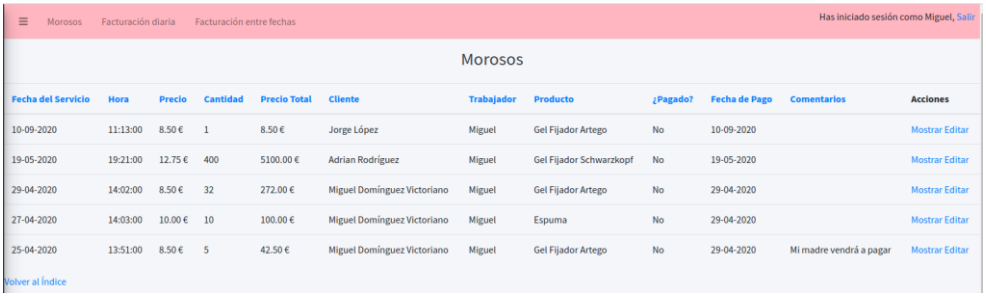
En el apartado del cobro de productos tenemos un gran número de consultas realizadas, algunas se usan para llevar a cabo la facturación diaria, otras para llevar a cabo la facturación entre fechas, otras para actualizar el stock cuando se realizan compras y otras para poder realizar las gráficas.

La primera consulta es exactamente igual a las otras anteriormente vistas, por tanto, me voy a centrar en explicar únicamente la sentencia realizada.

```
public function findAllPaymentProduct(): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentProduct.id, paymentProduct.hour, paymentProduct.dateProduct, paymentProduct.price,
           paymentProduct.quantity, paymentProduct.sumPrice, paymentProduct.paid, paymentProduct.datePaid,
           paymentProduct.comments, client.clientName, user.username, product.productName
    FROM App:PaymentProduct paymentProduct, App:Client client, App:User user, App:Product product
    WHERE paymentProduct.client = client.id
    AND paymentProduct.user = user.id
    AND paymentProduct.product = product.id
    ORDER BY paymentProduct.dateProduct DESC, paymentProduct.hour DESC
    '
    );
    // returns an array of Product objects

    return $query->getResult();

}
```

En esta sentencia intervienen las entidades **PaymentProduct**, **Client**, **User** y **Product**; se muestra la hora, la fecha de venta, el precio, la cantidad, el precio total, si se ha pagado el producto, la fecha de pago, el cliente que ha comprado el producto, el vendedor y el nombre del producto, ordenado por fecha y hora de compra.

La siguiente consulta creada sirve para saber qué productos no han sido pagados por los clientes, siendo muy parecida a la consulta anterior, únicamente se añade una cláusula para buscar los pagos no cobrados.




```php
public function findAllPaymentProductNotPaid(): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentProduct.id, paymentProduct.hour, paymentProduct.dateProduct, paymentProduct.price,
           paymentProduct.quantity, paymentProduct.sumPrice, paymentProduct.paid, paymentProduct.datePaid,
           paymentProduct.comments, client.clientName, user.username, product.productName
    FROM App:PaymentProduct paymentProduct, App:Client client, App:User user, App:Product product
    WHERE paymentProduct.client = client.id
    AND paymentProduct.user = user.id
    AND paymentProduct.product = product.id
    AND paymentProduct.paid = 0
    ORDER BY paymentProduct.dateProduct DESC
    '
    );

    // returns an array of Product objects
    return $query->getResult();
}
```

Lo siguiente a explicar es la facturación diaria de los productos, este apartado involucra a 3 consultas, **findPaymentByDay**, **sumPaymentByDay** y **sumPaymentByDayInCash**.

La primera de estas tres consultas muestra toda la facturación de la fecha que se selecciona, la segunda de las consultas muestra el montante total obtenido en la fecha seleccionada y la tercera consulta muestra el montante obtenido ese día en metálico, por tanto, dinero que hay en caja.

```php
public function findPaymentByDay($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentProduct.id, paymentProduct.hour, paymentProduct.dateProduct, paymentProduct.price,
           paymentProduct.quantity, paymentProduct.sumPrice, paymentProduct.paid, paymentProduct.datePaid,
           paymentProduct.comments, client.clientName, user.username, product.productName
    FROM App:PaymentProduct paymentProduct, App:Client client, App:User user, App:Product product
    WHERE paymentProduct.client = client.id
    AND paymentProduct.user = user.id
    AND paymentProduct.product = product.id
    AND paymentProduct.dateProduct = :date
    ORDER BY paymentProduct.dateProduct DESC
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Product objects
    return $query->getResult();
}

public function sumPaymentByDay($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice)
    FROM App:PaymentProduct paymentProduct
    WHERE paymentProduct.dateProduct = :date
    AND paymentProduct.paid = 1
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Product objects
    return $query->getResult();
}
```

```php
public function sumPaymentByDayinCash($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice)
    FROM App:PaymentProduct paymentProduct
    WHERE paymentProduct.dateProduct = :date
    AND paymentProduct.paid = 1
    AND paymentProduct.paymentType = :efectivo
    '
    )->setParameter( key: 'date', $date)
        ->setParameter( key: 'efectivo', value: 'efectivo');

    // returns an array of Product objects
    return $query->getResult();
}
```

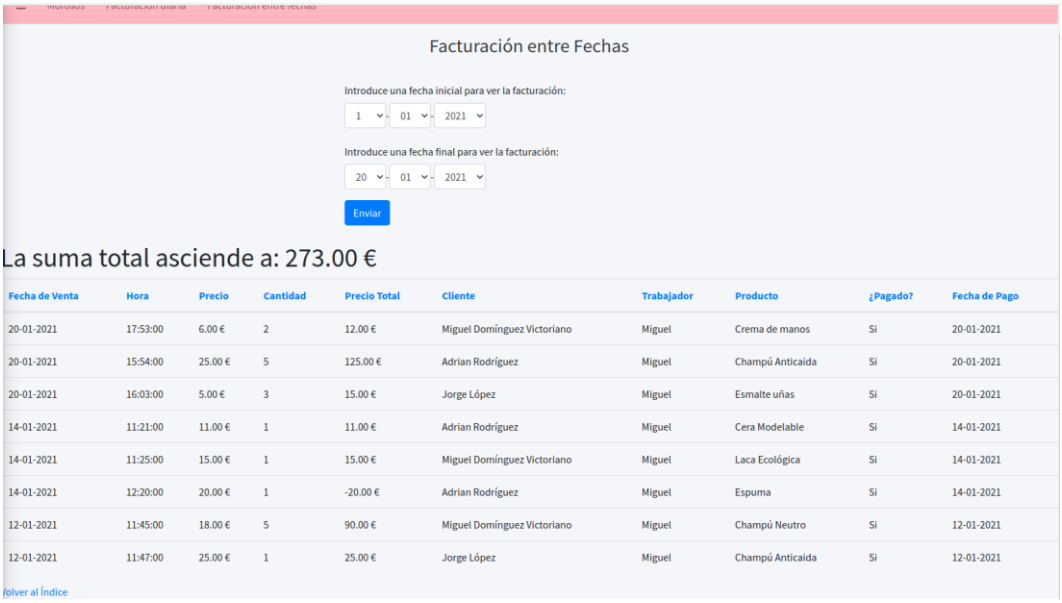
Esta primera consulta es exactamente igual que la consulta **findAllPaymentProduct**, únicamente se le pasa como parámetro desde el controlador la fecha en la que queremos que aparezca la facturación.

La segunda consulta realiza la suma de todas las ventas realizadas en la fecha que se pasa como parámetro y que se hayan pagado.

La tercera consulta es exactamente igual que la anterior pero además también poniendo como condición que se haya pagado en efectivo.

A continuación, tenemos el apartado de facturación entre fechas, este apartado involucra a dos consultas, **findPymentBetweenDays** y **sumPaymentBetweenDays**.

Estas consultas siguen el mismo patrón que las consultas anteriores, únicamente se le pasan dos fechas en vez de una para tener un intervalo de fechas para mostrar los datos.




```php
public function findPaymentBetweenDays($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentProduct.id, paymentProduct.hour, paymentProduct.dateProduct, paymentProduct.price,
           paymentProduct.quantity, paymentProduct.sumPrice, paymentProduct.paid, paymentProduct.datePaid,
           paymentProduct.comments, client.clientName, user.username, product.productName
    FROM App:PaymentProduct paymentProduct, App:Client client, App:User user, App:Product product
    WHERE paymentProduct.client = client.id
    AND paymentProduct.user = user.id
    AND paymentProduct.product = product.id
    AND paymentProduct.dateProduct BETWEEN :initDate AND :endDate
    ORDER BY paymentProduct.dateProduct DESC
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}

public function sumPaymentBetweenDays($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice)
    FROM App:PaymentProduct paymentProduct
    WHERE paymentProduct.dateProduct BETWEEN :initDate AND :endDate
    AND paymentProduct.paid = 1

    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    // returns an array of Product objects
    return $query->getResult();
}
```

Esta primera consulta es exactamente igual que la consulta **findAllPaymentProduct**, únicamente se le pasan como parámetros desde el controlador un intervalo de fechas para mostrar la facturación en ese intervalo de fechas indicado.

La segunda consulta realiza la suma de todas las ventas realizadas entre el rango de fecha que se pasan como parámetros y que se hayan pagado.

También tenemos el apartado de la realización de las compras, estas consultas sirven para saber si tenemos suficiente stock para vender un producto, para actualizar el stock y para saber el precio actual de un producto y aplicarlo.

```
public function findPriceById($id): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
        SELECT product.salePrice
        FROM App:Product product
        WHERE product.id = :id
    '
    )->setParameter( key: 'id', $id);

    // returns an array of Product objects
    return $query->getResult();
}

public function findAmountById($id): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
        SELECT product.currentAmount
        FROM App:Product product
        WHERE product.id = :id
    '
    )->setParameter( key: 'id', $id);

    // returns an array of Product objects
    return $query->getResult();
}
```

```
public function findMinAmountById($id): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
        SELECT product.minAmount
        FROM App:Product product
        WHERE product.id = :id
    '
    )->setParameter( key: 'id', $id);

    // returns an array of Product objects
    return $query->getResult();
}


public function updateStock($amountDiscount, $productId)
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    UPDATE App:Product product SET product.currentAmount = product.currentAmount - :amountDiscount
    WHERE product.id = :productId
    '
    )->setParameters(array('amountDiscount' => $amountDiscount,
        'productId' => $productId));

    // returns an array of Schedule objects
    return $query->execute();
}
```

La sentencia **findPriceById** consulta el precio de venta actual de un producto y lo copia cuando se está creando un nuevo cobro de productos.

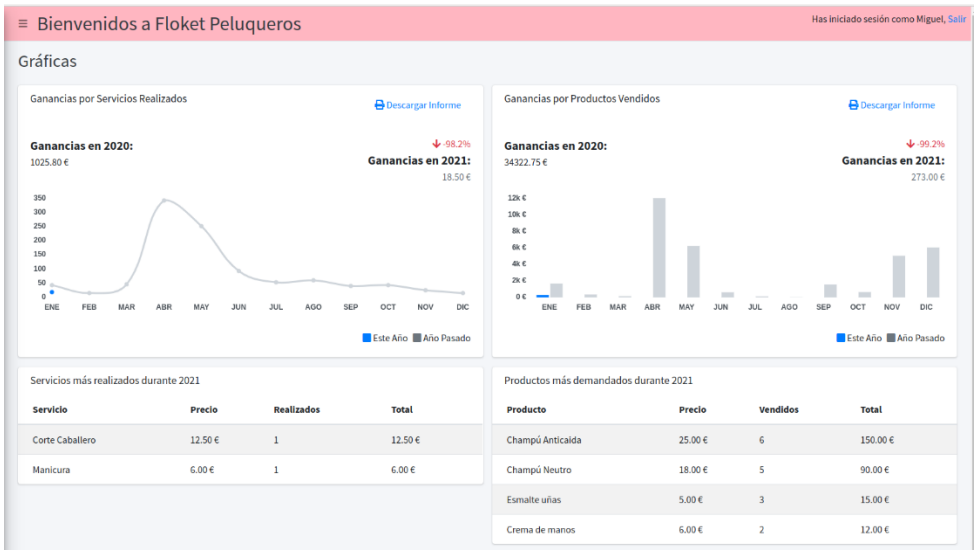
Las sentencias **findAmountById** y **findMinAmountById** viéndolo ahora podría ser una única sentencia, pero me supone más trabajo cambiar todo ahora. Estas dos sentencias preguntan por el stock y el stock mínimo de un producto para saber si a la hora de realizar una venta se tiene suficiente stock o no. Se compara el stock actual con la cantidad de unidades de un producto que se demandan y según si tenemos stock o no se realiza la venta y si se llega al stock mínimo, se muestra un mensaje de advertencia para que se reponga ese stock.

La sentencia **updatestock** es la única sentencia que actualiza los datos de la base de datos, con ella se descuentan las unidades que se venden de un producto.

```
if ($amount - $amountDiscount < $minAmount && $amount - $amountDiscount >= 0) {
    $this->addFlash( type: 'stock',  message: 'Revisar el stock para este producto');

    $entityManager->getRepository( className: PaymentProduct::class)->updateStock($amountDiscount, $productId);

    $entityManager->persist($paymentProduct);
    $entityManager->flush();
    return $this->redirectToRoute( route: 'payment_product_new');

} elseif ($amount - $amountDiscount >= 0) {
    $entityManager->getRepository( className: PaymentProduct::class)->updateStock($amountDiscount, $productId);

    $entityManager->persist($paymentProduct);
    $entityManager->flush();
    return $this->redirectToRoute( route: 'payment_product_index');
} else {
    $this->addFlash( type: 'error',  message: 'No se puede comprar el producto porque no hay suficiente stock');
    return $this->redirectToRoute( route: 'payment_product_new');
}
```

Aquí podemos ver un trozo de código perteneciente al controlador que hace todo el proceso que he explicado para saber si se tiene stock de un producto y en caso afirmativo descontar las unidades vendidas.

Por último, tenemos el apartado de las gráficas, este apartado es muy potente ya que de un vistazo podemos observar la evolución de las ventas de productos en el año actual respecto del anterior y saber los productos más demandados.



```php
public function findProfitbyMonths($i, $year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice)
    FROM App:PaymentProduct paymentProduct
    WHERE MONTH(paymentProduct.datePaid) = :month
    AND YEAR(paymentProduct.datePaid) = :year
    '
    )->setParameter( key: 'month', $i)
        ->setParameter( key: 'year', $year);
    return $query->getResult();
}

public function findProfitbyYears($year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice) as benefits
    FROM App:PaymentProduct paymentProduct
    WHERE YEAR(paymentProduct.datePaid) = :year
    '
    )->setParameter( key: 'year', $year);
    return $query->getResult();
}
```

```php
public function mostDemandedProductsPerYear($year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentProduct.sumPrice) as productSum, product.salePrice,
           SUM(paymentProduct.quantity) as productCount, product.productName
    FROM App:PaymentProduct paymentProduct, App:Product product
    WHERE paymentProduct.product = product.id
    AND YEAR(paymentProduct.datePaid) = :year
    GROUP BY paymentProduct.product
    ORDER BY SUM(paymentProduct.quantity) DESC
    '
    )->setParameter( key: 'year', $year)
        ->setMaxResults( maxResults: 4);
    return $query->getResult();
}
```

Las consultas **findProfitByMonths** y **findProfitByYears** se usan para saber los beneficios totales que ha habido en cada mes del año que se selecciona, a estas consultas se les pasa como parámetro el mes y el año o únicamente el año para obtener los beneficios de dicho período, posteriormente esos datos se pasan al controlador, el controlador monta esos datos de forma que el lenguaje JavaScript lo entienda y pinte las gráficas.

La consulta **mostDemandedProductsPerYear** muestra de forma acotada los cuatro productos más vendidos, para ello intervienen las entidades **PaymentProduct** y **Product**; mostrando el nombre del producto, el precio, las unidades vendidas de ese producto y la suma total obtenida por la venta de ese producto, ordenado la cantidad de unidades vendidas de un producto de forma descendente para que aparezcan los productos más vendidos.

PaymentService (Pago de Servicios)

- **findAllPaymentService** (Buscar pago de servicios)
- **findAllPaymentServiceNotPaid** (Buscar pagos de servicios no cobrados)
- **findPaymentByDay** (Buscar pagos de un día)
- **sumPaymentByDay** (Suma de los pagos de un día)
- **sumPaymentByDayInCash** (Suma de los pagos de un día en efectivo)
- **findPaymentBetweenDays** (Buscar pagos realizados entre un rango de días)
- **sumPaymentBetweenDays** (Suma de los pagos realizados entre un rango de días)
- **findProfitByMonths** (Buscar los beneficios de cada mes)
- **findProfitByYears** (Buscar los beneficios de un año)
- **mostDemandedProductsPerYear** (Productos más demandados del año)
- **findPriceById** (Buscar precio por su identificador)

En el apartado del cobro de servicios tenemos un gran número de consultas realizadas, algunas se usan para llevar a cabo la facturación diaria, otras para llevar a cabo la facturación entre fechas y otras para poder realizar las gráficas.

La primera consulta es exactamente igual a las otras anteriormente vistas, por tanto, me voy a centrar en explicar únicamente la sentencia realizada.

```
public function findAllPaymentService(): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentService.id, paymentService.hour, paymentService.dateService, paymentService.datePaid, paymentService.price,
           paymentService.paid, paymentService.comments, client.clientName, user.username, service.serviceName
    FROM App:PaymentService paymentService, App:Client client, App:User user, App:Service service
    WHERE paymentService.client = client.id
    AND paymentService.user = user.id
    AND paymentService.service = service.id
    ORDER BY paymentService.dateService DESC, paymentService.hour DESC
    '
    );
    // returns an array of Product objects
    return $query->getResult();
}
```

En esta sentencia intervienen las entidades **PaymentService**, **Client**, **User** y **Product**; se muestra la hora, la fecha de realización del servicio, el precio, si se ha pagado el servicio, la fecha de pago, el cliente al que se le ha realizado el servicio, el trabajador y el nombre del servicio, ordenado por fecha y hora de compra.

La siguiente consulta creada sirve para saber qué servicio no han sido pagados por los clientes, siendo muy parecida a la consulta anterior, únicamente se añade una cláusula para buscar los servicios no cobrados.

```
public function findAllPaymentServiceNotPaid(): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentService.id, paymentService.hour, paymentService.dateService, paymentService.datePaid, paymentService.price,
           paymentService.paid, paymentService.comments, client.clientName, user.username, service.serviceName
    FROM App:PaymentService paymentService, App:Client client, App:User user, App:Service service
    WHERE paymentService.client = client.id
    AND paymentService.user = user.id
    AND paymentService.service = service.id
    AND paymentService.paid = 0
    ORDER BY paymentService.dateService DESC
    '
    );

    // returns an array of Product objects
    return $query->getResult();
}
```

Lo siguiente a explicar es la facturación diaria de los servicios, este apartado involucra a 3 consultas, **findPaymentByDay**, **sumPaymentByDay** y **sumPaymentByDayInCash**.

La primera de estas tres consultas muestra toda la facturación de la fecha que se selecciona, la segunda de las consultas muestra el montante total obtenido en la fecha seleccionada y la tercera consulta muestra el montante obtenido ese día en metálico, por tanto, dinero que hay en caja.

```
public function findPaymentByDay($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentService.id, paymentService.hour, paymentService.dateService, paymentService.datePaid, paymentService.price,
           paymentService.paid, paymentService.comments, client.clientName, user.username, service.serviceName
    FROM App:PaymentService paymentService, App:Client client, App:User user, App:Service service
    WHERE paymentService.client = client.id
    AND paymentService.user = user.id
    AND paymentService.service = service.id
    AND paymentService.dateService = :date
    ORDER BY paymentService.dateService DESC
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Product objects
    return $query->getResult();
}

public function sumPaymentByDay($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price)
    FROM App:PaymentService paymentService
    WHERE paymentService.dateService = :date
    '
    )->setParameter( key: 'date', $date);

    // returns an array of Product objects
    return $query->getResult();
}
```

```
public function sumPaymentByDayinCash($date): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price)
    FROM App:PaymentService paymentService
    WHERE paymentService.dateService = :date
    AND paymentService.paymentType = :efectivo
    '
    )->setParameter( key: 'date', $date)
    ->setParameter( key: 'efectivo',  value: 'efectivo');

    // returns an array of Product objects
    return $query->getResult();
}
```

Esta primera consulta es exactamente igual que la consulta **findAllPaymentService**, únicamente se le pasa como parámetro desde el controlador la fecha en la que queremos que aparezca la facturación.

La segunda consulta realiza la suma de todos los servicios realizados en la fecha que se pasa como parámetro y que se hayan pagado.

La tercera consulta es exactamente igual que la anterior pero además también poniendo como condición que se haya pagado en efectivo.

A continuación, tenemos el apartado de facturación entre fechas, este apartado involucra a dos consultas, **findPymentBetweenDays** y **sumPaymentBetweenDays**.

Estas consultas siguen el mismo patrón que las consultas anteriores, únicamente se le pasan dos fechas en vez de una para tener un intervalo de fechas para mostrar los datos.

```
public function findPaymentBetweenDays($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT paymentService.id, paymentService.hour, paymentService.dateService, paymentService.datePaid, paymentService.price,
           paymentService.paid, paymentService.comments, client.clientName, user.username, service.serviceName
    FROM App:PaymentService paymentService, App:Client client, App:User user, App:Service service
    WHERE paymentService.client = client.id
    AND paymentService.user = user.id
    AND paymentService.service = service.id
    AND paymentService.dateService BETWEEN :initDate AND :endDate
    ORDER BY paymentService.dateService DESC
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}

public function sumPaymentBetweenDays($initDate, $endDate): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price)
    FROM App:PaymentService paymentService
    WHERE paymentService.dateService BETWEEN :initDate AND :endDate
    '
    )->setParameter( key: 'initDate', $initDate)
        ->setParameter( key: 'endDate', $endDate);

    return $query->getResult();
}
```

Esta primera consulta es exactamente igual que la consulta **findAllPaymentService**, únicamente se le pasan como parámetros desde el controlador un intervalo de fechas para mostrar la facturación en ese intervalo de fechas indicado.

La segunda consulta realiza la suma de los beneficios de todos los servicios realizados entre el rango de fecha que se pasan como parámetros y que se hayan pagado.

En el caso del apartado del cobro de servicios, evidentemente no tenemos que actualizar ningún stock, únicamente tenemos la consulta **findServicePriceById** para copiar el precio actual del servicio a la hora de cobrar un nuevo servicio.

```
public function findServicePriceById($id): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
        SELECT service.price
        FROM App:Service service
        WHERE service.id = :id
    '
    )->setParameter( key: 'id', $id);

    // returns an array of Product objects
    return $query->getResult();
}
```

Por último, tenemos el apartado de las gráficas, este apartado es muy potente ya que de un vistazo podemos observar la evolución de la demanda de servicios en el año actual respecto del anterior y saber los servicios más demandados.

```
public function findProfitbyMonths($i, $year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price)
    FROM App:PaymentService paymentService
    WHERE MONTH(paymentService.datePaid) = :month
    AND YEAR(paymentService.datePaid) = :year
    '
    )->setParameter( key: 'month', $i)
        ->setParameter( key: 'year', $year);
    return $query->getResult();
}

public function findProfitbyYears($year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price) as benefits
    FROM App:PaymentService paymentService
    WHERE YEAR(paymentService.datePaid) = :year
    '
    )->setParameter( key: 'year', $year);
    return $query->getResult();
}
```

```
public function mostDemandedServicesPerYear($year): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery('
    SELECT SUM(paymentService.price) as serviceSum, service.price,
           COUNT(paymentService.service) as serviceCount, service.serviceName
    FROM App:PaymentService paymentService, App:Service service
    WHERE paymentService.service = service.id
    AND YEAR(paymentService.datePaid) = :year
    GROUP BY paymentService.service
    ORDER BY COUNT(paymentService.service) DESC
    '
    )->setParameter( key: 'year', $year)
        ->setMaxResults( maxResults: 4);
    return $query->getResult();
}
```

Las consultas **findProfitByMonths** y **findProfitByYears** se usan para saber los beneficios totales que ha habido en cada mes del año que se selecciona, a estas consultas se les pasa como parámetro el mes y el año o únicamente el año para obtener los beneficios de dicho período, posteriormente esos datos se pasan al controlador, el controlador monta esos datos de forma que el lenguaje JavaScript lo entienda y pinte las gráficas.

La consulta **mostDemandedProductsPerYear** muestra de forma acotada los cuatro servicios más demandados, para ello intervienen las entidades **PaymentService** y **Service**; mostrando el nombre del servicio, el precio actual, las demandas de ese servicio y la suma total obtenida por la realización de ese servicio, ordenado la cantidad de veces que se ha realizado el servicio de forma descendente para que aparezcan los servicios más demandados.

3.2.2.- Capa Vista

La **Vista** es la parte en que el usuario interactúa (un motor de plantillas es parte de esta capa). En Symfony, la vista es principalmente la capa de plantillas PHP.

Twig es un motor de plantilla para el lenguaje de programación PHP. Su sintaxis origina de Jinja y las plantillas Django. Es un producto de código abierto autorizado bajo Licencia BSD y mantenido por Fabien Potencier.

El motor de plantillas TWIG forma parte del framework Symfony por defecto.

Twig define tres clases de delimitadores:

- {% ... %}, se utiliza para ejecutar declaraciones, como pueden ser los bucles for.
- {{ ... }}, se utiliza para imprimir el contenido de variables o el resultado de evaluar una expresión.
- {# ... #}, se utiliza para añadir comentarios en las plantillas. Estos comentarios no son incluidos en la página renderizada.

Características:

- Flujo de control complejo
- Escapado automático
- Plantillas heredadas
- Filtros de variables
- Soporte i18n (Gettext)
- Macros
- Completamente extensible

Estas plantillas se organizan en la carpeta **templates**, dentro de esta carpeta tenemos una subcarpeta con el mismo nombre que el de los controladores de la aplicación.

Twig nos permite **extender** o **heredar** partes del código que tenemos en otras páginas, esto resulta muy útil por si tenemos partes de código que se repiten en diferentes lugares de nuestra aplicación.

Para ello tenemos las declaraciones {% extends '' %} y {% include '' %}, mediante estas declaraciones podemos extender o incluir código que está creado en otra parte de la aplicación. Por ejemplo, todas las declaraciones de los css o javascript se puede hacer en el archivo **{% extends 'base.html.twig' %}** y extenderse a todas las demás páginas; o también los navbar o los siderbar de la aplicación que son iguales para todas las páginas se pueden incluir mediante la declaración **{% include 'sidebar.html.twig' %}**. También se pueden crear bloques de código mediante la declaración **{%block %}** para separar bloques y de esta forma tener el código más estructurado.

Se pueden crear tantos **Controladores** como se necesite, estos controladores son los encargados de toda la lógica de la aplicación y también de renderizan las plantillas de twig creando todo el entramado de rutas de la aplicación, estas plantillas generan código HTML, CSS, JavaScript, etc.

Para el diseño de la aplicación he usado **Bootstrap** y **Admin LTE** porque proveen de un conjunto de herramientas para diseñar de forma muy fácil y rápida un sitio web.

Para la creación de este proyecto, como ya he mencionado anteriormente tenemos unas funciones básicas que tiene que cumplir todo software para poder persistir datos, estas funciones básicas son llamadas **CRUD** son el acrónimo de **Create**, **Read**, **Update** y **Delete**, en español Crear, Leer, Actualizar y Borrar, el framework Symfony provee de una función para crear un esqueleto de plantillas twig con el que poder tener todas estas funciones. Evidentemente sobre estas plantillas creadas, tenemos que modificar el código para moldearlas a nuestras necesidades.

Estas plantillas son:

- **_delete_form.html.twig** (Borrar un registro mediante su id)
- **_form.html.twig** (Monta el formulario para registrar datos)
- **edit.html.twig** (Monta la estructura para borrar o modificar un registro)
- **index.html.twig** (Apartado principal donde se muestran todos las datos de ese apartado)
- **new.html.twig** (Se monta el formulario para registrar nuevos datos)
- **show.html.twig** (Se muestran los datos de un registro en concreto)

A partir de esta estructura he creado más vistas para que la aplicación y el uso que se hace de ella sea mucho más completo.

Voy a explicar todo el flujo que se crea en la aplicación centrándome en la parte de las plantillas para ver el manejo de las diferentes plantillas, controladores, formularios de registro, …

En el apartado de clientes al acceder a él, lo primero que vemos es el índice (**index.html.twig**) para ello previamente se llama al controlador **ClientController**, este controlador crea la ruta **/clientes** y de él cuelgan el resto de sus páginas (de las diferentes rutas hablaré en el apartado del controlador). Si nos fijamos, el controlador hace sus operaciones y renderiza la página **client/index.html.twig** pasándole todos los datos que ha obtenido de la consulta realizada.

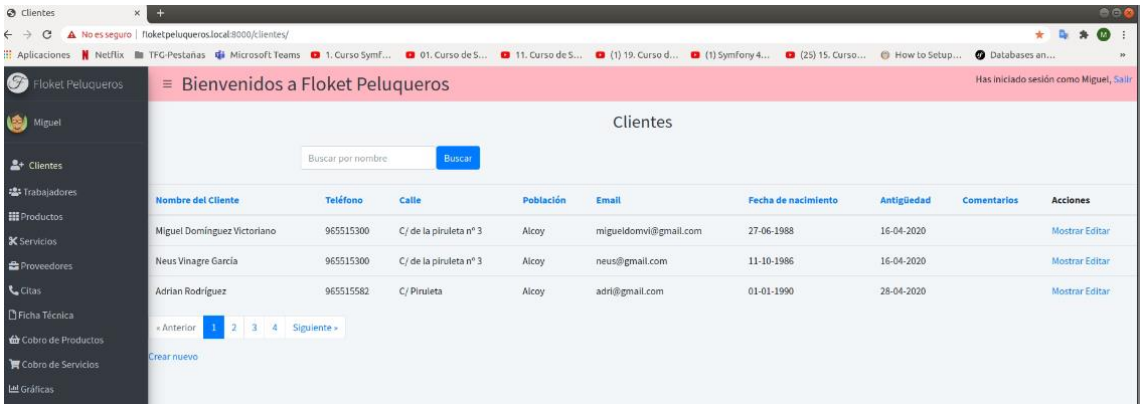
```php
/**
 * @Route("/clientes")
 */
class ClientController extends AbstractController
{
    /**
     * @Route("/", name="client_index", methods={"GET"})
     */
    public function index(PaginatorInterface $paginator, Request $request): Response
    {
        $clientname = $request->get( key: 'query');

        if (!empty($clientname)) {
            $em = $this->getDoctrine()->getManager();
            $query = $em->getRepository( classname: Client::class)->findClientByName($clientname);
        } else {
            $em = $this->getDoctrine()->getManager();
            $query = $em->getRepository( classname: Client::class)->findAllCli();
        }

        $pagination = $paginator->paginate(
            $query, /* query NOT result */
            $request->query->getInt( key: 'page',  default: 1), /*page number*/
             limit: 3 /*limit per page*/
        );

        return $this->render( view: 'client/index.html.twig', [
            'pagination' => $pagination,
        ]);
    }
```

Ahora la plantilla **client/index.html.twig** se encarga de mostrar los diferentes datos mediante código HTML, CSS, …

```
<table class="table">
    <thead>
    <tr>
        {# sorting of properties based on query components #}
        <th{% if pagination.isSorted('a.clientName') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Nombre del Cliente', 'a.clientName') }}
        </th>
        <th{% if pagination.isSorted('a.phone') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Teléfono', 'a.phone') }}
        </th>
        <th{% if pagination.isSorted('a.street') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Calle', 'a.street') }}
        </th>
        <th{% if pagination.isSorted('a.location') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Población', 'a.location') }}
        </th>
        <th{% if pagination.isSorted('a.mail') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Email', 'a.mail') }}
        </th>
        <th{% if pagination.isSorted('a.dob') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Fecha de nacimiento', 'a.dob') }}
        </th>
        <th{% if pagination.isSorted('a.antiquity') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Antigüedad', 'a.antiquity') }}
        </th>
        <th{% if pagination.isSorted('a.comments') %} class="sorted"{% endif %}>
            {{ knp_pagination_sortable(pagination, 'Comentarios', 'a.comments') }}
        </th>
        <th>Acciones</th>
    </tr>
    </thead>
    {# table body #}
    <tbody>
```

Obviando código, muestro la parte donde se crean los títulos de los datos que arroja la consulta y donde se crea el bucle que hace que aparezcan todos los datos, en este caso de los clientes, por otro lado, si nos fijamos podemos observar que se hace referencia a los apartados de Mostrar, Editar y Crear un nuevo registro. El apartado de Mostrar hace referencia a **client_show**, el apartado Editar hace referencia a **client_edit** y el apartado Crear hace referencia a **client_new**, cada uno de ellos son las diferentes plantillas mencionadas, con ellas se crean el flujo de la aplicación.

```
            {% for client in pagination %}
                <tr {% if loop.index is odd %}class="color"{% endif %}>
                    <td>{{ client.clientName }}</td>
                    <td>{{ client.phone }}</td>
                    <td>{{ client.street }}</td>
                    <td>{{ client.location }}</td>
                    <td>{{ client.mail }}</td>
                    <td>{{ client.dob ? client.dob|date('d-m-Y') : '' }}</td>
                    <td>{{ client.antiquity ? client.antiquity|date('d-m-Y') : '' }}</td>
                    <td>{{ client.comments }}</td>
                    <td>
                        <a href="{{ path('client_show', {'id': client.id}) }}">Mostrar</a>
                        <a href="{{ path('client_edit', {'id': client.id}) }}">Editar</a>
                    </td>
                </tr>
            {% else %}
                <tr>
                    <td colspan="10">No se encontraron registros</td>
                </tr>
            {% endfor %}
            </tbody>
        </table>
        {# display navigation #}
        <div class="navigation">
            {{ knp_pagination_render(pagination) }}
        </div>
        <a href="{{ path('client_new') }}">Crear nuevo</a>
        <!-- /.content-header -->
    </div>
    <!-- /.content-wrapper -->

    {% include 'footer.html.twig' %}
</div>
<!-- ./wrapper -->
{% endblock %}

{% include 'javascripts.html.twig' %}
</body>
</html>
```

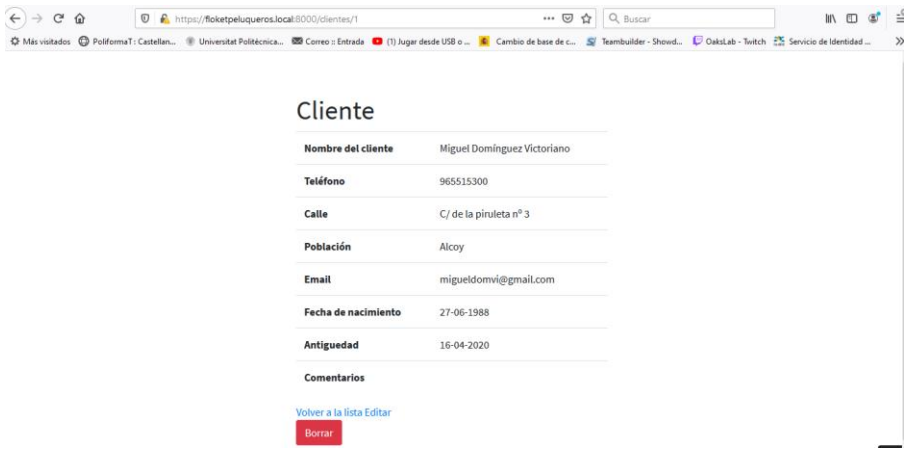


Al presionar sobre Mostrar, aparecerán los datos en concreto de ese cliente, para ello tenemos otro apartado en el controlador llamado **client_show**, este apartado lo que hace es pasar los datos únicamente de ese cliente mediante su identificador.

```php
/**
 * @Route("/{id}", name="client_show", methods={"GET"})
 */
public function show(Client $client): Response
{
    return $this->render( view: 'client/show.html.twig', [
        'client' => $client,
    ]);
}
```

El apartado del controlador **client_show** renderiza **client/show.html.twig.**

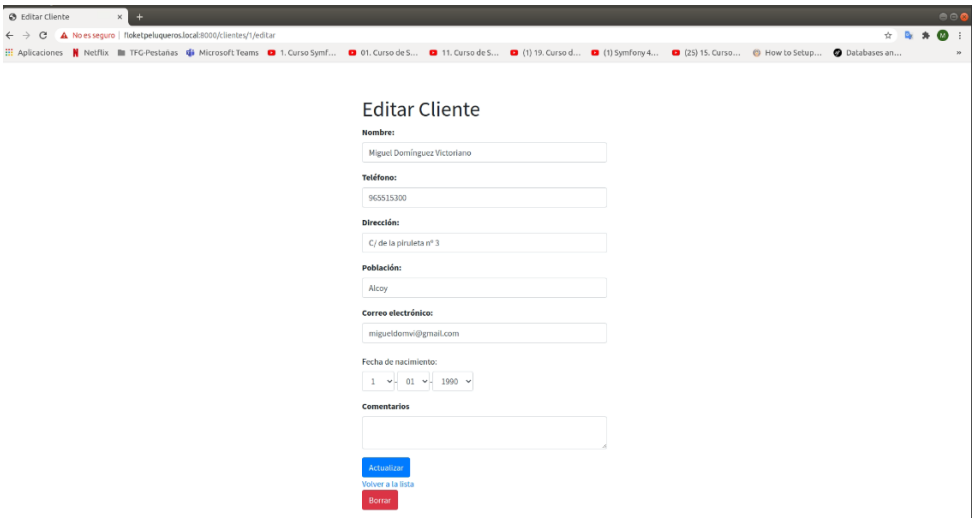
```twig
<table class="table">
    <tbody>
    <tr>
        <th>Nombre del cliente</th>
        <td>{{ client.clientName }}</td>
    </tr>
    <tr>
        <th>Teléfono</th>
        <td>{{ client.phone }}</td>
    </tr>
    <tr>
        <th>Calle</th>
        <td>{{ client.street }}</td>
    </tr>
    <tr>
        <th>Población</th>
        <td>{{ client.location }}</td>
    </tr>
    <tr>
        <th>Email</th>
        <td>{{ client.mail }}</td>
    </tr>
    <tr>
        <th>Fecha de nacimiento</th>
        <td>{{ client.dob ? client.dob|date('d-m-Y') : '' }}</td>
    </tr>
    <tr>
        <th>Antiguedad</th>
        <td>{{ client.antiquity ? client.antiquity|date('d-m-Y') : '' }}</td>
    </tr>
    <tr>
        <th>Comentarios</th>
        <td>{{ client.comments }}</td>
    </tr>
    </tbody>
</table>

<a href="{{ path('client_index') }}">Volver a la lista</a>

<a href="{{ path('client_edit', {'id': client.id}) }}">Editar</a>

{{ include('client/_delete_form.html.twig') }}
```



Desde esta pantalla podemos volver al índice, editar este cliente o borrarlo.

Al presionar sobre Editar, aparece el formulario para poder editar este cliente. El procedimiento es exactamente el mismo, el controlador ejecuta el apartado **client_edit** y renderiza **client/edit.html.twig**.

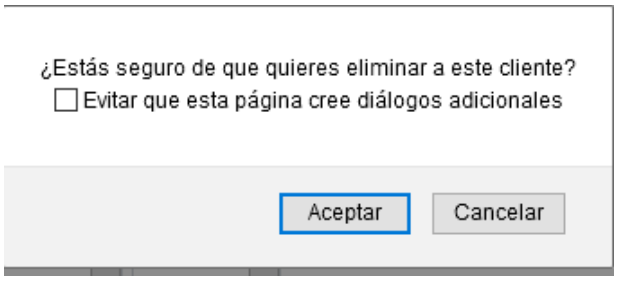
```
/**
 * @Route("/{id}/editar", name="client_edit", methods={"GET","POST"})
 */
public function edit(Request $request, Client $client): Response
{
    $form = $this->createForm( type: ClientType::class, $client);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'client_index');
    }

    return $this->render( view: 'client/edit.html.twig', [
        'client' => $client,
        'form' => $form->createView(),
    ]);
}
```



Al presionar sobre Borrar, aparece un diálogo para borrar ese cliente. El procedimiento es exactamente el mismo, el controlador ejecuta el apartado **client_delete** y borra el registro de ese cliente.

```php
/**
 * @Route("/{id}", name="client_delete", methods={"DELETE"})
 */
public function delete(Request $request, Client $client): Response
{
    if ($this->isCsrfTokenValid( id: 'delete'.$client->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($client);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'client_index');
}
```

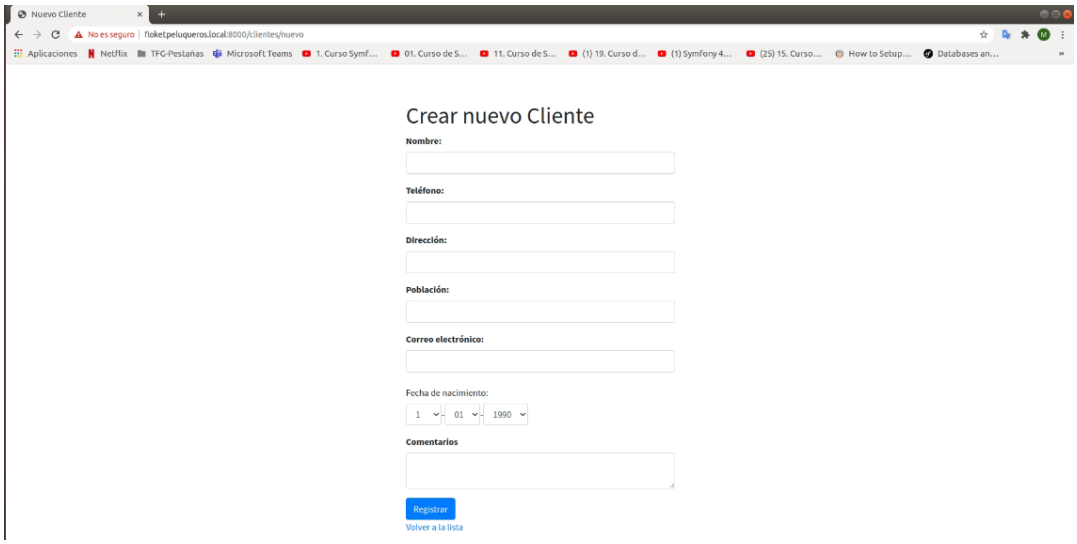
```twig
<form method="post" action="{{ path('client_delete', {'id': client.id}) }}" onsubmit="return confirm('¿Estás seguro de que quieres elim
    <input type="hidden" name="_method" value="DELETE">
    <input type="hidden" name="_token" value="{{ csrf_token('delete' ~ client.id) }}">
    <button class="btn btn-danger">Borrar</button>
</form>
```



Por último, tenemos la función de crear un nuevo cliente, para ello, se llama al apartado del controlador llamado **client_new** y renderiza **client/new.html.twig.**

```php
/**
 * @Route("/nuevo", name="client_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $client = new Client();
    $form = $this->createForm( type: ClientType::class, $client);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($client);
        $entityManager->flush();
        return $this->redirectToRoute( route: 'client_index');
    }

    return $this->render( view: 'client/new.html.twig', [
        'client' => $client,
        'form' => $form->createView(),
    ]);
}
```

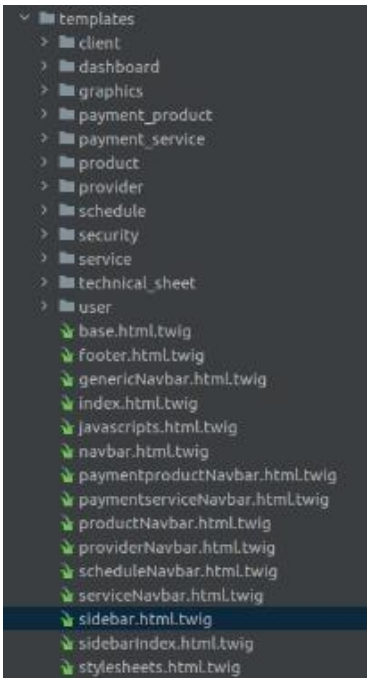
```twig
{% extends 'base.html.twig' %}

{% block title %}Nuevo Cliente{% endblock %}

{% block body %}
    {% for message in app.flashes('registrado') %}
        <div class="alert alert-success">
            {{ message }}
        </div>
    {% endfor %}
    <div class="container h-100">
        <div class="row h-100 justify-content-center align-items-center">
            <div class="col-6 rounded my-3 p-5">
                <h1>Crear nuevo Cliente</h1>

                {{ include('client/_form.html.twig') }}

                <a href="{{ path('client_index') }}">Volver a la lista</a>
            </div>
        </div>
    </div>
{% endblock %}
```



Una funcionalidad muy interesante de las plantillas **twig** es que desde una plantilla se puede llamar a un trozo de código ubicado en otra plantilla haciendo que el código sea reutilizable, de esta manera se puede tener en plantillas separadas las partes de la aplicación que son comunes, por ejemplo, la parte lateral donde aparece el menú, la parte de la cabecera o la parte del pie, …

Siguiendo este principio he creado todas las plantillas necesarias para separar los diferentes trozos de código e ir montando un puzle para crear la aplicación.

Como he dicho, todos los apartados de la aplicación siguen esta estructura para el control de los datos, aparte en diferentes apartados se han creado más plantillas para dotar de más funcionalidades a la aplicación, entre ellas se han creado las plantillas para mostrar por ejemplo el stock de los productos, la facturación tanto de servicios como de productos, las gráficas, … Dentro de los controladores he creado diferentes apartados para estos fines, incluso he creado nuevos controladores para separar un poco el control de cada apartado.



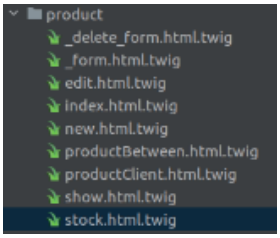
Ahora voy a explicar los apartados en los que he añadido más controladores y plantillas para crear todo el entramado de la aplicación.

Product (Productos)

En primer lugar, tenemos el apartado de productos, en este apartado se ha creado un submenú en la plantilla **productNavbar.html.twig** para acceder a falta de stock, productos demandados por un cliente y a los productos demandados entre fechas.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('product_stock') }}" class="nav-link">Falta de Stock</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('product_client') }}" class="nav-link">Productos demandados por cliente</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('product_between_dates') }}" class="nav-link">Productos demandados entre fechas</a>
    </li>
</ul>
```

El controlador lo expicaré en el siguiente punto. Se han creado las plantillas llamadas stock.html.twig, productBetween.html.twig y productClient.html.twig siguiendo el mismo proceso que los explicados anteriormente.

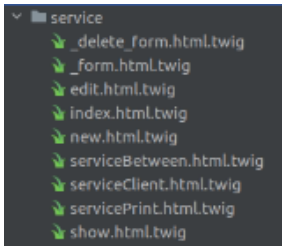


Service (Servicios)

A continuación, tenemos el apartado de servicios, en este apartado se ha creado un submenú en la plantilla **serviceNavbar.html.twig** para acceder a los servicios demandados por un cliente y a los servicios demandados entre fechas y a para imprimir los servicios.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_client') }}" class="nav-link">Servicios demandados por cliente</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_between_dates') }}" class="nav-link">Servicios demandados entre fechas</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_print') }}" class="nav-link">
            <i class="nav-icon fas fa-print"></i>
            Imprimir Servicios
        </a>
    </li>
</ul>
```

Se han creado las plantillas llamadas serviceBetween.html.twig, serviceClient.html.twig y servicePrint.html.twig

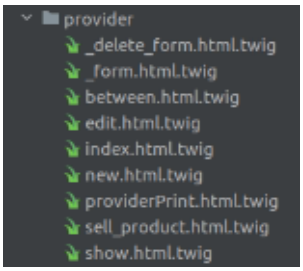


Provider (Proveedores)

También tenemos el apartado de proveedores, en este apartado se ha creado un submenú en la plantilla **providerNavbar.html.twig** para acceder a las compras que se han realizado a los proveedores, a las compras a proveedores entre fechas y para imprimir los datos de los proveedores.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('provider_sell') }}" class="nav-link">Compra a proveedores</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('provider_between') }}" class="nav-link">Compra a proveedores entre fechas</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('provider_print') }}" class="nav-link">
            <i class="nav-icon fas fa-print"></i>
            Imprimir Proveedores
        </a>
    </li>
</ul>
```

Se han creado las plantillas llamadas between.html.twig, sell_product.html.twig y providerPrint.html.twig

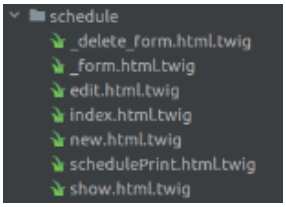


Schedule (Citas)

En el apartado de citas se ha creado un submenú en la plantilla **scheduleNavbar.html.twig** para imprimir las citas diarias.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('schedule_print') }}" class="nav-link">
            <i class="nav-icon fas fa-print"></i>
            Imprimir Citas
        </a>
    </li>
</ul>
```

Se ha creado la plantilla llamada schedulePrint.html.twig.

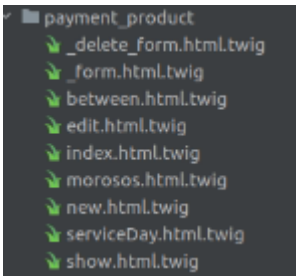


PaymentProduct (Cobro de productos)

En el apartado del pago de productos se ha creado un submenú en la plantilla **paymentproductoNavbar.html.twig** para acceder al apartado de morosos, para ver la facturación diaria y para ver la facturación entre fechas.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('payment_product_morosos') }}" class="nav-link">Morosos</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('product_billing') }}" class="nav-link">Facturación diaria</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('product_between') }}" class="nav-link">Facturación entre fechas</a>
    </li>
</ul>
```

Se han creado las plantillas llamadas between.html.twig, morosos.html.twig y serviceDay.html.twig.

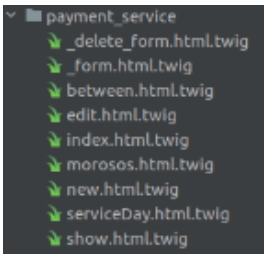


PaymentService (Cobro de servicios)

Por último, tenemos el apartado de los servicios, en este apartado se ha creado un submenú en la plantilla **paymentserviceNavbar.html.twig** para acceder al apartado de morosos, para ver la facturación diaria y para ver la facturación entre fechas.


```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" data-widget="pushmenu" href="#" role="button"><i class="fas fa-bars"></i></a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_client') }}" class="nav-link">Servicios demandados por cliente</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_between_dates') }}" class="nav-link">Servicios demandados entre fechas</a>
    </li>
    <li class="nav-item d-none d-sm-inline-block">
        <a href="{{ path('service_print') }}" class="nav-link">
            <i class="nav-icon fas fa-print"></i>
            Imprimir Servicios
        </a>
    </li>
</ul>
```

Se han creado las plantillas llamadas between.html.twig, morosos.html.twig y serviceDay.html.twig



3.2.3.- Capa Controlador

Un controlador es una función PHP que se encarga de obtener la información de la petición HTTP y de generar y devolver la respuesta HTTP (en forma de objeto de tipo `Response`). La respuesta puede ser una página HTML, un documento XML, un array JSON serializado, una imagen, una redirección a otra página, un error de tipo 404 o cualquier otra cosa que se nos ocurra. El controlador contiene toda la lógica que la aplicación necesita para generar el contenido de la página.

El objetivo de un controlador siempre es el mismo: crear y devolver un objeto `Response`. Para ello, a veces obtiene información de la petición, o busca un recurso en la base de datos, envía un correo electrónico, o guarda información en la sesión del usuario. Independientemente de lo que haga, el controlador siempre devuelve un objeto Response que se utiliza para generar la respuesta que se envía al usuario.

Para hacer un poco más fácil la generación de código de la aplicación, se ha usado un comando que crea automáticamente un esqueleto de cada controlador CRUD, es decir se crea el controlador **ClientController** por ejemplo, junto a las funciones para poder Crear, Leer, Actualizar y Borrar los objetos Client, también se crean las diferentes vistas de la aplicación y los formularios asociados. Evidentemente el código creado se tiene que adaptar a las necesidades de nuestra aplicación, pero ya tenemos un esqueleto funcional con el que empezar a desarrollar nuestra aplicación.

```
php bin/console make:crud
```

Mediante este comando nos ahorramos muchas horas de trabajo ya que si tuviésemos que crear a mano 10 controladores con sus respectivas vistas y su formulario tardaríamos unas cuantas horas.

```php
/**
 * @Route("/clientes")
 */
class ClientController extends AbstractController
{
    /**
     * @Route("/", name="client_index", methods={"GET"})
     */
    public function index(PaginatorInterface $paginator, Request $request): Response
    {
        $clientname = $request->get( key: 'query');

        if (!empty($clientname)) {
            $em = $this->getDoctrine()->getManager();
            $query = $em->getRepository( className: Client::class)->findClientByName($clientname);
        } else {
            $em = $this->getDoctrine()->getManager();
            $query = $em->getRepository( className: Client::class)->findAllCli();
        }

        $pagination = $paginator->paginate(
            $query, /* query NOT result */
            $request->query->getInt( key: 'page', default: 1), /*page number*/
             limit: 3 /*limit per page*/
        );

        return $this->render( view: 'client/index.html.twig', [
            'pagination' => $pagination,
        ]);
    }
```

A modo de ejemplo, podemos observar la función index del ClientController, en primer lugar, podemos observar que se crea la **ruta** clientes y de ella cuelgan todas sus subrutas.

El **nombre** es muy importante para llamar a esta función dentro del código, por ejemplo, twig nos permite dentro del código html poder hacer referencia a esta función.

Se hacen diferentes llamadas a la base de datos, en este caso si por el buscador se ha introducido algún nombre de un cliente se hace esa llamada a la base de datos, si el buscador no tiene ninguna búsqueda, se muestran los datos de todos los clientes. Aquí podemos decir que esta es la lógica de esta función.

Posteriormente esa consulta se pasa al paginador para que pagine los datos en caso de que haya más de 3 filas en la consulta.

Esta función hace que se renderice el archivo alojado en client/index.html.twig y le pasa como parámetros los datos de la consulta realizada para posteriormente poder en la Vista esos datos.

Por tanto, el **Controlador** se encarga de recibir las peticiones que le llegan desde la **Vista** mediante un formulario, por ejemplo, procesarlas, hacer las consultas a la base de datos mediante los **Repositorios**, obtener y procesar la información de la base datos y devolverle la respuesta a la **Vista** donde ésta pintará por pantalla los datos obtenidos.

En el caso de esta aplicación, hay funciones más complejas, pero en general no tienen demasiada complejidad.

En cuanto a las rutas, tenemos el **dominio floket.local:8000**, de este dominio cuelgan las diferentes rutas de la aplicación, que son:

Name	Method	Scheme	Host	Path
_preview_error	ANY	ANY	ANY	/_error/{code}.{_format}
_wdt	ANY	ANY	ANY	/_wdt/{token}
_profiler_home	ANY	ANY	ANY	/_profiler/
_profiler_search	ANY	ANY	ANY	/_profiler/search
_profiler_search_bar	ANY	ANY	ANY	/_profiler/search_bar
_profiler_phpinfo	ANY	ANY	ANY	/_profiler/phpinfo
_profiler_search_results	ANY	ANY	ANY	/_profiler/{token}/search/results
_profiler_open_file	ANY	ANY	ANY	/_profiler/open
_profiler	ANY	ANY	ANY	/_profiler/{token}
_profiler_router	ANY	ANY	ANY	/_profiler/{token}/router
_profiler_exception	ANY	ANY	ANY	/_profiler/{token}/exception
_profiler_exception_css	ANY	ANY	ANY	/_profiler/{token}/exception.css
service_billing	GET|POST	ANY	ANY	/cobro_servicios/facturacion_diaria
service_between	GET|POST	ANY	ANY	/cobro_servicios/facturacion_entre_fechas
product_billing	GET|POST	ANY	ANY	/cobro_productos/facturacion_diaria
product_between	GET|POST	ANY	ANY	/cobro_productos/facturacion_entre_fechas
provider_sell	GET|POST	ANY	ANY	/proveedores/compras_proveedor
provider_between	GET|POST	ANY	ANY	/proveedores/compras_entre_fechas
client_index	GET	ANY	ANY	/clientes/
client_new	GET|POST	ANY	ANY	/clientes/nuevo
client_show	GET	ANY	ANY	/clientes/{id}
client_edit	GET|POST	ANY	ANY	/clientes/{id}/editar
client_delete	DELETE	ANY	ANY	/clientes/{id}
index	ANY	ANY	ANY	/
dashboard	ANY	ANY	ANY	/dashboard
service_client	GET|POST	ANY	ANY	/servicios/demandas_del_cliente
service_between_dates	GET|POST	ANY	ANY	/servicios/demanda_entre_fechas
product_client	GET|POST	ANY	ANY	/productos/demandas_del_cliente
product_between_dates	GET|POST	ANY	ANY	/productos/demanda_entre_fechas
graficas	GET|POST	ANY	ANY	/graficas
payment_product_index	GET	ANY	ANY	/cobro_productos/
payment_product_morosos	GET|POST	ANY	ANY	/cobro_productos/morosos
payment_product_new	GET|POST	ANY	ANY	/cobro_productos/nuevo
payment_product_show	GET	ANY	ANY	/cobro_productos/{id}
payment_product_edit	GET|POST	ANY	ANY	/cobro_productos/{id}/editar
payment_product_delete	DELETE	ANY	ANY	/cobro_productos/{id}

```
payment_service_morosos     GET|POST   ANY      ANY    /cobro_servicios/morosos
payment_service_new         GET|POST   ANY      ANY    /cobro_servicios/nuevo
payment_service_show        GET        ANY      ANY    /cobro_servicios/{id}
payment_service_edit        GET|POST   ANY      ANY    /cobro_servicios/{id}/editar
payment_service_delete      DELETE     ANY      ANY    /cobro_servicios/{id}
service_print               GET        ANY      ANY    /servicios/service_print
provider_print              GET        ANY      ANY    /proveedores/provider_print
schedule_print              GET        ANY      ANY    /citas/citas_print
graficas_servicios_print    GET        ANY      ANY    /graficas/servicios_print
graficas_productos_print    GET        ANY      ANY    /graficas/productos_print
product_index               GET        ANY      ANY    /productos/
product_stock               GET        ANY      ANY    /productos/stock
product_new                 GET|POST   ANY      ANY    /productos/nuevo
product_show                GET        ANY      ANY    /productos/{id}
product_edit                GET|POST   ANY      ANY    /productos/{id}/editar
product_delete              DELETE     ANY      ANY    /productos/{id}
provider_index              GET        ANY      ANY    /proveedores/
provider_new                GET|POST   ANY      ANY    /proveedores/nuevo
provider_show               GET        ANY      ANY    /proveedores/{id}
provider_edit               GET|POST   ANY      ANY    /proveedores/{id}/editar
provider_delete             DELETE     ANY      ANY    /proveedores/{id}
schedule_index              GET|POST   ANY      ANY    /citas/
schedule_new                GET|POST   ANY      ANY    /citas/nuevo
schedule_show               GET        ANY      ANY    /citas/{id}
schedule_edit               GET|POST   ANY      ANY    /citas/{id}/editar
schedule_delete             DELETE     ANY      ANY    /citas/{id}
app_login                   ANY        ANY      ANY    /login
app_logout                  ANY        ANY      ANY    /logout
service_index               GET        ANY      ANY    /servicios/
service_new                 GET|POST   ANY      ANY    /servicios/nuevo
service_show                GET        ANY      ANY    /servicios/{id}
service_edit                GET|POST   ANY      ANY    /servicios/{id}/editar
service_delete              DELETE     ANY      ANY    /servicios/{id}
technical_sheet_index       GET|POST   ANY      ANY    /ficha_tecnica/
technical_sheet_paginated   GET|POST   ANY      ANY    /ficha_tecnica/paginar/{id}
technical_sheet_new         GET|POST   ANY      ANY    /ficha_tecnica/nuevo
technical_sheet_show        GET        ANY      ANY    /ficha_tecnica/{id}
technical_sheet_edit        GET|POST   ANY      ANY    /ficha_tecnica/{id}/editar
technical_sheet_delete      DELETE     ANY      ANY    /ficha_tecnica/{id}
```

```
user_index                  GET        ANY      ANY    /trabajadores/
user_new                    GET|POST   ANY      ANY    /trabajadores/nuevo
user_show                   GET        ANY      ANY    /trabajadores/{id}
user_edit                   GET|POST   ANY      ANY    /trabajadores/{id}/editar
user_delete                 DELETE     ANY      ANY    /trabajadores/{id}
--------------------------- ---------- -------- ------ ----------------------------------------
```

Para realizar el control de acceso a la aplicación, tenemos el archivo **security.yaml** ubicado en **config/packages**, en este archivo podemos restringir el acceso a las diferentes páginas de la aplicación según el rol del trabajador.

```yaml
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    - { path: ^/dashboard, roles: [ROLE_ADMIN, ROLE_USER] }
    - { path: ^/clientes, roles: [ROLE_ADMIN, ROLE_USER] }
    - { path: ^/trabajadores, roles: ROLE_ADMIN }
    - { path: ^/productos, roles: ROLE_ADMIN }
    - { path: ^/proveedores, roles: ROLE_ADMIN }
    - { path: ^/servicios, roles: ROLE_ADMIN }
    - { path: ^/horario, roles: [ROLE_ADMIN, ROLE_USER] }
    - { path: ^/ficha_tecnica, roles: [ROLE_ADMIN, ROLE_USER] }
    - { path: ^/pago_productos, roles: ROLE_ADMIN }
    - { path: ^/pago_servicios, roles: ROLE_ADMIN }
    - { path: ^/graficas, roles: ROLE_ADMIN }
```

Como se puede observar, el trabajador con rol de administrador puede acceder a todas las rutas en cambio el trabajador con el rol de usuario solamente tiene acceso a unas pocas rutas de la aplicación.

He creado un total de 15 controladores, cada uno de ellos se encarga de controlar una parte de la lógica de la aplicación siendo este apartado el corazón o la parte más importante del desarrollo realizado.

Cada uno de estos controladores contiene funciones que son las que realmente se encargan de controlar cada apartado de la aplicación. Voy a ir explicando uno a uno todos los controladores creados y sus funciones.

BillingController

- serviceDay
- serviceBetween
- productDay
- productBetween
- providerSellProduct
- providerSellBetween

ClientController

- index
- new
- show
- edit
- delete

DashboardController

- index
- Dashboard

DemandController

- serviceClient
- serviceBetween
- productClient
- productBetween

GraphicsController

- graficas

PaymentProductController

- index
- morosos

- new
- show
- edit
- delete

PaymentServiceController

- index
- morosos
- new
- show
- edit
- delete

PrintController

- printServices
- printProviders
- printSchedule
- printComparativeServices
- printComparativeProducts

ProductController

- index
- stock
- new
- show
- edit
- delete

ProviderController

- index
- new
- show
- edit
- delete

ScheduleController

- index
- stock
- new
- show

- edit
- delete

SecurityController

- login
- logout

ServiceController

- index
- new
- show
- edit
- delete

TechnicalSheetController

- index
- paginated
- new
- show
- edit
- delete

UserController

- index
- new
- show
- edit
- delete

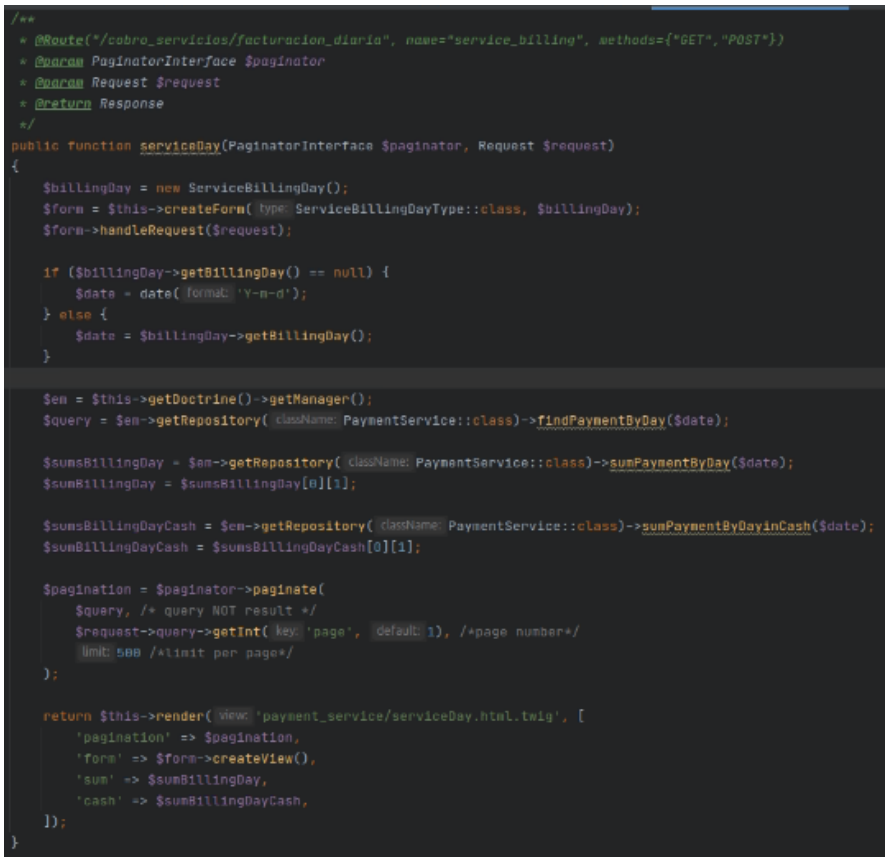
Cada uno de los controladores contiene funciones que son muy similares o prácticamente iguales, por eso únicamente voy a explicar esas funciones una única vez.

BillingController

Este controlador se encarga de la facturación de los servicios, de los productos y de los proveedores. Las funciones creadas en este controlador en el caso de los servicios y de los productos es exactamente igual, la diferencia es que cada uno de ellos apunta a su respectivo repositorio para realizar las consultas.

serviceDay o productDay

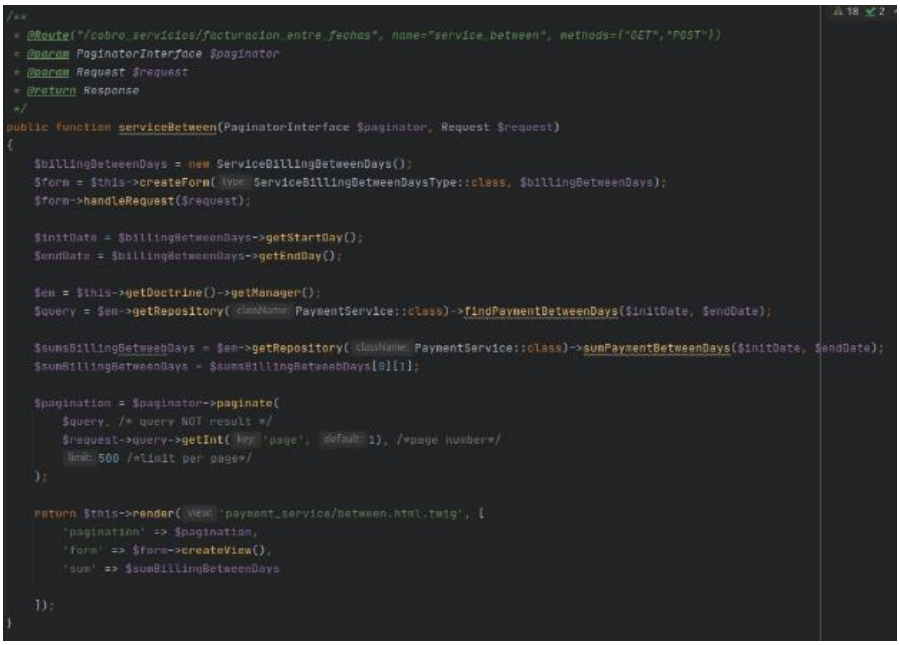
Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación realiza tres consultas a los repositorios de la base de datos para mostrar la facturación diaria, por defecto muestra la facturación del día actual si no se ha seleccionado ninguna fecha, por otro lado, muestra la suma de los beneficios totales del día seleccionado y la suma en efectivo de los beneficios del día seleccionado; como he dicho anteriormente, los datos generados se pasan a la plantilla twig correspondiente; esta función es exactamente igual para obtener la facturación diaria tanto de los servicios como de los productos por tanto, solo he explicado esta función pero es exactamente igual en el caso del cobro de productos.



```
/**
 * @Route("/cobro_servicios/facturacion_diaria", name="service_billing", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function serviceDay(PaginatorInterface $paginator, Request $request)
{
    $billingDay = new ServiceBillingDay();
    $form = $this->createForm( type: ServiceBillingDayType::class, $billingDay);
    $form->handleRequest($request);

    if ($billingDay->getBillingDay() == null) {
        $date = date( format: 'Y-m-d');
    } else {
        $date = $billingDay->getBillingDay();
    }

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: PaymentService::class)->findPaymentByDay($date);

    $sumsBillingDay = $em->getRepository( className: PaymentService::class)->sumPaymentByDay($date);
    $sumBillingDay = $sumsBillingDay[0][1];

    $sumsBillingDayCash = $em->getRepository( className: PaymentService::class)->sumPaymentByDayinCash($date);
    $sumBillingDayCash = $sumsBillingDayCash[0][1];

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 500 /*limit per page*/
    );

    return $this->render( view: 'payment_service/serviceDay.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'sum' => $sumBillingDay,
        'cash' => $sumBillingDayCash,
    ]);
}
```

serviceBetween o productBetween

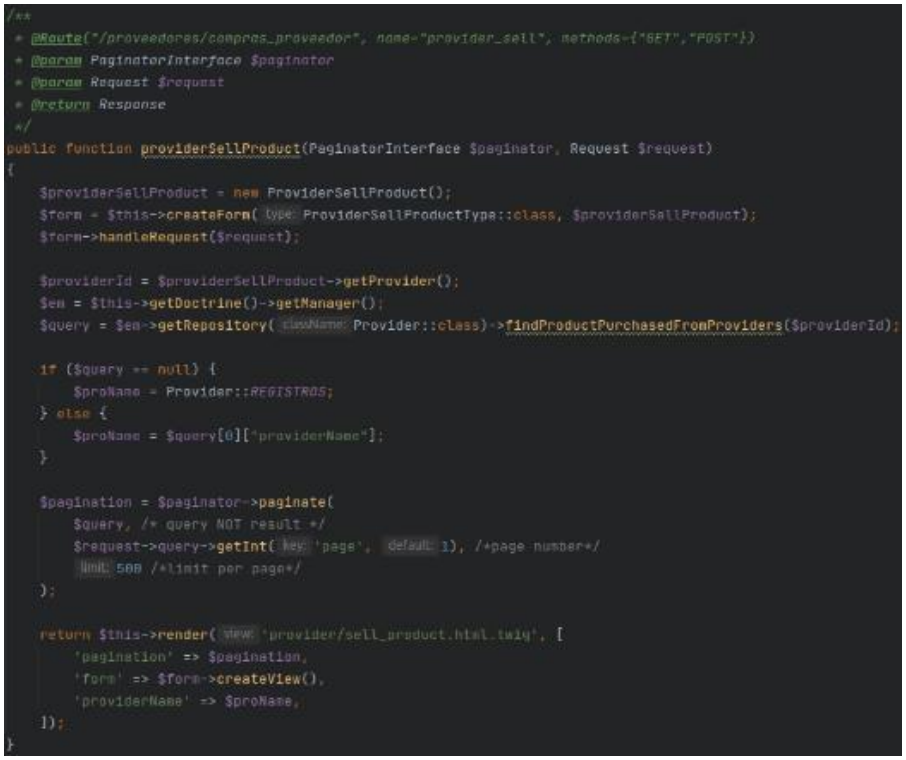
Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación realiza dos consultas a los repositorios de la base de datos para mostrar la facturación entre las fechas seleccionadas, por otro lado, muestra la suma de los beneficios totales del rango de fechas seleccionado; como he dicho anteriormente, los datos generados se pasan a la plantilla twig correspondiente; esta función es exactamente igual para obtener la facturación entre fechas tanto de los servicios como de los productos por tanto, solo he explicado esta función pero es exactamente igual en el caso del cobro de productos.



```php
/**
 * @Route("/cobro_servicios/facturacion_entre_fechas", name="service_between", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function serviceBetween(PaginatorInterface $paginator, Request $request)
{
    $billingBetweenDays = new ServiceBillingBetweenDays();
    $form = $this->createForm( type: ServiceBillingBetweenDaysType::class, $billingBetweenDays);
    $form->handleRequest($request);

    $initDate = $billingBetweenDays->getStartDay();
    $endDate = $billingBetweenDays->getEndDay();

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: PaymentService::class)->findPaymentBetweenDays($initDate, $endDate);

    $sumsBillingBetweenDays = $em->getRepository( className: PaymentService::class)->sumPaymentBetweenDays($initDate, $endDate);
    $sumBillingBetweenDays = $sumsBillingBetweenDays[0][1];

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 500 /*limit per page*/
    );

    return $this->render( view: 'payment_service/between.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'sum' => $sumBillingBetweenDays

    ]);
}
```

providerSellProduct

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación realiza la consulta a los repositorios de la base de datos para mostrar las compras al proveedor que se haya seleccionado; como he dicho anteriormente, los datos generados se pasan a la plantilla twig correspondiente.



```php
/**
 * @Route("/proveedores/compras_proveedor", name="provider_sell", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function providerSellProduct(PaginatorInterface $paginator, Request $request)
{
    $providerSellProduct = new ProviderSellProduct();
    $form = $this->createForm( type: ProviderSellProductType::class, $providerSellProduct);
    $form->handleRequest($request);

    $providerId = $providerSellProduct->getProvider();
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Provider::class)->findProductPurchasedFromProviders($providerId);

    if ($query == null) {
        $proName = Provider::REGISTROS;
    } else {
        $proName = $query[0]["providerName"];
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 500 /*limit per page*/
    );

    return $this->render( view: 'provider/sell_product.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'providerName' => $proName,
    ]);
}
```

providerSellBetween

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación realiza una consulta a los repositorios de la base de datos para mostrar la facturación entre las fechas seleccionadas, como he dicho anteriormente, los datos generados se pasan a la plantilla twig correspondiente.

```php
/**
 * @Route("/proveedores/compras_entre_fechas", name="provider_between", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function providerSellBetween(PaginatorInterface $paginator, Request $request)
{
    $providerSellBetween = new ProviderSellBetweenDates();
    $form = $this->createForm( type: ProviderSellBetweenDatesType::class, $providerSellBetween);
    $form->handleRequest($request);

    $initDate = $providerSellBetween->getStartDay();
    $endDate = $providerSellBetween->getEndDay();

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Provider::class)->findProductPurchasedBetweenDates($initDate, $endDate);

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
        limit: 500 /*limit per page*/
    );

    return $this->render( view: 'provider/between.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
    ]);
}
```

ClientController

Este controlador se encarga de la creación, muestra y manipulación de los clientes, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, según si buscamos a un cliente o no hacemos ninguna búsqueda, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a los clientes buscados o a todos los cliente de la aplicación, en este caso los datos obtenidos se van a mostrar con un límite de tres líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/", name="client_index", methods={"GET"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $clientname = $request->get( key: 'query');

    if (!empty($clientname)) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Client::class)->findClientByName($clientname);
    } else {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Client::class)->findAllCli();
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );

    return $this->render( view: 'client/index.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```php
/**
 * @Route("/nuevo", name="client_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $client = new Client();
    $form = $this->createForm( type: ClientType::class, $client);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($client);
        $entityManager->flush();
        return $this->redirectToRoute( route: 'client_index');
    }

    return $this->render( view: 'client/new.html.twig', [
        'client' => $client,
        'form' => $form->createView(),
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'clientName', type: TextType::class, [
            'label' => 'Nombre: '
        ])
        ->add( child: 'phone', type: NumberType::class, [
            'label' => 'Teléfono: '
        ])
        ->add( child: 'street', type: TextType::class, [
            'label' => 'Dirección: '
        ])
        ->add( child: 'location', type: TextType::class, [
            'label' => 'Población: '
        ])
        ->add( child: 'mail', type: EmailType::class, [
            'label' => 'Correo electrónico: '
        ])
        ->add( child: 'dob', type: BirthdayType::class, [
            'label' => 'Fecha de nacimiento: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( dateTime: "01-01-1990")
        ])
        ->add( child: 'comments', type: TextareaType::class,
            [
                'label' => 'Comentarios',
                'required' => false,
            ])
    ;
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este cliente y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="client_show", methods={"GET"})
 */
public function show(Client $client): Response
{
    return $this->render( view: 'client/show.html.twig', [
        'client' => $client,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este cliente y los muestra en el formulario de edición para modificar los datos de este cliente, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```php
/**
 * @Route("/{id}/editar", name="client_edit", methods={"GET","POST"})
 */
public function edit(Request $request, Client $client): Response
{
    $form = $this->createForm( type: ClientType::class, $client);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'client_index');
    }

    return $this->render( view: 'client/edit.html.twig', [
        'client' => $client,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del cliente que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

```php
/**
 * @Route("/{id}", name="client_delete", methods={"DELETE"})
 */
public function delete(Request $request, Client $client): Response
{
    if ($this->isCsrfTokenValid( id: 'delete'.$client->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($client);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'client_index');
}
```

DashboardController

Este controlador se encarga de mostrar la página principal de la aplicación y de la página del menú una vez validado.

Como se puede observar, este controlador únicamente renderiza la plantilla twig del índice principal de la aplicación y de la página del menú una vez se ha validado con usuario y contraseña.

DemandController

Este controlador se encarga de saber qué productos y servicios han sido demandados por un cliente y de qué productos y servicios han sido demandados entre un rango de fechas.

Las funciones creadas en este controlador en el caso de los servicios y de los productos es exactamente igual, la diferencia es que cada uno de ellos apunta a su respectivo repositorio para realizar las consultas.

serviceClient o productClient

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, mediante un desplegable se obtiene el nombre del cliente a buscar, una vez seleccionado el cliente, aparecen los datos de los productos comprados o los servicios demandados junto al precio gastado en cada producto o servicio, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/servicios/demandas_del_cliente", name="service_client", methods={"GET","POST"})
 * @param Request $request
 * @return Response
 */
public function serviceClient(PaginatorInterface $paginator, Request $request): Response
{
    $serviceClient = new ServiceDemandClient();
    $form = $this->createForm( type: ServiceDemandClientType::class, $serviceClient);
    $form->handleRequest($request);

    $clientId = $serviceClient->getClient();
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Service::class)->findDemandForServicesByAClient($clientId);

    if ($query == null) {
        $cliName = Service::REGISTROS;
    } else {
        $cliName = $query[0]["clientName"];
    }
    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
        limit: 1000 /*limit per page*/
    );

    return $this->render( view: 'service/serviceClient.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'clientName' => $cliName,
    ]);
}
```

serviceBetween o productBetween

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, mediante dos desplegables se selecciona el rango de fechas, una vez seleccionado el rango, aparecen los datos de los productos comprados o los servicios demandados junto al precio gastado en cada producto o servicio en ese rango de fechas, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrarlos.

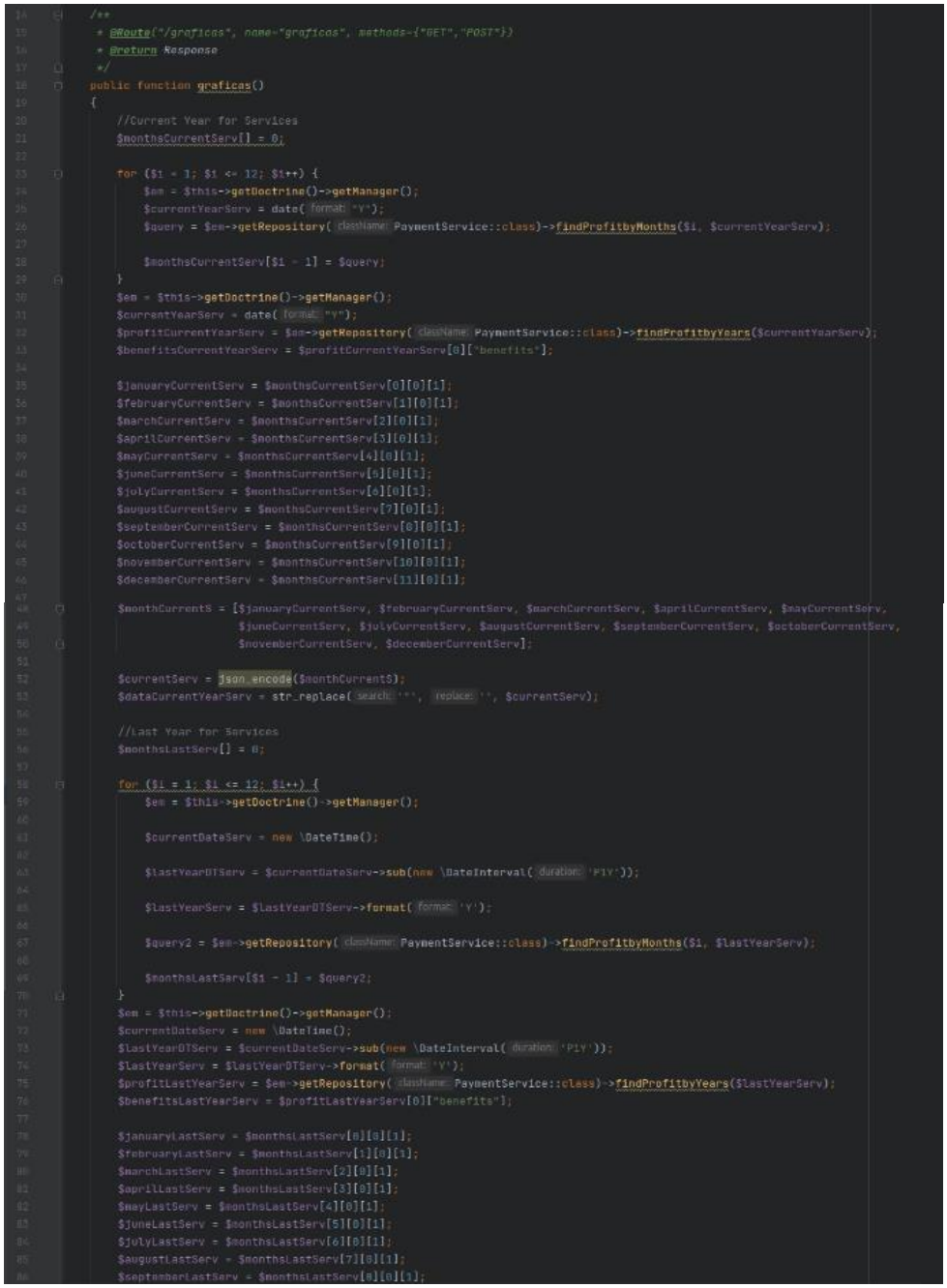
```php
/**
 * @Route("/productos/demandas_del_cliente", name="product_client", methods={"GET","POST"})
 * @param Request $request
 * @return Response
 */
public function productClient(PaginatorInterface $paginator, Request $request): Response
{
    $productClient = new ProductDemandClient();
    $form = $this->createForm( type: ProductDemandClientType::class, $productClient);
    $form->handleRequest($request);

    $clientId = $productClient->getClient();
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Product::class)->findDemandForProductsByAClient($clientId);

    if ($query == null) {
        $cliName = Product::REGISTROS;
    } else {
        $cliName = $query[0]["clientName"];
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
        limit: 1000 /*limit per page*/
    );

    return $this->render( view: 'product/productClient.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'clientName' => $cliName,
    ]);
}
```

GraphicsController

Este controlador se encarga de obtener todos los datos de las ventas de los productos y servicios del año actual y del año anterior desglosados mes a mes y de los productos y servicios más demandados por los clientes para poder realizar unas gráficas donde poder observar de forma visual la evolución de las ventas en el salón de belleza.

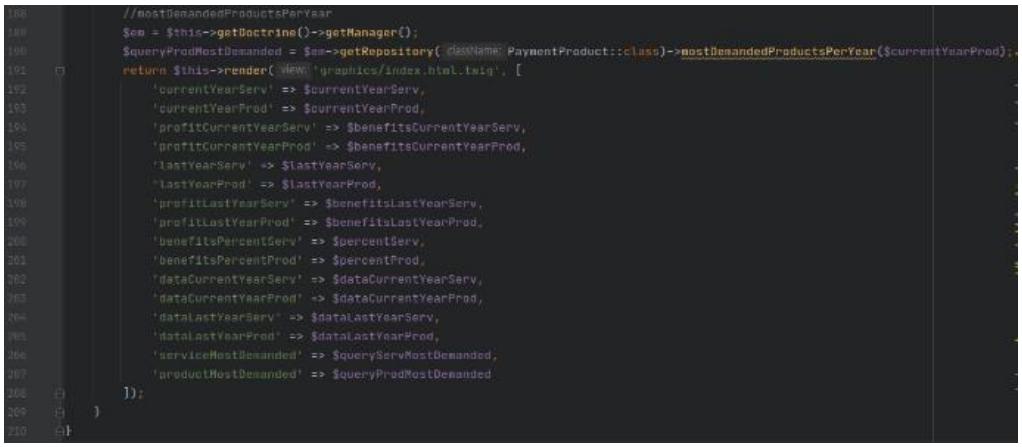
graficas

En este controlador tenemos una única función que se encarga de realizar las consultas a la base de datos para obtener los beneficios de los servicios y productos de cada mes del año actual y del anterior, también se obtienen los 5 servicios y productos más demandados en el año actual para realizar estadísticas.



```
/**
 * @Route("/graficas", name="graficas", methods={"GET","POST"})
 * @return Response
 */
public function graficas()
{
    //Current Year for Services
    $monthsCurrentServ[] = 0;

    for ($i = 1; $i <= 12; $i++) {
        $em = $this->getDoctrine()->getManager();
        $currentYearServ = date( format: "Y");
        $query = $em->getRepository( className: PaymentService::class)->findProfitbyMonths($i, $currentYearServ);

        $monthsCurrentServ[$i - 1] = $query;
    }
    $em = $this->getDoctrine()->getManager();
    $currentYearServ = date( format: "Y");
    $profitCurrentYearServ = $em->getRepository( className: PaymentService::class)->findProfitbyYears($currentYearServ);
    $benefitsCurrentYearServ = $profitCurrentYearServ[0]["benefits"];

    $januaryCurrentServ = $monthsCurrentServ[0][0][1];
    $februaryCurrentServ = $monthsCurrentServ[1][0][1];
    $marchCurrentServ = $monthsCurrentServ[2][0][1];
    $aprilCurrentServ = $monthsCurrentServ[3][0][1];
    $mayCurrentServ = $monthsCurrentServ[4][0][1];
    $juneCurrentServ = $monthsCurrentServ[5][0][1];
    $julyCurrentServ = $monthsCurrentServ[6][0][1];
    $augustCurrentServ = $monthsCurrentServ[7][0][1];
    $septemberCurrentServ = $monthsCurrentServ[8][0][1];
    $octoberCurrentServ = $monthsCurrentServ[9][0][1];
    $novemberCurrentServ = $monthsCurrentServ[10][0][1];
    $decemberCurrentServ = $monthsCurrentServ[11][0][1];

    $monthCurrentS = [$januaryCurrentServ, $februaryCurrentServ, $marchCurrentServ, $aprilCurrentServ, $mayCurrentServ,
                      $juneCurrentServ, $julyCurrentServ, $augustCurrentServ, $septemberCurrentServ, $octoberCurrentServ,
                      $novemberCurrentServ, $decemberCurrentServ];

    $currentServ = json_encode($monthCurrentS);
    $dataCurrentYearServ = str_replace( search: '"', replace: '', $currentServ);

    //Last Year for Services
    $monthsLastServ[] = 0;

    for ($i = 1; $i <= 12; $i++) {
        $em = $this->getDoctrine()->getManager();

        $currentDateServ = new \DateTime();

        $lastYearDTServ = $currentDateServ->sub(new \DateInterval( duration: 'P1Y'));

        $lastYearServ = $lastYearDTServ->format( format: 'Y');

        $query2 = $em->getRepository( className: PaymentService::class)->findProfitbyMonths($i, $lastYearServ);

        $monthsLastServ[$i - 1] = $query2;
    }
    $em = $this->getDoctrine()->getManager();
    $currentDateServ = new \DateTime();
    $lastYearDTServ = $currentDateServ->sub(new \DateInterval( duration: 'P1Y'));
    $lastYearServ = $lastYearDTServ->format( format: 'Y');
    $profitLastYearServ = $em->getRepository( className: PaymentService::class)->findProfitbyYears($lastYearServ);
    $benefitsLastYearServ = $profitLastYearServ[0]["benefits"];

    $januaryLastServ = $monthsLastServ[0][0][1];
    $februaryLastServ = $monthsLastServ[1][0][1];
    $marchLastServ = $monthsLastServ[2][0][1];
    $aprilLastServ = $monthsLastServ[3][0][1];
    $mayLastServ = $monthsLastServ[4][0][1];
    $juneLastServ = $monthsLastServ[5][0][1];
    $julyLastServ = $monthsLastServ[6][0][1];
    $augustLastServ = $monthsLastServ[7][0][1];
    $septemberLastServ = $monthsLastServ[8][0][1];
```

```
        $octoberLastServ = $monthsLastServ[9][0][1];
        $novemberLastServ = $monthsLastServ[10][0][1];
        $decemberLastServ = $monthsLastServ[11][0][1];

        $monthLast = [$januaryLastServ, $februaryLastServ, $marchLastServ, $aprilLastServ, $mayLastServ, $juneLastServ,
            $julyLastServ, $augustLastServ, $septemberLastServ, $octoberLastServ, $novemberLastServ, $decemberLastServ];

        $LastServ = json_encode($monthLast);
        $dataLastYearServ = str_replace( search: '"', replace: '', $LastServ);

        //Percent of Benefits
        $benefitsServ = $benefitsCurrentYearServ - $benefitsLastYearServ;
        $benefitsPercentServ = ($benefitsServ / $benefitsLastYearServ) * 100;
        $percentServ = round($benefitsPercentServ, precision: 2);

        //Current Year for Products
        $monthsCurrentProd[] = 0;

        for ($i = 1; $i <= 12; $i++) {
            $em = $this->getDoctrine()->getManager();
            $currentYearProd = date( format: "Y");
            $queryProd1 = $em->getRepository( className: PaymentProduct::class)->findProfitbyMonths($i, $currentYearProd);

            $monthsCurrentProd[$i - 1] = $queryProd1;
        }
        $em = $this->getDoctrine()->getManager();
        $currentYearProd = date( format: "Y");
        $profitCurrentYearProd = $em->getRepository( className: PaymentProduct::class)->findProfitbyYears($currentYearProd);
        $benefitsCurrentYearProd = $profitCurrentYearProd[0]["benefits"];

        $januaryCurrentProd = $monthsCurrentProd[0][0][1];
        $februaryCurrentProd = $monthsCurrentProd[1][0][1];
        $marchCurrentProd = $monthsCurrentProd[2][0][1];
        $aprilCurrentProd = $monthsCurrentProd[3][0][1];
        $mayCurrentProd = $monthsCurrentProd[4][0][1];
        $juneCurrentProd = $monthsCurrentProd[5][0][1];
        $julyCurrentProd = $monthsCurrentProd[6][0][1];
        $augustCurrentProd = $monthsCurrentProd[7][0][1];
        $septemberCurrentProd = $monthsCurrentProd[8][0][1];
        $octoberCurrentProd = $monthsCurrentProd[9][0][1];
        $novemberCurrentProd = $monthsCurrentProd[10][0][1];
        $decemberCurrentProd = $monthsCurrentProd[11][0][1];

        $monthCurrentP = [$januaryCurrentProd, $februaryCurrentProd, $marchCurrentProd, $aprilCurrentProd, $mayCurrentProd,
            $juneCurrentProd, $julyCurrentProd, $augustCurrentProd, $septemberCurrentProd, $octoberCurrentProd,
            $novemberCurrentProd, $decemberCurrentProd];

        $currentProd = json_encode($monthCurrentP);
        $dataCurrentYearProd = str_replace( search: '"', replace: '', $currentProd);

        //Last Year for Products
        $monthsLastProd[] = 0;

        for ($i = 1; $i <= 12; $i++) {
            $em = $this->getDoctrine()->getManager();

            $currentDateProd = new \DateTime();

            $LastYearDTProd = $currentDateProd->sub(new \DateInterval( duration: 'P1Y'));

            $lastYearDTProd = $LastYearDTProd->format( format: 'Y');

            $queryProd2 = $em->getRepository( className: PaymentProduct::class)->findProfitbyMonths($i, $lastYearDTProd);

            $monthsLastProd[$i - 1] = $queryProd2;
        }
        $em = $this->getDoctrine()->getManager();
        $currentDateProd = new \DateTime();
        $LastYearDTProd = $currentDateProd->sub(new \DateInterval( duration: 'P1Y'));
        $lastYearProd = $LastYearDTProd->format( format: 'Y');
        $profitLastYearProd = $em->getRepository( className: PaymentProduct::class)->findProfitbyYears($lastYearProd);
        $benefitsLastYearProd = $profitLastYearProd[0]["benefits"];
        $januaryLastProd = $monthsLastProd[0][0][1];
        $februaryLastProd= $monthsLastProd[1][0][1];
        $marchLastProd = $monthsLastProd[2][0][1];
        $aprilLastProd = $monthsLastProd[3][0][1];
        $mayLastProd = $monthsLastProd[4][0][1];
        $juneLastProd = $monthsLastProd[5][0][1];
        $julyLastProd = $monthsLastProd[6][0][1];
        $augustLastProd = $monthsLastProd[7][0][1];
        $septemberLastProd = $monthsLastProd[8][0][1];
        $octoberLastProd = $monthsLastProd[9][0][1];
        $novemberLastProd = $monthsLastProd[10][0][1];
        $decemberLastProd = $monthsLastProd[11][0][1];

        $monthLastProd = [$januaryLastProd, $februaryLastProd, $marchLastProd, $aprilLastProd, $mayLastProd, $juneLastProd,
            $julyLastProd, $augustLastProd, $septemberLastProd, $octoberLastProd, $novemberLastProd, $decemberLastProd];

        $lastProd = json_encode($monthLastProd);
        $dataLastYearProd = str_replace( search: '"', replace: '', $LastProd);

        //Percent of Benefits
        $benefitsProd = $benefitsCurrentYearProd - $benefitsLastYearProd;
        $benefitsPercentProd = ($benefitsProd / $benefitsLastYearProd) * 100;
        $percentProd = round($benefitsPercentProd, precision: 2);

        //mostDemandedServicesPerYear
        $em = $this->getDoctrine()->getManager();
        $queryServMostDemanded = $em->getRepository( className: PaymentService::class)->mostDemandedServicesPerYear($currentYearServ);
```



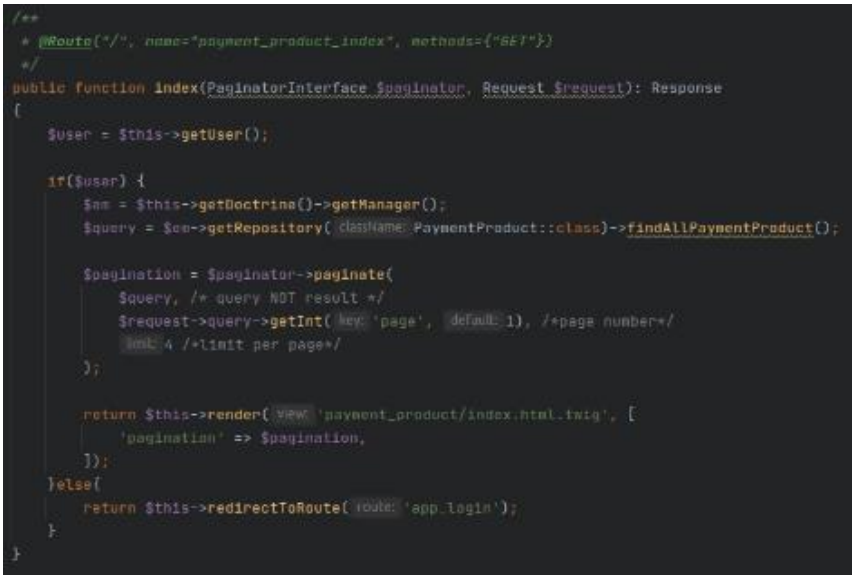
PaymentProductController

Este controlador se encarga de la creación, muestra y manipulación del pago de los productos, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

Además, se ha añadido la función morosos para saber qué clientes no han pagado un producto.

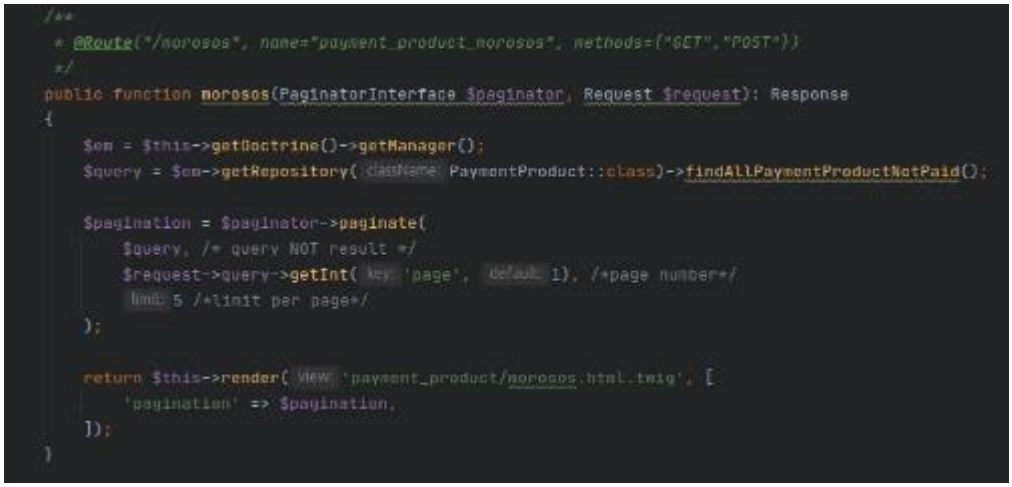
Index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a todos los pagos de los productos, en este caso los datos obtenidos se van a mostrar con un límite de cuatro líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.



morosos

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a todos los productos que no han sido pagados por un cliente, en este caso los datos obtenidos se van a mostrar con un límite de cinco líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.



```
/**
 * @Route("/morosos", name="payment_product_morosos", methods={"GET","POST"})
 */
public function morosos(PaginatorInterface $paginator, Request $request): Response
{
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: PaymentProduct::class)->findAllPaymentProductNotPaid();

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );

    return $this->render( view: 'payment_product/morosos.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación

Esta función se encarga de validar si en el momento de la compra existen suficientes unidades de ese producto, si las hay se comprueba si después de la compra su stock será igual o inferior al stock mínimo para mostrar un mensaje de advertencia, si todo es correcto se valida la compra y se actualiza en la base de datos el stock de ese producto y la compra realizada.

```
/**
 * @Route("/nuevo", name="payment_product_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $paymentProduct = new PaymentProduct();
    $form = $this->createForm( type: PaymentProductType::class, $paymentProduct);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $user = $this->getUser();
        $paymentProduct->setUser($user);

        $entityManager = $this->getDoctrine()->getManager();

        $id = $paymentProduct->getProduct();

        $prices = $this->getDoctrine()
            ->getRepository( persistentObject: PaymentProduct::class)
            ->findPriceById($id);

        $price = $prices[0]["salePrice"];
```

```
        $quantity = $paymentProduct->getQuantity();
        $sumPrice = $price * $quantity;

        $paymentProduct->setPrice($price);
        $paymentProduct->setSumPrice($sumPrice);

        $amountDiscount = $quantity;
        $productId = $id;

        $amounts = $this->getDoctrine()
            ->getRepository( persistentObject: PaymentProduct::class)
            ->findAmountById($id);

        $amount = $amounts[0]["currentAmount"];

        $minAmounts = $this->getDoctrine()
            ->getRepository( persistentObject: PaymentProduct::class)
            ->findMinAmountById($id);

        $minAmount = $minAmounts[0]["minAmount"];

        if ($amount - $amountDiscount < $minAmount && $amount - $amountDiscount >= 0) {
            $this->addFlash( type: 'stock',  message: 'Revisar el stock para este producto');

            $entityManager->getRepository( className: PaymentProduct::class)->updateStock($amountDiscount, $productId);

            $entityManager->persist($paymentProduct);
            $entityManager->flush();
            return $this->redirectToRoute( route: 'payment_product_new');

        } elseif ($amount - $amountDiscount >= 0) {
            $entityManager->getRepository( className: PaymentProduct::class)->updateStock($amountDiscount, $productId);

            $entityManager->persist($paymentProduct);
            $entityManager->flush();
            return $this->redirectToRoute( route: 'payment_product_index');
        } else {
            $this->addFlash( type: 'error',  message: 'No se puede comprar el producto porque no hay suficiente stock');
            return $this->redirectToRoute( route: 'payment_product_new');
        }
    }
```

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('price', MoneyType::class, [
            'label' => 'Precio: ',
        ])
        ->add( child: 'quantity',  type: IntegerType::class, [
            'label' => 'Cantidad: ',
        ])
        ->add( child: 'client',  type: EntityType::class,
            [
                'class' => Client::class,
                'label' => 'Cliente: ',
                'choice_label' => function (Client $client) {
                    return sprintf( format: ' %s', $client->getClientName());
                },
                'placeholder' => 'Selecciona un cliente',
                'choices' => $this->clientRepository->findAllClients(),
            ])
        ->add( child: 'product',  type: EntityType::class,
            [
                'class' => Product::class,
                'label' => 'Producto: ',
                'choice_label' => function (Product $product) {
                    return sprintf( format: '%s - %s unidades en stock', $product->getProductName(), $product->getCurrentAmount());
                },
                'placeholder' => 'Selecciona un producto',
                'choices' => $this->productRepository->findAllProducts(),
            ])
        ->add( child: 'dateProduct',  type: DateType::class, [
            'label' => 'Fecha del Producto: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'hour',  type: TimeType::class, [
            'label' => 'Hora: ',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'datePaid',  type: DateType::class, [
            'label' => 'Fecha de Pago: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
```

```
->add( child: 'paid', type: CheckboxType::class,
    [
        'label' => '¿Se ha pagado el producto?',
        'required' => false,
    ])
->add( child: 'paymentType', type: ChoiceType::class, [
    'label' => 'Tipo de pago ',
    'choices' => [
        'Tipo de pago' => [
            'Efectivo' => 'efectivo',
            'Tarjeta' => 'tarjeta',
            'No se ha pagado' => 'no'
        ],
    ]
])
->add( child: 'comments', type: TextareaType::class,
    [
        'label' => 'Comentarios',
        'required' => false,
    ]);
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este pago de producto y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```
/**
 * @Route("/{id}", name="payment_product_show", methods={"GET"})
 */
public function show(PaymentProduct $paymentProduct): Response
{
    return $this->render( view: 'payment_product/show.html.twig', [
        'payment_product' => $paymentProduct,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este pago de producto y los muestra en el formulario de edición para modificar los datos de este pago, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```
/**
 * @Route("/{id}/editar", name="payment_product_edit", methods={"GET","POST"})
 */
public function edit(Request $request, PaymentProduct $paymentProduct): Response
{
    $form = $this->createForm( type: PaymentProductType::class, $paymentProduct);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'payment_product_index');
    }

    return $this->render( view: 'payment_product/edit.html.twig', [
        'payment_product' => $paymentProduct,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del pago del producto que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

```
/**
 * @Route("/{id}", name="payment_product_delete", methods={"DELETE"})
 */
public function delete(Request $request, PaymentProduct $paymentProduct): Response
{
    if ($this->isCsrfTokenValid( id: 'delete'.$paymentProduct->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($paymentProduct);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'payment_product_index');
}
```

PaymentServiceController

Este controlador se encarga de la creación, muestra y manipulación del pago de los servicios, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

Además, se ha añadido la función morosos para saber qué clientes no han pagado un servicio.

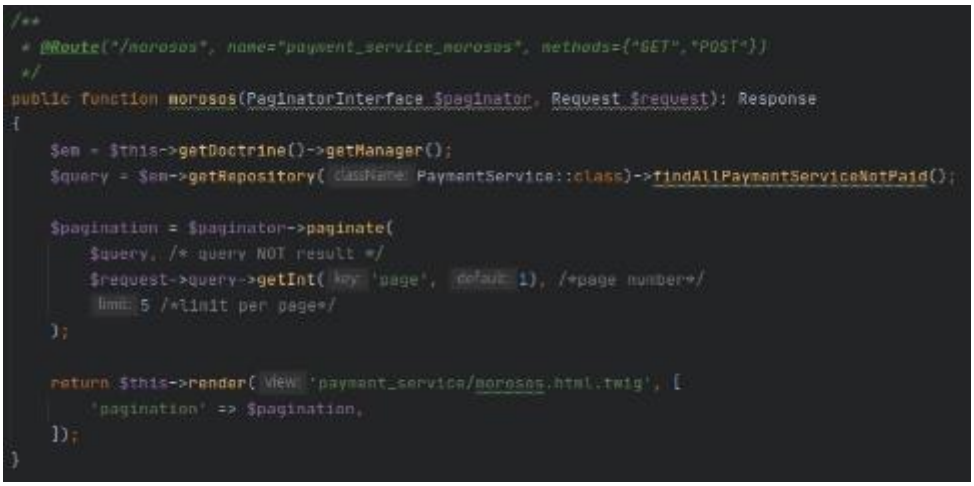
index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a todos los pagos de los servicios, en este caso los datos obtenidos se van a mostrar con un límite de seis líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```
/**
 * @Route("/", name="payment_service_index", methods={"GET","POST"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $user = $this->getUser();

    if($user) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: PaymentService::class)->findAllPaymentService();

        $pagination = $paginator->paginate(
            $query, /* query NOT result */
            $request->query->getInt( key: 'page', default: 1), /*page number*/
             limit: 6 /*limit per page*/
        );

        return $this->render( view: 'payment_service/index.html.twig', [
            'pagination' => $pagination,
        ]);
    }else{
        return $this->redirectToRoute( route: 'app_login');
    }
}
```

morosos

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a todos los servicios que no han sido pagados por un cliente, en este caso los datos obtenidos se van a mostrar con un límite de cinco líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.



```php
/**
 * @Route("/morosos", name="payment_service_morosos", methods={"GET","POST"})
 */
public function morosos(PaginatorInterface $paginator, Request $request): Response
{
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: PaymentService::class)->findAllPaymentServiceNotPaid();

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
        limit: 5 /*limit per page*/
    );

    return $this->render( view: 'payment_service/morosos.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación

```php
/**
 * @Route("/nuevo", name="payment_service_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $paymentService = new PaymentService();
    $form = $this->createForm( type: PaymentServiceType::class, $paymentService);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $user = $this->getUser();
        $paymentService->setUser($user);
        $entityManager = $this->getDoctrine()->getManager();

        $id = $paymentService->getService();

        $prices = $this->getDoctrine()
            ->getRepository( persistentObject: PaymentService::class)
            ->findServicePriceById($id);

        $price = $prices[0]["price"];

        $paymentService->setPrice($price);

        $entityManager->persist($paymentService);
        $entityManager->flush();
        return $this->redirectToRoute( route: 'payment_service_index');
    }

    return $this->render( view: 'payment_service/new.html.twig', [
        'payment_service' => $paymentService,
        'form' => $form->createView(),
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        /*->add('price', MoneyType::class, [
            'label' => 'Precio: ',
        ])*/
        ->add( child: 'client', type: EntityType::class,
            [
                'class' => Client::class,
                'label' => 'Cliente: ',
                'choice_label' => function (Client $client) {
                    return sprintf( format: '%s', $client->getClientName());
                },
                'placeholder' => 'Selecciona un cliente',
                'choices' => $this->clientRepository->findAllClients(),
            ])
        ->add( child: 'service', type: EntityType::class,
            [
                'class' => Service::class,
                'label' => 'Servicio: ',
                'choice_label' => function (Service $service) {
                    return sprintf( format: '%s - %s€', $service->getServiceName(), $service->getPrice());
                },
                'placeholder' => 'Selecciona un servicio',
                'choices' => $this->serviceRepository->findAllServices(),
            ])
        ->add( child: 'dateService', type: DateType::class, [
                'label' => 'Fecha del Servicio: ',
                'widget' => 'choice',
                'format' => 'd-MM-y',
                'data' => new \DateTime( dateTime: "now")
            ]
        )
        ->add( child: 'hour', type: TimeType::class, [
            'label' => 'Hora: ',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'datePaid', type: DateType::class, [
            'label' => 'Fecha de Pago: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'paid', type: CheckboxType::class,
            [
                'label' => '¿Se ha pagado el servicio?',
                'required' => false,
            ])
        ->add( child: 'paymentType', type: ChoiceType::class, [
            'label' => 'Tipo de pago ',
            'choices' => [
                'Tipo de pago' => [
                    'Efectivo' => 'efectivo',
                    'Tarjeta' => 'tarjeta',
                    'No se ha pagado' => 'no'
                ],
            ]
        ])
        ->add( child: 'comments', type: TextareaType::class,
            [
                'label' => 'Comentarios',
                'required' => false,

            ]);
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este pago de servicio y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="payment_service_show", methods={"GET"})
 */
public function show(PaymentService $paymentService): Response
{
    return $this->render( view: 'payment_service/show.html.twig', [
        'payment_service' => $paymentService,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este pago de servicio y los muestra en el formulario de edición para modificar los datos de este pago, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```php
/**
 * @Route("/{id}/editar", name="payment_service_edit", methods={"GET","POST"})
 */
public function edit(Request $request, PaymentService $paymentService): Response
{
    $form = $this->createForm( type: PaymentServiceType::class, $paymentService);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'payment_service_index');
    }

    return $this->render( view: 'payment_service/edit.html.twig', [
        'payment_service' => $paymentService,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del pago del servicio que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

```php
/**
 * @Route("/{id}", name="payment_service_delete", methods={"DELETE"})
 */
public function delete(Request $request, PaymentService $paymentService): Response
{
    if ($this->isCsrfTokenValid( id: 'delete' . $paymentService->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($paymentService);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'payment_service_index');
}
```

PrintController

Este controlador se encarga de crear los diferentes documentos que se pueden descargar en la aplicación para saber, por ejemplo, el precio de los diferentes servicios que se ofrecen en el salón de belleza, de los proveedores del salón de belleza, de las citas diarias del salón de belleza y de las comparativas de ganancias de servicios y productos del año actual y del anterior.

El procedimiento de actuación de estas funciones es muy similar a las anteriores descritas, se obtienen datos de la base de datos y se pasan a las plantillas twig pero en vez de mostrarlas en las diferentes páginas de la aplicación, se crea un documento en pdf para poder ser descargado.

printService

Esta primera función se encarga de obtener todos los datos de los servicios que se ofrecen en el salón de belleza y de crear un documento en pdf para descargar.

```php
/**
 * @Route("/servicios/service_print", name="service_print", methods={"GET"})
 */
public function printServices(): Response
{
    // Configure Dompdf according to your needs
    $pdfOptions = new Options();
    $pdfOptions->set('defaultFont', 'Arial');

    // Instantiate Dompdf with our options
    $dompdf = new Dompdf($pdfOptions);

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Service::class)->findAllServ();

    $currentYear = date( format: "d-m-Y");

    // Retrieve the HTML generated in our twig file
    $html = $this->renderView( view: 'service/servicePrint.html.twig', [
        'pagination' => $query,
        'date' => $currentYear,
    ]);

    // Load HTML to Dompdf
    $dompdf->loadHtml($html);

    // (Optional) Setup the paper size and orientation 'portrait' or 'portrait'
    $dompdf->setPaper( size: 'A4', orientation: 'portrait');

    // Render the HTML as PDF
    $dompdf->render();

    // Output the generated PDF to Browser (force download)
    $dompdf->stream( filename: "servicios.pdf", [
        "Attachment" => true
    ]);
}
```

printProviders

De igual forma que la función anterior, esta función se encarga de obtener todos los datos de los proveedores del salón de belleza y de crear un documento en pdf para descargar.

```
/**
 * @Route("/proveedores/provider_print", name="provider_print", methods={"GET"})
 */
public function printProviders(): Response
{
    // Configure Dompdf according to your needs
    $pdfOptions = new Options();
    $pdfOptions->set('defaultFont', 'Arial');

    // Instantiate Dompdf with our options
    $dompdf = new Dompdf($pdfOptions);

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Provider::class)->findAllProvid();

    $currentYear = date( format: "d-m-Y");

    // Retrieve the HTML generated in our twig file
    $html = $this->renderView( view: 'provider/providerPrint.html.twig', [
        'pagination' => $query,
        'date' => $currentYear,
    ]);

    // Load HTML to Dompdf
    $dompdf->loadHtml($html);

    // (Optional) Setup the paper size and orientation 'portrait' or 'portrait'
    $dompdf->setPaper( size: 'A4', orientation: 'portrait');

    // Render the HTML as PDF
    $dompdf->render();

    // Output the generated PDF to Browser (force download)
    $dompdf->stream( filename: "proveedores.pdf", [
        "Attachment" => true
    ]);
}
```

printSchedule

Esta tercera función es exactamente igual que las anteriores, se encarga de obtener todos los datos de las citas diarias del salón de belleza y de crear un documento en pdf para descargar.

```
/**
 * @Route("/citas/citas_print", name="schedule_print", methods={"GET"})
 */
public function printSchedule(): Response
{
    // Configure Dompdf according to your needs
    $pdfOptions = new Options();
    $pdfOptions->set('defaultFont', 'Arial');

    // Instantiate Dompdf with our options
    $dompdf = new Dompdf($pdfOptions);

    $date = date( format: 'Y-m-d');

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Schedule::class)->findScheduleByDate($date);

    $currentYear = date( format: "d-m-Y");

    // Retrieve the HTML generated in our twig file
    $html = $this->renderView( view: 'schedule/schedulePrint.html.twig', [
        'pagination' => $query,
        'date' => $currentYear,

    ]);

    // Load HTML to Dompdf
    $dompdf->loadHtml($html);

    // (Optional) Setup the paper size and orientation 'portrait' or 'portrait'
    $dompdf->setPaper( size: 'A4', orientation: 'portrait');

    // Render the HTML as PDF
    $dompdf->render();

    // Output the generated PDF to Browser (force download)
    $dompdf->stream( filename: "horario.pdf", [
        "Attachment" => true
    ]);
}
```

printComparativeServices

Esta función se encarga de obtener todos los beneficios de los servicios de cada mes del año actual y del anterior para crear un documento en pdf donde se muestra una tabla comparativa.

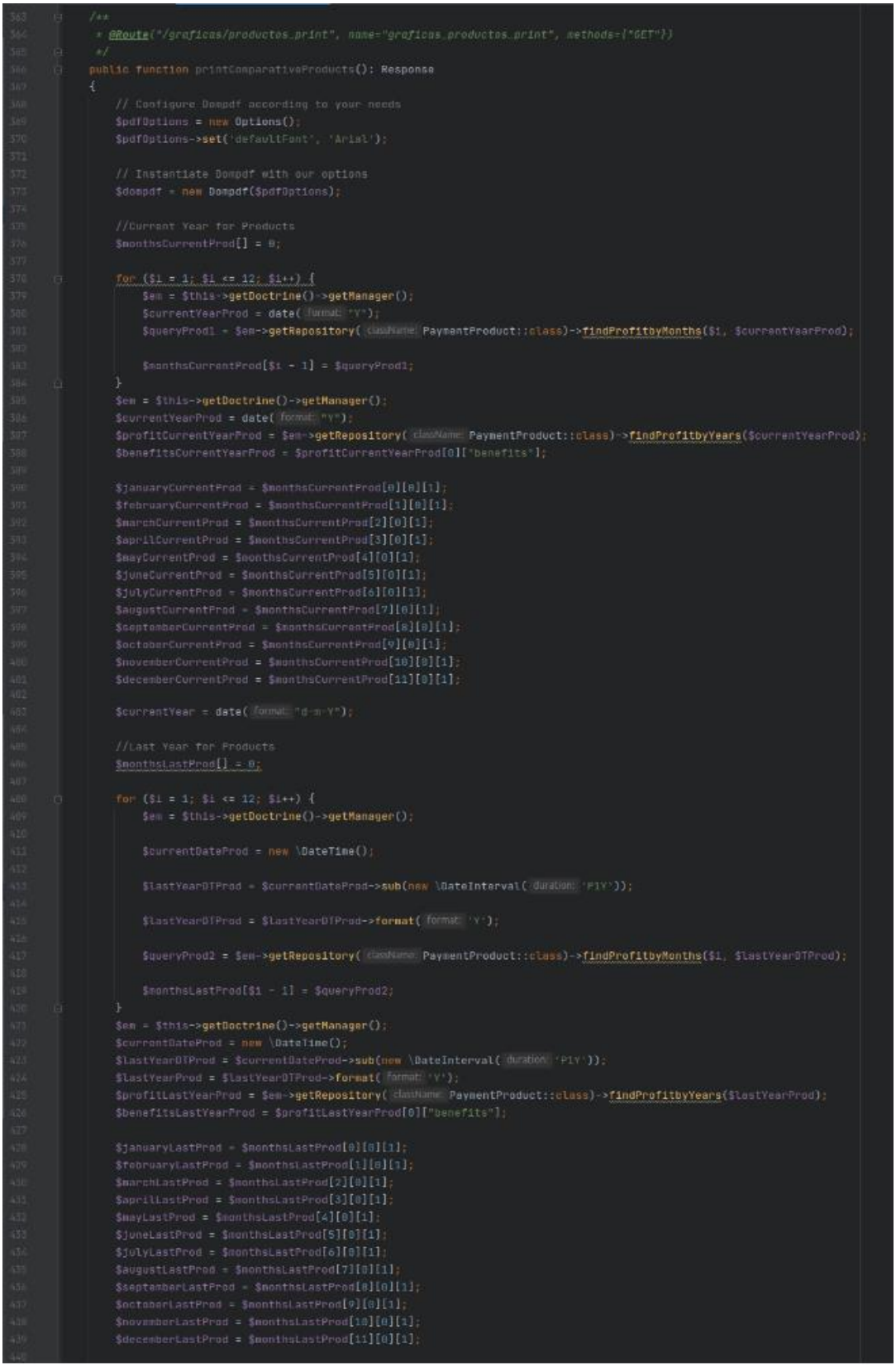
```php
/**
 * @Route("/graficas/servicios_print", name="graficas_servicios_print", methods={"GET"})
 */
public function printComparativeServices(): Response
{
    // Configure Dompdf according to your needs
    $pdfOptions = new Options();
    $pdfOptions->set('defaultFont', 'Arial');

    // Instantiate Dompdf with our options
    $dompdf = new Dompdf($pdfOptions);

    //Current Year for Services
    $monthsCurrentServ[] = 0;

    for ($i = 1; $i <= 12; $i++) {
        $em = $this->getDoctrine()->getManager();
        $currentYearServ = date( format: "Y");
        $query = $em->getRepository( className: PaymentService::class)->findProfitbyMonths($i, $currentYearServ);

        $monthsCurrentServ[$i - 1] = $query;
    }
    $em = $this->getDoctrine()->getManager();
    $currentYearServ = date( format: "Y");
    $profitCurrentYearServ = $em->getRepository( className: PaymentService::class)->findProfitbyYears($currentYearServ);
    $benefitsCurrentYearServ = $profitCurrentYearServ[0]["benefits"];

    $januaryCurrentServ = $monthsCurrentServ[0][0][1];
    $februaryCurrentServ = $monthsCurrentServ[1][0][1];
    $marchCurrentServ = $monthsCurrentServ[2][0][1];
    $aprilCurrentServ = $monthsCurrentServ[3][0][1];
    $mayCurrentServ = $monthsCurrentServ[4][0][1];
    $juneCurrentServ = $monthsCurrentServ[5][0][1];
    $julyCurrentServ = $monthsCurrentServ[6][0][1];
    $augustCurrentServ = $monthsCurrentServ[7][0][1];
    $septemberCurrentServ = $monthsCurrentServ[8][0][1];
    $octoberCurrentServ = $monthsCurrentServ[9][0][1];
    $novemberCurrentServ = $monthsCurrentServ[10][0][1];
    $decemberCurrentServ = $monthsCurrentServ[11][0][1];

    $currentYear = date( format: "d-m-Y");

    //Last Year for Services
    $monthsLastServ[] = 0;

    for ($i = 1; $i <= 12; $i++) {
        $em = $this->getDoctrine()->getManager();

        $currentDateServ = new \DateTime();

        $lastYearDTServ = $currentDateServ->sub(new \DateInterval( duration: 'P1Y'));

        $lastYearServ = $lastYearDTServ->format( format: 'Y');

        $query2 = $em->getRepository( className: PaymentService::class)->findProfitbyMonths($i, $lastYearServ);

        $monthsLastServ[$i - 1] = $query2;
    }
    $em = $this->getDoctrine()->getManager();
    $currentDateServ = new \DateTime();
    $lastYearDTServ = $currentDateServ->sub(new \DateInterval( duration: 'P1Y'));
    $lastYearServ = $lastYearDTServ->format( format: 'Y');
    $profitLastYearServ = $em->getRepository( className: PaymentService::class)->findProfitbyYears($lastYearServ);
    $benefitsLastYearServ = $profitLastYearServ[0]["benefits"];

    $januaryLastServ = $monthsLastServ[0][0][1];
    $februaryLastServ = $monthsLastServ[1][0][1];
    $marchLastServ = $monthsLastServ[2][0][1];
    $aprilLastServ = $monthsLastServ[3][0][1];
    $mayLastServ = $monthsLastServ[4][0][1];
    $juneLastServ = $monthsLastServ[5][0][1];
    $julyLastServ = $monthsLastServ[6][0][1];
    $augustLastServ = $monthsLastServ[7][0][1];
    $septemberLastServ = $monthsLastServ[8][0][1];
    $octoberLastServ = $monthsLastServ[9][0][1];
    $novemberLastServ = $monthsLastServ[10][0][1];
    $decemberLastServ = $monthsLastServ[11][0][1];
```

```php
        //Benefits per month
        $benefitsJanuary = $januaryCurrentServ - $januaryLastServ;
        if ($januaryLastServ > 0) {
            $benefitsJanuaryPercent = round((($benefitsJanuary / $januaryLastServ) * 100), precision: 2);
        } else {
            $benefitsJanuaryPercent = 0;
        }

        $benefitsFebruary = $februaryCurrentServ - $februaryLastServ;
        if ($februaryLastServ > 0) {
            $benefitsFebruaryPercent = round((($benefitsFebruary / $februaryLastServ) * 100), precision: 2);
        } else {
            $benefitsFebruaryPercent = 0;
        }

        $benefitsMarch = $marchCurrentServ - $marchLastServ;
        if ($marchLastServ > 0) {
            $benefitsMarchPercent = round((($benefitsMarch / $marchLastServ) * 100), precision: 2);
        } else {
            $benefitsMarchPercent = 0;
        }

        $benefitsApril = $aprilCurrentServ - $aprilLastServ;
        if ($aprilLastServ > 0) {
            $benefitsAprilPercent = round((($benefitsApril / $aprilLastServ) * 100), precision: 2);
        } else {
            $benefitsAprilPercent = 0;
        }

        $benefitsMay = $mayCurrentServ - $mayLastServ;
        if ($mayLastServ > 0) {
            $benefitsMayPercent = round((($benefitsMay / $mayLastServ) * 100), precision: 2);
        } else {
            $benefitsMayPercent = 0;
        }

        $benefitsJune = $juneCurrentServ - $juneLastServ;
        if ($juneLastServ > 0) {
            $benefitsJunePercent = round((($benefitsJune / $juneLastServ) * 100), precision: 2);
        } else {
            $benefitsJunePercent = 0;
        }

        $benefitsJuly = $julyCurrentServ - $julyLastServ;
        if ($julyLastServ > 0) {
            $benefitsJulyPercent = round((($benefitsJuly / $julyLastServ) * 100), precision: 2);
        } else {
            $benefitsJulyPercent = 0;
        }

        $benefitsAugust = $augustCurrentServ - $augustLastServ;
        if ($augustLastServ > 0) {
            $benefitsAugustPercent = round((($benefitsAugust / $augustLastServ) * 100), precision: 2);
        } else {
            $benefitsAugustPercent = 0;
        }

        $benefitsSeptember = $septemberCurrentServ - $septemberLastServ;
        if ($septemberLastServ > 0) {
            $benefitsSeptemberPercent = round((($benefitsSeptember / $septemberLastServ) * 100), precision: 2);
        } else {
            $benefitsSeptemberPercent = 0;
        }

        $benefitsOctober = $octoberCurrentServ - $octoberLastServ;
        if ($octoberLastServ > 0) {
            $benefitsOctoberPercent = round((($benefitsOctober / $octoberLastServ) * 100), precision: 2);
        } else {
            $benefitsOctoberPercent = 0;
        }
```

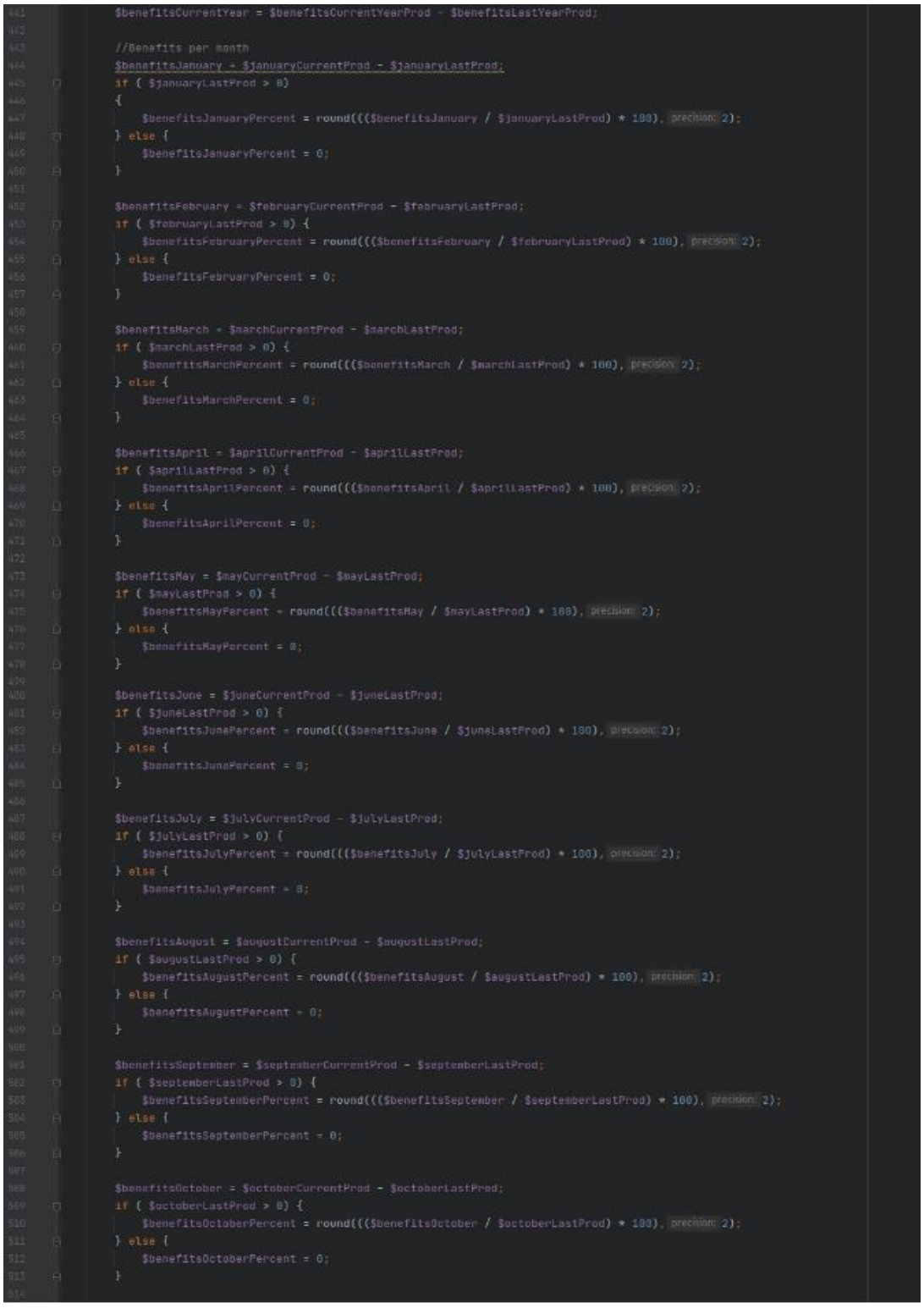
```php
        $benefitsNovember = $novemberCurrentServ - $novemberLastServ;
        if ($novemberLastServ > 0) {
            $benefitsNovemberPercent = round((($benefitsNovember / $novemberLastServ) * 100), precision: 2);
        } else {
            $benefitsNovemberPercent = 0;
        }

        $benefitsDecember = $decemberCurrentServ - $decemberLastServ;
        if ($decemberLastServ > 0) {
            $benefitsDecemberPercent = round((($benefitsDecember / $decemberLastServ) * 100), precision: 2);
        } else {
            $benefitsDecemberPercent = 0;
        }

        $benefitsCurrentYear = $benefitsCurrentYearServ - $benefitsLastYearServ;
        // Retrieve the HTML generated in our twig file
        $html = $this->renderView( view: 'graphics/servicePrint.html.twig', [
            'date' => $currentYear,
            'currentYear' => $currentYearServ,
            'lastYear' => $lastYearServ,
            'benefitsCurrentYear' => $benefitsCurrentYearServ,
            'benefitsLastYear' => $benefitsLastYearServ,
            'benefits' => $benefitsCurrentYear,

            'januaryCurrent' => $januaryCurrentServ,
            'februaryCurrent' => $februaryCurrentServ,
            'marchCurrent' => $marchCurrentServ,
            'aprilCurrent' => $aprilCurrentServ,
            'mayCurrent' => $mayCurrentServ,
            'juneCurrent' => $juneCurrentServ,
            'julyCurrent' => $julyCurrentServ,
            'augustCurrent' => $augustCurrentServ,
            'septemberCurrent' => $septemberCurrentServ,
            'octoberCurrent' => $octoberCurrentServ,
            'novemberCurrent' => $novemberCurrentServ,
            'decemberCurrent' => $decemberCurrentServ,

            'januaryLast' => $januaryLastServ,
            'februaryLast' => $februaryLastServ,
            'marchLast' => $marchLastServ,
            'aprilLast' => $aprilLastServ,
            'mayLast' => $mayLastServ,
            'juneLast' => $juneLastServ,
            'julyLast' => $julyLastServ,
            'augustLast' => $augustLastServ,
            'septemberLast' => $septemberLastServ,
            'octoberLast' => $octoberLastServ,
            'novemberLast' => $novemberLastServ,
            'decemberLast' => $decemberLastServ,

            'benefitsJanuary' => $benefitsJanuaryPercent,
            'benefitsFebruary' => $benefitsFebruaryPercent,
            'benefitsMarch' => $benefitsMarchPercent,
            'benefitsApril' => $benefitsAprilPercent,
            'benefitsMay' => $benefitsMayPercent,
            'benefitsJune' => $benefitsJunePercent,
            'benefitsJuly' => $benefitsJulyPercent,
            'benefitsAugust' => $benefitsAugustPercent,
            'benefitsSeptember' => $benefitsSeptemberPercent,
            'benefitsOctober' => $benefitsOctoberPercent,
            'benefitsNovember' => $benefitsNovemberPercent,
            'benefitsDecember' => $benefitsDecemberPercent,
        ]);

        // Load HTML to Dompdf
        $dompdf->loadHtml($html);

        // (Optional) Setup the paper size and orientation 'portrait' or 'portrait'
        $dompdf->setPaper( size: 'A4', orientation: 'portrait');

        // Render the HTML as PDF
        $dompdf->render();

        // Output the generated PDF to Browser (force download)
        $dompdf->stream( filename: "informeServicios.pdf", [
            "Attachment" => true
        ]);
    }
```

printComparativeProducts

Esta función se encarga de obtener todos los beneficios de los productos de cada mes del año actual y del anterior para crear un documento en pdf donde se muestra una una tabla comparativa.



```
    /**
     * @Route("/graficas/productos_print", name="graficas_productos_print", methods={"GET"})
     */
    public function printComparativeProducts(): Response
    {
        // Configure Dompdf according to your needs
        $pdfOptions = new Options();
        $pdfOptions->set('defaultFont', 'Arial');

        // Instantiate Dompdf with our options
        $dompdf = new Dompdf($pdfOptions);

        //Current Year for Products
        $monthsCurrentProd[] = 0;

        for ($i = 1; $i <= 12; $i++) {
            $em = $this->getDoctrine()->getManager();
            $currentYearProd = date( format: "Y");
            $queryProd1 = $em->getRepository( className: PaymentProduct::class)->findProfitbyMonths($i, $currentYearProd);

            $monthsCurrentProd[$i - 1] = $queryProd1;
        }
        $em = $this->getDoctrine()->getManager();
        $currentYearProd = date( format: "Y");
        $profitCurrentYearProd = $em->getRepository( className: PaymentProduct::class)->findProfitbyYears($currentYearProd);
        $benefitsCurrentYearProd = $profitCurrentYearProd[0]['benefits'];

        $januaryCurrentProd = $monthsCurrentProd[0][0][1];
        $februaryCurrentProd = $monthsCurrentProd[1][0][1];
        $marchCurrentProd = $monthsCurrentProd[2][0][1];
        $aprilCurrentProd = $monthsCurrentProd[3][0][1];
        $mayCurrentProd = $monthsCurrentProd[4][0][1];
        $juneCurrentProd = $monthsCurrentProd[5][0][1];
        $julyCurrentProd = $monthsCurrentProd[6][0][1];
        $augustCurrentProd = $monthsCurrentProd[7][0][1];
        $septemberCurrentProd = $monthsCurrentProd[8][0][1];
        $octoberCurrentProd = $monthsCurrentProd[9][0][1];
        $novemberCurrentProd = $monthsCurrentProd[10][0][1];
        $decemberCurrentProd = $monthsCurrentProd[11][0][1];

        $currentYear = date( format: "d-m-Y");

        //Last Year for Products
        $monthsLastProd[] = 0;

        for ($i = 1; $i <= 12; $i++) {
            $em = $this->getDoctrine()->getManager();

            $currentDateProd = new \DateTime();

            $lastYearDTProd = $currentDateProd->sub(new \DateInterval( duration: 'P1Y'));

            $lastYearDTProd = $lastYearDTProd->format( format: 'Y');

            $queryProd2 = $em->getRepository( className: PaymentProduct::class)->findProfitbyMonths($i, $lastYearDTProd);

            $monthsLastProd[$i - 1] = $queryProd2;
        }
        $em = $this->getDoctrine()->getManager();
        $currentDateProd = new \DateTime();
        $lastYearDTProd = $currentDateProd->sub(new \DateInterval( duration: 'P1Y'));
        $lastYearProd = $lastYearDTProd->format( format: 'Y');
        $profitLastYearProd = $em->getRepository( className: PaymentProduct::class)->findProfitbyYears($lastYearProd);
        $benefitsLastYearProd = $profitLastYearProd[0]["benefits"];

        $januaryLastProd = $monthsLastProd[0][0][1];
        $februaryLastProd = $monthsLastProd[1][0][1];
        $marchLastProd = $monthsLastProd[2][0][1];
        $aprilLastProd = $monthsLastProd[3][0][1];
        $mayLastProd = $monthsLastProd[4][0][1];
        $juneLastProd = $monthsLastProd[5][0][1];
        $julyLastProd = $monthsLastProd[6][0][1];
        $augustLastProd = $monthsLastProd[7][0][1];
        $septemberLastProd = $monthsLastProd[8][0][1];
        $octoberLastProd = $monthsLastProd[9][0][1];
        $novemberLastProd = $monthsLastProd[10][0][1];
        $decemberLastProd = $monthsLastProd[11][0][1];
```



```
        $benefitsCurrentYear = $benefitsCurrentYearProd - $benefitsLastYearProd;

        //Benefits per month
        $benefitsJanuary = $januaryCurrentProd - $januaryLastProd;
        if ( $januaryLastProd > 0)
        {
            $benefitsJanuaryPercent = round((($benefitsJanuary / $januaryLastProd) * 100), precision: 2);
        } else {
            $benefitsJanuaryPercent = 0;
        }

        $benefitsFebruary = $februaryCurrentProd - $februaryLastProd;
        if ( $februaryLastProd > 0) {
            $benefitsFebruaryPercent = round((($benefitsFebruary / $februaryLastProd) * 100), precision: 2);
        } else {
            $benefitsFebruaryPercent = 0;
        }

        $benefitsMarch = $marchCurrentProd - $marchLastProd;
        if ( $marchLastProd > 0) {
            $benefitsMarchPercent = round((($benefitsMarch / $marchLastProd) * 100), precision: 2);
        } else {
            $benefitsMarchPercent = 0;
        }

        $benefitsApril = $aprilCurrentProd - $aprilLastProd;
        if ( $aprilLastProd > 0) {
            $benefitsAprilPercent = round((($benefitsApril / $aprilLastProd) * 100), precision: 2);
        } else {
            $benefitsAprilPercent = 0;
        }

        $benefitsMay = $mayCurrentProd - $mayLastProd;
        if ( $mayLastProd > 0) {
            $benefitsMayPercent = round((($benefitsMay / $mayLastProd) * 100), precision: 2);
        } else {
            $benefitsMayPercent = 0;
        }

        $benefitsJune = $juneCurrentProd - $juneLastProd;
        if ( $juneLastProd > 0) {
            $benefitsJunePercent = round((($benefitsJune / $juneLastProd) * 100), precision: 2);
        } else {
            $benefitsJunePercent = 0;
        }

        $benefitsJuly = $julyCurrentProd - $julyLastProd;
        if ( $julyLastProd > 0) {
            $benefitsJulyPercent = round((($benefitsJuly / $julyLastProd) * 100), precision: 2);
        } else {
            $benefitsJulyPercent = 0;
        }

        $benefitsAugust = $augustCurrentProd - $augustLastProd;
        if ( $augustLastProd > 0) {
            $benefitsAugustPercent = round((($benefitsAugust / $augustLastProd) * 100), precision: 2);
        } else {
            $benefitsAugustPercent = 0;
        }

        $benefitsSeptember = $septemberCurrentProd - $septemberLastProd;
        if ( $septemberLastProd > 0) {
            $benefitsSeptemberPercent = round((($benefitsSeptember / $septemberLastProd) * 100), precision: 2);
        } else {
            $benefitsSeptemberPercent = 0;
        }

        $benefitsOctober = $octoberCurrentProd - $octoberLastProd;
        if ( $octoberLastProd > 0) {
            $benefitsOctoberPercent = round((($benefitsOctober / $octoberLastProd) * 100), precision: 2);
        } else {
            $benefitsOctoberPercent = 0;
        }
```

```
        $benefitsNovember = $novemberCurrentProd - $novemberLastProd;
        if ( $novemberLastProd > 0) {
            $benefitsNovemberPercent = round((($benefitsNovember / $novemberLastProd) * 100), precision: 2);
        } else {
            $benefitsNovemberPercent = 0;
        }

        $benefitsDecember = $decemberCurrentProd - $decemberLastProd;
        if ( $decemberLastProd > 0) {
            $benefitsDecemberPercent = round((($benefitsDecember / $decemberLastProd) * 100), precision: 2);
        } else {
            $benefitsDecemberPercent = 0;
        }

        // Retrieve the HTML generated in our twig file
        $html = $this->renderView( view: 'graphics/productPrint.html.twig', [
            'date' => $currentYear,
            'currentYear' => $currentYearProd,
            'lastYear' => $lastYearProd,
            'benefitsCurrentYear' => $benefitsCurrentYearProd,
            'benefitsLastYear' => $benefitsLastYearProd,
            'benefits' => $benefitsCurrentYear,

            'januaryCurrent' => $januaryCurrentProd,
            'februaryCurrent' => $februaryCurrentProd,
            'marchCurrent' => $marchCurrentProd,
            'aprilCurrent' => $aprilCurrentProd,
            'mayCurrent' => $mayCurrentProd,
            'juneCurrent' => $juneCurrentProd,
            'julyCurrent' => $julyCurrentProd,
            'augustCurrent' => $augustCurrentProd,
            'septemberCurrent' => $septemberCurrentProd,
            'octoberCurrent' => $octoberCurrentProd,
            'novemberCurrent' => $novemberCurrentProd,
            'decemberCurrent' => $decemberCurrentProd,

            'januaryLast' => $januaryLastProd,
            'februaryLast' => $februaryLastProd,
            'marchLast' => $marchLastProd,
            'aprilLast' => $aprilLastProd,
            'mayLast' => $mayLastProd,
            'juneLast' => $juneLastProd,
            'julyLast' => $julyLastProd,
            'augustLast' => $augustLastProd,
            'septemberLast' => $septemberLastProd,
            'octoberLast' => $octoberLastProd,
            'novemberLast' => $novemberLastProd,
            'decemberLast' => $decemberLastProd,

            'benefitsJanuary' => $benefitsJanuaryPercent,
            'benefitsFebruary' => $benefitsFebruaryPercent,
            'benefitsMarch' => $benefitsMarchPercent,
            'benefitsApril' => $benefitsAprilPercent,
            'benefitsMay' => $benefitsMayPercent,
            'benefitsJune' => $benefitsJunePercent,
            'benefitsJuly' => $benefitsJulyPercent,
            'benefitsAugust' => $benefitsAugustPercent,
            'benefitsSeptember' => $benefitsSeptemberPercent,
            'benefitsOctober' => $benefitsOctoberPercent,
            'benefitsNovember' => $benefitsNovemberPercent,
            'benefitsDecember' => $benefitsDecemberPercent,
        ]);

        // Load HTML to Dompdf
        $dompdf->loadHtml($html);

        // (Optional) Setup the paper size and orientation 'portrait' or 'portrait'
        $dompdf->setPaper( size: 'A4', orientation: 'portrait');

        // Render the HTML as PDF
        $dompdf->render();

        // Output the generated PDF to Browser (force download)
        $dompdf->stream( filename: "InformeProductos.pdf", [
            "Attachment" => true
        ]);
    }
}
```

ProductController

Este controlador se encarga de la creación, muestra y manipulación de los productos, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, según si buscamos un producto o no hacemos ninguna búsqueda, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a los productos buscados o a todos los productos de la aplicación, en este caso los datos obtenidos se van a mostrar con un límite de cinco líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/", name="product_index", methods={"GET"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $productname = $request->get( key: 'query');

    if (!empty($productname)) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Product::class)->findProductByName($productname);
    } else {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Product::class)->findAllProds();
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );

    return $this->render( view: 'product/index.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

stock

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, para saber la falta de stock de los diferentes productos realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a los productos con falta de stock, los datos obtenidos se van a mostrar con un límite de cinco líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/stock", name="product_stock", methods={"GET"})
 */
public function stock(PaginatorInterface $paginator, Request $request): Response
{
    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: Product::class)->findAllStock();

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );

    return $this->render( view: 'product/stock.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```php
/**
 * @Route("/nuevo", name="product_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $product = new Product();
    $form = $this->createForm( type: ProductType::class, $product);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $currentAmount = $product->getAmount();
        $product->setCurrentAmount($currentAmount);
        $totalPurchase = $product->getAmount() * $product->getPurchasePrice();
        $product->setTotalPurchase($totalPurchase);
        $entityManager->persist($product);
        $entityManager->flush();

        return $this->redirectToRoute( route: 'product_index');
    }

    return $this->render( view: 'product/new.html.twig', [
        'product' => $product,
        'form' => $form->createView(),
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'productName',  type: TextType::class, [
            'label' => 'Nombre: '
        ])
        ->add( child: 'sale_price',  type: MoneyType::class, [
            'label' => 'Precio de venta: '
        ])
        ->add( child: 'purchase_price',  type: MoneyType::class, [
            'label' => 'Precio del proveedor: '
        ])
        ->add( child: 'amount',  type: NumberType::class, [
            'label' => 'Unidades: '
        ])
        ->add( child: 'min_amount',  type: NumberType::class, [
            'label' => 'Cantidad Mínima: '
        ])
        ->add( child: 'datePurchase',  type: DateType::class, [
            'label' => 'Fecha de compra: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'provider',  type: EntityType::class,
            [
                'class' => Provider::class,
                'label' => 'Proveedor: ',
                'choice_label' => function (Provider $provider) {
                    return sprintf( format: '%s - %s', $provider->getProviderName(), $provider->getBrand());
                },
                'placeholder' => 'Elije un proveedor',
                'choices' => $this->providerRepository->findAllProviders(),
            ]);
}
```

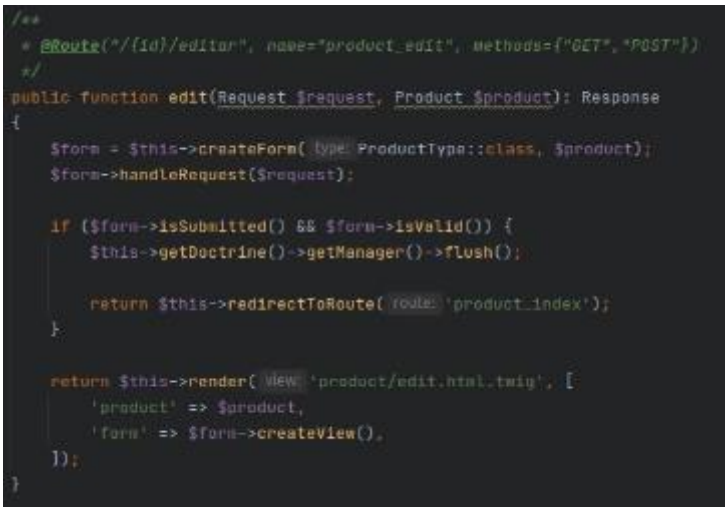
show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este producto y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="product_show", methods={"GET"})
 */
public function show(Product $product): Response
{
    return $this->render( view: 'product/show.html.twig', [
        'product' => $product,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este producto y los muestra en el formulario de edición para modificar los datos de este pago, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.



delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del producto que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

ProviderController

Este controlador se encarga de la creación, muestra y manipulación de los proveedores, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, según si buscamos un proveedor por su nombre o marca o no hacemos ninguna búsqueda, realiza una consulta a los repositorios de la base de datos para mostrar los datos referentes a los proveedores buscados o a todos los proveedores de la aplicación, en este caso los datos obtenidos se van a mostrar con un límite de cinco líneas por página, creando tantas páginas como sean necesarias para que los datos se visualicen de forma ordenada, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```php
/**
 * @Route("/nuevo", name="provider_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $provider = new Provider();
    $form = $this->createForm( type: ProviderType::class, $provider);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($provider);
        $entityManager->flush();
        return $this->redirectToRoute( route: 'provider_index');
    }

    return $this->render( view: 'provider/new.html.twig', [
        'provider' => $provider,
        'form' => $form->createView(),
    ]);
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este proveedor y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="provider_show", methods={"GET"})
 */
public function show(Provider $provider): Response
{
    return $this->render( view: 'provider/show.html.twig', [
        'provider' => $provider,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este proveedor y los muestra en el formulario de edición para modificar los datos de este proveedor, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```php
/**
 * @Route("/{id}/editar", name="provider_edit", methods={"GET","POST"})
 */
public function edit(Request $request, Provider $provider): Response
{
    $form = $this->createForm( type: ProviderType::class, $provider);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'provider_index');
    }

    return $this->render( view: 'provider/edit.html.twig', [
        'provider' => $provider,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del proveedor que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

```php
/**
 * @Route("/{id}", name="provider_delete", methods={"DELETE"})
 */
public function delete(Request $request, Provider $provider): Response
{
    if ($this->isCsrfTokenValid( id: 'delete'.$provider->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($provider);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'provider_index');
}
```

ScheduleController

Este controlador se encarga de la creación, muestra y manipulación de las citas del salón de belleza, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, por defecto aparecen las citas del día actual, pero podemos ver las citas de cualquier día seleccionándolo, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/", name="schedule_index", methods={"GET","POST"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $user = $this->getUser();

    if ($user) {
        $scheduleDate = new ScheduleDate();
        $form = $this->createForm( type: ScheduleDateType::class, $scheduleDate);
        $form->handleRequest($request);

        if ($scheduleDate->getDate() == null) {
            $date = date( format: 'Y-m-d');
        } else {
            $date = $scheduleDate->getDate();

        }
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Schedule::class)->findScheduleByDate($date);

        $pagination = $paginator->paginate(
            $query, /* query NOT result */
            $request->query->getInt( key: 'page',  default: 1), /*page number*/
             limit: 100 /*limit per page*/
        );

        return $this->render( view: 'schedule/index.html.twig', [
            'pagination' => $pagination,
            'form' => $form->createView(),
            'date' => $date
        ]);
    } else {
        return $this->redirectToRoute( route: 'app_login');
    }

}
```

new

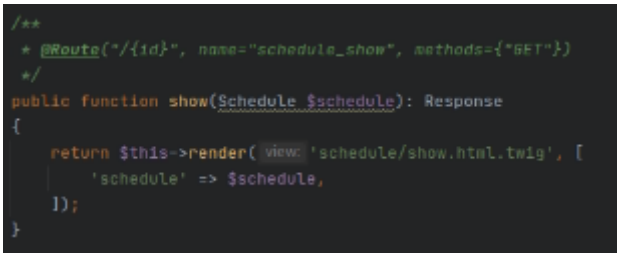
Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'date', type: DateType::class, [
            'label' => 'Fecha: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'hour', type: TimeType::class, [
            'label' => 'Hora: ',
            'data' => new \DateTime( datetime: "now")

        ])
        ->add( child: 'client', type: EntityType::class,
            [
                'class' => Client::class,
                'label' => 'Cliente: ',
                'choice_label' => function (Client $client) {
                    return sprintf( format: '%s', $client->getClientName());
                },
                'placeholder' => 'Selecciona un cliente',
                'choices' => $this->clientRepository->findAllClients(),
            ])
        ->add( child: 'service', type: EntityType::class,
            [
                'class' => Service::class,
                'label' => 'Servicio: ',
                'choice_label' => function (Service $service) {
                    return sprintf( format: '%s - %s€', $service->getServiceName(), $service->getPrice());
                },
                'placeholder' => 'Selecciona un servicio',
                'choices' => $this->serviceRepository->findAllServices(),
            ])
        ->add( child: 'user', type: EntityType::class,
            [
                'class' => User::class,
                'label' => 'Empleado: ',
                'choice_label' => function (User $user) {
                    return sprintf( format: '%s', $user->getUsername());
                },
                'placeholder' => 'Selecciona un trabajador',
                'choices' => $this->userRepository->findAllUsers()
            ]
        );
}
```

```
/**
 * @Route("/nuevo", name="schedule_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $schedule = new Schedule();
    $form = $this->createForm( type: ScheduleType::class, $schedule);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($schedule);
        $entityManager->flush();

        return $this->redirectToRoute( route: 'schedule_index');
    }

    return $this->render( view: 'schedule/new.html.twig', [
        'schedule' => $schedule,
        'form' => $form->createView(),
    ]);
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de esta cita y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.



edit

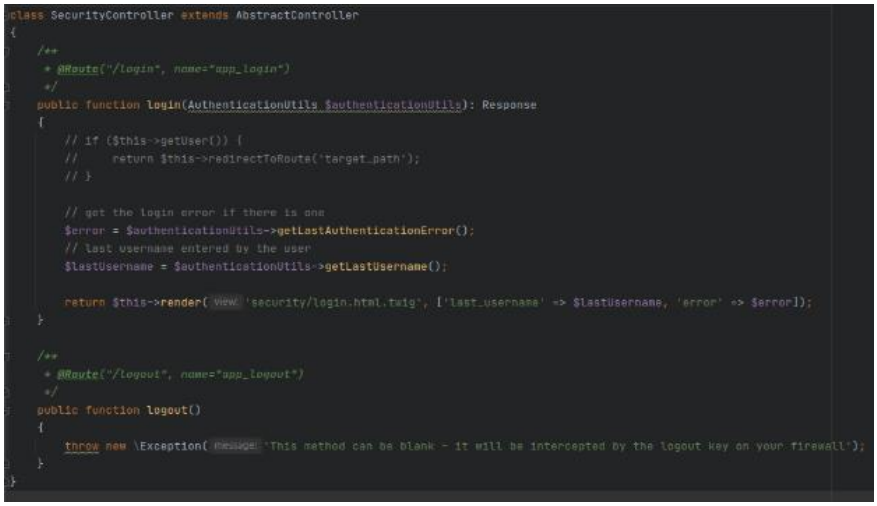
Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de esta cita y los muestra en el formulario de edición para modificar los datos de esta cita, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id de la cita que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

SecurityController

Este controlador es especial porque se encarga del control de acceso a la aplicación autenticando a los usuarios que intentan acceder a ella. En concreto se encarga de la entrada y la salida de los usuarios en la aplicación.



```
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // if ($this->getUser()) {
        //     return $this->redirectToRoute('target_path');
        // }

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render( view: 'security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error]);
    }

    /**
     * @Route("/logout", name="app_logout")
     */
    public function logout()
    {
        throw new \Exception( message: 'This method can be blank - it will be intercepted by the logout key on your firewall');
    }
}
```

ServiceController

Este controlador se encarga de la creación, muestra y manipulación de los servicios del salón de belleza, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, tenemos la posibilidad de buscar el nombre de un servicio en concreto o por el contrario queremos que se muestren todos los servicios, según nuestra elección se ejecutará una consulta en la base de datos u otra, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/", name="service_index", methods={"GET"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $servicename = $request->get( key: 'query');

    if (!empty($servicename)) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Service::class)->findServiceByName($servicename);
    } else {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: Service::class)->findAllServ();
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page', default: 1), /*page number*/
         limit: 3 /*limit per page*/
    );

    return $this->render( view: 'service/index.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```php
/**
 * @Route("/nuevo", name="service_new", methods={"GET","POST"})
 */
public function new(Request $request): Response
{
    $service = new Service();
    $form = $this->createForm( type: ServiceType::class, $service);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($service);
        $entityManager->flush();

        return $this->redirectToRoute( route: 'service_index');
    }

    return $this->render( view: 'service/new.html.twig', [
        'service' => $service,
        'form' => $form->createView(),
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'serviceName', type: TextType::class, [
            'label' => 'Nombre: '
        ])
        ->add( child: 'description', type: TextareaType::class, [
            'label' => 'Descripción: '
        ])
        ->add( child: 'price', type: MoneyType::class, [
            'label' => 'Precio: '
        ])
    ;
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este servicio y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="service_show", methods={"GET"})
 */
public function show(Service $service): Response
{
    return $this->render( view: 'service/show.html.twig', [
        'service' => $service,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este servicio y los muestra en el formulario de edición para modificar los datos de este servicio, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```php
/**
 * @Route("/{id}/editar", name="service_edit", methods={"GET","POST"})
 */
public function edit(Request $request, Service $service): Response
{
    $form = $this->createForm( type: ServiceType::class, $service);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'service_index');
    }

    return $this->render( view: 'service/edit.html.twig', [
        'service' => $service,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del servicio que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

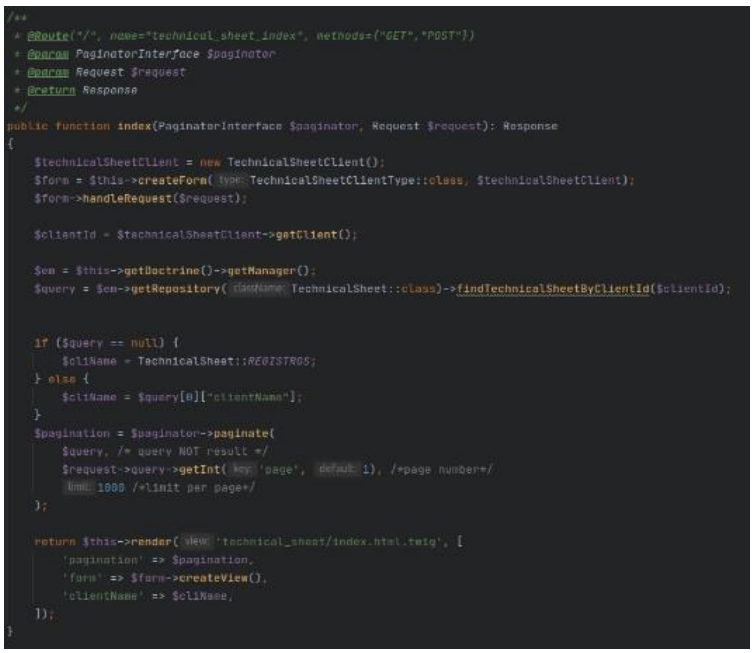
```php
/**
 * @Route("/{id}", name="service_delete", methods={"DELETE"})
 */
public function delete(Request $request, Service $service): Response
{
    if ($this->isCsrfTokenValid( id: 'delete' , $service->getId(), $request->request->get( key: ' token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($service);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'service_index');
}
```

TechnicalSheetController

Este controlador se encarga de la creación, muestra y manipulación de las fichas técnicas de los clientes del salón de belleza, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, mediante el desplegable podemos seleccionar el nombre de un cliente para que aparezca su ficha técnica, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos



```
/**
 * @Route("/", name="technical_sheet_index", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $technicalSheetClient = new TechnicalSheetClient();
    $form = $this->createForm( type: TechnicalSheetClientType::class, $technicalSheetClient);
    $form->handleRequest($request);

    $clientId = $technicalSheetClient->getClient();

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: TechnicalSheet::class)->findTechnicalSheetByClientId($clientId);


    if ($query == null) {
        $cliName = TechnicalSheet::REGISTROS;
    } else {
        $cliName = $query[0]["clientName"];
    }
    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 1000 /*limit per page*/
    );

    return $this->render( view: 'technical_sheet/index.html.twig', [
        'pagination' => $pagination,
        'form' => $form->createView(),
        'clientName' => $cliName,
    ]);
}
```

paginated

Esta función es exactamente igual que la de index, la única diferencia es que tenemos la posibilidad de que los datos aparezcan en diferentes páginas para poder visualizar los datos de forma más ordenada. Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, mediante el desplegable podemos seleccionar el nombre de un cliente para que aparezca su ficha técnica, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos.

```php
/**
 * @Route("/paginar/{id}", name="technical_sheet_paginated", methods={"GET","POST"})
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */
public function paginated(PaginatorInterface $paginator, Request $request, $id): Response
{

    $em = $this->getDoctrine()->getManager();
    $query = $em->getRepository( className: TechnicalSheet::class)->findTechnicalSheetByClientId($id);


    if ($query == null) {
        $cliName = TechnicalSheet::REGISTROS;
    } else {
        $cliName = $query[0]["clientName"];
    }
    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 1 /*limit per page*/
    );

    return $this->render( view: 'technical_sheet/index2.html.twig', [
        'pagination' => $pagination,
        'clientName' => $cliName,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

```php
/**
 * @Route("/nuevo", name="technical_sheet_new", methods={"GET","POST"})
 */
public function new(Request $request, SluggerInterface $slugger): Response
{
    $technicalSheet = new TechnicalSheet();
    $form = $this->createForm( type: TechnicalSheetType::class, $technicalSheet);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {

        $brochureFile = $form->get('photo')->getData();

        if ($brochureFile) {
            $originalFilename = pathinfo($brochureFile->getClientOriginalName(),  flags: PATHINFO_FILENAME);
            // this is needed to safely include the file name as part of the URL
            $safeFilename = $slugger->slug($originalFilename);
            $newFilename = $safeFilename . '-' . uniqid() . '.' . $brochureFile->guessExtension();

            // Move the file to the directory where brochures are stored
            try {
                $brochureFile->move(
                    $this->getParameter( name: 'photos_directory'),
                    $newFilename
                );
            } catch (FileException $e) {
                throw new \Exception( message: 'Ha ocurrido un error');
            }

            // updates the 'brochureFilename' property to store the PDF file name
            // instead of its contents
            $technicalSheet->setPhoto($newFilename);
        }

        $user = $this->getUser();
        $technicalSheet->setUser($user);

        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($technicalSheet);
        $entityManager->flush();

        return $this->redirectToRoute( route: 'technical_sheet_index');
    }

    return $this->render( view: 'technical_sheet/new.html.twig', [
        'technical_sheet' => $technicalSheet,
        'form' => $form->createView(),
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'date', type: DateType::class, [
            'label' => 'Fecha: ',
            'widget' => 'choice',
            'format' => 'd-MM-y',
            'data' => new \DateTime( datetime: "now")
        ])
        ->add( child: 'description', type: TextareaType::class, [
            'label' => 'Trabajo realizado: '
        ])
        ->add( child: 'photo', type: FileType::class, [
            'label' => 'Selecciona una imagen:',
            'mapped' => false,
            'required' => false
        ])
        ->add( child: 'client', type: EntityType::class,
            [
                'class' => Client::class,
                'label' => 'Cliente: ',
                'choice_label' => function (Client $client) {
                    return sprintf( format: '%s', $client->getClientName());
                },
                'placeholder' => 'Selecciona un cliente',
                'choices' => $this->clientRepository->findAllClients(),
            ])
        ->add( child: 'service', type: EntityType::class,
            [
                'class' => Service::class,
                'label' => 'Servicio: ',
                'choice_label' => function (Service $service) {
                    return sprintf( format: '%s', $service->getServiceName());
                },
                'placeholder' => 'Selecciona un servicio',
                'choices' => $this->serviceRepository->findAllServices(),
            ])
    ;
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este registro en la ficha técnica y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="technical_sheet_show", methods={"GET"})
 */
public function show(TechnicalSheet $technicalSheet): Response
{
    return $this->render( view: 'technical_sheet/show.html.twig', [
        'technical_sheet' => $technicalSheet,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este registro de la ficha técnica y los muestra en el formulario de edición para modificar los datos de este registro de la ficha técnica, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```
/**
 * @Route("/{id}/editar", name="technical_sheet_edit", methods={"GET","POST"})
 */
public function edit(Request $request, TechnicalSheet $technicalSheet, SluggerInterface $slugger): Response
{
    $form = $this->createForm( type: TechnicalSheetType::class, $technicalSheet);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {

        $brochureFile = $form->get('photo')->getData();

        if ($brochureFile) {
            $originalFilename = pathinfo($brochureFile->getClientOriginalName(), flags: PATHINFO_FILENAME);
            // this is needed to safely include the file name as part of the URL
            $safeFilename = $slugger->slug($originalFilename);
            $newFilename = $safeFilename . '-' . uniqid() . '.' . $brochureFile->guessExtension();

            // Move the file to the directory where brochures are stored
            try {
                $brochureFile->move(
                    $this->getParameter( name: 'photos_directory'),
                    $newFilename
                );
            } catch (FileException $e) {
                throw new \Exception( message: 'Ha ocurrido un error');
            }

            // updates the 'brochureFilename' property to store the PDF file name
            // instead of its contents
            $technicalSheet->setPhoto($newFilename);
        }

        $this->getDoctrine()->getManager()->flush();

        return $this->redirectToRoute( route: 'technical_sheet_index');
    }

    return $this->render( view: 'technical_sheet/edit.html.twig', [
        'technical_sheet' => $technicalSheet,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del registro de la ficha técnica que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

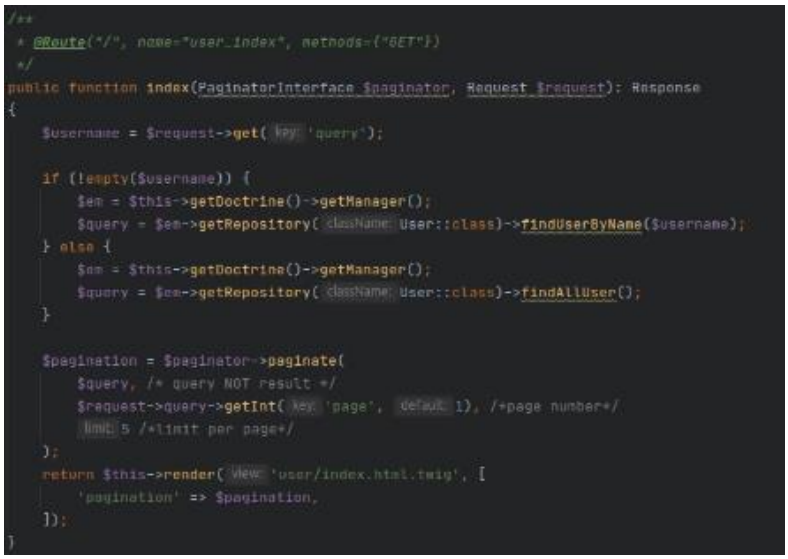
```
/**
 * @Route("/{id}", name="technical_sheet_delete", methods={"DELETE"})
 */
public function delete(Request $request, TechnicalSheet $technicalSheet): Response
{
    if ($this->isCsrfTokenValid( id: 'delete' . $technicalSheet->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($technicalSheet);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'technical_sheet_index');
}
```

UserController

Este controlador se encarga de la creación, muestra y manipulación de los trabajadores del salón de belleza, estas funciones han sido creadas automáticamente cuando se ha creado el controlador adaptando a posteriori cada función a las necesidades de la aplicación.

index

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, mediante el buscador podemos buscar el nombre de un trabajador para que aparezcan sus datos o no buscar a ningún trabajador y aparecerán los datos de todos los trabajadores, los datos generados se pasan a la plantilla twig correspondiente y ésta se encarga de mostrar los datos obtenidos



```
/**
 * @Route("/", name="user_index", methods={"GET"})
 */
public function index(PaginatorInterface $paginator, Request $request): Response
{
    $username = $request->get( key: 'query');

    if (!empty($username)) {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: User::class)->findUserByName($username);
    } else {
        $em = $this->getDoctrine()->getManager();
        $query = $em->getRepository( className: User::class)->findAllUser();
    }

    $pagination = $paginator->paginate(
        $query, /* query NOT result */
        $request->query->getInt( key: 'page',  default: 1), /*page number*/
         limit: 5 /*limit per page*/
    );
    return $this->render( view: 'user/index.html.twig', [
        'pagination' => $pagination,
    ]);
}
```

new

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener todos los datos que se han rellenado en el formulario y a validarlos, si todos los datos son correctos se persisten en la base de datos de la aplicación.

Esta función, al igual que la de la ficha técnica es un poco especial porque se sube una foto en este caso del trabajador para que aparezca en la página web junto a su nombre, por tanto, para ello se crea un directorio donde se almacenan las fotografías de los trabajadores, por otro lado, tenemos que encriptar la contraseña para que absolutamente nadie pueda verla en texto plano ni manipularla.

```php
/**
 * @Route("/nuevo", name="user_new", methods={"GET","POST"})
 */
public function new(Request $request, UserPasswordEncoderInterface $passwordEncoder, SluggerInterface $slugger): Response
{
    $user = new User();
    $form = $this->createForm( type: UserType::class, $user);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()){

        $brochureFile = $form->get('photo')->getData();

        if ($brochureFile) {
            $originalFilename = pathinfo($brochureFile->getClientOriginalName(), flags: PATHINFO_FILENAME);
            // this is needed to safely include the file name as part of the URL
            $safeFilename = $slugger->slug($originalFilename);
            $newFilename = $safeFilename.'-'.uniqid().'.'.$brochureFile->guessExtension();

            // Move the file to the directory where brochures are stored
            try {
                $brochureFile->move(
                    $this->getParameter( name: 'photos_users_directory'),
                    $newFilename
                );
            } catch (FileException $e) {
                throw new \Exception( message: 'Ha ocurrido un error');
            }

            // updates the 'brochureFilename' property to store the PDF file name
            // instead of its contents
            $user->setPhoto($newFilename);
        }


        $em = $this->getDoctrine()->getManager();
        if($user->getIsAdmin() === true){
            $user->setRoles(['ROLE_ADMIN']);
        } else {
            $user->setRoles(['ROLE_USER']);
        }
        $user->setPassword($passwordEncoder->encodePassword($user, $form['password']->getData()));
        $em->persist($user);
        $em->flush();
        return $this->redirectToRoute( route: 'user_index');
    }

    return $this->render( view: 'usar/new.html.twig', [
        'user' => $user,
        'form' => $form->createView()
    ]);
}
```

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add( child: 'username',  type: TextType::class,  [
            'label' => 'Nombre: '
        ])
        ->add( child: 'password',  type: PasswordType::class,  [
            'label' => 'Contrasena: '
        ])
        ->add( child: 'isAdmin',  type: CheckboxType::class, [
            'label' => '¿Administrador? ',
            'required' => false,
        ])
        ->add( child: 'photo',  type: FileType::class, [
            'label' => 'Selecciona una imagen:',
            'mapped' => false,
            'required' => false
        ])
    ;
}
```

show

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este trabajador y se los pasa a la plantilla twig y ésta se encarga de mostrarlos.

```php
/**
 * @Route("/{id}", name="user_show", methods={"GET"})
 */
public function show(User $user): Response
{
    return $this->render( view: 'user/show.html.twig', [
        'user' => $user,
    ]);
}
```

edit

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, se encarga de obtener los datos referentes al id de este trabajador y los muestra en el formulario de edición para modificar los datos de este trabajador, al guardar estos datos el formulario es validado, si los datos son correctos se persisten en la base de datos.

```php
/**
 * @Route("/{id}/editar", name="user_edit", methods={"GET","POST"})
 */
public function edit(Request $request, User $user, UserPasswordEncoderInterface $passwordEncoder, SluggerInterface $slugger): Response
{
    $form = $this->createForm( type: UserType::class, $user);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {

        $brochureFile = $form->get('photo')->getData();

        if ($brochureFile) {
            $originalFilename = pathinfo($brochureFile->getClientOriginalName(), flags: PATHINFO_FILENAME);
            // this is needed to safely include the file name as part of the URL
            $safeFilename = $slugger->slug($originalFilename);
            $newFilename = $safeFilename.'-'.uniqid().'.'.$brochureFile->guessExtension();

            // Move the file to the directory where brochures are stored
            try {
                $brochureFile->move(
                    $this->getParameter( name: 'photos_users_directory'),
                    $newFilename
                );
            } catch (FileException $e) {
                throw new \Exception( message: 'Ha ocurrido un error');
            }

            // updates the 'brochureFilename' property to store the PDF file name
            // instead of its contents
            $user->setPhoto($newFilename);
        }


        $em = $this->getDoctrine()->getManager();
        if ($user->getIsAdmin() === true) {
            $user->setRoles(['ROLE_ADMIN']);
        } else {
            $user->setRoles(['ROLE_USER']);
        }
        $user->setPassword($passwordEncoder->encodePassword($user, $form['password']->getData()));
        $em->persist($user);
        $em->flush();
        return $this->redirectToRoute( route: 'user_index');
    }

    return $this->render( view: 'user/edit.html.twig', [
        'user' => $user,
        'form' => $form->createView(),
    ]);
}
```

delete

Esta función en primer lugar crea la ruta de acceso, el nombre de la función para referenciarla en diferentes partes de la aplicación y los métodos de acceso a ella, a continuación, obtiene el id del trabajador que se quiere borrar y accede a la base de datos para eliminar sus datos, una vez borrados se guarda el estado de la base de datos.

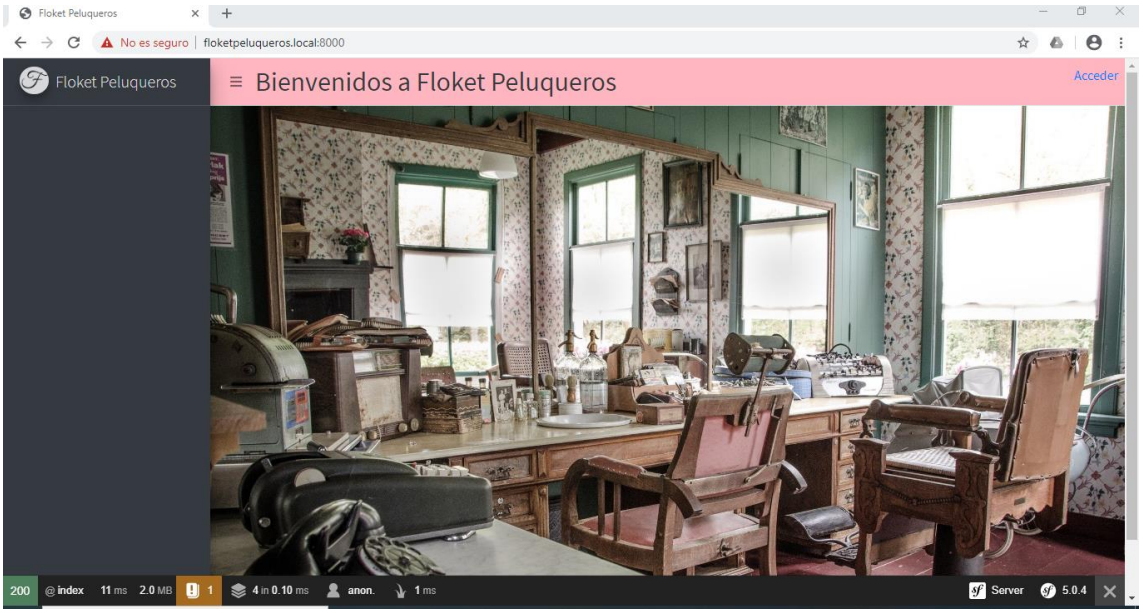
```php
/**
 * @Route("/{id}", name="user_delete", methods={"DELETE"})
 */
public function delete(Request $request, User $user): Response
{
    if ($this->isCsrfTokenValid( id: 'delete'.$user->getId(), $request->request->get( key: '_token'))) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($user);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'user_index');
}
```

4.- Tour por la Aplicación

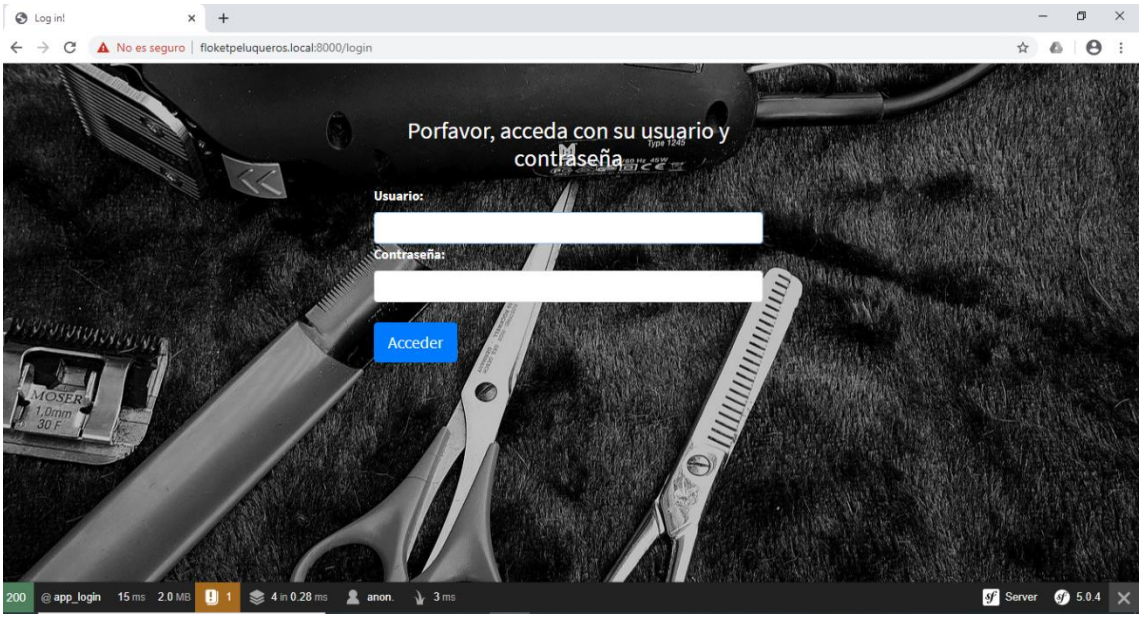
A continuación, voy a hacer una demo de la aplicación mediante capturas de pantalla y explicando todo aquello relevante de cada parte de la aplicación.

4.1.- HomePage y Acceso a la Aplicación

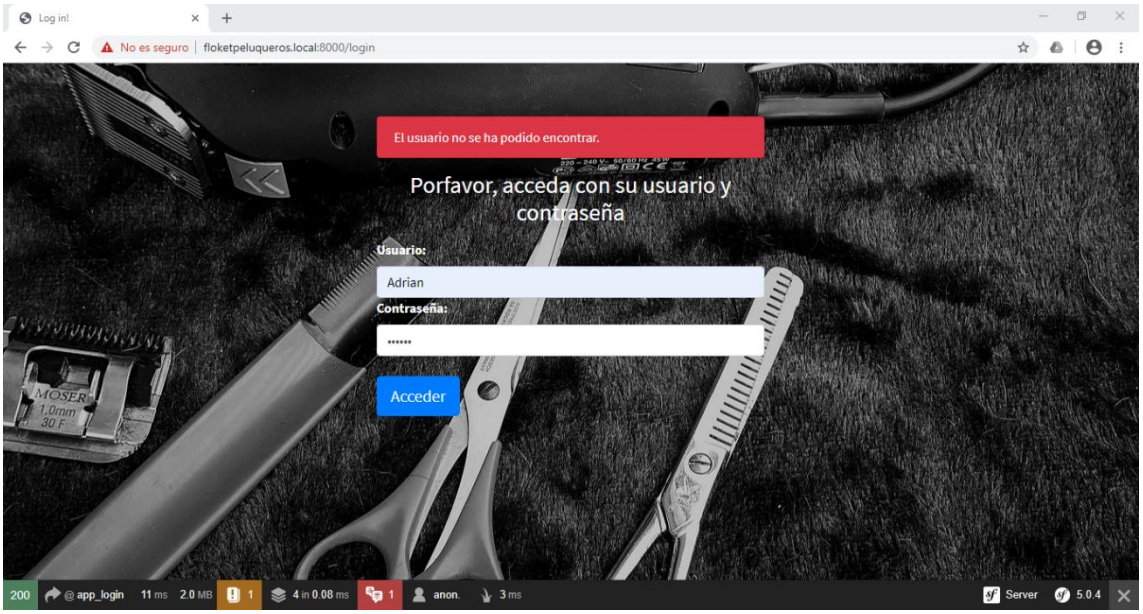
Para acceder a la aplicación, desde cualquier navegador web introducimos el dominio floketpeluqueros.local:8000, a continuación, aparecerá la página principal de la aplicación, también conocida como homepage.



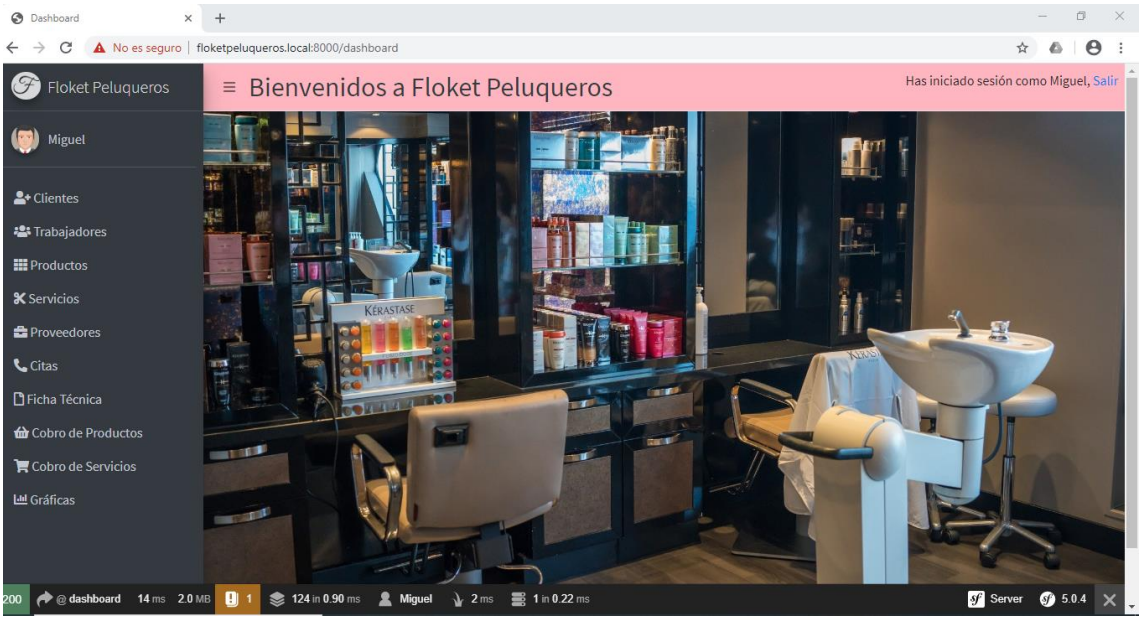
Una vez hayamos accedido a la página principal, podemos acceder a la aplicación mediante el botón **Acceder**, a continuación, aparecerá una página llamada **login** para realizar el acceso mediante usuario y contraseña a la aplicación.



Si se introduce mal el usuario o la contraseña salta un aviso de advertencia.



Según el tipo de usuario que quiera acceder a la aplicación, se le mostrarán unos apartados u otros.

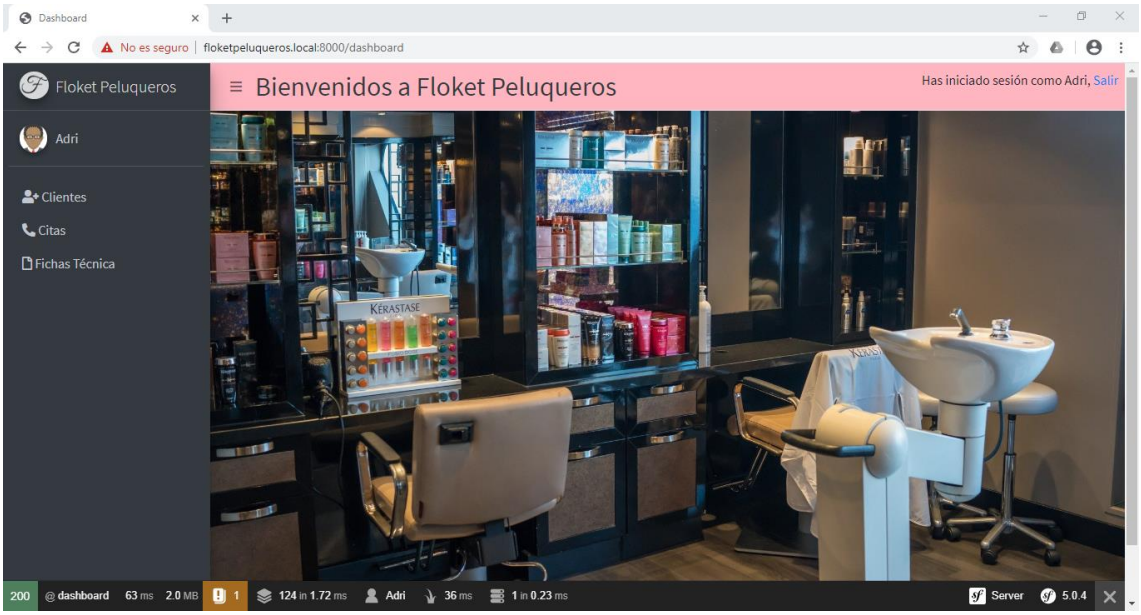


En este caso, hemos accedido con el usuario Miguel que tiene el rol de administrador y puede acceder a todas las partes de la aplicación.

A la izquierda podemos ver el nombre del trabajador que accede junto a su foto, a la derecha también el trabajador que ha iniciado sesión y un enlace para salir de su sesión.

Si pulsamos sobre Salir, el actual usuario sale de su sesión y se vuelve a la página principal de la aplicación.

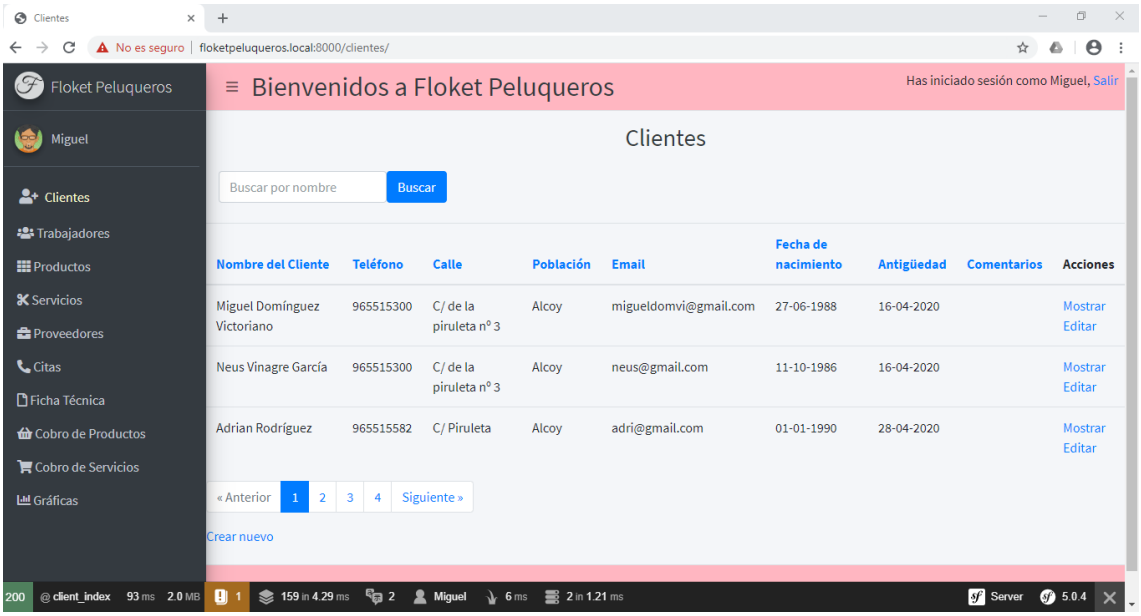
Por el contrario, al acceder con el usuario Adri que tiene el rol de usuario, solamente podemos tener acceso a los apartados imprescindibles para realizar el trabajo, no tiene acceso a los apartados de cobros, ventas o estadísticas.

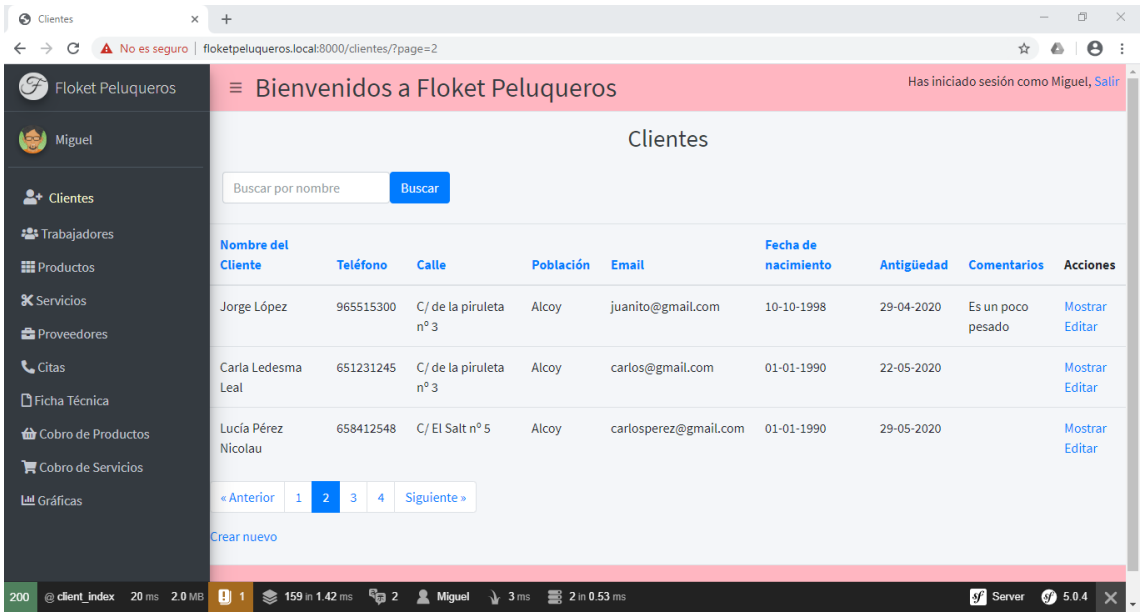


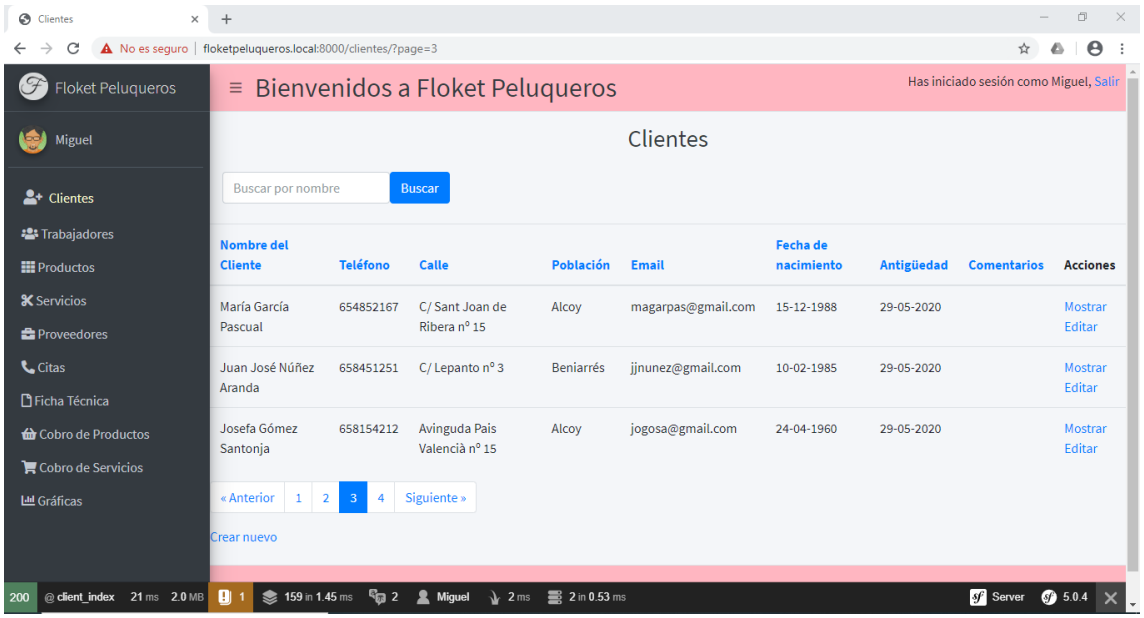
4.2.- Clientes

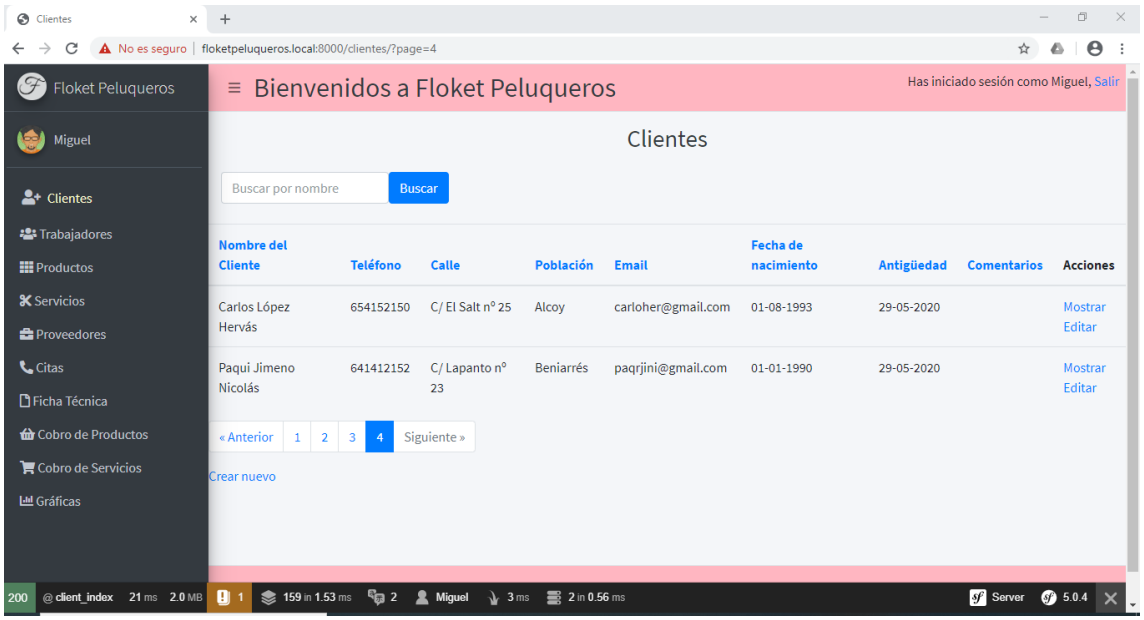
En primer lugar, voy a explicar el apartado referente a los clientes, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a los clientes.

Al acceder a apartado **Clientes**, vemos un **listado** con todos los **clientes** de nuestra aplicación paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.



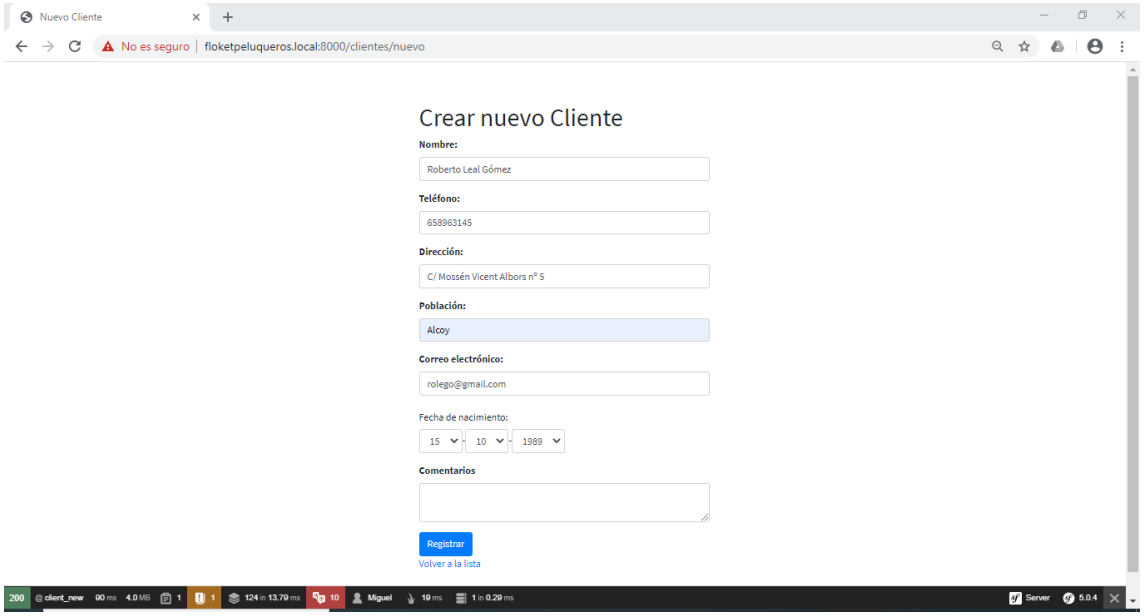

Clientes
No es seguro | floketpeluqueros.local:8000/clientes/?page=2
Floket Peluqueros
Bienvenidos a Floket Peluqueros
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Clientes
Buscar por nombre
Buscar
Nombre del Cliente | Teléfono | Calle | Población | Email | Fecha de nacimiento | Antigüedad | Comentarios | Acciones
Jorge López | 965515300 | C/ de la piruleta nº 3 | Alcoy | juanito@gmail.com | 10-10-1998 | 29-04-2020 | Es un poco pesado | Mostrar Editar
Carla Ledesma Leal | 651231245 | C/ de la piruleta nº 3 | Alcoy | carlos@gmail.com | 01-01-1990 | 22-05-2020 | Mostrar Editar
Lucía Pérez Nicolau | 658412548 | C/ El Salt nº 5 | Alcoy | carlosperez@gmail.com | 01-01-1990 | 29-05-2020 | Mostrar Editar
« Anterior 1 2 3 4 Siguiente »
Crear nuevo
200 @client_index 20 ms 2.0 MB 1 159 in 1.42 ms 2 Miguel 3 ms 2 in 0.53 ms Server 5.0.4


Clientes
No es seguro | floketpeluqueros.local:8000/clientes/?page=3
Floket Peluqueros
Bienvenidos a Floket Peluqueros
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Clientes
Buscar por nombre
Buscar
Nombre del Cliente | Teléfono | Calle | Población | Email | Fecha de nacimiento | Antigüedad | Comentarios | Acciones
María García Pascual | 654852167 | C/ Sant Joan de Ribera nº 15 | Alcoy | magarpas@gmail.com | 15-12-1988 | 29-05-2020 | Mostrar Editar
Juan José Núñez Aranda | 658451251 | C/ Lepanto nº 3 | Beniarrés | jjnunez@gmail.com | 10-02-1985 | 29-05-2020 | Mostrar Editar
Josefa Gómez Santonja | 658154212 | Avinguda País Valencià nº 15 | Alcoy | jogosa@gmail.com | 24-04-1960 | 29-05-2020 | Mostrar Editar
« Anterior 1 2 3 4 Siguiente »
Crear nuevo
200 @client_index 21 ms 2.0 MB 1 159 in 1.45 ms 2 Miguel 2 ms 2 in 0.53 ms Server 5.0.4

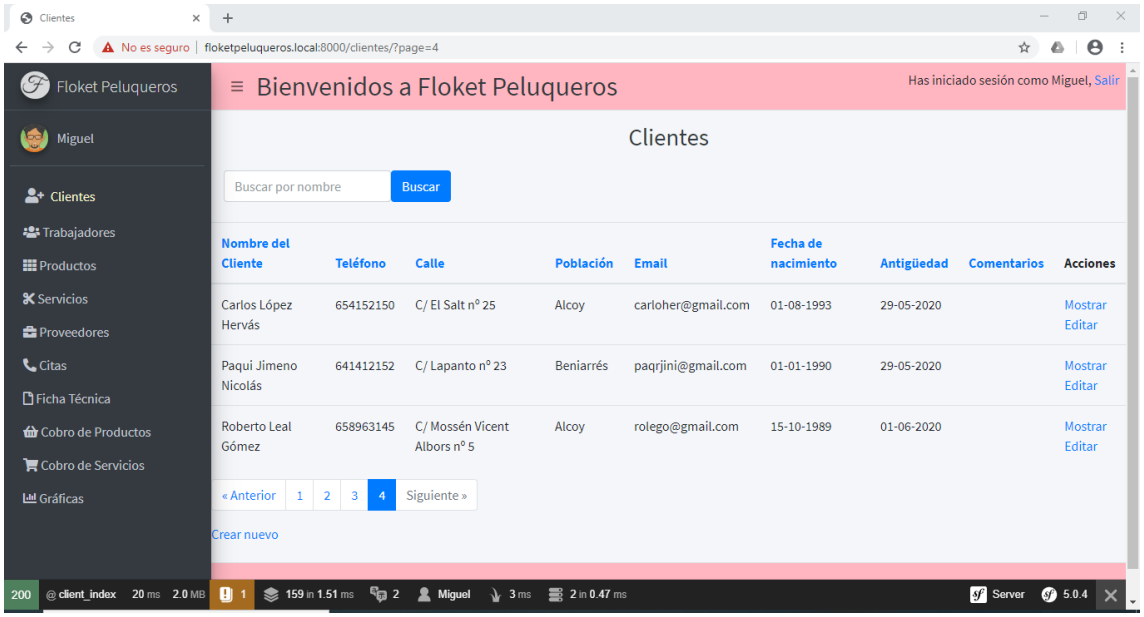

Clientes
No es seguro | floketpeluqueros.local:8000/clientes/?page=4
Floket Peluqueros
Bienvenidos a Floket Peluqueros
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Clientes
Buscar por nombre
Buscar
Nombre del Cliente | Teléfono | Calle | Población | Email | Fecha de nacimiento | Antigüedad | Comentarios | Acciones
Carlos López Hervás | 654152150 | C/ El Salt nº 25 | Alcoy | carloher@gmail.com | 01-08-1993 | 29-05-2020 | Mostrar Editar
Paqui Jimeno Nicolás | 641412152 | C/ Lapanto nº 23 | Beniarrés | paqrjini@gmail.com | 01-01-1990 | 29-05-2020 | Mostrar Editar
« Anterior 1 2 3 4 Siguiente »
Crear nuevo
200 @client_index 21 ms 2.0 MB 1 159 in 1.53 ms 2 Miguel 3 ms 2 in 0.56 ms Server 5.0.4

Podemos observar que tenemos 4 páginas de clientes donde podemos ver todos sus datos para posteriormente trabajar con ellos.

Para crear un **nuevo cliente** debemos pulsar sobre Crear nuevo y nos aparecerá una nueva ventana para la creación del nuevo cliente.

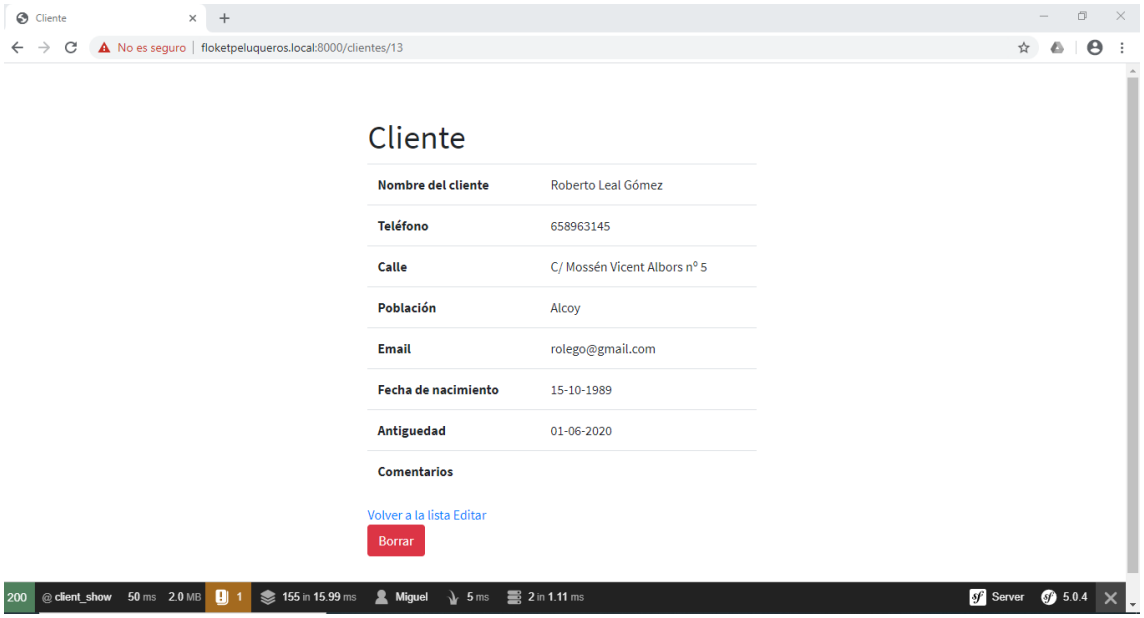


Al introducir todos los datos del nuevo cliente correctamente podemos crearlo al presionar el botón **Registrar**.

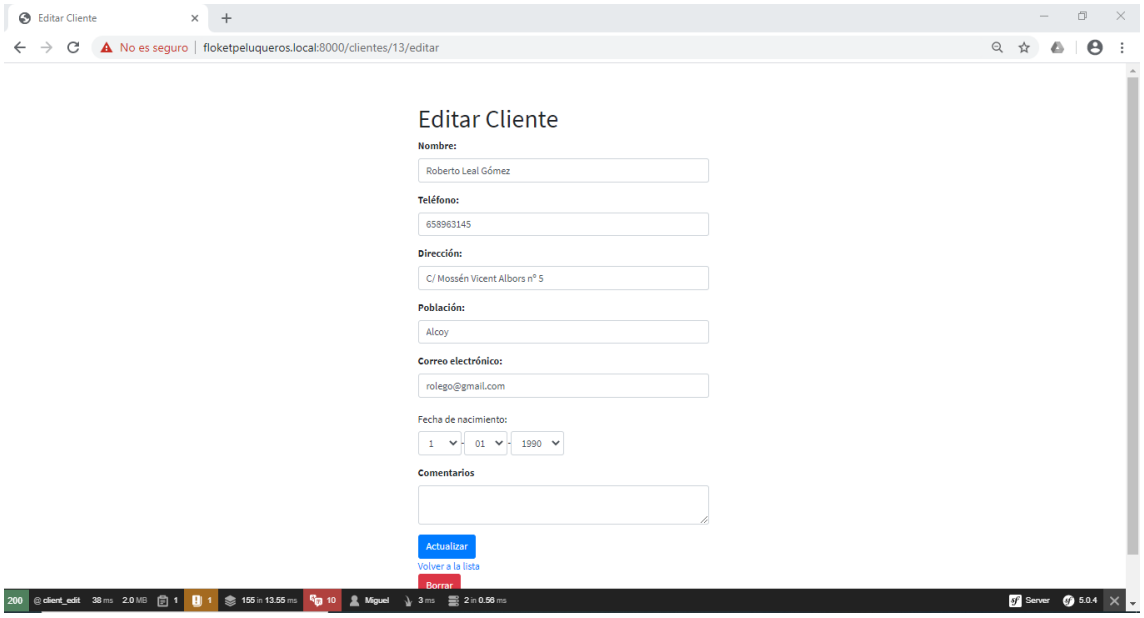


Ya nos aparece el nuevo cliente en el listado.

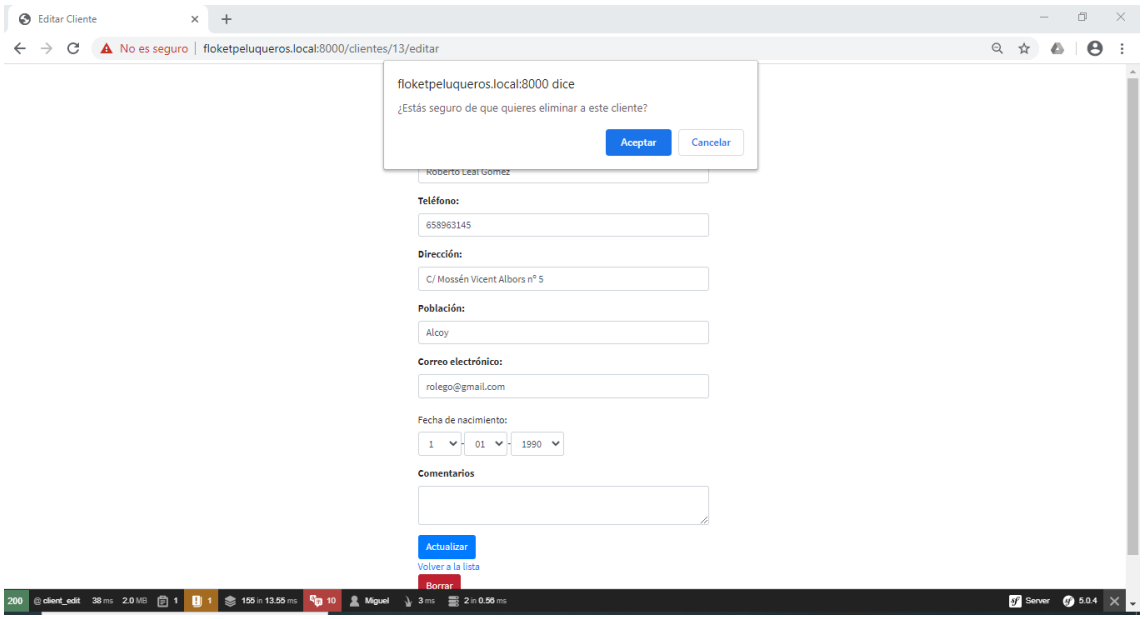
También podemos **ver** los datos del cliente en otra ventana de forma más organizada si presionamos sobre **Mostrar** en el apartado de **Acciones**.

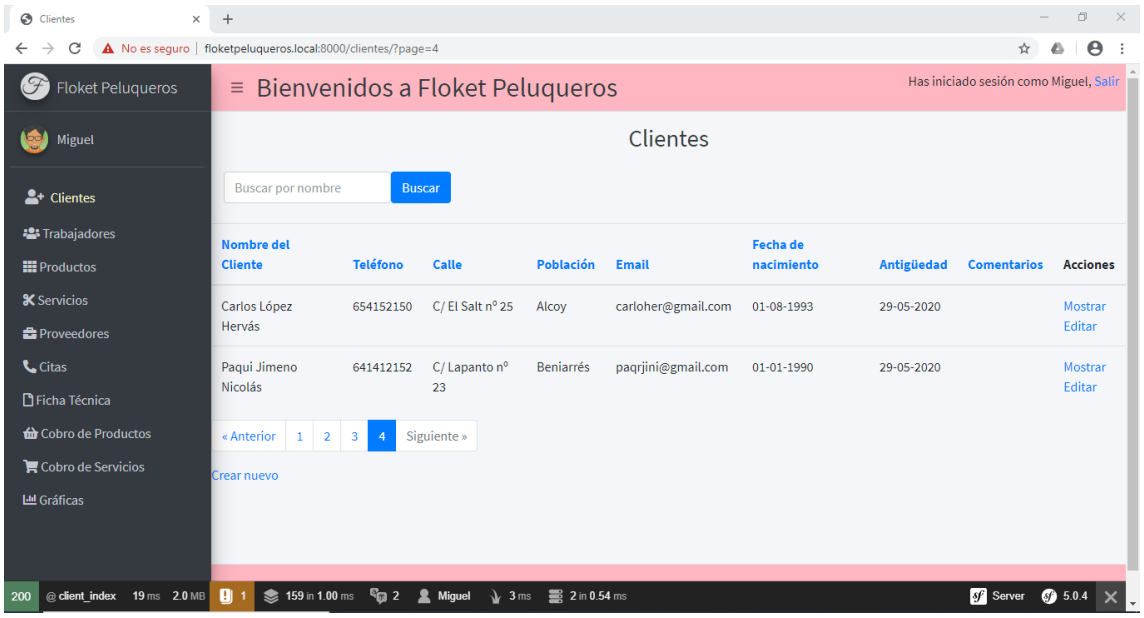


Si por la razón que sea, hemos introducido mal un dato del cliente, lo podemos **modificar** en el apartado de **Acciones** con el botón **Editar**.



Incluso si queremos **eliminar** al **cliente** podemos eliminarlo mediante el botón **Borrar**, al presionar el botón Borrar nos aparecerá un cuadro de diálogo preguntándonos si realmente queremos eliminar a este cliente.

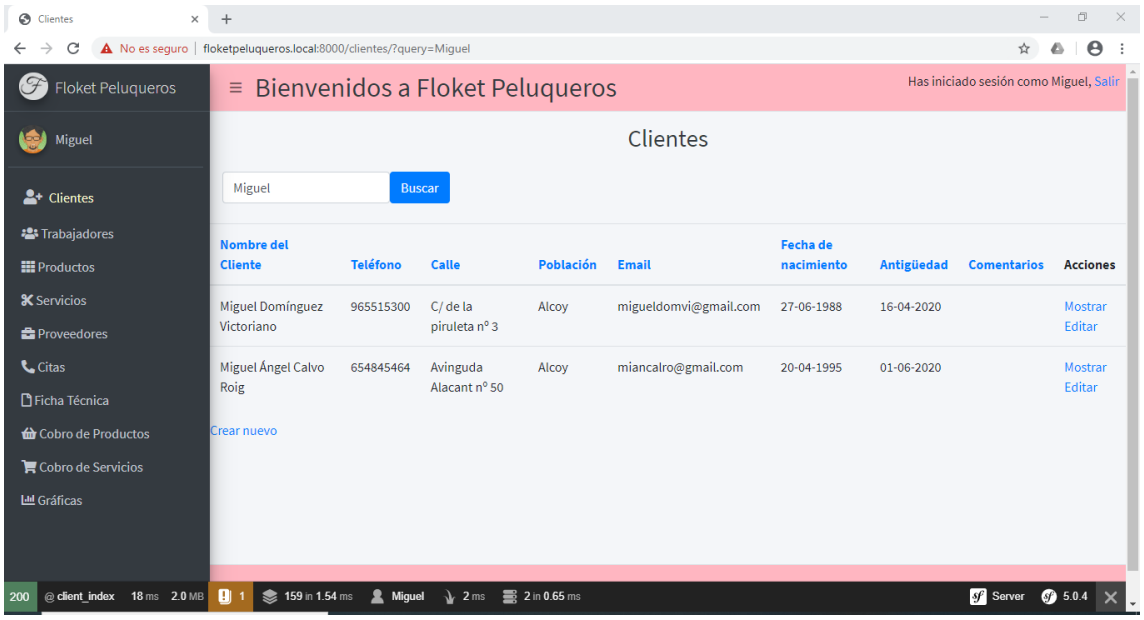


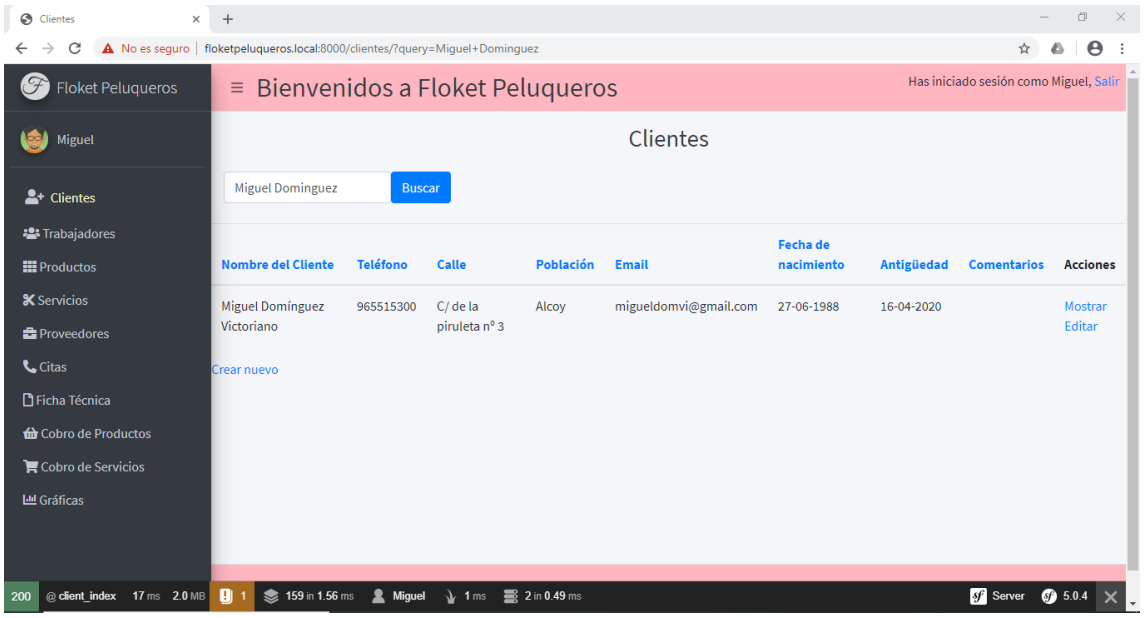


Al borrar al cliente, realmente podemos ver que ya no aparece en el listado de clientes.

Si el listado de clientes se hace muy grande y queremos buscar a un cliente en particular, podemos realizar una **búsqueda** por el **nombre** del cliente.

Si al realizar la búsqueda por el nombre, cabe la posibilidad de que aparezcan varios resultados, para ello podemos realizar la búsqueda por nombre y apellido para acotar la búsqueda; el buscador discrimina los acentos, las mayúsculas y las minúsculas.


Clientes
No es seguro | floketpeluqueros.local:8000/clientes/?query=Miguel
Floket Peluqueros
Bienvenidos a Floket Peluqueros
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Clientes
Miguel
Buscar
Nombre del Cliente
Teléfono
Calle
Población
Email
Fecha de nacimiento
Antigüedad
Comentarios
Acciones
Miguel Domínguez Victoriano
965515300
C/ de la piruleta nº 3
Alcoy
migueldomvi@gmail.com
27-06-1988
16-04-2020
Mostrar
Editar
Miguel Ángel Calvo Roig
654845464
Avinguda Alacant nº 50
Alcoy
miancalro@gmail.com
20-04-1995
01-06-2020
Mostrar
Editar
Crear nuevo
200 @ client_index 18 ms 2.0 MB 1 159 in 1.54 ms Miguel 2 ms 2 in 0.65 ms Server 5.0.4

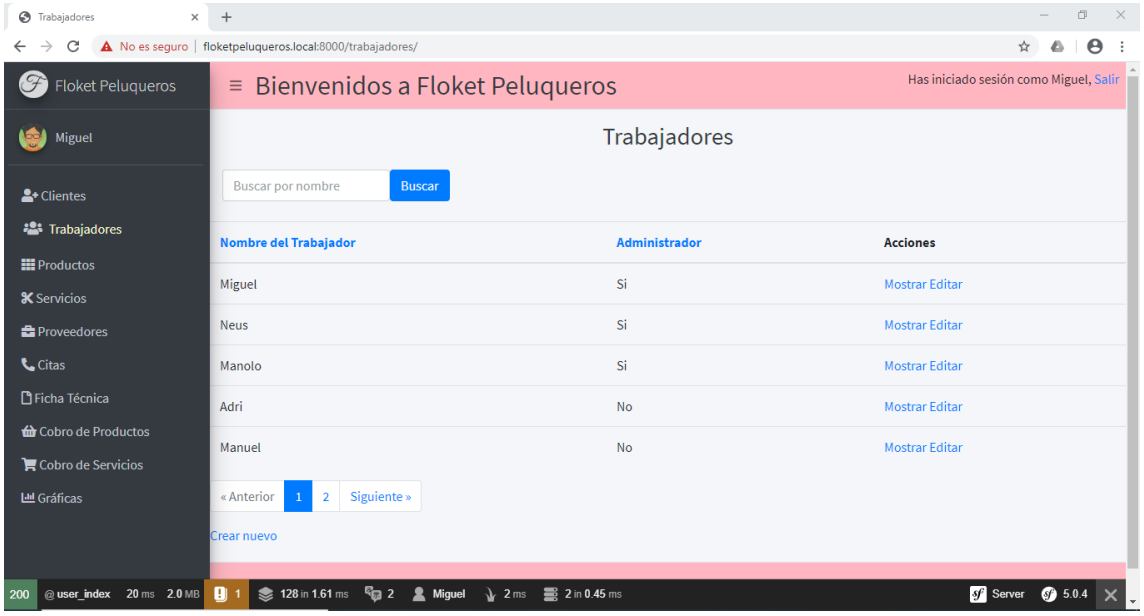

Clientes
No es seguro | floketpeluqueros.local:8000/clientes/?query=Miguel+Dominguez
Floket Peluqueros
Bienvenidos a Floket Peluqueros
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Clientes
Miguel Dominguez
Buscar
Nombre del Cliente
Teléfono
Calle
Población
Email
Fecha de nacimiento
Antigüedad
Comentarios
Acciones
Miguel Domínguez Victoriano
965515300
C/ de la piruleta nº 3
Alcoy
migueldomvi@gmail.com
27-06-1988
16-04-2020
Mostrar
Editar
Crear nuevo
200 @ client_index 17 ms 2.0 MB 1 159 in 1.56 ms Miguel 1 ms 2 in 0.49 ms Server 5.0.4

4.3.- Trabajadores

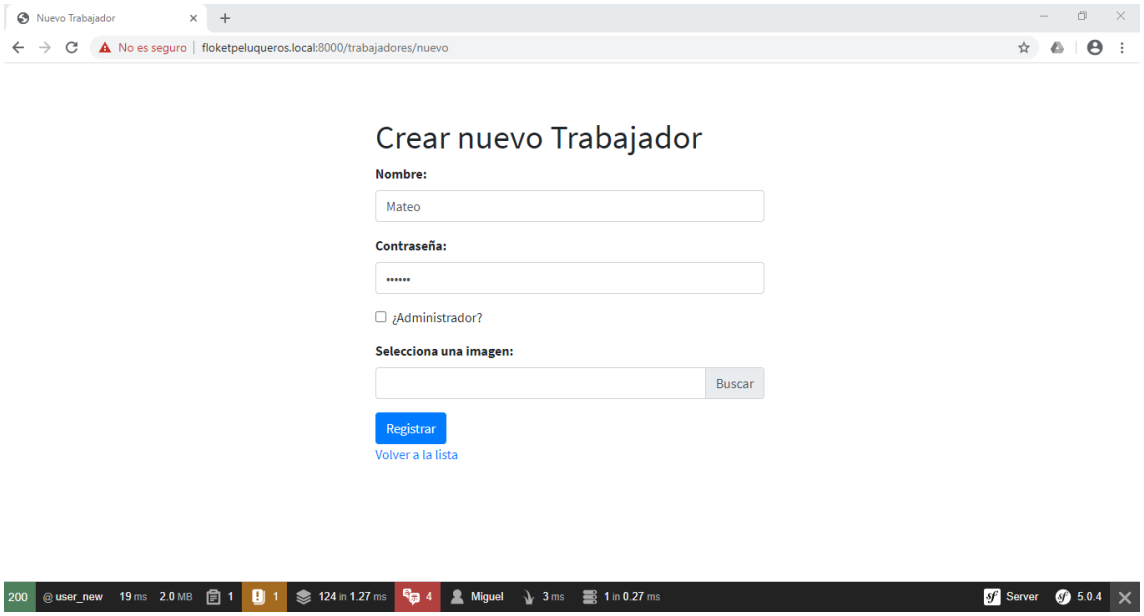
Ahora, voy a explicar el apartado referente a los trabajadores, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a los trabajadores.

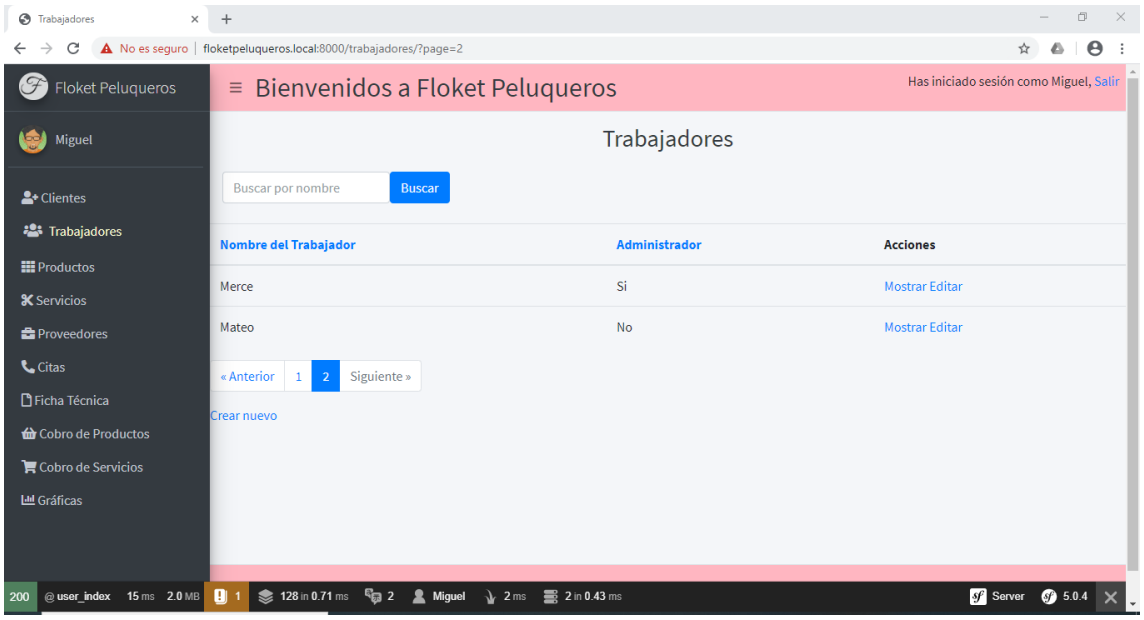
Al acceder a apartado **Trabajadores**, vemos un **listado** con todos los **trabajadores** de nuestra aplicación paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.

El procedimiento es el mismo que en el apartado de clientes, por tanto, no voy a realizar lo mismo, voy a mostrar lo relevante en este apartado de la aplicación.

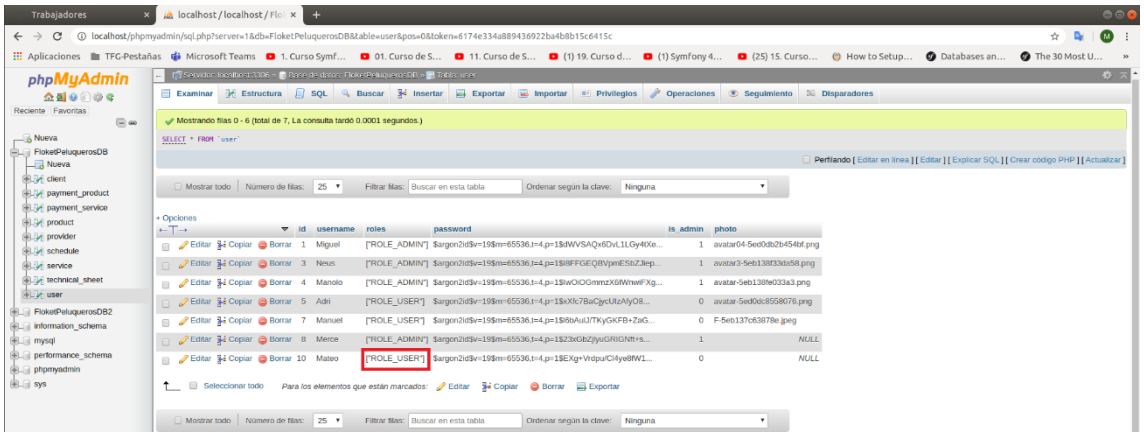


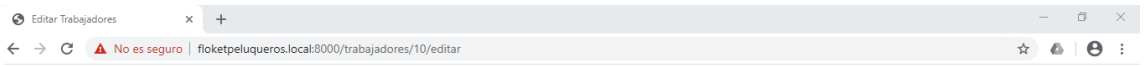
Al **crear** un nuevo **trabajador**, tenemos que decidir el tipo de trabajador a crear, si será un trabajador con rol de usuario o un trabajador con rol de administrador.





Al **crear** un nuevo **trabajador** con rol de usuario, este solamente tendría acceso a unos pocos apartados de la aplicación, pero si cambiamos el rol a administrador, tendrá acceso a todos los paneles de la aplicación.





Editar Trabajadores

Nombre:

Mateo

Contraseña:

••••••

¿Administrador?

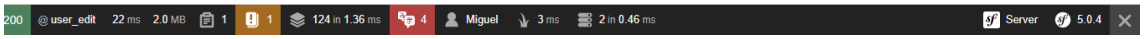
Selecciona una imagen:

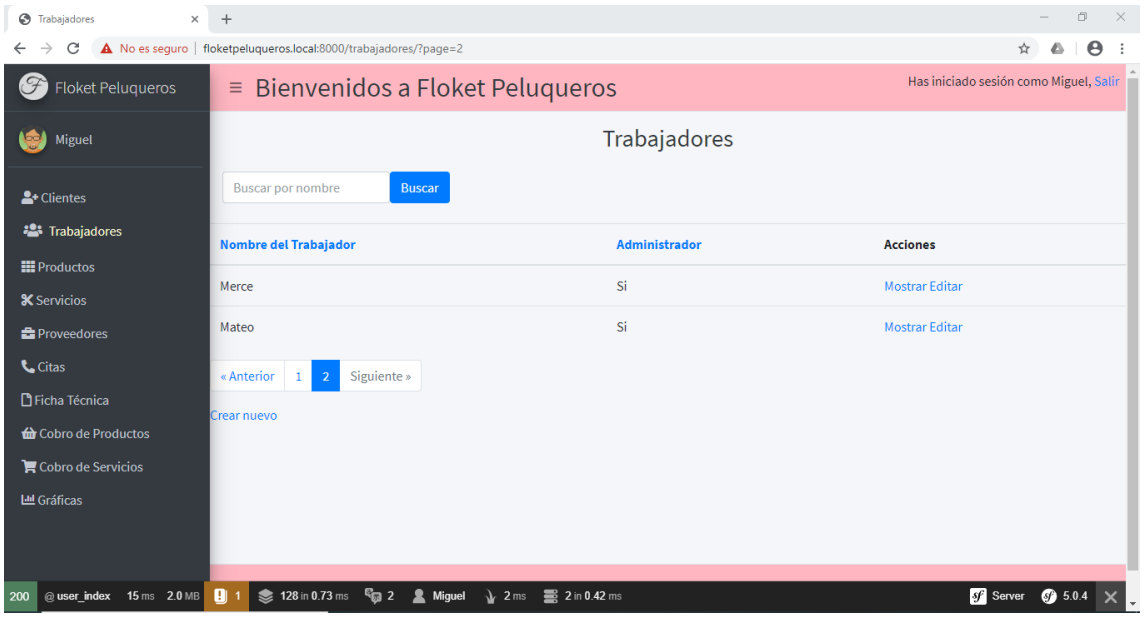
Buscar

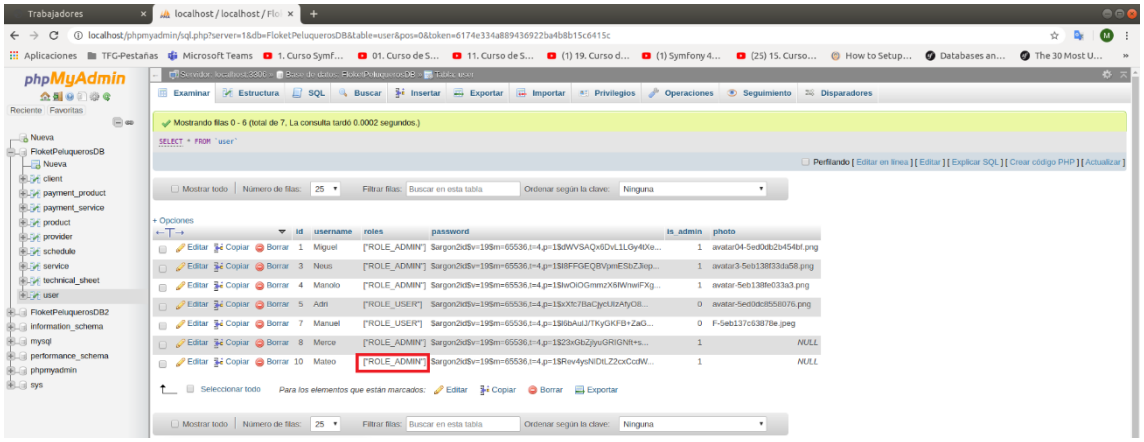
Actualizar

Volver a la lista

Borrar



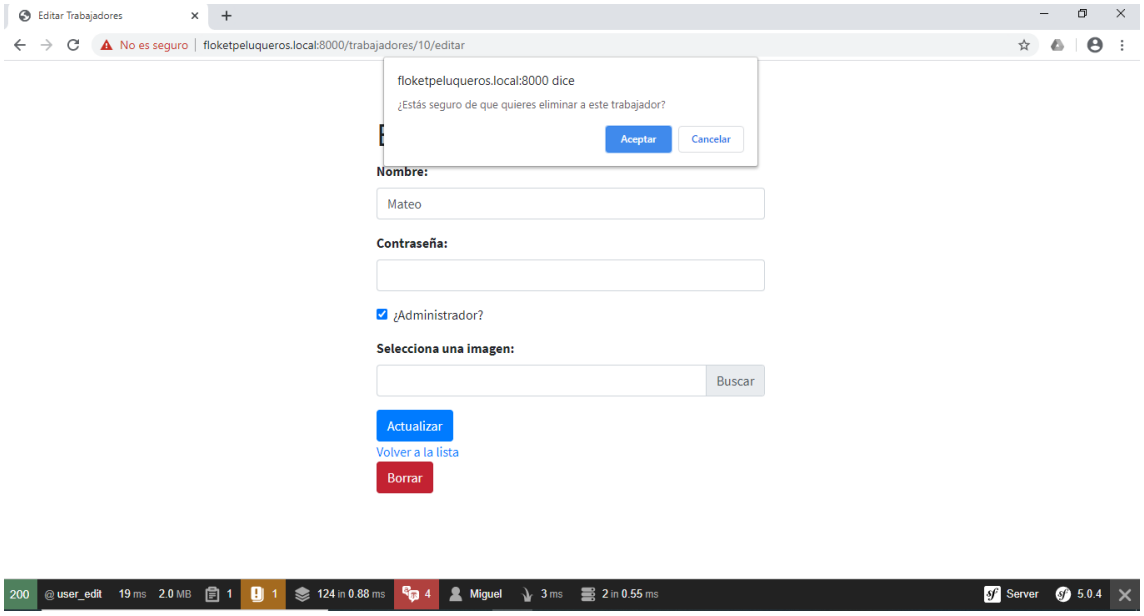




Vemos que, al **modificar** el rol del usuario, realmente ese cambio se está realizando en el registro de ese trabajador dentro de la base de datos.

También podemos **buscar** a un trabajador por su nombre para ver sus datos.

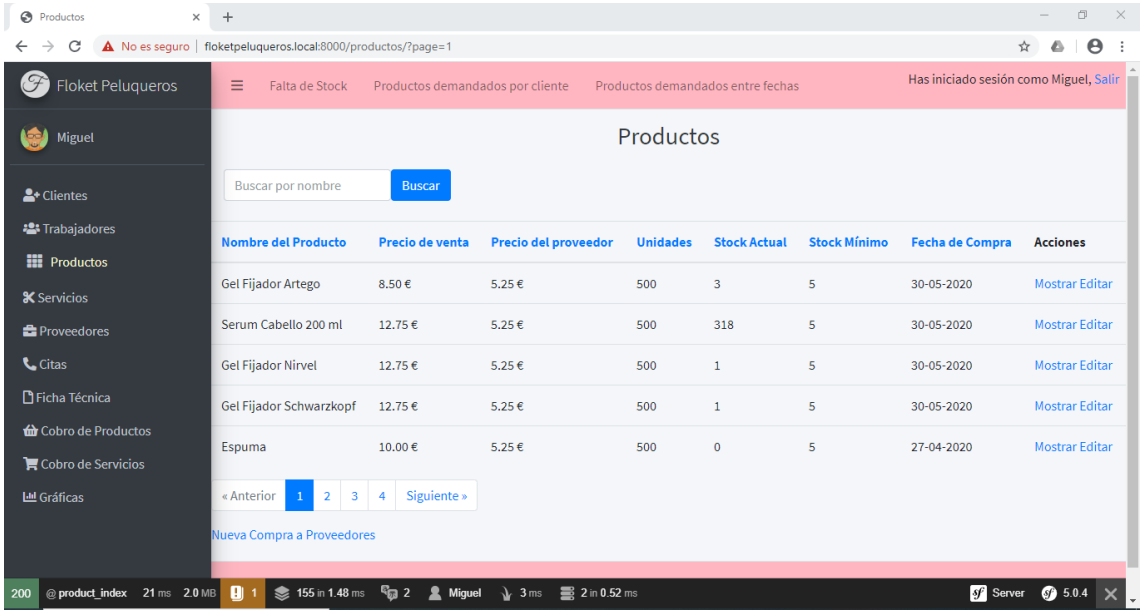
Y por último, podemos **borrar** a este trabajador, cuando se le haya terminado el contrato o se le haya despedido.

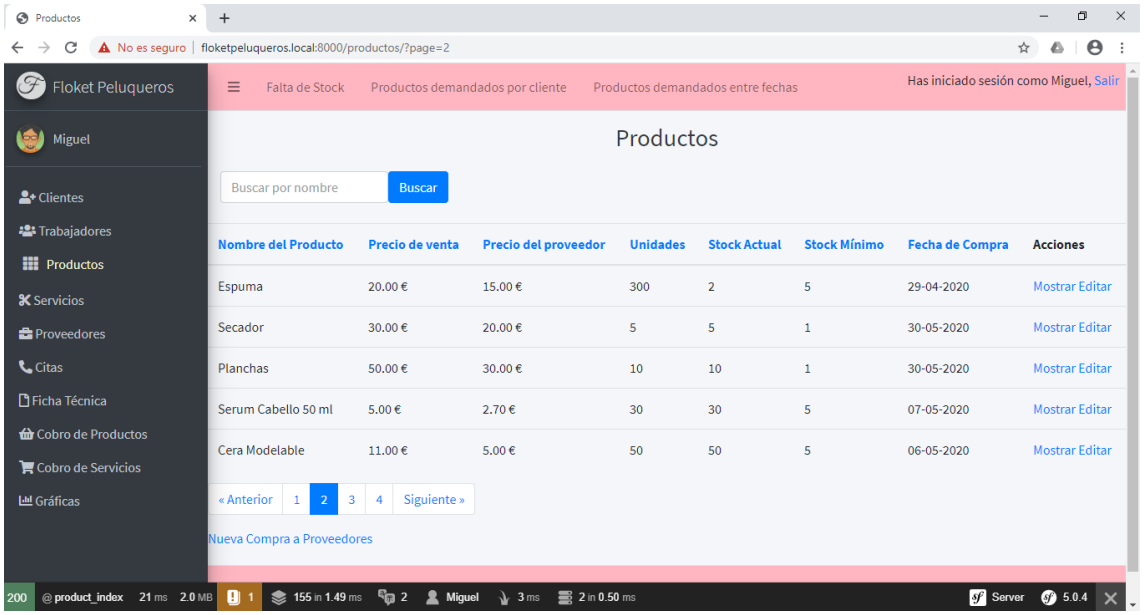


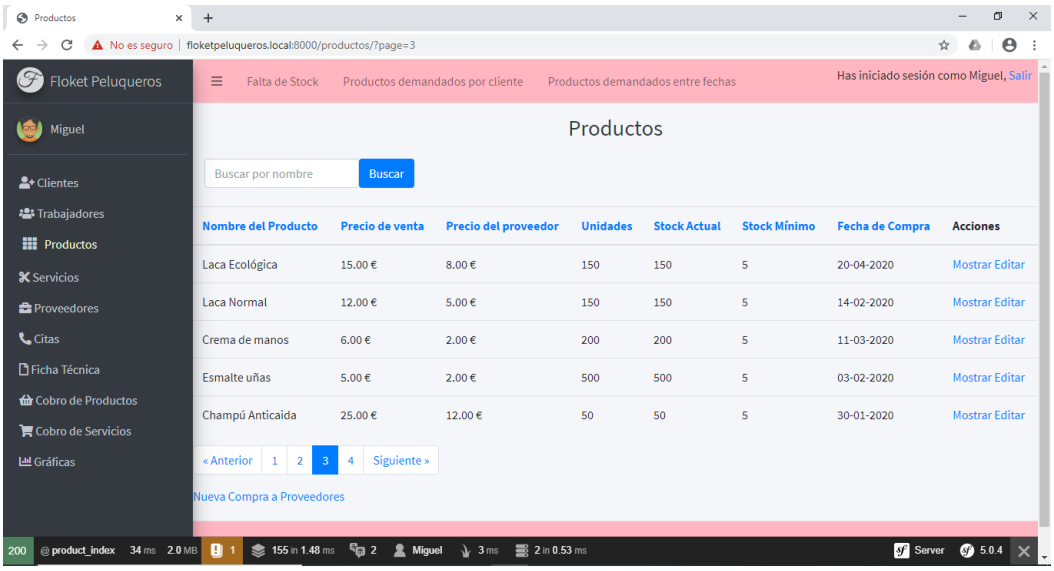
4.4.- Productos

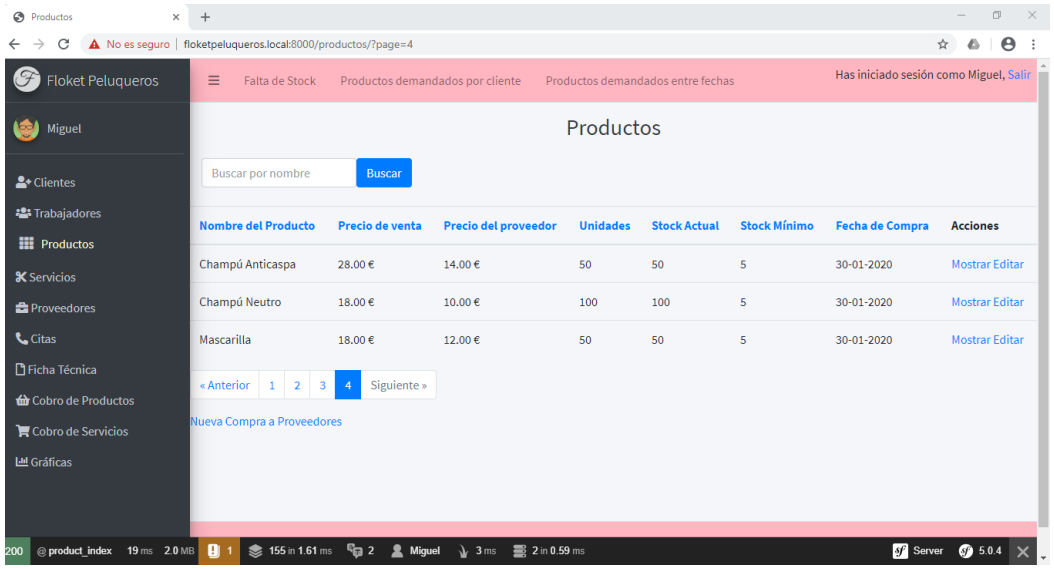
A continuación, voy a explicar el apartado referente a los productos, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a los productos.

Al acceder a apartado **Productos**, vemos un **listado** con todos los **productos** que se venden en el salón de belleza paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.

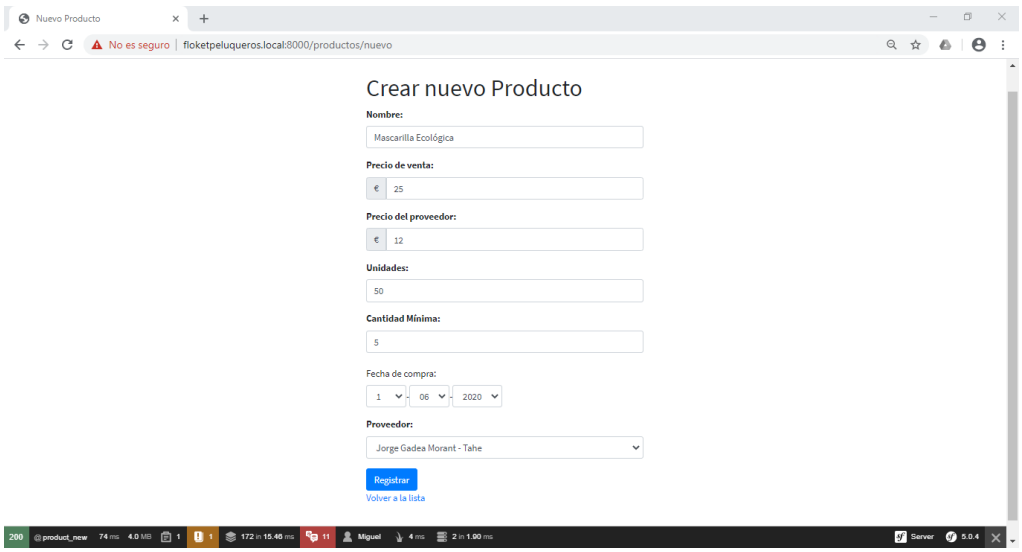




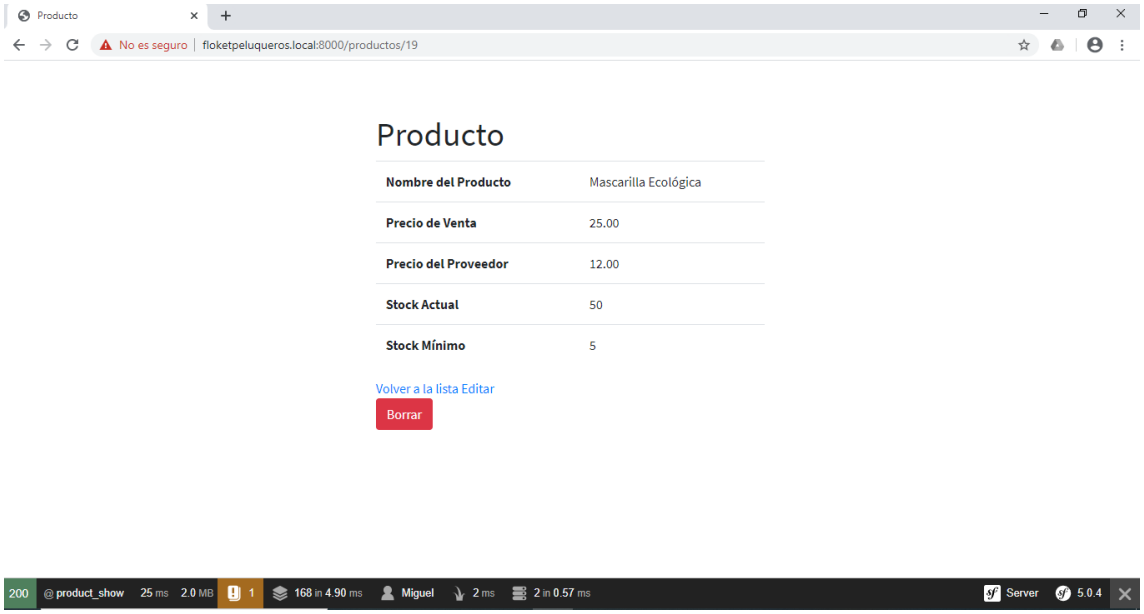


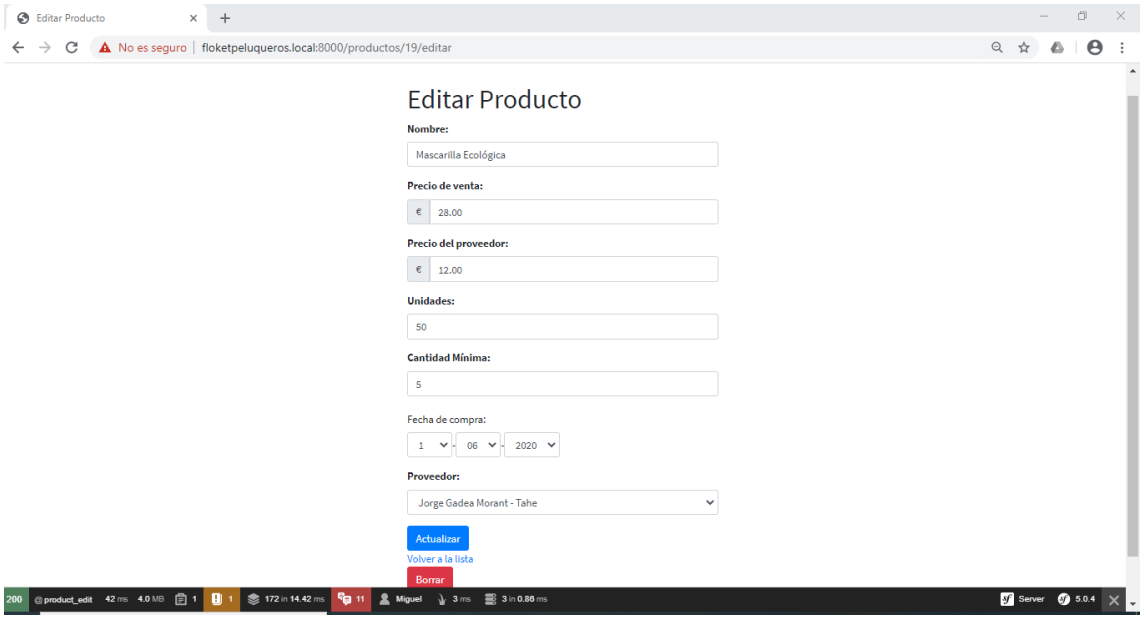


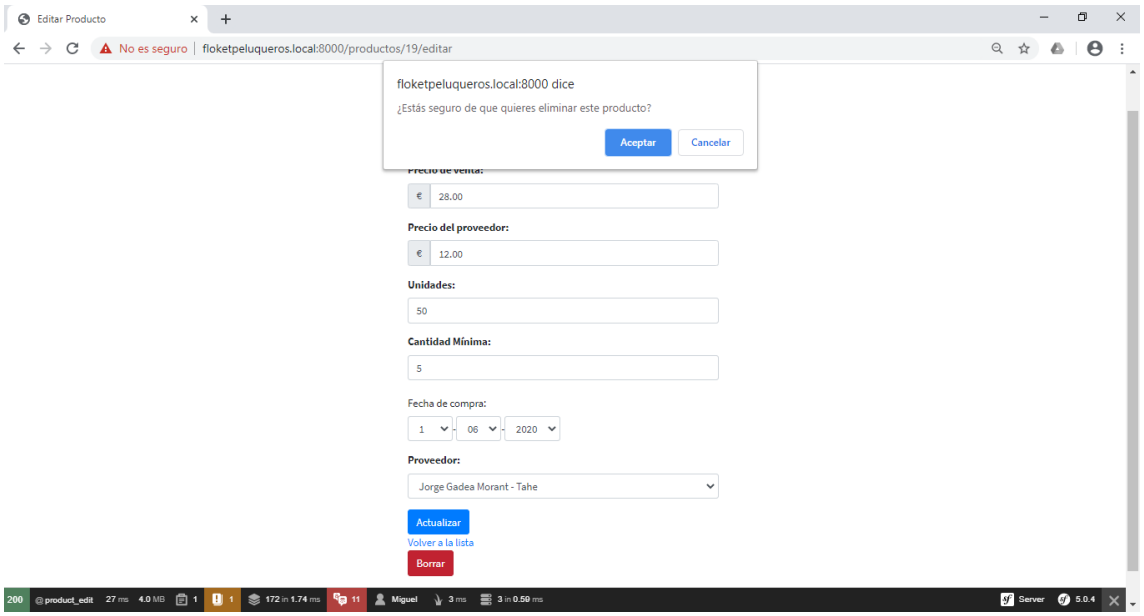
Para crear nuevos productos, evidentemente primero hay que realizar un pedido al proveedor, en este caso para hay que acceder a Nueva Compra a Proveedores para adquirir los nuevos productos.



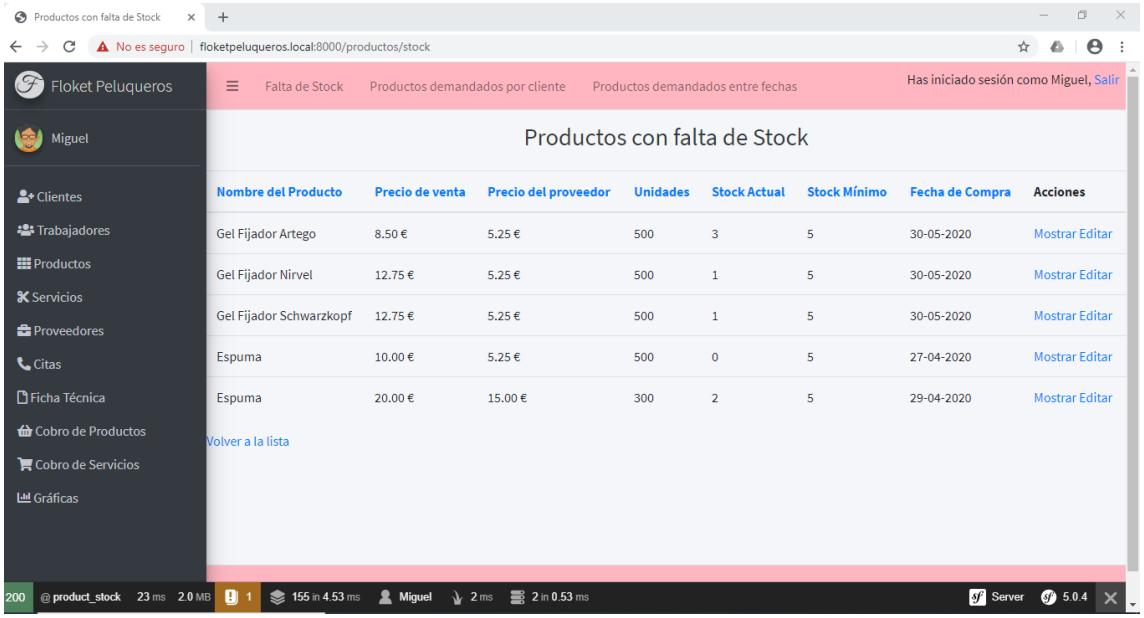
Podemos **Mostrar** el producto accediento al apartado de Acciones en el panel principal de los productos, también podemos modificar algún dato del producto, por ejemplo si queremos subir o bajar su precio y también borrar un producto.



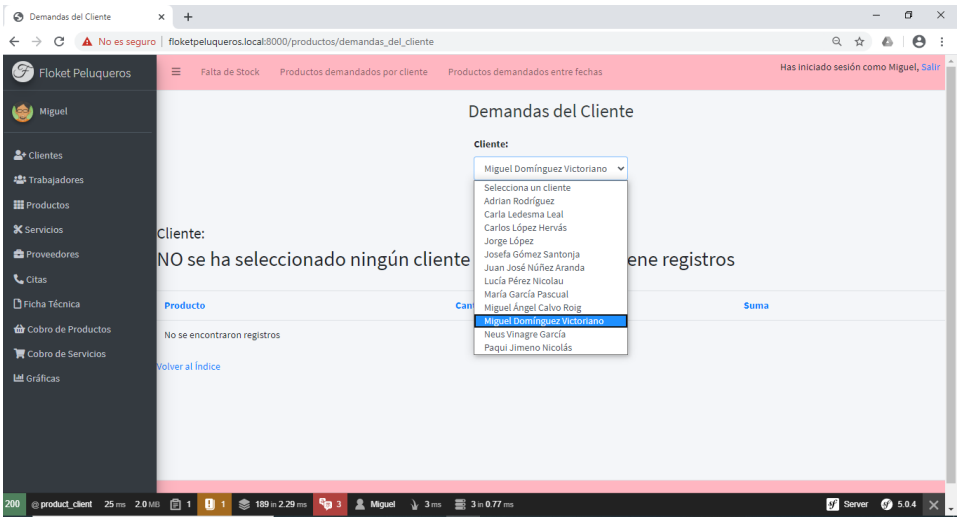


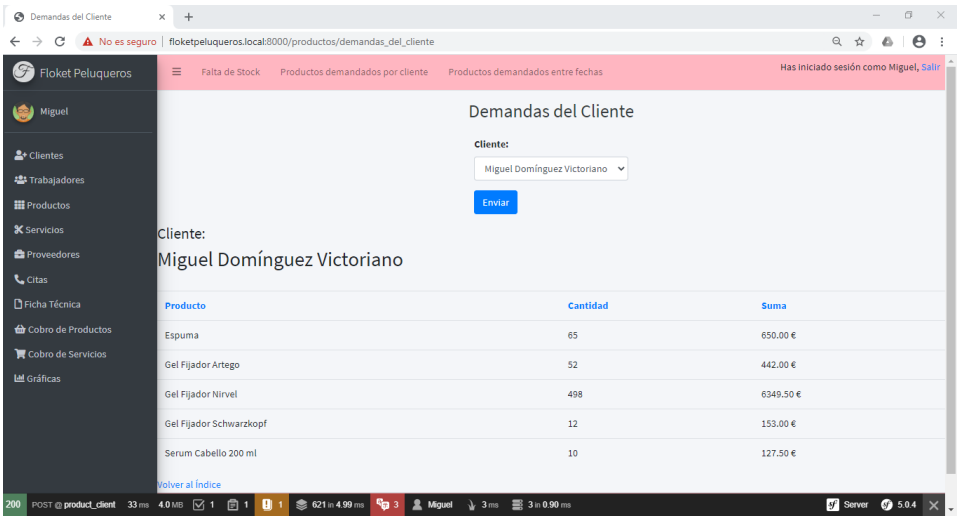


También podemos ver los productos que tenemos con poco o sin stock accediendo a la pestaña **Falta de stock**, esta pestaña muestra un listado de los productos que deberíamos pedir a los proveedores si lo vemos necesario.



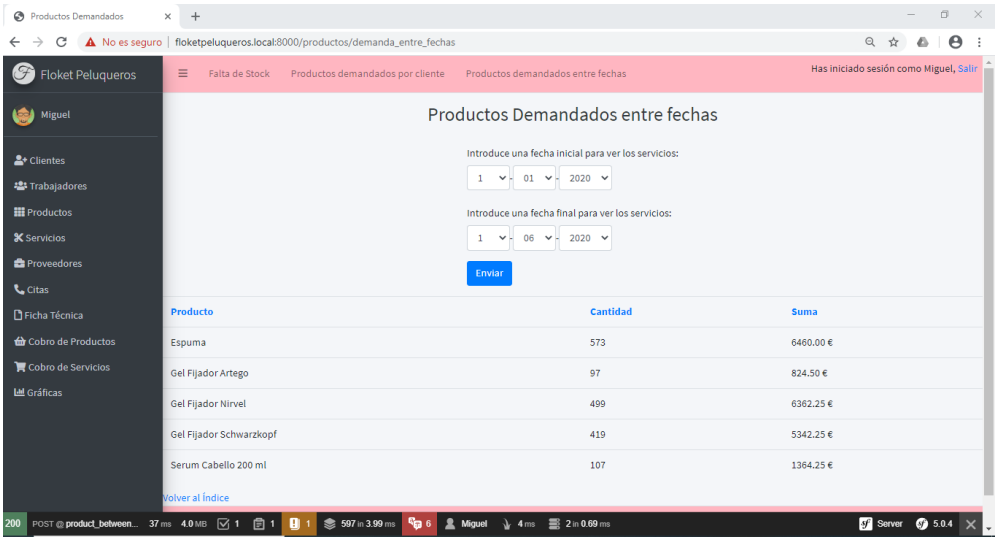
Para saber los productos que ha comprado un cliente, se puede acceder a la pestaña **Productos demandados por cliente**, en esta pestaña podemos seleccionar el nombre de un cliente desde el formulario desplegable, posteriormente nos aparecerá un listado con todos los productos que ha comprado el cliente seleccionado.



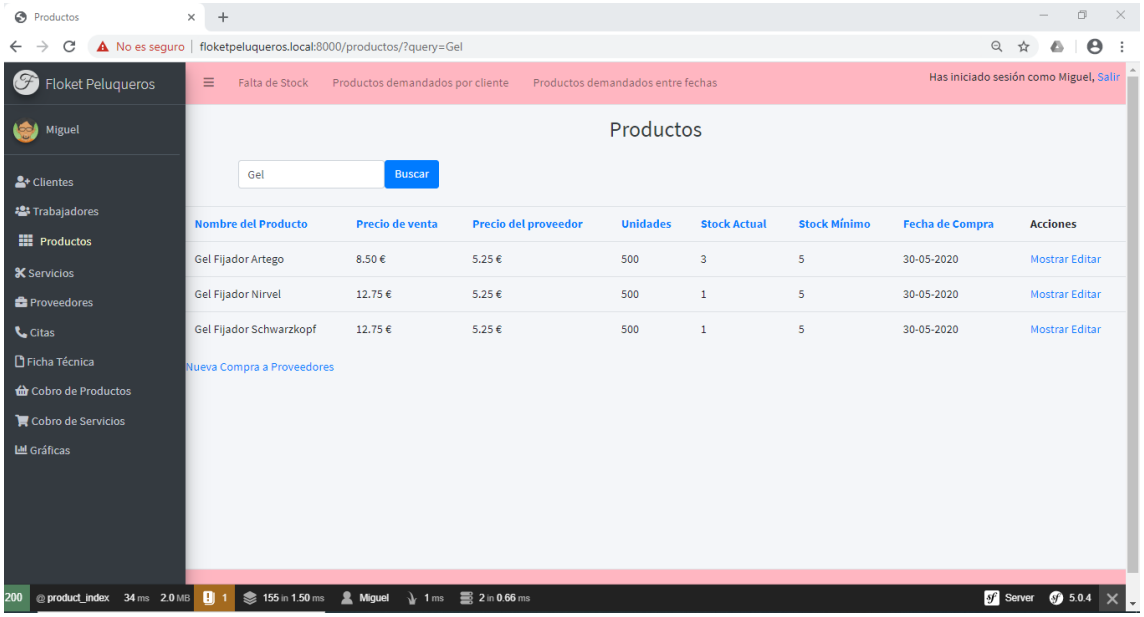


También podemos saber los productos que han sido comprados en un rango de fechas, para así poder prever las necesidades de stock que tendremos en un rango de fechas similar.

Por ejemplo, desde principios de año hasta ahora podemos ver todos los productos que se han comprado en el salón de belleza seleccionado estas fechas.



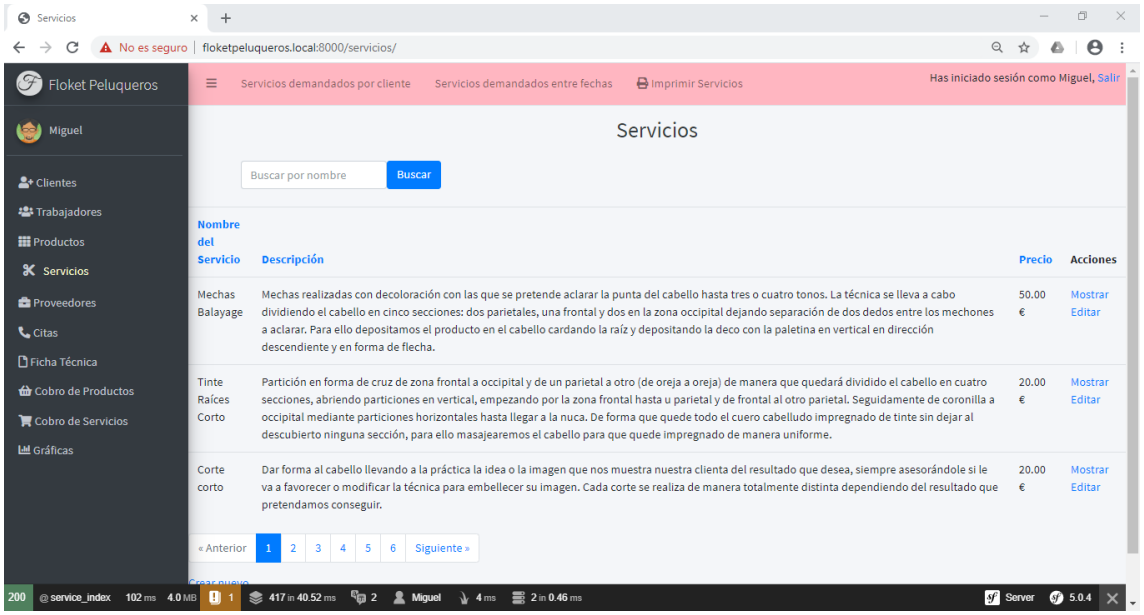
En el panel principal de los productos también existe un buscador para poder buscar los diferentes productos por su nombre.

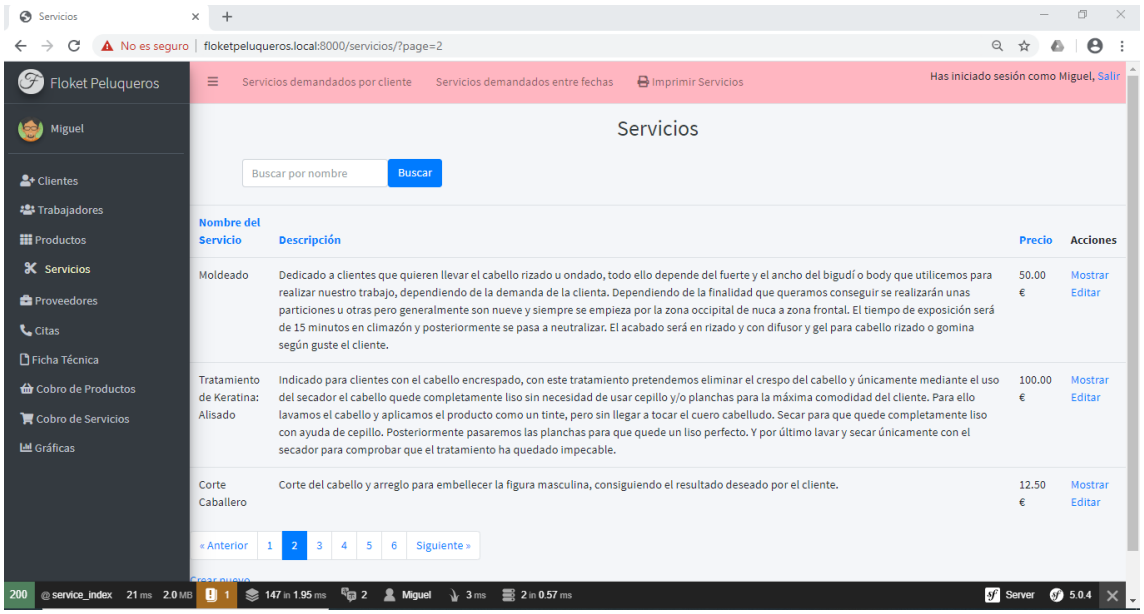


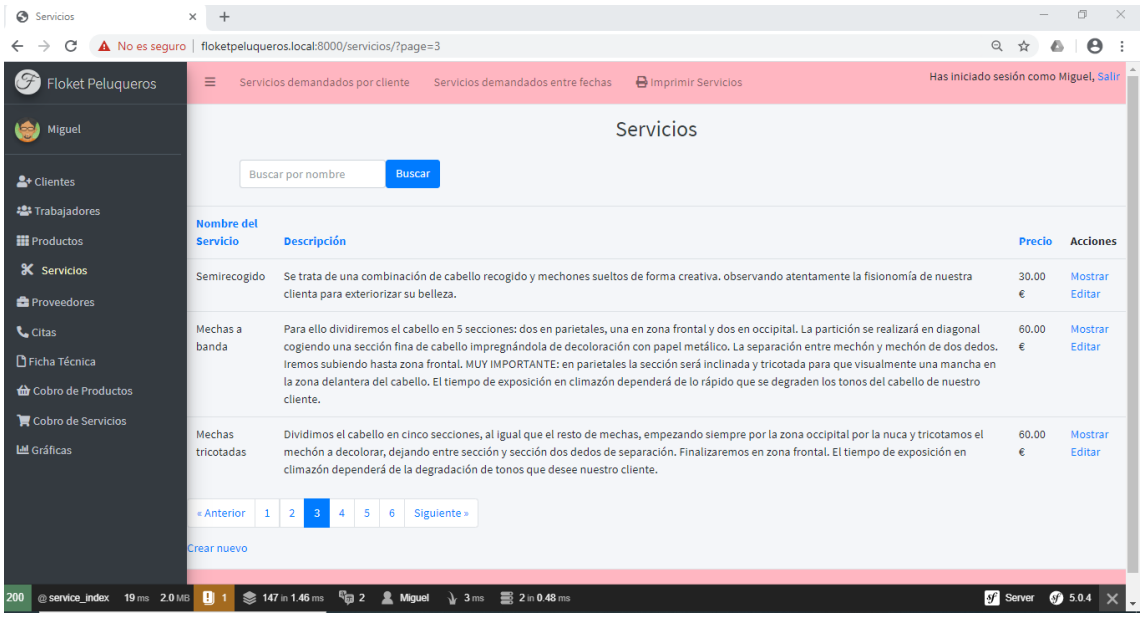
4.5.- Servicios

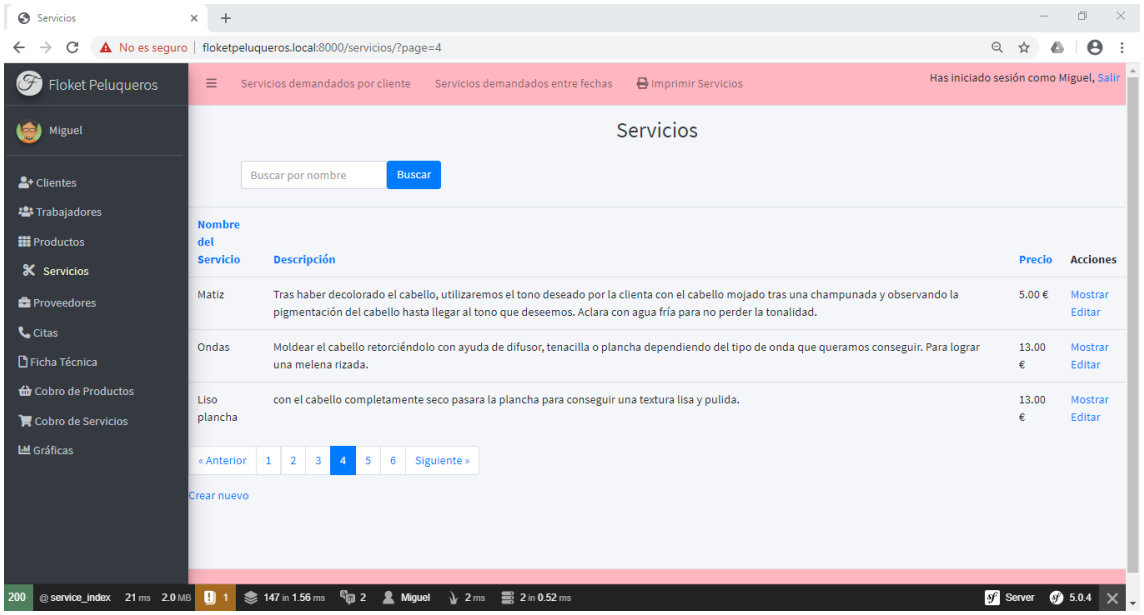
A continuación, voy a explicar el apartado referente a los servicios, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a los servicios.

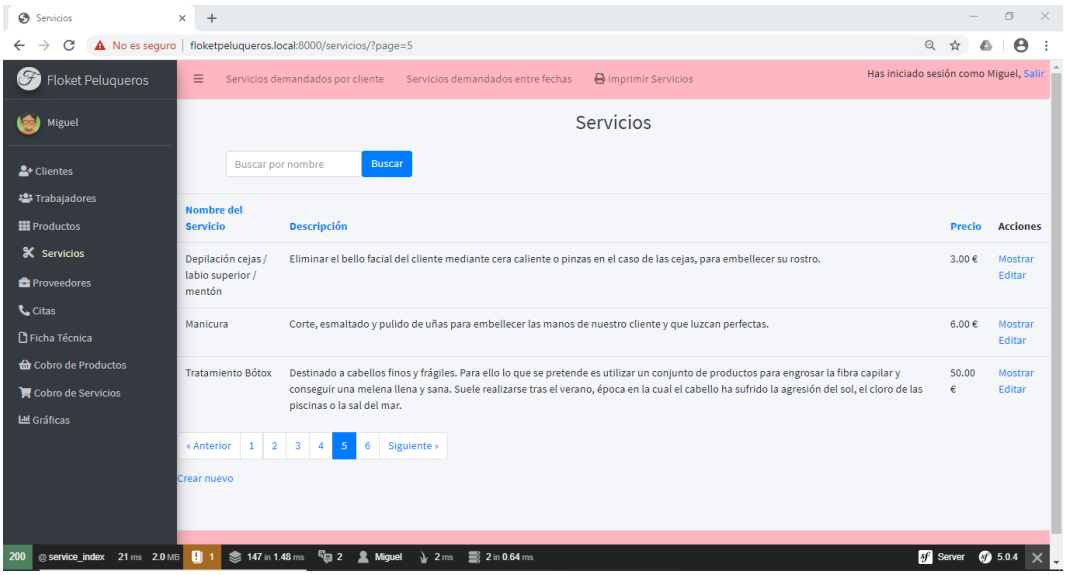
Al acceder a apartado **Servicios**, vemos un **listado** con todos los **servicios** que se ofrecen en el salón de belleza paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.

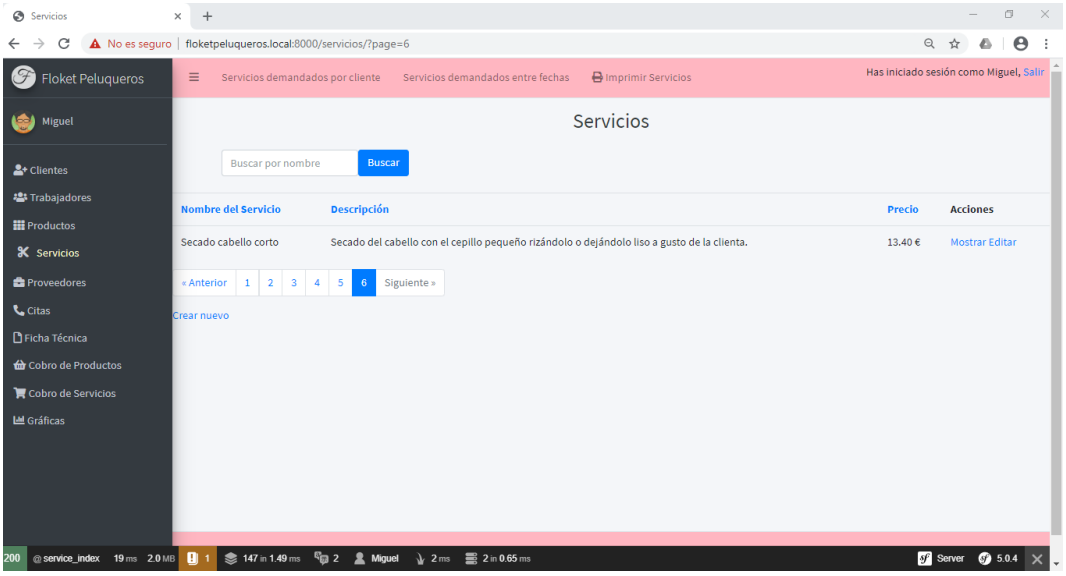



Servicios
No es seguro | floketpeluqueros.local:8000/servicios/?page=2
Floket Peluqueros
Servicios demandados por cliente
Servicios demandados entre fechas
Imprimir Servicios
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Servicios
Buscar por nombre
Buscar
Nombre del Servicio
Descripción
Precio
Acciones
Moldeado
Dedicado a clientes que quieren llevar el cabello rizado u ondado, todo ello depende del fuerte y el ancho del bigudí o body que utilicemos para realizar nuestro trabajo, dependiendo de la demanda de la clienta. Dependiendo de la finalidad que queramos conseguir se realizarán unas particiones u otras pero generalmente son nueve y siempre se empieza por la zona occipital de nuca a zona frontal. El tiempo de exposición será de 15 minutos en climazón y posteriormente se pasa a neutralizar. El acabado será en rizado y con difusor y gel para cabello rizado o gomina según guste el cliente.
50.00 €
Mostrar
Editar
Tratamiento de Keratina: Alisado
Indicado para clientes con el cabello encrespado, con este tratamiento pretendemos eliminar el crespo del cabello y únicamente mediante el uso del secador el cabello quede completamente liso sin necesidad de usar cepillo y/o planchas para la máxima comodidad del cliente. Para ello lavamos el cabello y aplicamos el producto como un tinte, pero sin llegar a tocar el cuero cabelludo. Secar para que quede completamente liso con ayuda de cepillo. Posteriormente pasaremos las planchas para que quede un liso perfecto. Y por último lavar y secar únicamente con el secador para comprobar que el tratamiento ha quedado impecable.
100.00 €
Mostrar
Editar
Corte Caballero
Corte del cabello y arreglo para embellecer la figura masculina, consiguiendo el resultado deseado por el cliente.
12.50 €
Mostrar
Editar
« Anterior
1
2
3
4
5
6
Siguiente »
Crear nuevo
200
service_index
21 ms
2.0 MB
147 in 1.95 ms
2
Miguel
3 ms
2 in 0.57 ms
Server
5.0.4

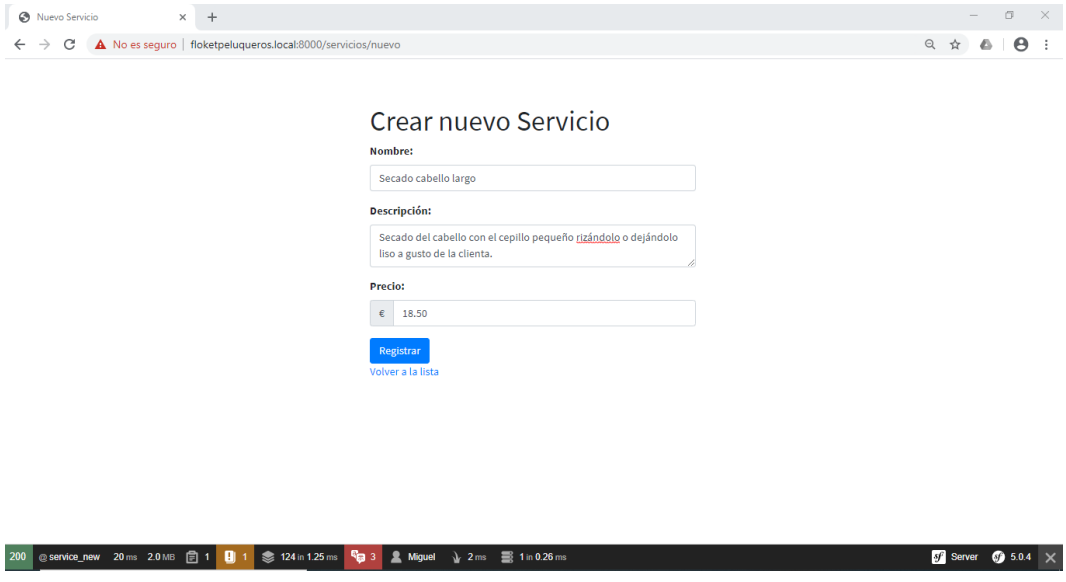

Servicios
No es seguro | floketpeluqueros.local:8000/servicios/?page=3
Floket Peluqueros
Servicios demandados por cliente
Servicios demandados entre fechas
Imprimir Servicios
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Servicios
Buscar por nombre
Buscar
Nombre del Servicio
Descripción
Precio
Acciones
Semirecogido
Se trata de una combinación de cabello recogido y mechones sueltos de forma creativa. observando atentamente la fisionomía de nuestra clienta para exteriorizar su belleza.
30.00 €
Mostrar
Editar
Mechas a banda
Para ello dividiremos el cabello en 5 secciones: dos en parietales, una en zona frontal y dos en occipital. La partición se realizará en diagonal cogiendo una sección fina de cabello impregnándola de decoloración con papel metálico. La separación entre mechón y mechón de dos dedos. Iremos subiendo hasta zona frontal. MUY IMPORTANTE: en parietales la sección será inclinada y tricotada para que visualmente una mancha en la zona delantera del cabello. El tiempo de exposición en climazón dependerá de lo rápido que se degraden los tonos del cabello de nuestro cliente.
60.00 €
Mostrar
Editar
Mechas tricotadas
Dividimos el cabello en cinco secciones, al igual que el resto de mechas, empezando siempre por la zona occipital por la nuca y tricotamos el mechón a decolorar, dejando entre sección y sección dos dedos de separación. Finalizaremos en zona frontal. El tiempo de exposición en climazón dependerá de la degradación de tonos que desee nuestro cliente.
60.00 €
Mostrar
Editar
« Anterior
1
2
3
4
5
6
Siguiente »
Crear nuevo
200
service_index
19 ms
2.0 MB
147 in 1.46 ms
2
Miguel
3 ms
2 in 0.48 ms
Server
5.0.4

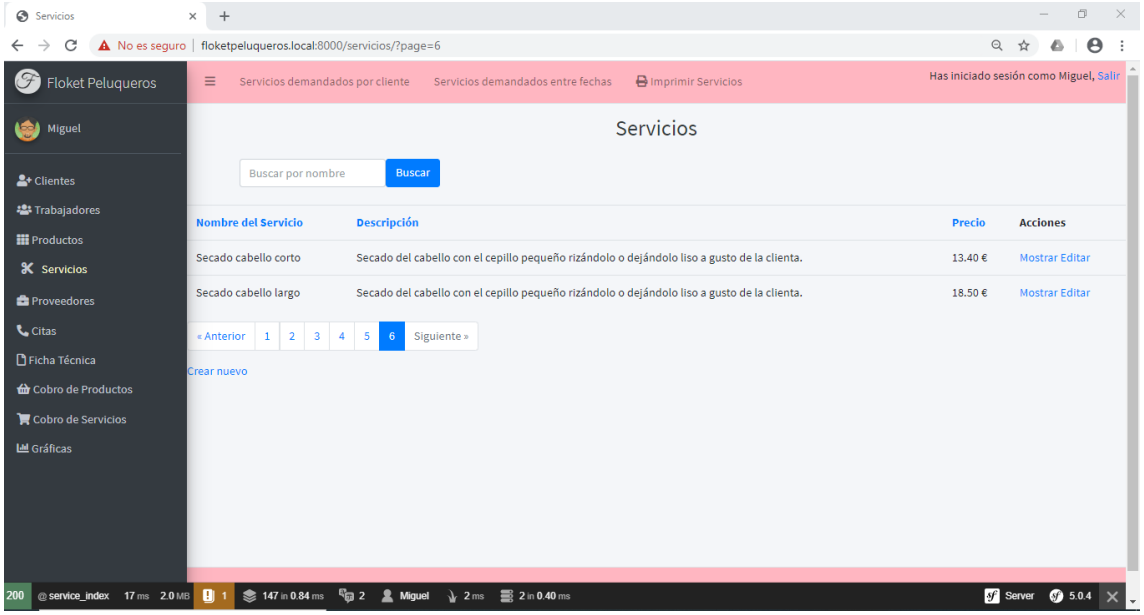

Servicios
No es seguro | floketpeluqueros.local:8000/servicios/?page=4
Floket Peluqueros
Servicios demandados por cliente
Servicios demandados entre fechas
Imprimir Servicios
Has iniciado sesión como Miguel, Salir
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Servicios
Buscar por nombre
Buscar
Nombre del Servicio
Descripción
Precio
Acciones
Matiz
Tras haber decolorado el cabello, utilizaremos el tono deseado por la clienta con el cabello mojado tras una champunada y observando la pigmentación del cabello hasta llegar al tono que deseemos. Aclara con agua fría para no perder la tonalidad.
5.00 €
Mostrar
Editar
Ondas
Moldear el cabello retorciéndolo con ayuda de difusor, tenacilla o plancha dependiendo del tipo de onda que queramos conseguir. Para lograr una melena rizada.
13.00 €
Mostrar
Editar
Liso plancha
con el cabello completamente seco pasara la plancha para conseguir una textura lisa y pulida.
13.00 €
Mostrar
Editar
« Anterior
1
2
3
4
5
6
Siguiente »
Crear nuevo
200
service_index
21 ms
2.0 MB
147 in 1.56 ms
2
Miguel
2 ms
2 in 0.52 ms
Server
5.0.4



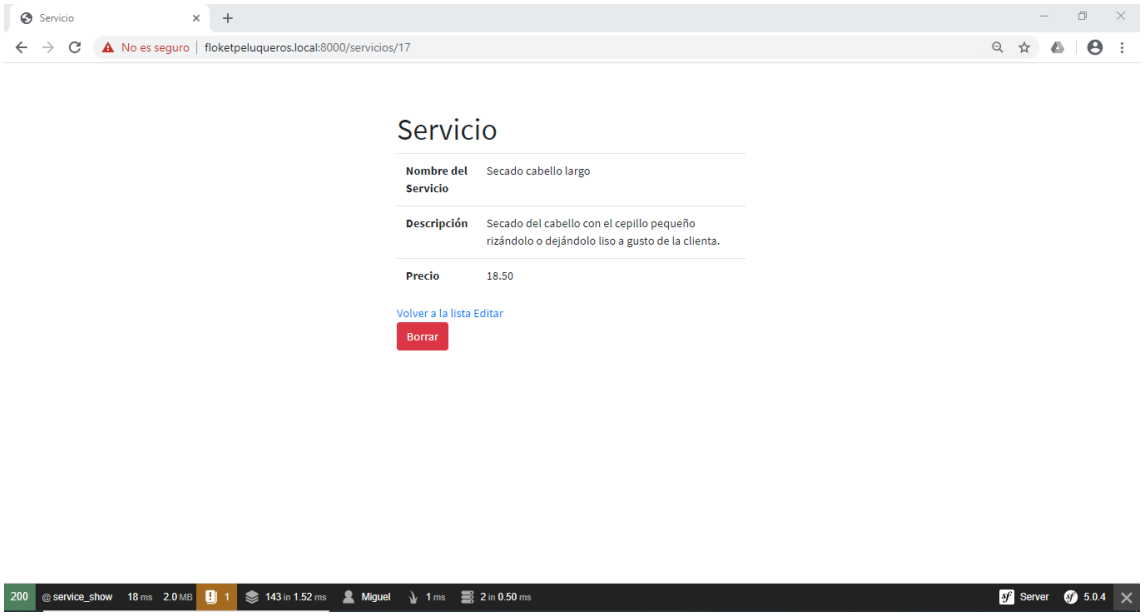


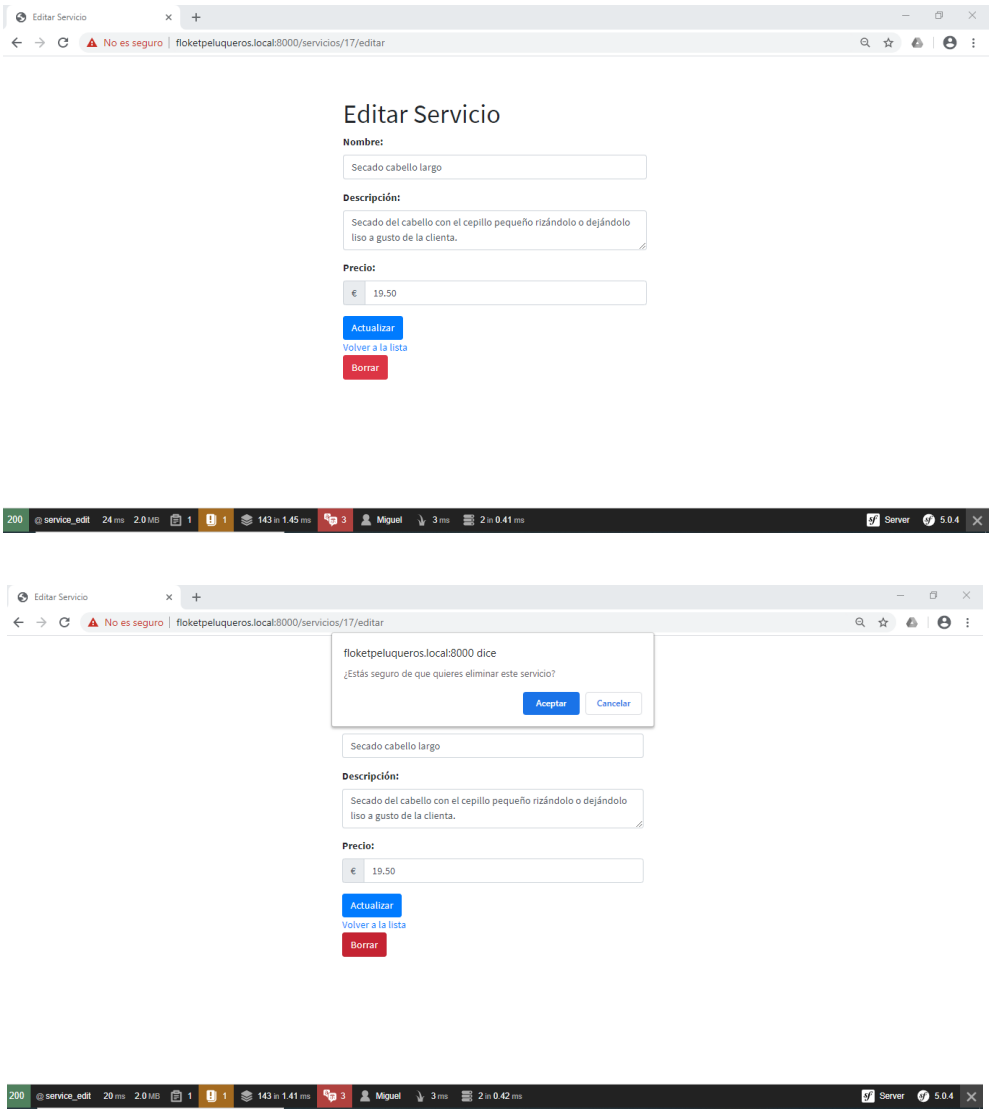
Para crear nuevos servicios, hay que acceder a Crear nuevo.



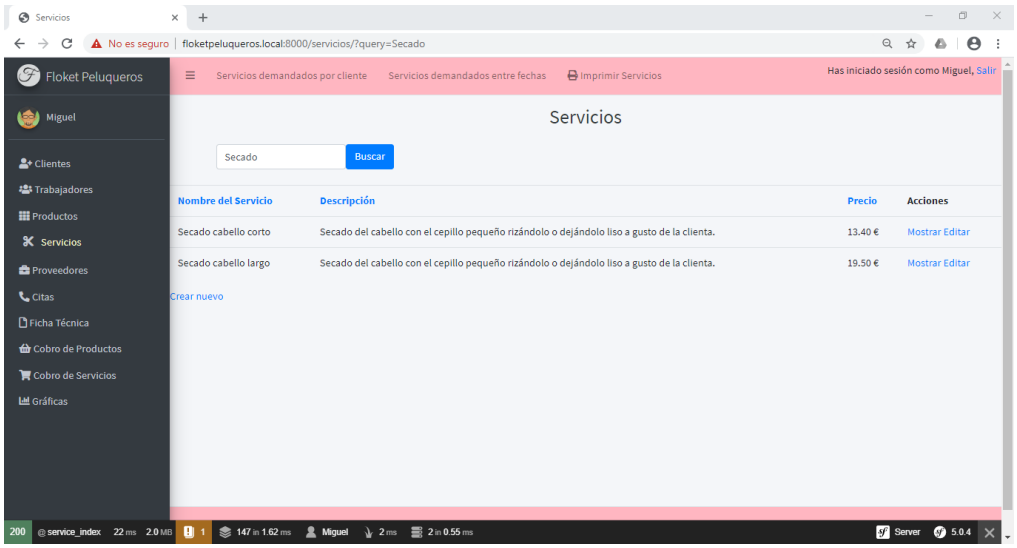


Podemos **Mostrar** los servicios accediento al apartado de Acciones en el panel principal, también podemos modificar algún dato, por ejemplo si queremos subir o bajar su precio y eliminar un servicio.

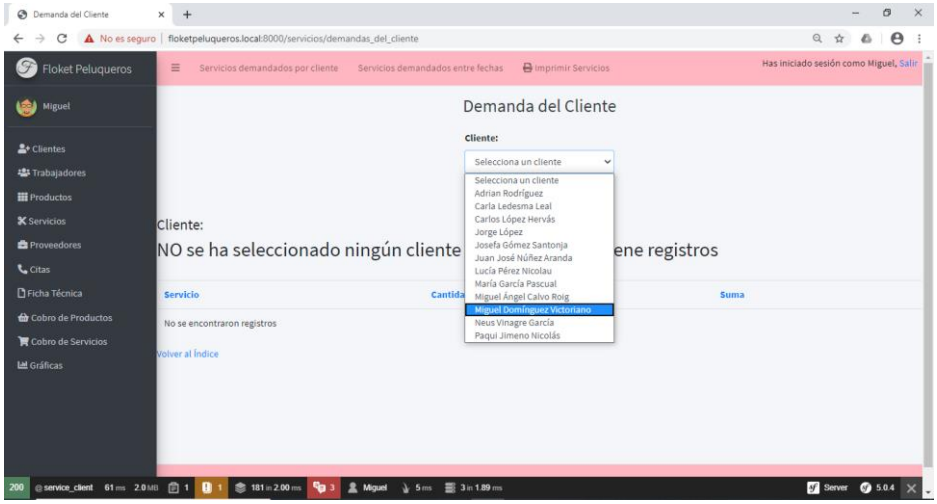


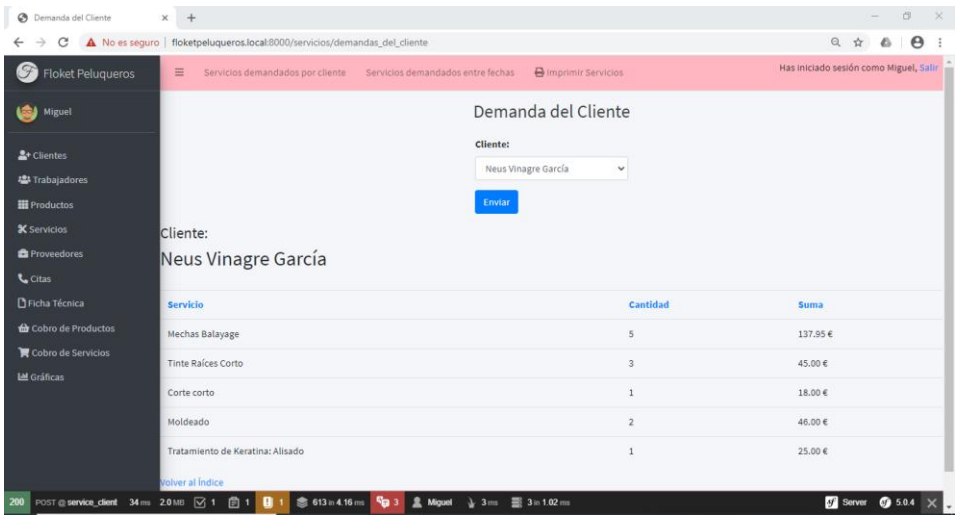


También se puede usar el buscador para buscar un servicio por su nombre.



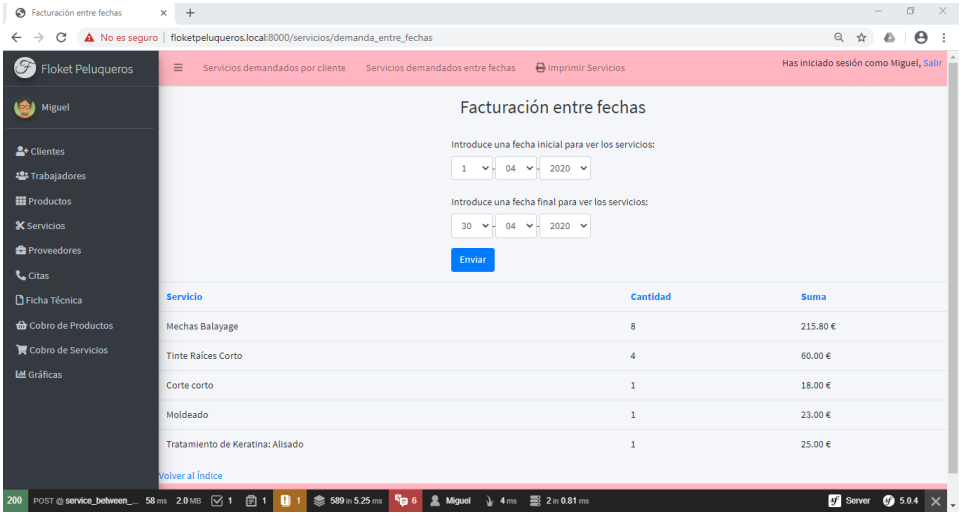
Para saber los servicios que ha demandado un cliente, se puede acceder a la pestaña **Servicios demandados por cliente**, en esta pestaña podemos seleccionar el nombre de un cliente desde el formulario desplegable, posteriormente nos aparecerá un listado con todos los servicios que se le han realizado al cliente.





También podemos saber los **Servicios demandados entre fechas**, para así poder tener una estimación del trabajo que tendremos en un rango de fechas similar.

Por ejemplo, en el mes de abril que son las fiestas patronales.



Por último, tenemos la posibilidad de generar un documento PDF con todos los servicios disponibles en el salón de belleza, este documento lo podemos generar pinchando en la pestaña **Imprimir Servicios**.

Floket Peluqueros

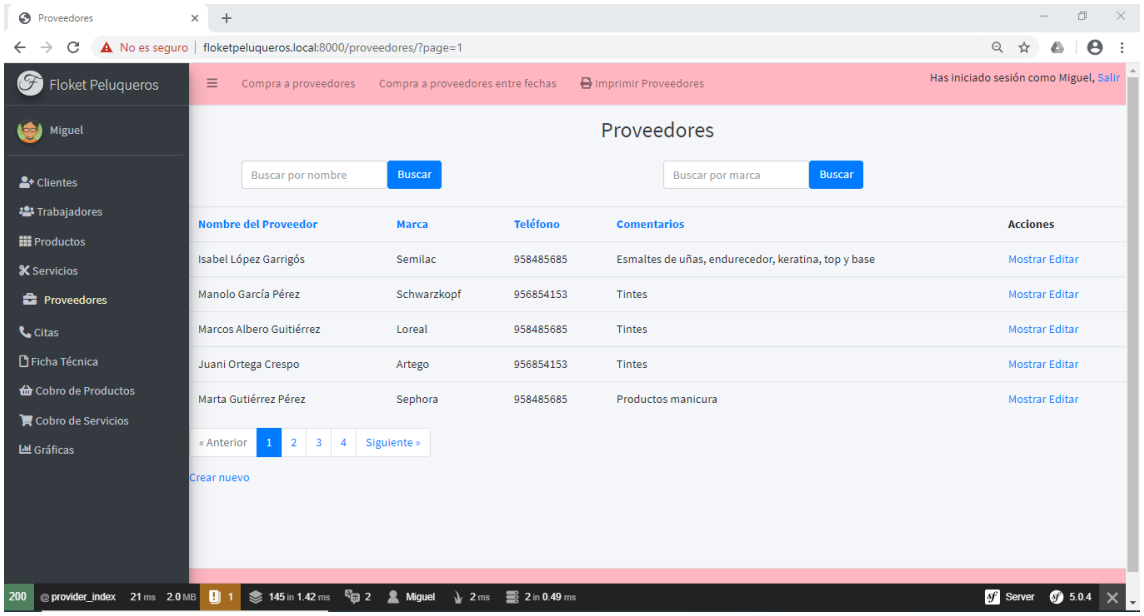
Listado de Servicios

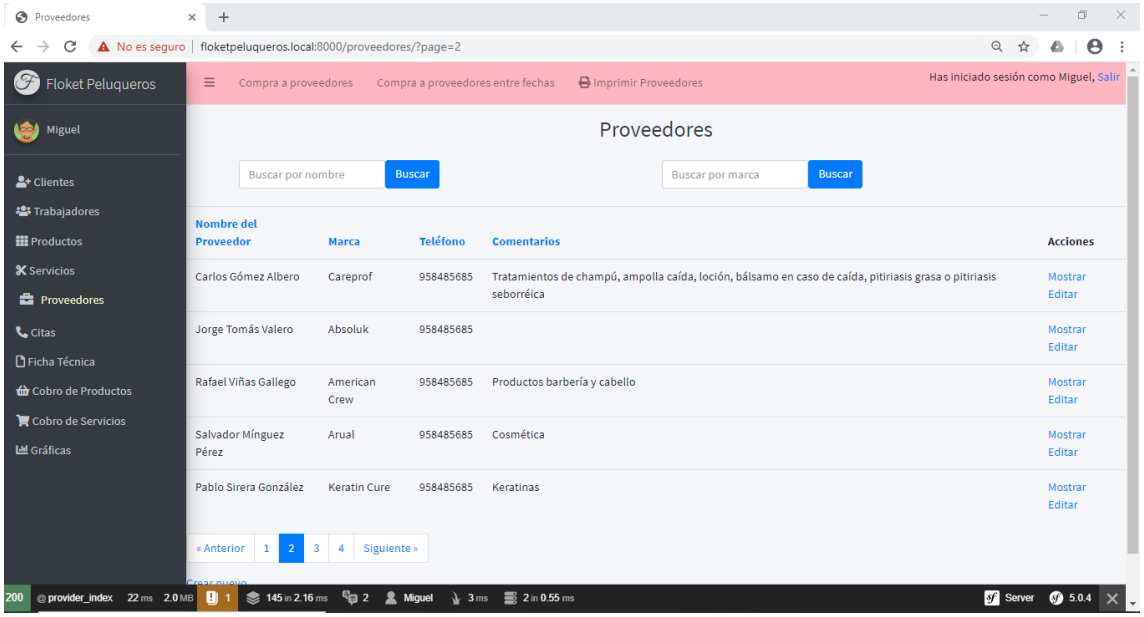
Nombre del Servicio	Precio
Mechas Balayage	50.00 €
Tinte Raíces Corto	20.00 €
Corte corto	20.00 €
Moldeado	50.00 €
Tratamiento de Keratina: Alisado	100.00 €
Corte Caballero	12.50 €
Semirecogido	30.00 €
Mechas a banda	60.00 €
Mechas tricotadas	60.00 €
Matiz	5.00 €
Ondas	13.00 €
Liso plancha	13.00 €
Depilación cejas / labio superior / mentón	3.00 €
Manicura	6.00 €
Tratamiento Bótox	50.00 €
Secado cabello corto	13.40 €
Secado cabello largo	19.50 €

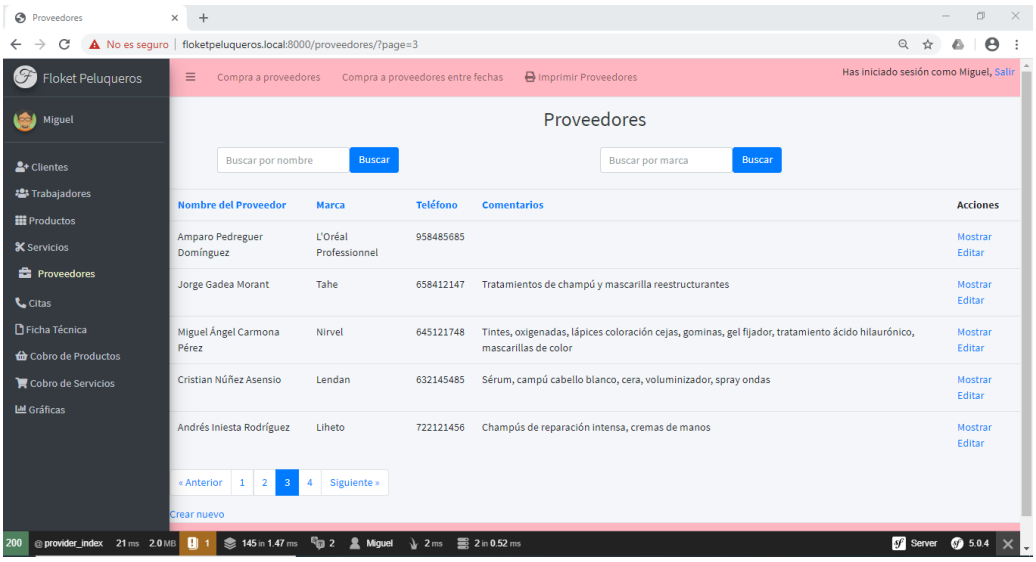
4.6.- Proveedores

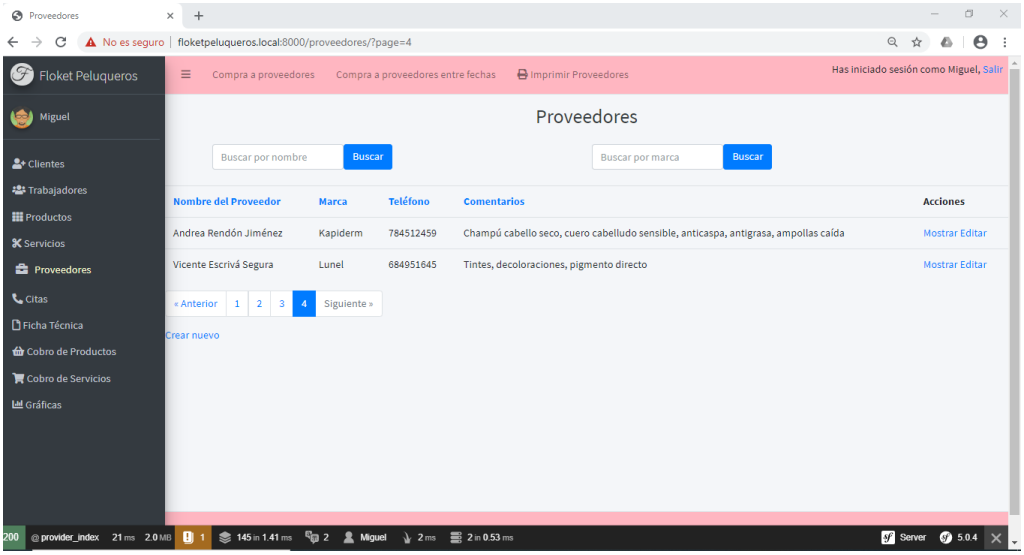
Ahora, voy a explicar el apartado referente a los proveedores, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a los proveedores.

Al acceder a apartado **Proveedores**, vemos un **listado** con todos los **proveedores** que nos proveen de productos en el salón de belleza paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.

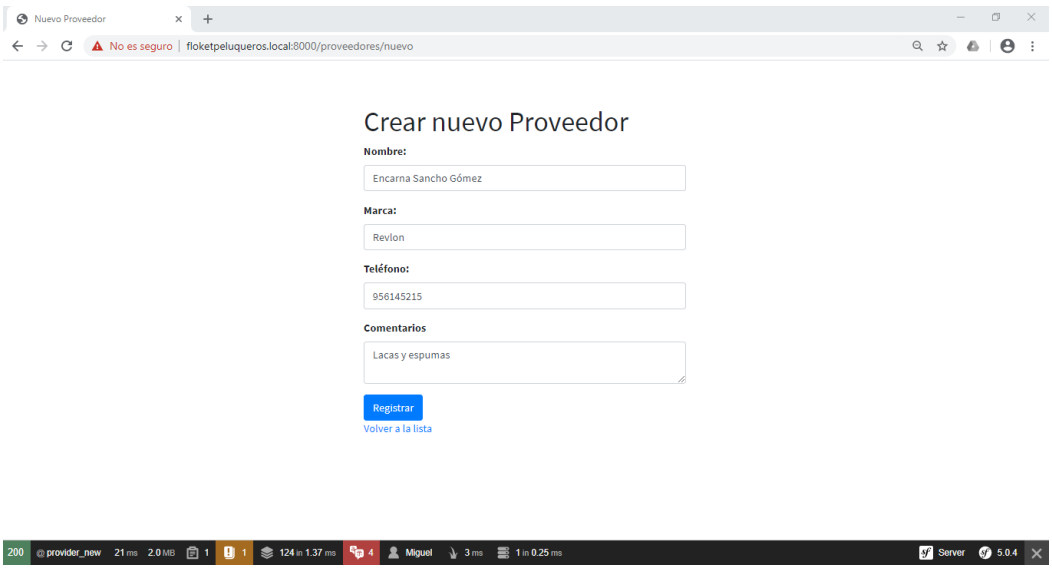




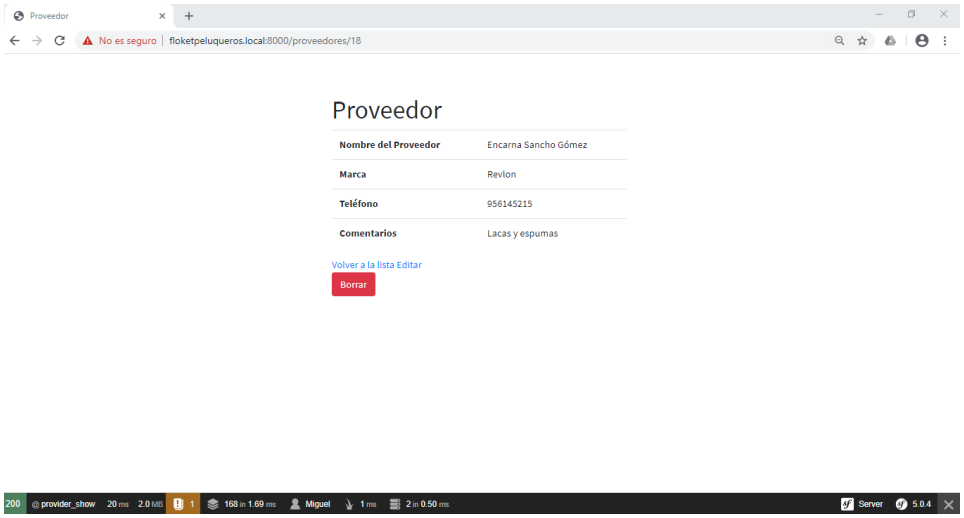


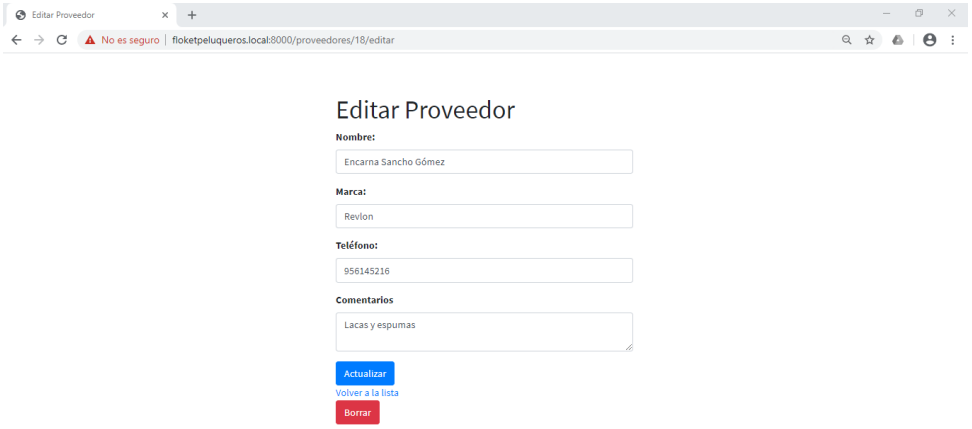


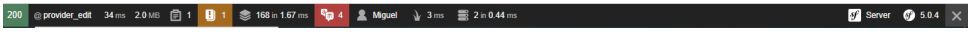
Para crear un nuevo proveedor, hay que acceder a Crear nuevo.

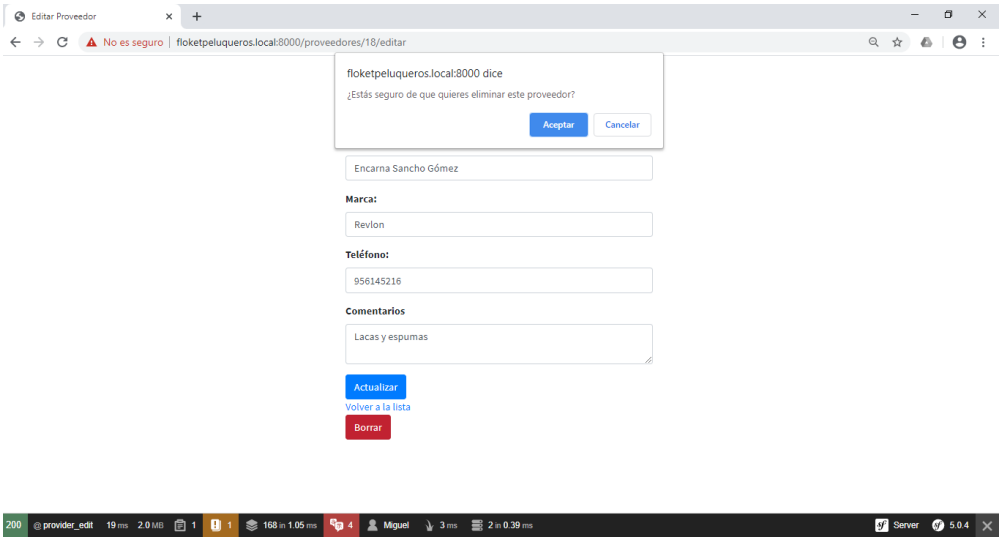


Podemos **Mostrar** a un proveedor accediento al apartado de Acciones en el panel principal, también podemos modificar algún dato, por ejemplo el teléfono de contacto y eliminar un servicio.

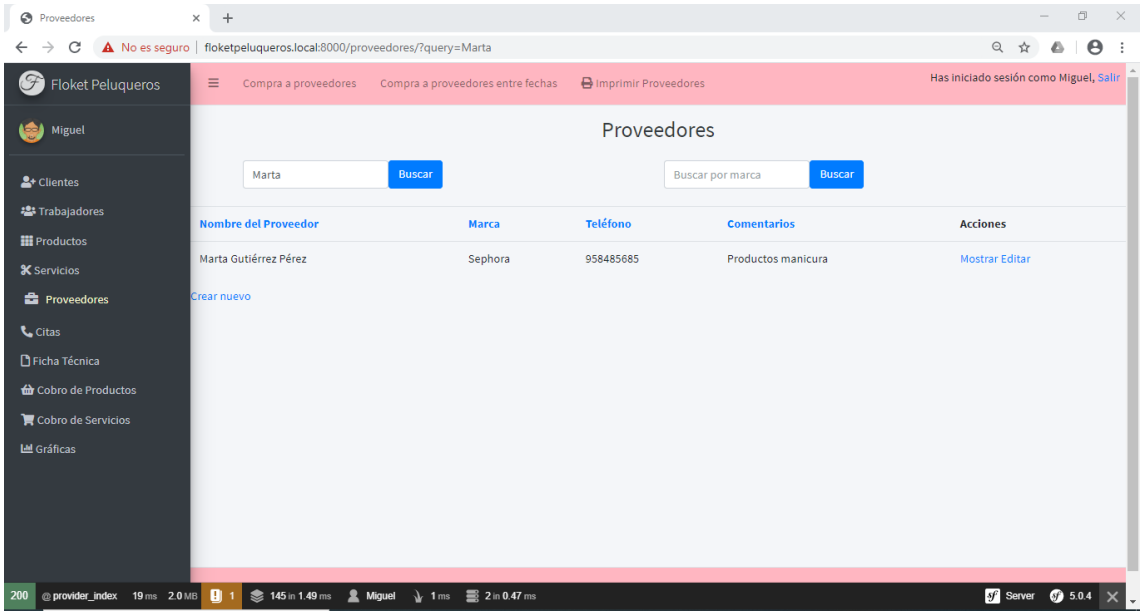


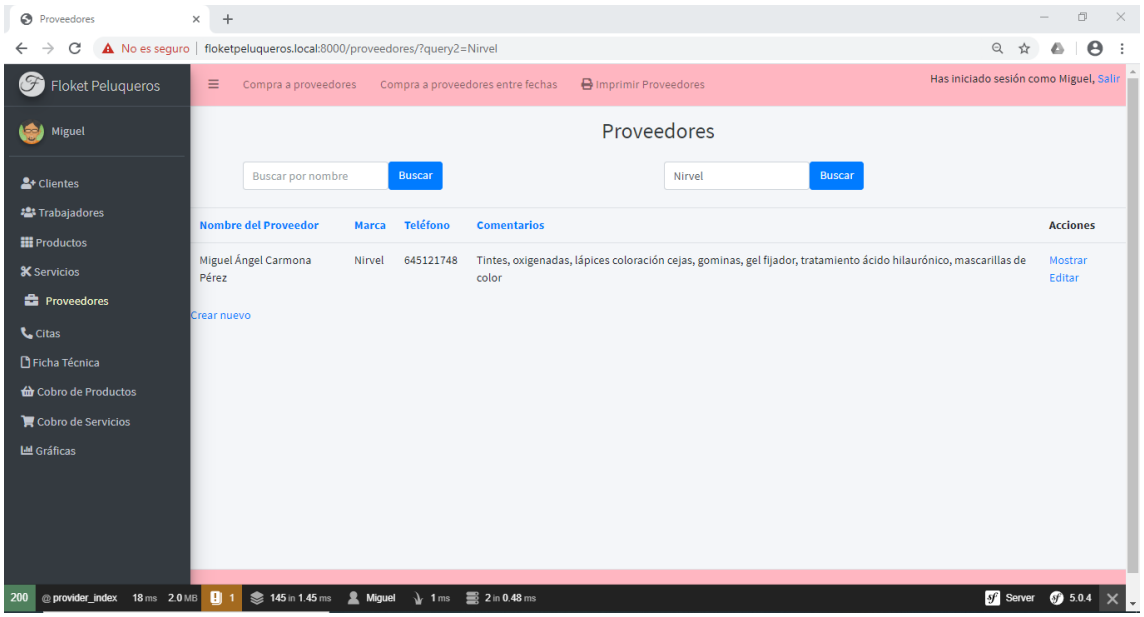




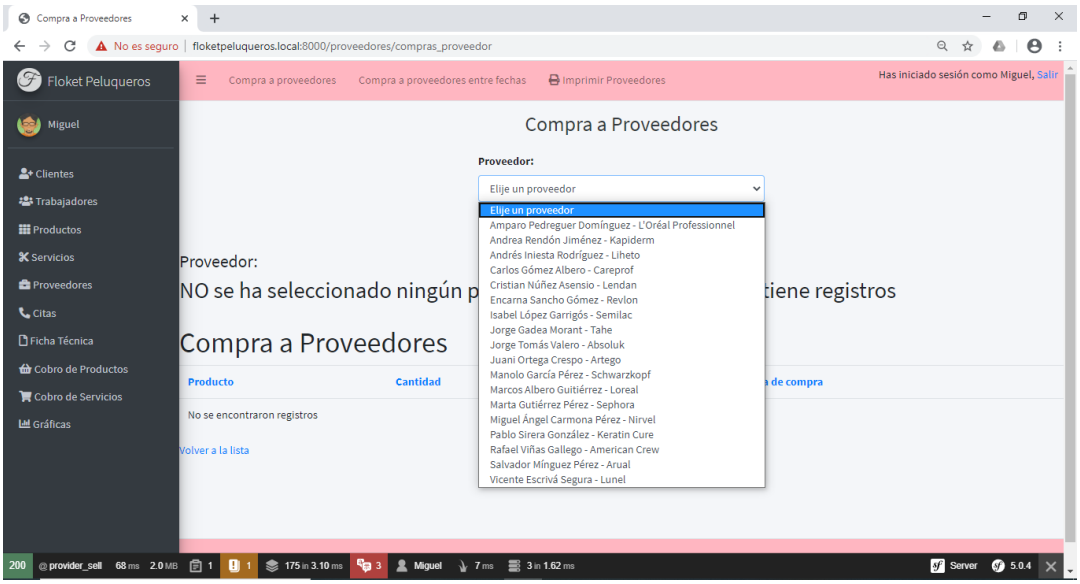


En el caso de los proveedores, hay habilitados dos buscadores diferentes, uno para buscar por el nombre del proveedor y otro para buscar por la marca a la que representa.

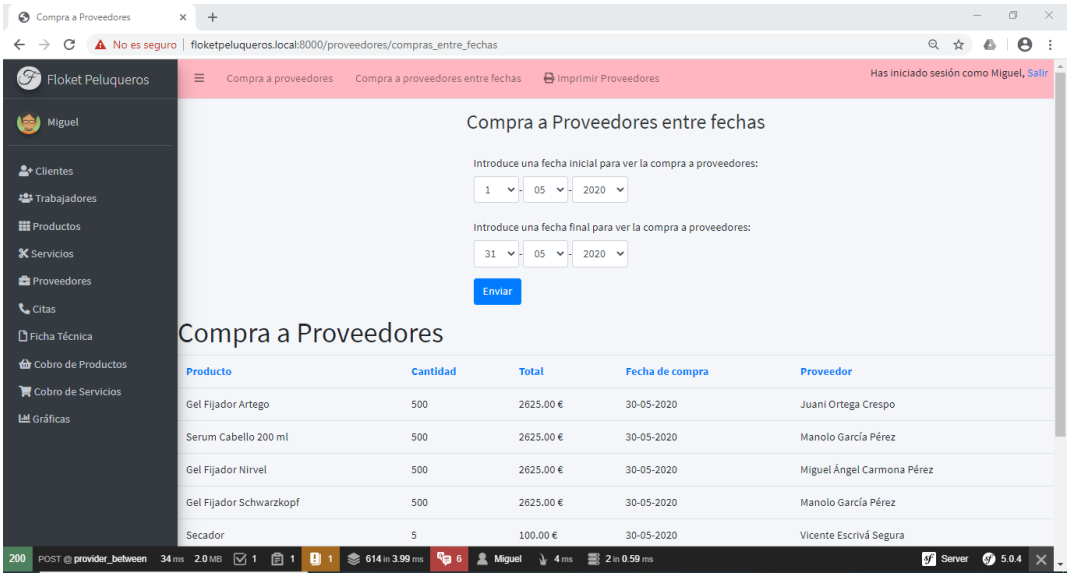




Para saber los productos que se le ha comprado a un proveedor, se puede acceder a la pestaña **Compra a proveedores**, en esta pestaña podemos seleccionar el nombre de un proveedor desde el formulario desplegable, posteriormente nos aparecerá un listado con todos los productos que se le han comprado al proveedor.



También podemos saber la **Compra a proveedores entre fechas**, para así poder tener una estimación de los productos que nos pueden hacer falta en un rango de fechas similar.



Por último, tenemos la posibilidad de generar un documento PDF con todos los proveedores, este documento lo podemos generar pinchando en la pestaña **Imprimir Proveedores**.

Floket Peluqueros

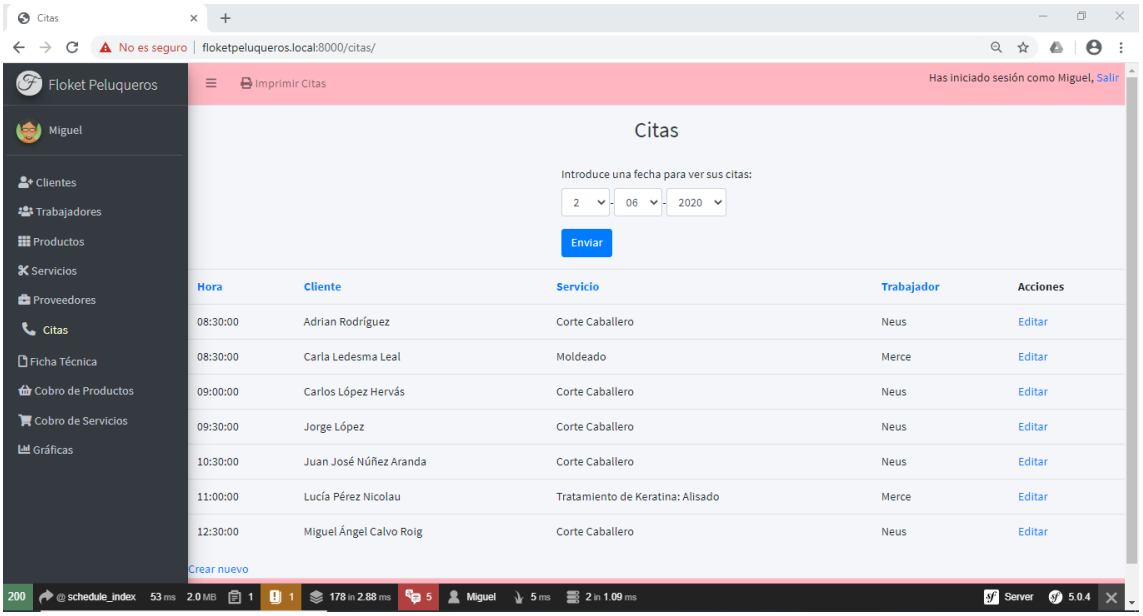
Lista de Proveedores a fecha: 02-06-2020

Nombre del Proveedor	Marca	Teléfono
Isabel López Garrigós	Semilac	958485685
Manolo García Pérez	Schwarzkopf	956854153
Marcos Albero Guitiérrez	Loreal	958485685
Juani Ortega Crespo	Artego	956854153
Marta Gutiérrez Pérez	Sephora	958485685
Carlos Gómez Albero	Careprof	958485685
Jorge Tomás Valero	Absoluk	958485685
Rafael Viñas Gallego	American Crew	958485685
Salvador Mínguez Pérez	Arual	958485685
Pablo Sirera González	Keratin Cure	958485685
Amparo Pedreguer Domínguez	L'Oréal Professionnel	958485685
Jorge Gadea Morant	Tahe	658412147
Miguel Ángel Carmona Pérez	Nirvel	645121748
Cristian Núñez Asensio	Lendan	632145485
Andrés Iniesta Rodríguez	Liheto	722121456
Andrea Rendón Jiménez	Kapiderm	784512459
Vicente Escrivá Segura	Lunel	684951645
Encarna Sancho Gómez	Revlon	956145216

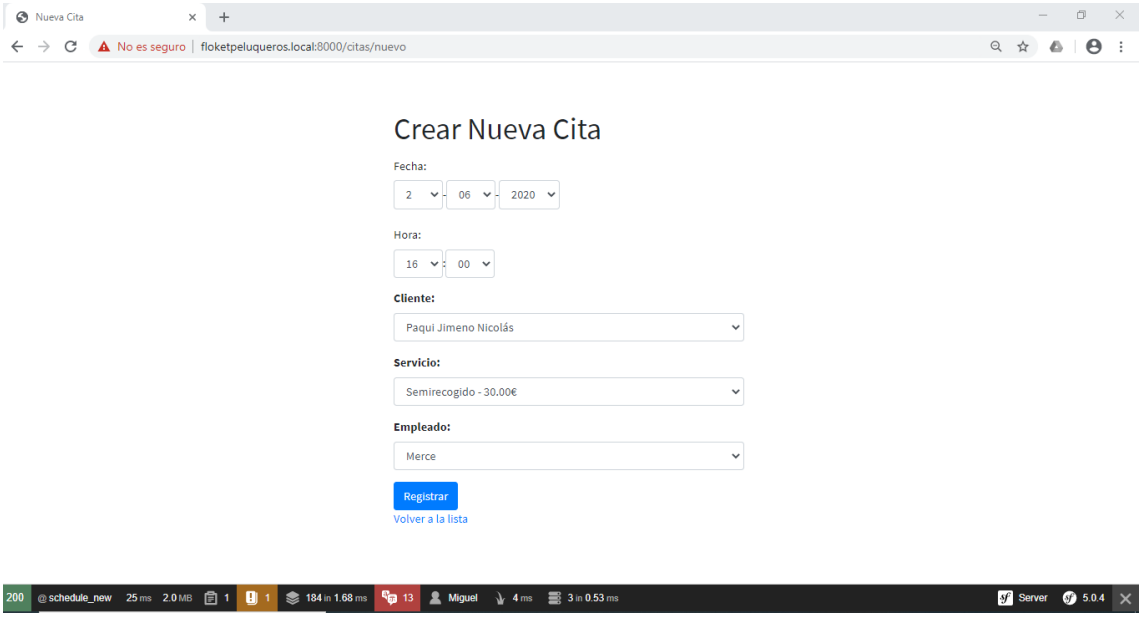
4.7.- Citas

Ahora, voy a explicar el apartado referente a las citas, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a las citas que nos piden los clientes.

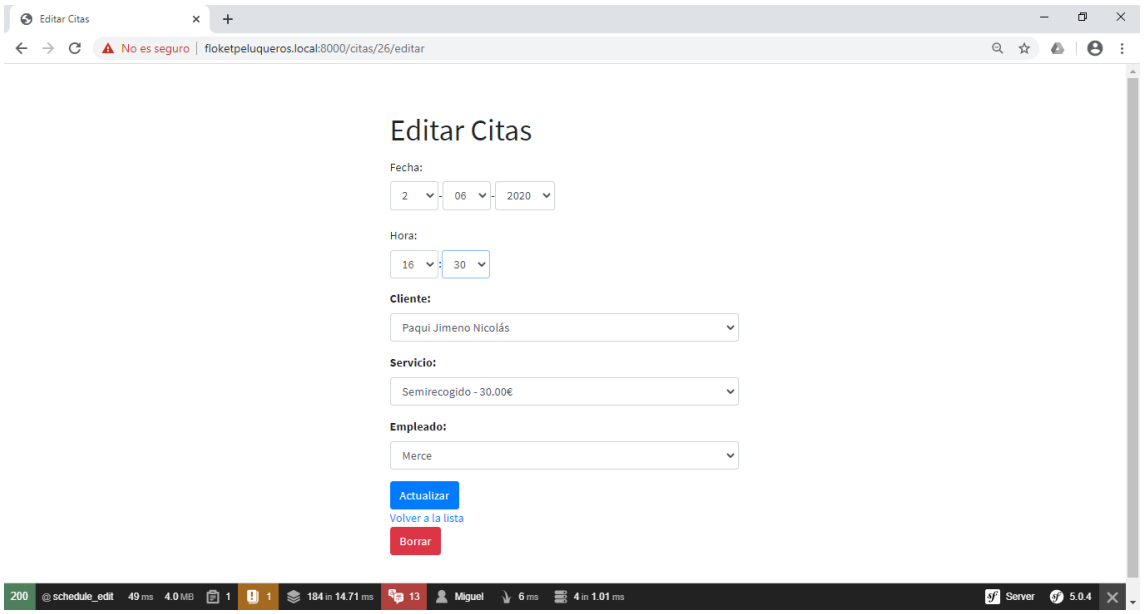
Al acceder a apartado **Citas**, vemos un **listado** con todas las citas que nos han pedido los clientes en la fecha actual.

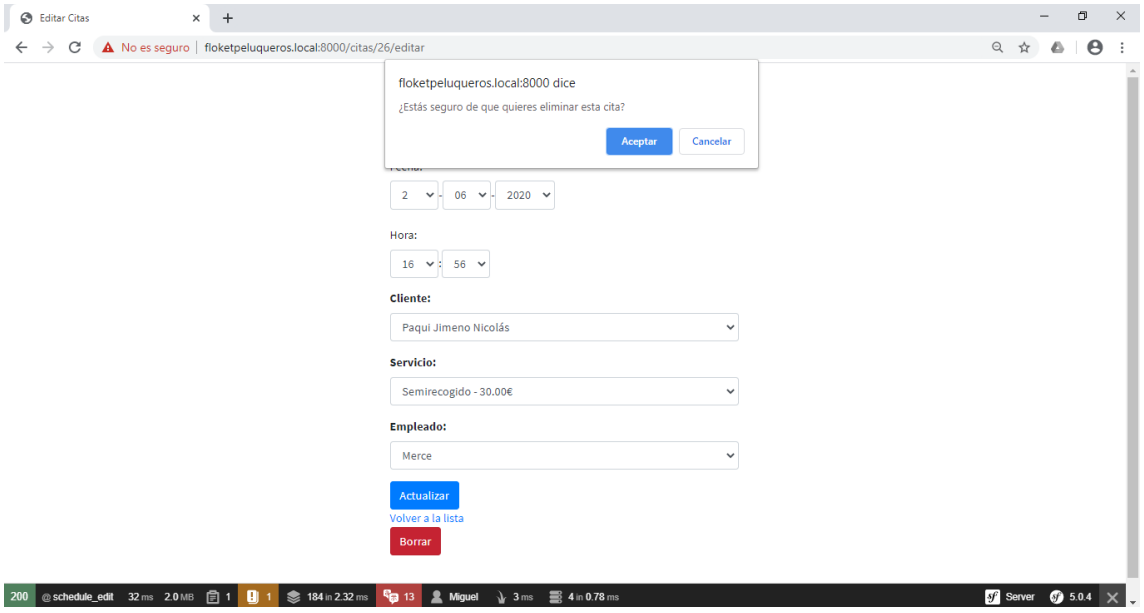


Para crear una nueva cita, hay que acceder a Crear nuevo.

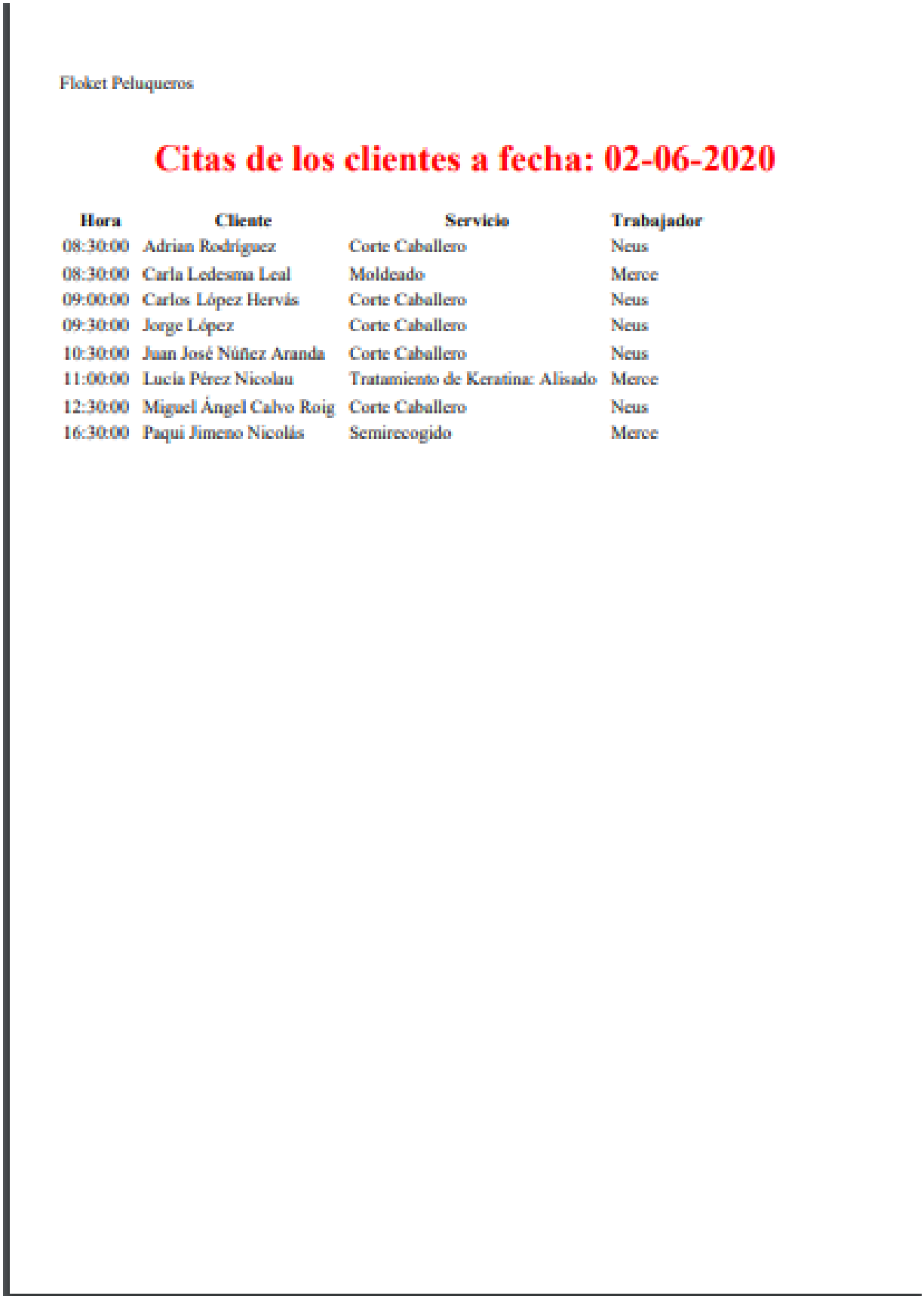


Podemos modificar una cita accediento al apartado de Acciones en el panel principal, también podemos eliminar una cita si por lo que sea el cliente al final la cancela.





Por último, tenemos la posibilidad de generar un documento PDF con todas las citas del día, este documento lo podemos generar pinchando en la pestaña **Imprimir Citas**.



Floket Peluqueros

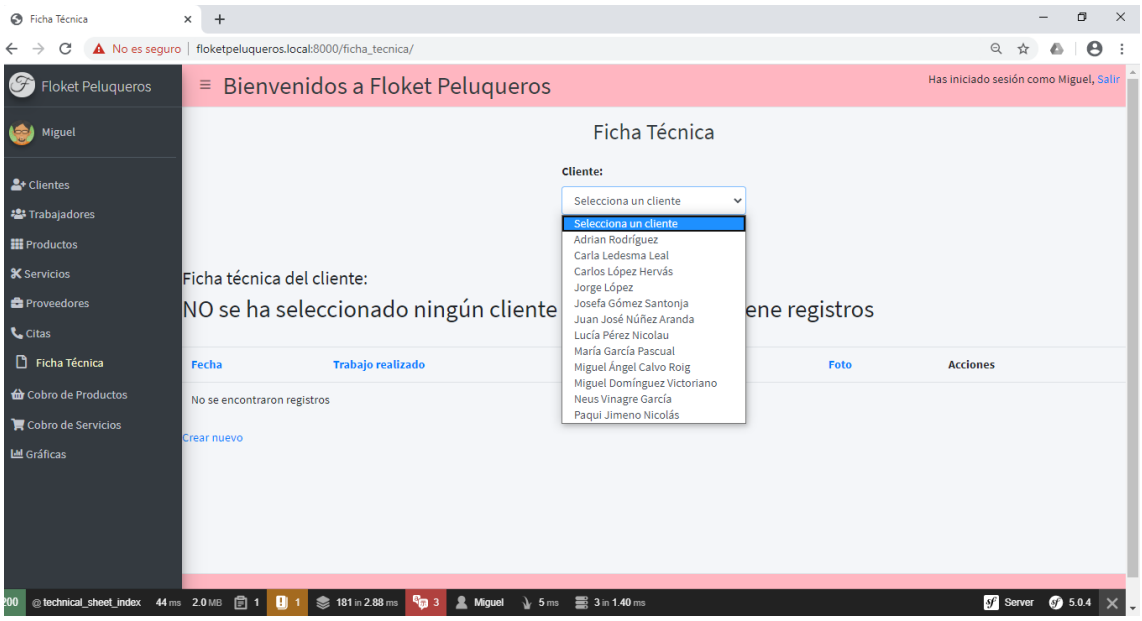
Citas de los clientes a fecha: 02-06-2020

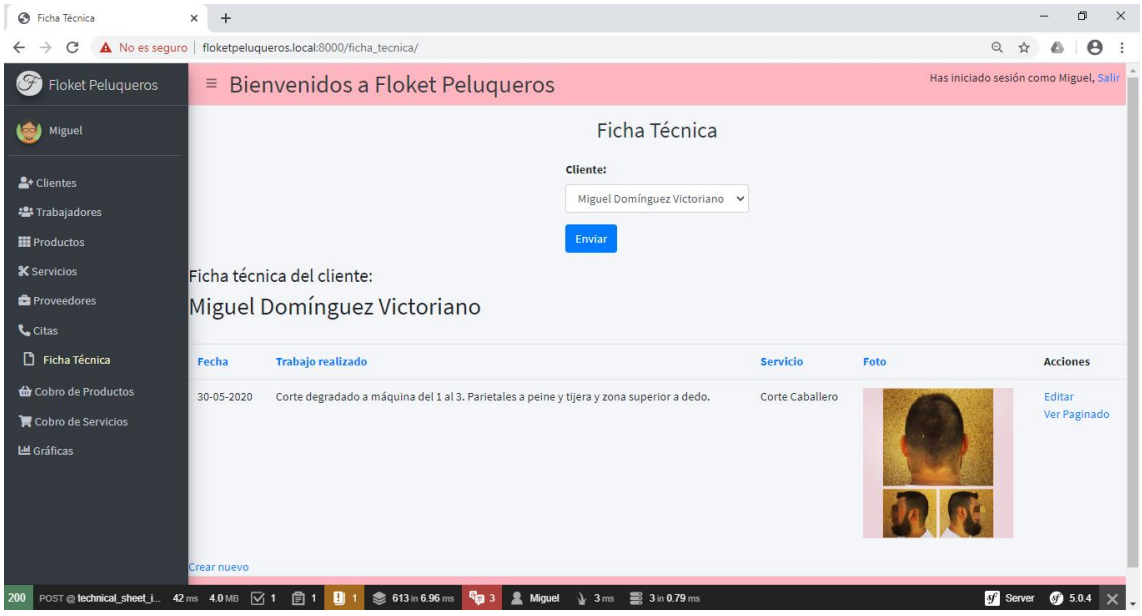
Hora	Cliente	Servicio	Trabajador
08:30:00	Adrian Rodríguez	Corte Caballero	Neus
08:30:00	Carla Ledesma Leal	Moldeado	Merce
09:00:00	Carlos López Hervás	Corte Caballero	Neus
09:30:00	Jorge López	Corte Caballero	Neus
10:30:00	Juan José Núñez Aranda	Corte Caballero	Neus
11:00:00	Lucía Pérez Nicolau	Tratamiento de Keratina: Alisado	Merce
12:30:00	Miguel Ángel Calvo Roig	Corte Caballero	Neus
16:30:00	Paqui Jimeno Nicolás	Semirecogido	Merce

4.8.- Ficha Técnica

Ahora, voy a explicar el apartado referente a la ficha técnica, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente a la ficha técnica de los clientes.

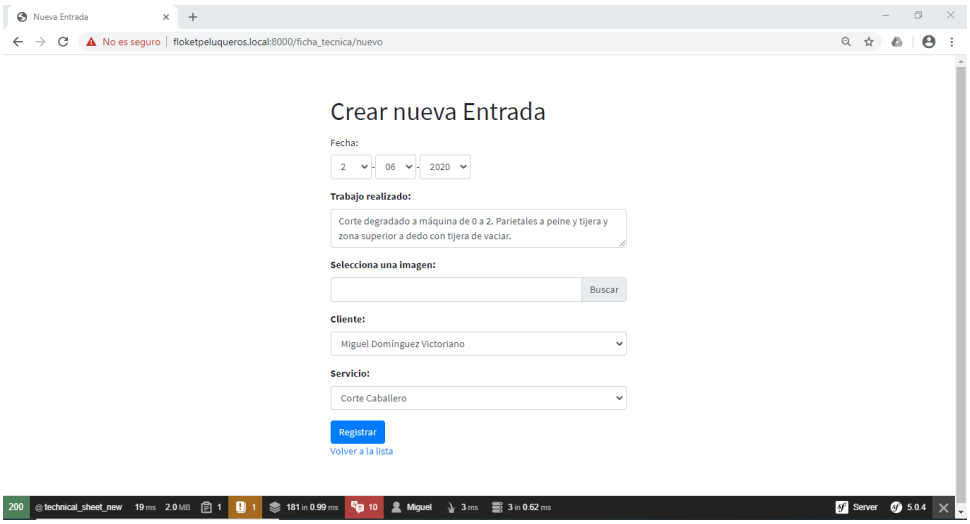
Al acceder al apartado **Ficha técnica**, tenemos que seleccionar un cliente del formulario desplegable para visualizar los registros de un cliente.

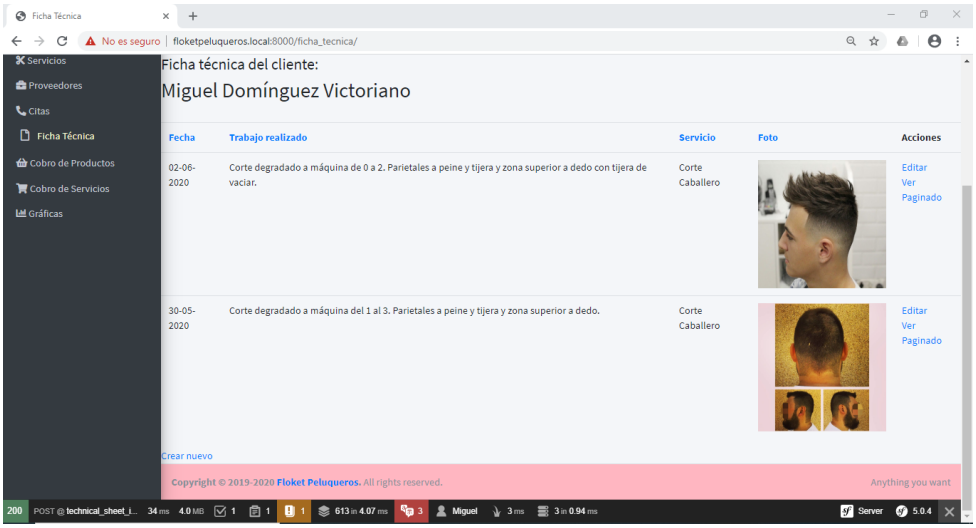




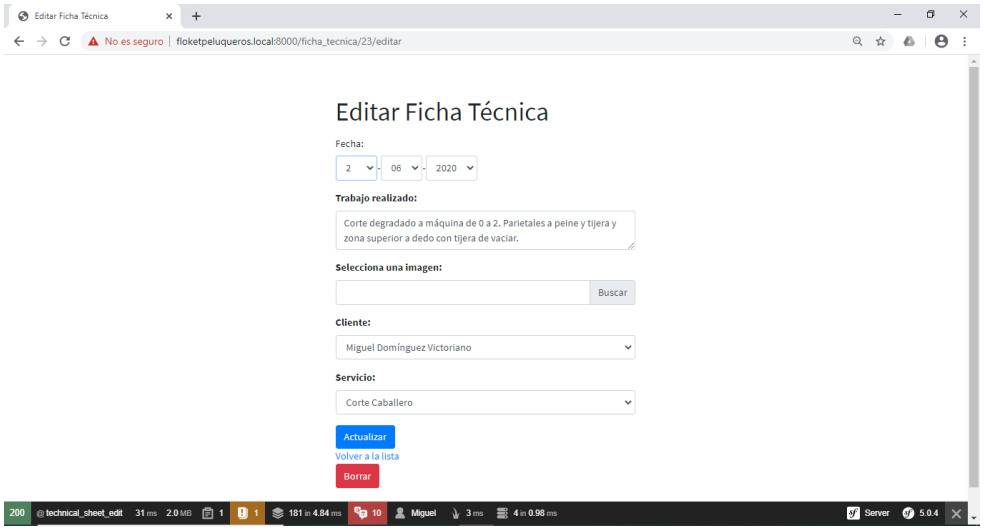
Al ver los registros de la ficha técnica de un cliente, podemos ver los trabajos que se le han realizado con anterioridad.

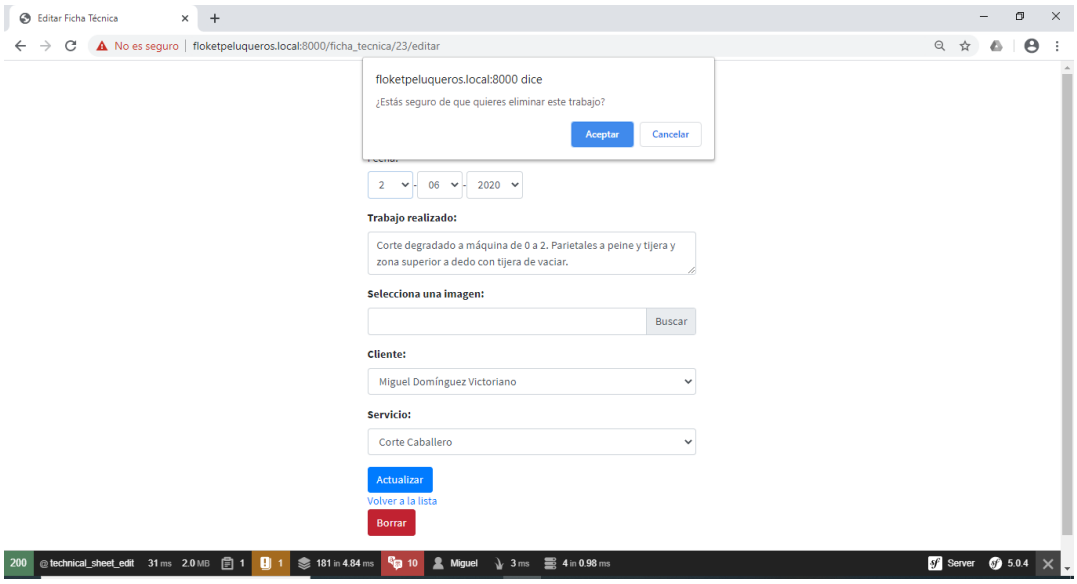
Para crear un nuevo registro, hay que acceder a Crear nuevo.





Podemos modificar un registro accediendo al apartado de Acciones en el panel principal, también podemos eliminar un registro.

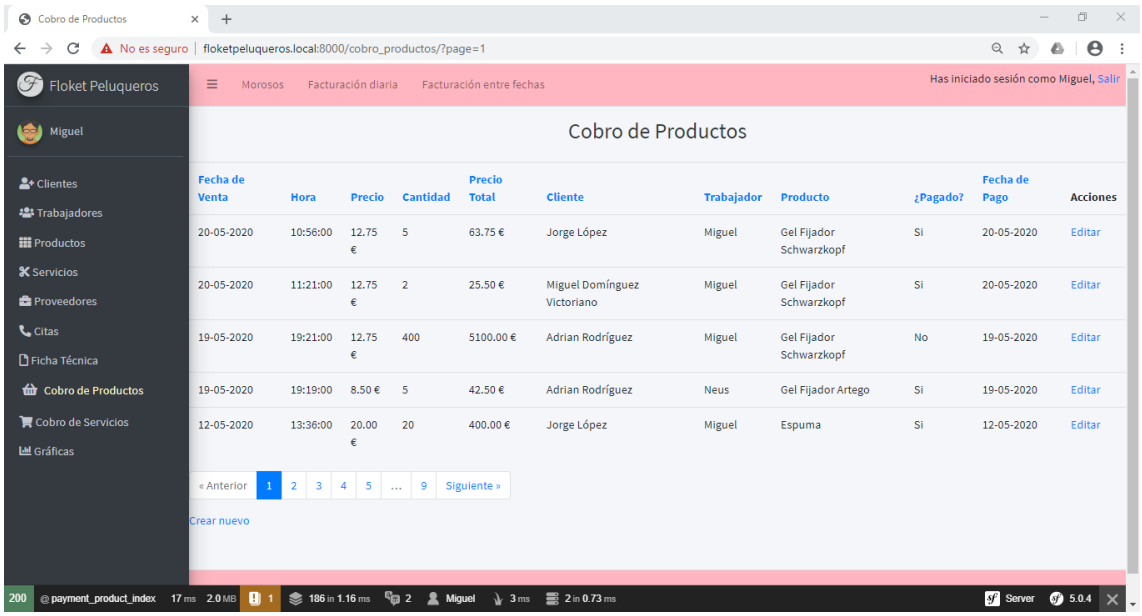


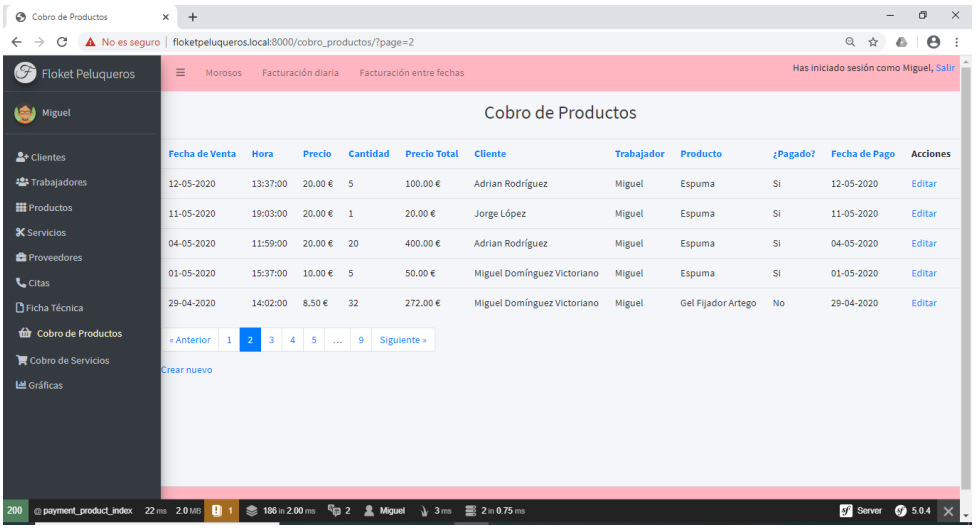


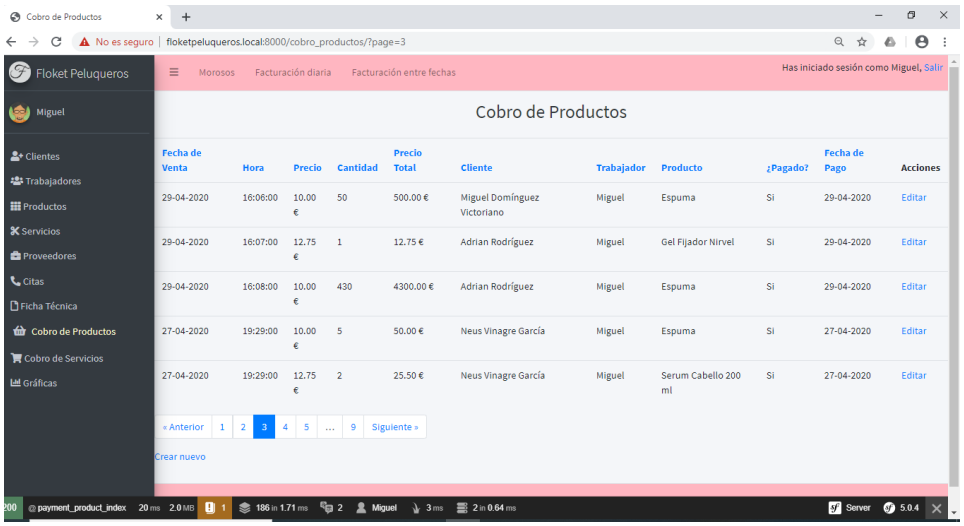
4.9.- Cobro de Productos

Ahora, voy a explicar el apartado referente al cobro de productos, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente al cobro de productos a los clientes.

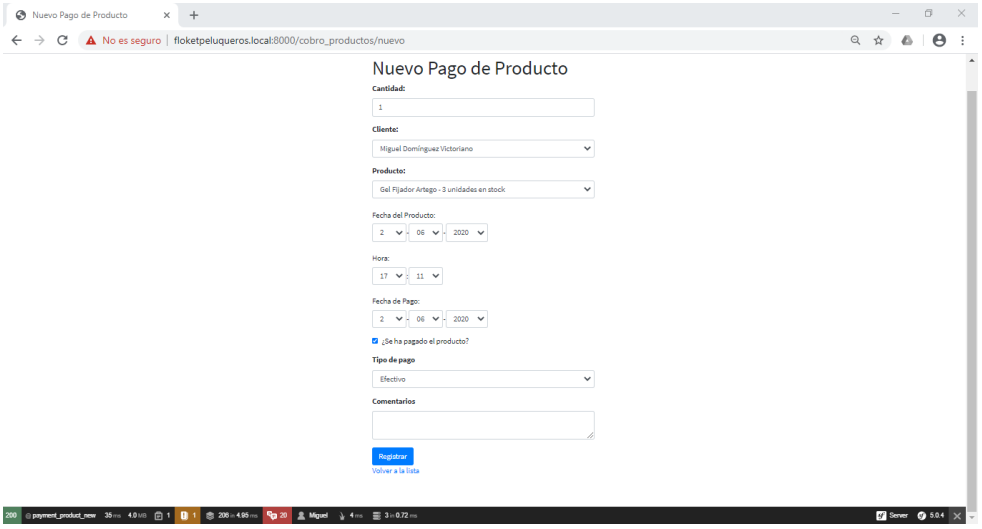
Al acceder al apartado **Cobro de productos**, podemos ver todos los cobros que se le han realizado a los clientes paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.

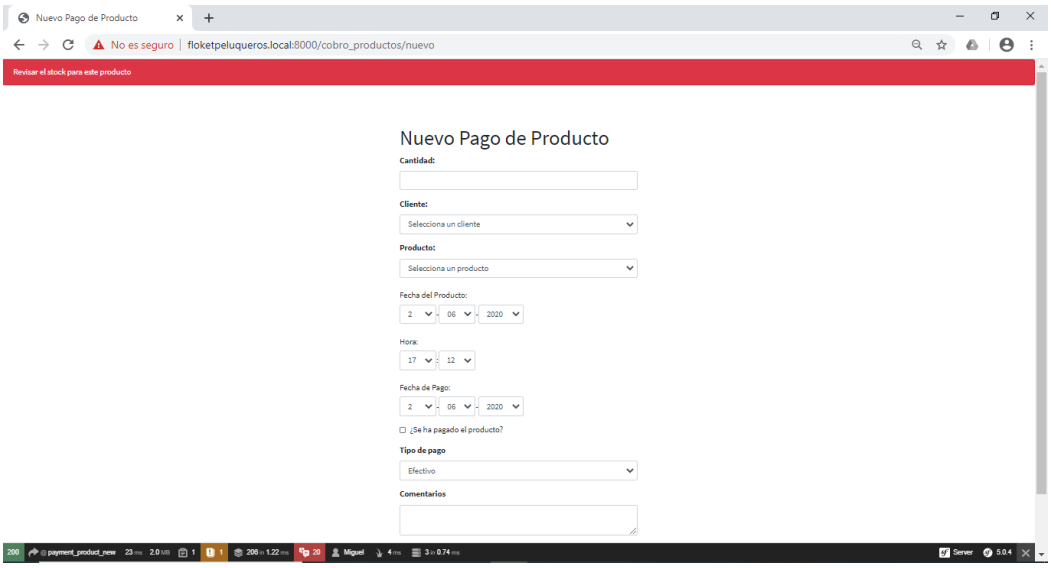


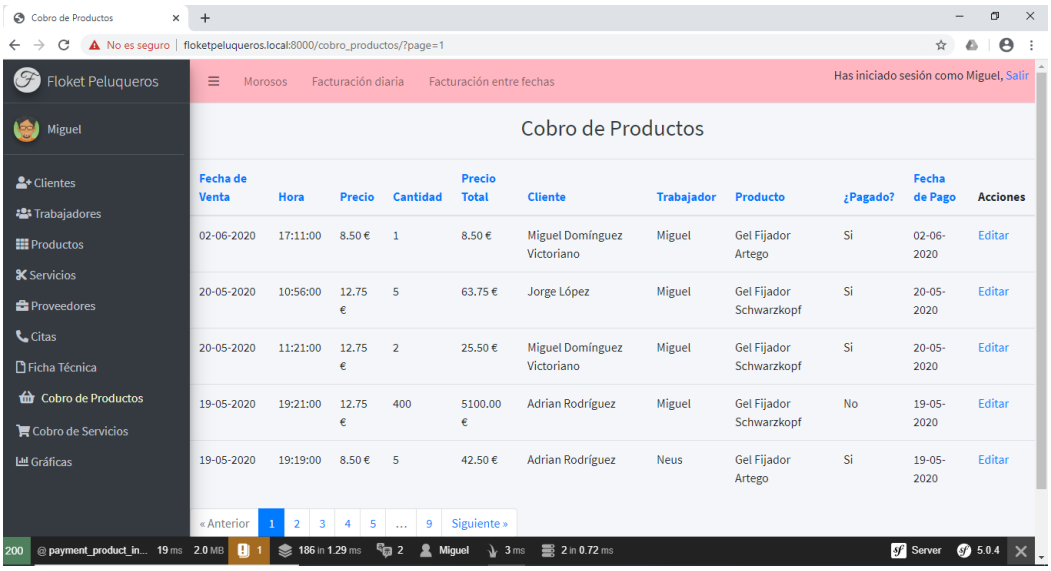


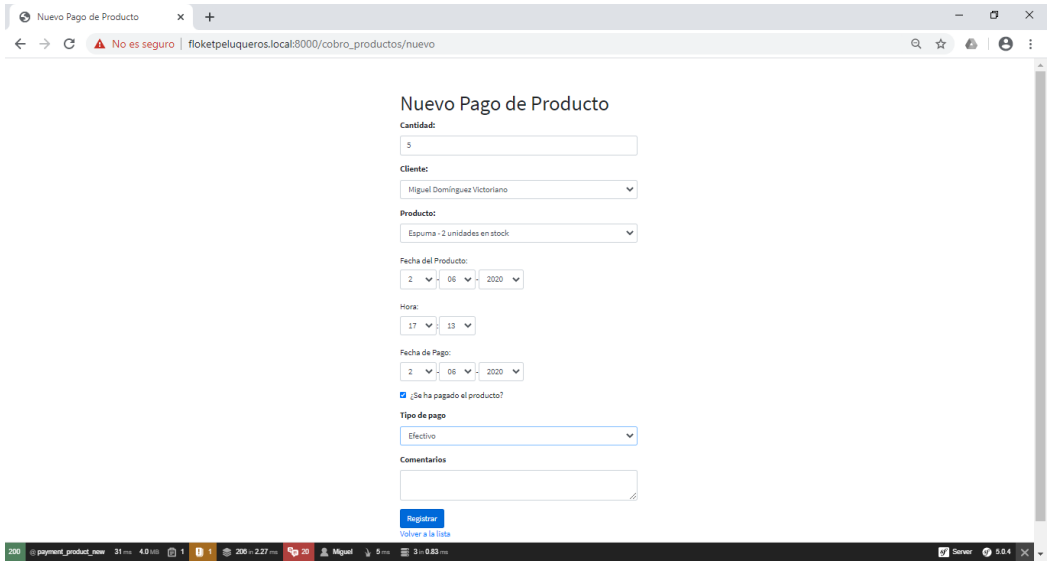


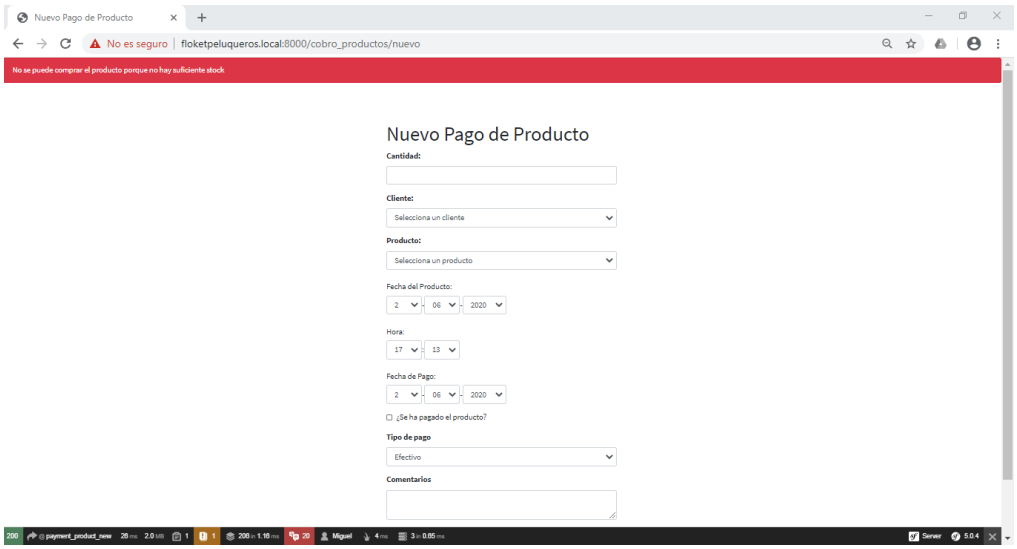
Para crear un nuevo cobro de un producto, hay que acceder a Crear nuevo, si hay stock del producto la compra se realiza sin problemas, si al comprar el producto queda poco stock salta un mensaje de advertencia y si no hay suficiente stock no podemos vender el producto.



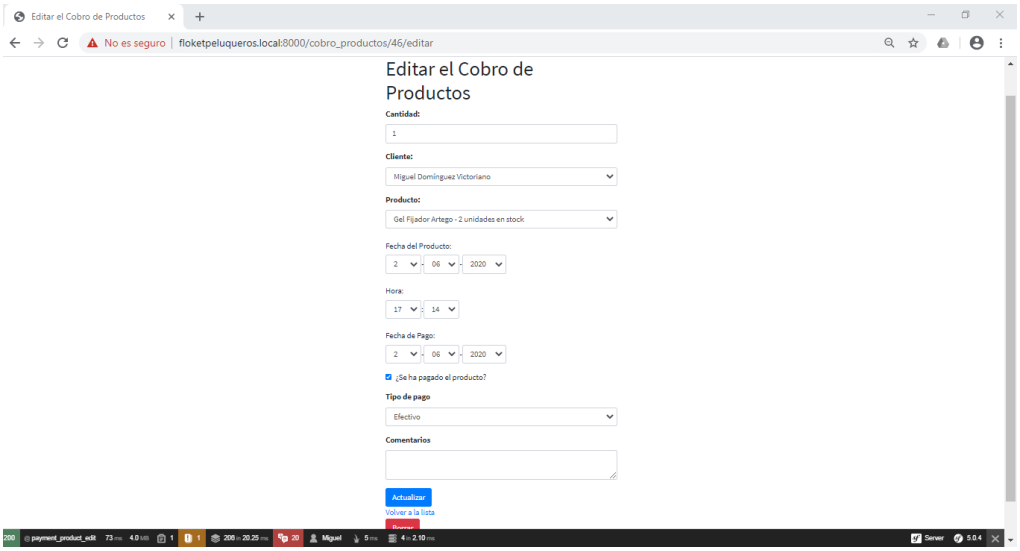

Nuevo Pago de Producto
No es seguro | floketpeluqueros.local:8000/cobro_productos/nuevo
Revisar el stock para este producto
Nuevo Pago de Producto
Cantidad:
Cliente:
Selecciona un cliente
Producto:
Selecciona un producto
Fecha del Producto:
Hora:
Fecha de Pago:
¿Se ha pagado el producto?
Tipo de pago
Efectivo
Comentarios

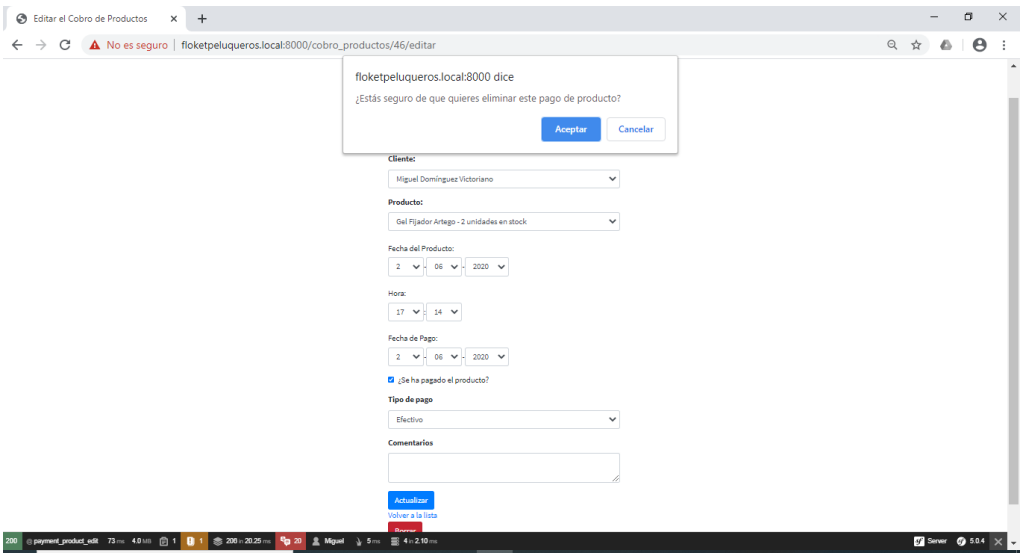

Cobro de Productos
No es seguro | floketpeluqueros.local:8000/cobro_productos/?page=1
Floket Peluqueros
Miguel
Clientes
Trabajadores
Productos
Servicios
Proveedores
Citas
Ficha Técnica
Cobro de Productos
Cobro de Servicios
Gráficas
Morosos
Facturación diaria
Facturación entre fechas
Has iniciado sesión como Miguel, Salir
Cobro de Productos
Fecha de Venta
Hora
Precio
Cantidad
Precio Total
Cliente
Trabajador
Producto
¿Pagado?
Fecha de Pago
Acciones
02-06-2020 17:11:00 8.50 € 1 8.50 € Miguel Domínguez Victoriano Miguel Gel Fijador Artego Si 02-06-2020 Editar
20-05-2020 10:56:00 12.75 € 5 63.75 € Jorge López Miguel Gel Fijador Schwarzkopf Si 20-05-2020 Editar
20-05-2020 11:21:00 12.75 € 2 25.50 € Miguel Domínguez Victoriano Miguel Gel Fijador Schwarzkopf Si 20-05-2020 Editar
19-05-2020 19:21:00 12.75 € 400 5100.00 € Adrian Rodríguez Miguel Gel Fijador Schwarzkopf No 19-05-2020 Editar
19-05-2020 19:19:00 8.50 € 5 42.50 € Adrian Rodríguez Neus Gel Fijador Artego Si 19-05-2020 Editar
« Anterior 1 2 3 4 5 … 9 Siguiente »


Nuevo Pago de Producto
No es seguro | floketpeluqueros.local:8000/cobro_productos/nuevo
Nuevo Pago de Producto
Cantidad:
5
Cliente:
Miguel Domínguez Victoriano
Producto:
Espuma - 2 unidades en stock
Fecha del Producto:
Hora:
Fecha de Pago:
¿Se ha pagado el producto?
Tipo de pago
Efectivo
Comentarios
Registrar
Volver a la lista

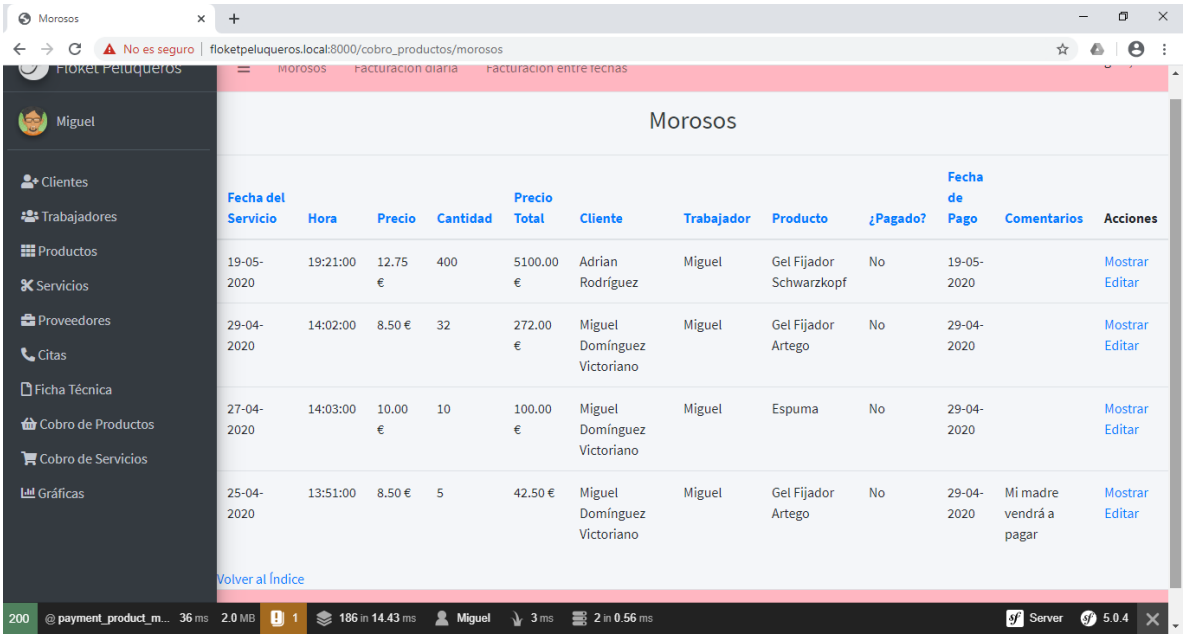


Podemos modificar un registro accediendo al apartado de Acciones en el panel principal, también podemos eliminar un registro.



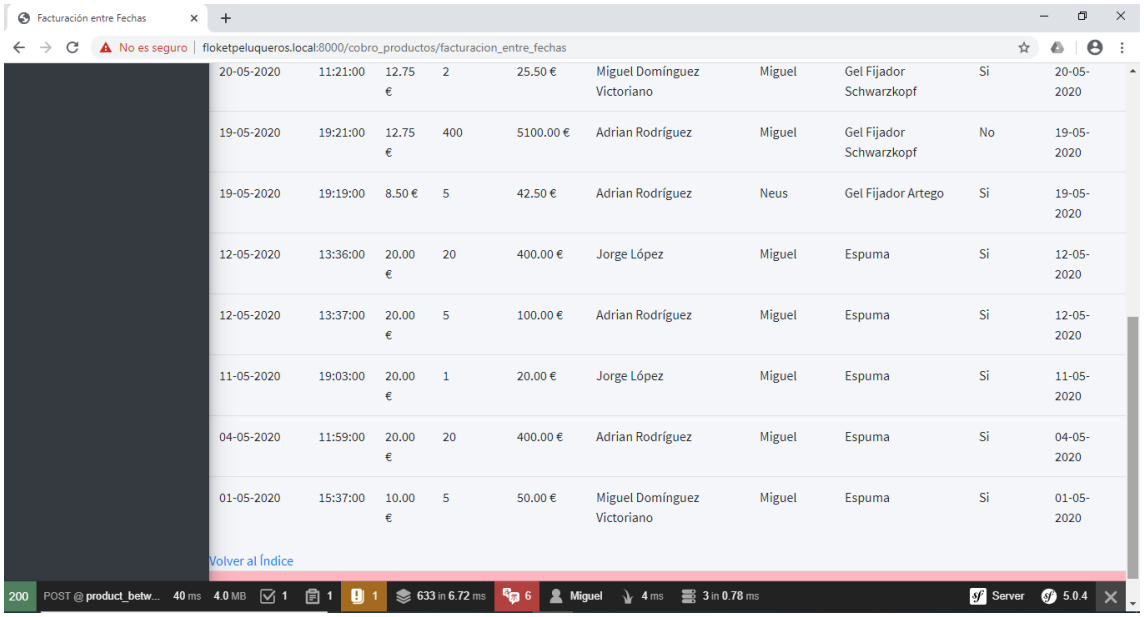


Para tener un control de la gente que no ha pagado un producto, podemos acceder al apartado de **Morosos**, en el podemos ver los productos que no han sido pagados por un clientes.



Para saber la facturación que se ha generado en ese dia, podemos acceder al apartado **Facturación diaria**.

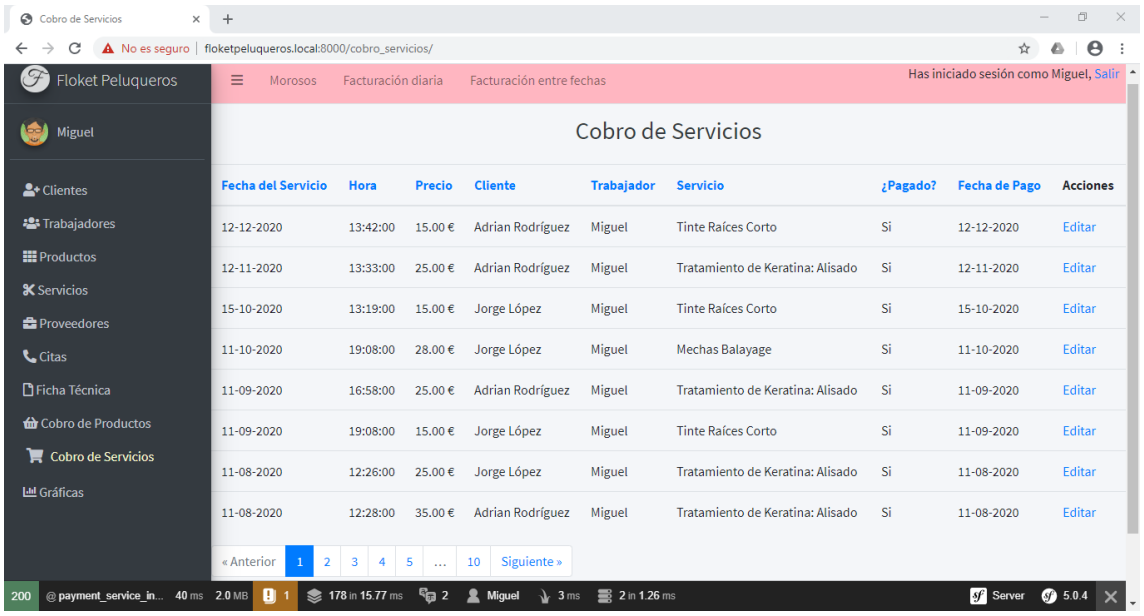
Por último, también tenemos acceso a la **Facturación entre fechas**, donde podemos seleccionar un rango de fechas para ver su facturación.

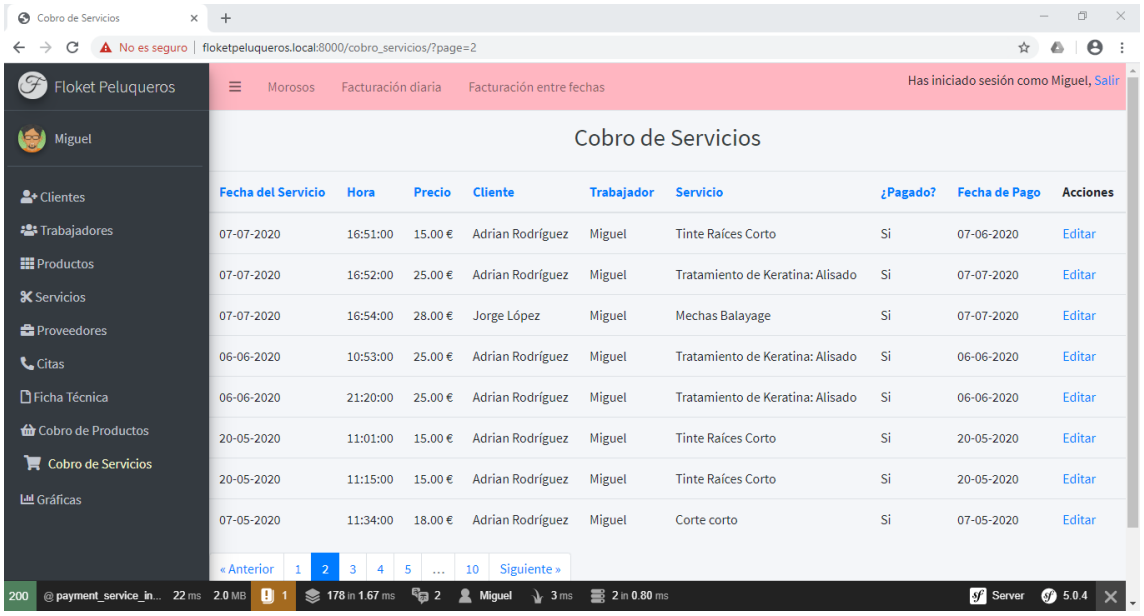


4.10.- Cobro de Servicios

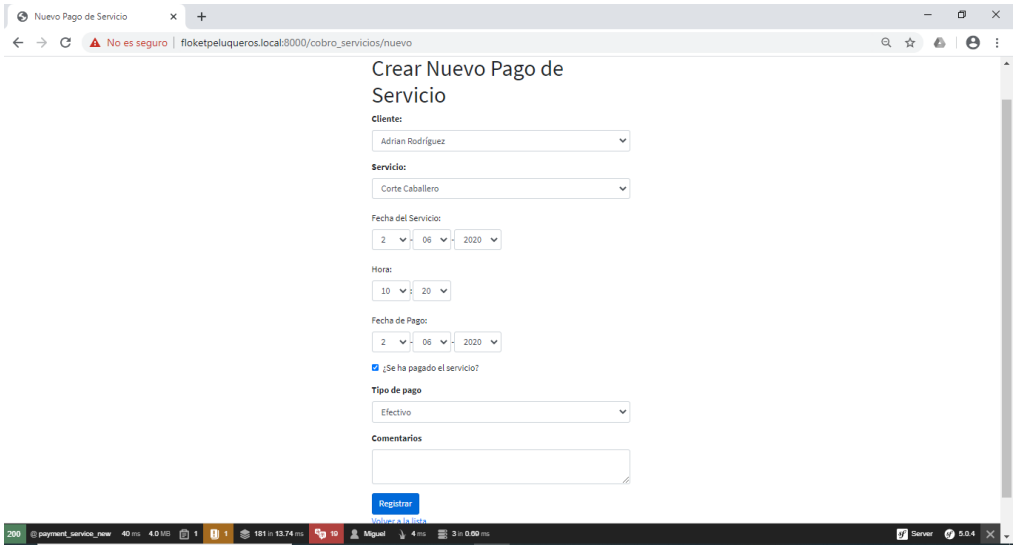
Ahora, voy a explicar el apartado referente al cobro de servicios, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos realizar toda la gestión referente al cobro de servicios a los clientes.

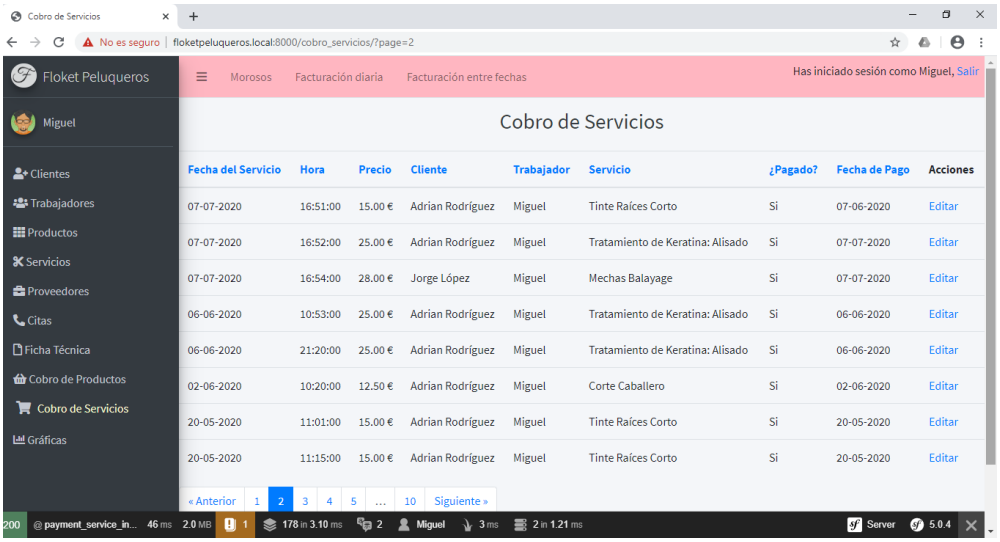
Al acceder al apartado **Cobro de servicios**, podemos ver todos los cobros que se le han realizado a los clientes paginados en diferentes páginas para no tener que hacer interminables desplazamientos hacia abajo.



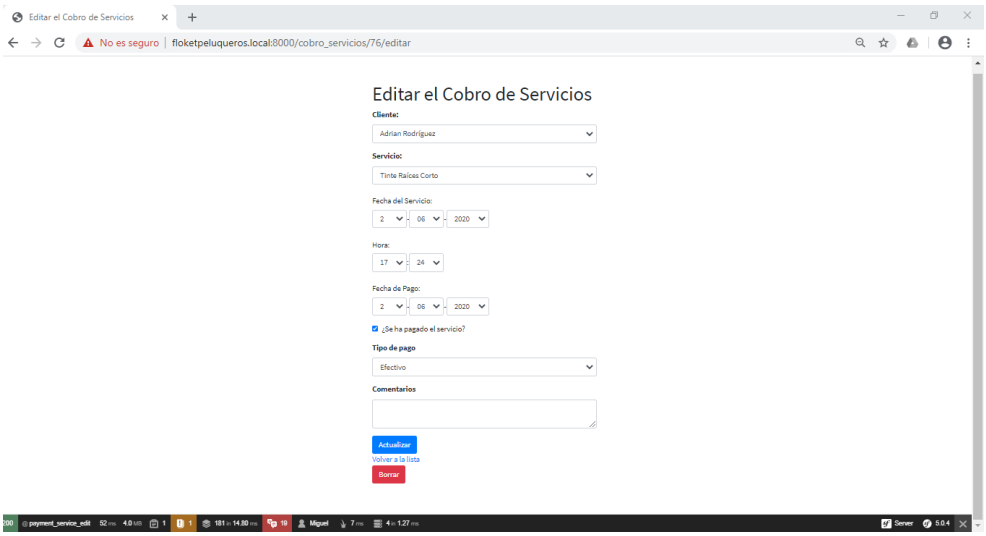


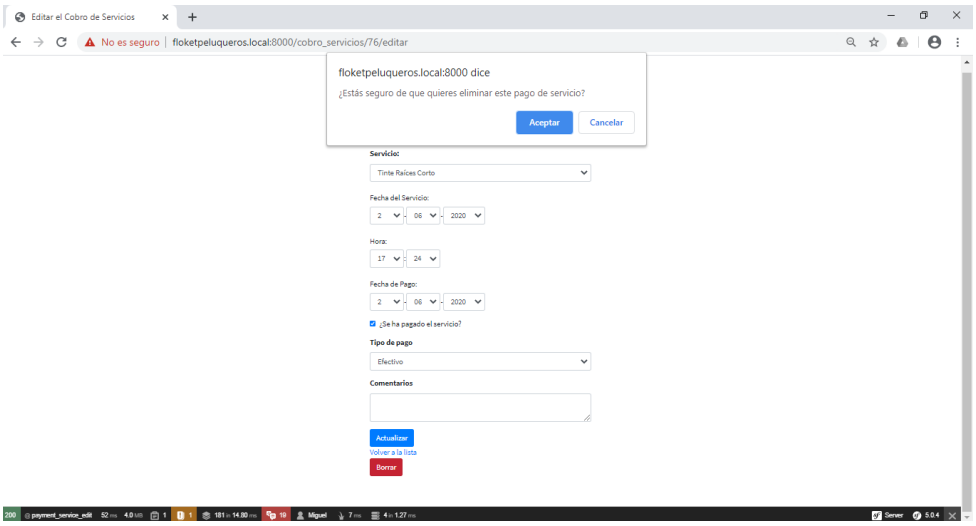
Para crear un nuevo cobro de un servicio, hay que acceder a Crear nuevo.



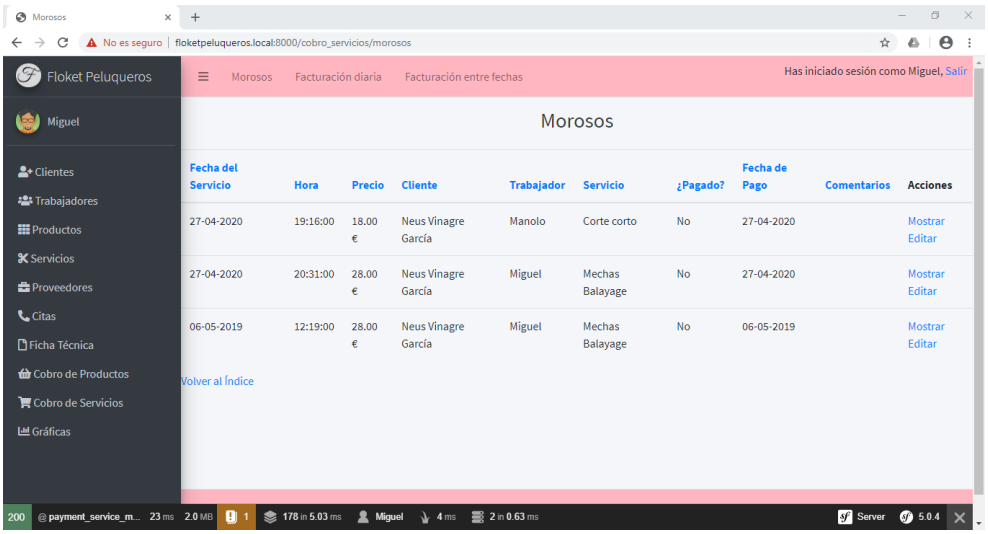


Podemos modificar un registro accediendo al apartado de Acciones en el panel principal, también podemos eliminar un registro.



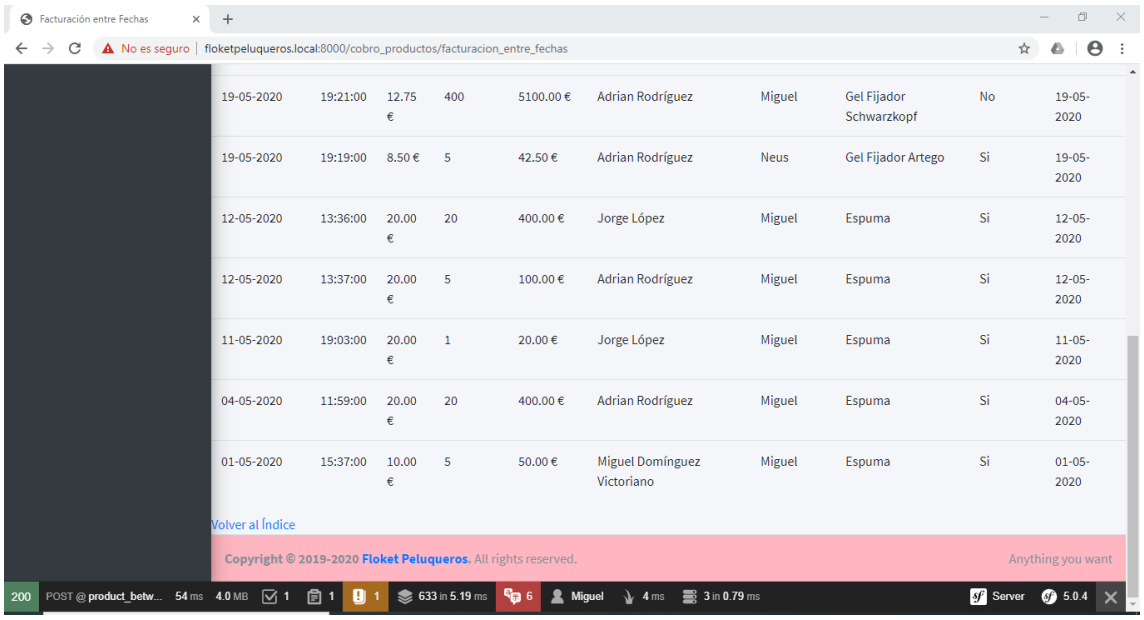


Para tener un control de la gente que no ha pagado un servicio, podemos acceder al apartado de **Morosos**, en el podemos ver los servicios que no han sido pagados por un clientes.



Para saber la facturación que se ha generado en ese dia, podemos acceder al apartado **Facturación diaria**.

Por último, también tenemos acceso a la **Facturación entre fechas**, donde podemos seleccionar un rango de fechas para ver su facturación.

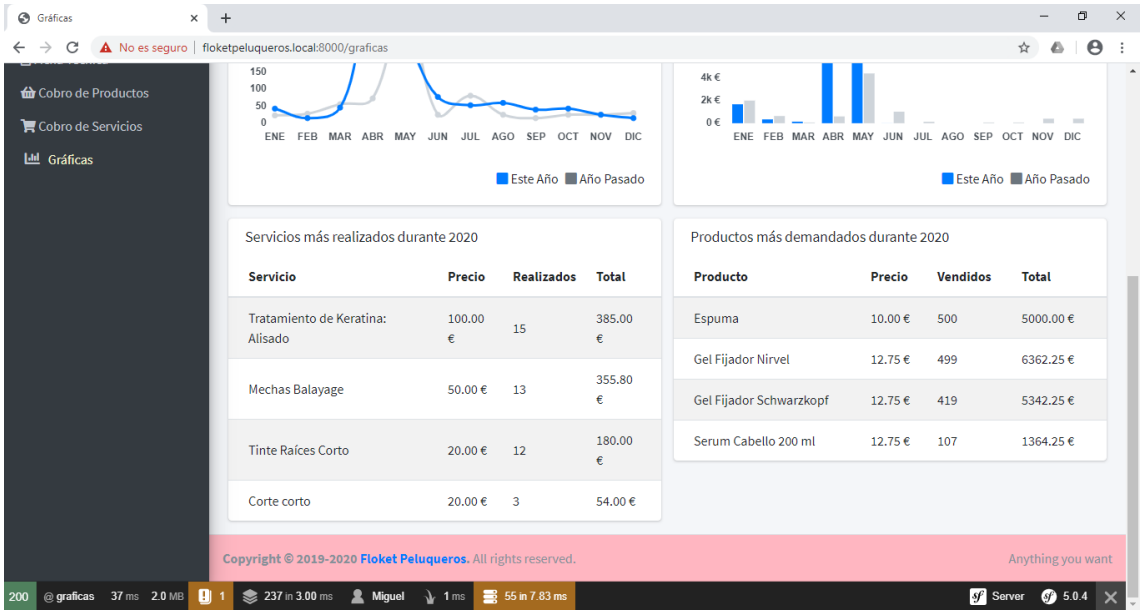


4.11.- Gráficas

Ahora, voy a explicar el apartado referente a las gráficas, aquí como ya explicamos y desarrollamos en profundidad en el apartado 2 Análisis del problema, podemos visualizar toda la gestión del salón de belleza.

En primer lugar, podemos ver las gráficas de líneas y de barras que representan las ganancias de productos y servicios comparándolos con el año anterior.

A continuación, podemos ver un listado de los productos y servicios más demandados en el salón de belleza.



Por último, podemos descargar unos documentos en PDF con las ganancias de los productos y de los servicios desglosadas por meses del año anterior y del año actual.

Floket Peluqueros

Informe de Ganancias por Servicios Prestados a fecha de: 02-06-2020

Año 2019		Año 2020		% Beneficios
Mes	Ganancias	Mes	Ganancias	
Enero	23.00 €	Enero	43.00 €	86.96 %
Febrero	28.00 €	Febrero	15.00 €	-46.43 %
Marzo	56.00 €	Marzo	46.00 €	-17.86 %
Abril	73.00 €	Abril	341.80 €	368.22 %
Mayo	367.00 €	Mayo	251.00 €	-31.61 %
Junio	25.00 €	Junio	77.50 €	210 %
Julio	81.00 €	Julio	53.00 €	-34.57 %
Agosto	25.00 €	Agosto	60.00 €	140 %
Septiembre	15.00 €	Septiembre	40.00 €	166.67 %
Octubre	25.00 €	Octubre	43.00 €	72 %
Noviembre	25.00 €	Noviembre	25.00 €	0 %
Diciembre	30.00 €	Diciembre	15.00 €	-50 %
773.00 €		1010.30 €		237.3 €

Floket Peluqueros

Informe de Ganancias por Productos Vendidos a fecha de: 02-06-2020

Año 2019		Año 2020		% Beneficios
Mes	Ganancias	Mes	Ganancias	
Enero	2000.00 €	Enero	1675.00 €	-16.25 %
Febrero	637.50 €	Febrero	340.00 €	-46.67 %
Marzo	63.75 €	Marzo	140.00 €	119.61 %
Abril	600.00 €	Abril	11996.50 €	1899.42 %
Mayo	4400.00 €	Mayo	6201.75 €	40.95 %
Junio	1020.00 €	Junio	8.50 €	-99.17 %
Julio	127.50 €	Julio	0 €	-100 %
Agosto	0 €	Agosto	0 €	0 %
Septiembre	63.75 €	Septiembre	0 €	-100 %
Octubre	63.75 €	Octubre	0 €	-100 %
Noviembre	400.00 €	Noviembre	0 €	-100 %
Diciembre	400.00 €	Diciembre	0 €	-100 %
9776.25 €		20361.75 €		10585.5 €

5.- Conclusiones y trabajos futuros

Personalmente estoy muy contento del trabajo que he realizado porque considero que la aplicación que he desarrollado es muy completa y se podría usar perfectamente en un ámbito real, una vez terminada la aplicación se la mostré a la jefa de mi pareja y se quedó muy sorprendida del resultado ya que se ajusta muy bien a las demandas del salón de belleza.

Con este Proyecto Final de Carrera he conseguido aquello que me infundía mucho respeto, que era manejarme en el mundo de la programación, ya que es mi talón de Aquiles desde que empecé a estudiar Informática hace 10 años, he evolucionado mucho en este mundo ya que he ido aprendiendo a mi ritmo y además como era un proyecto personal moldeado a mi gusto, lo he disfrutado muchísimo.

Evidentemente ha habido días en los que me frustraba no conseguir algo de lo que me proponía para ir haciendo la aplicación y mi cabeza le daba mil vueltas hasta que se me iluminaba la bombilla y me iba corriendo al ordenador para programarlo, creo que este mundo engancha por eso. Hay días en los que sientes que no sabes hacer nada y otros días en que crees que podrías conquistar el mundo.

Se que me queda muchísimo por aprender y espero poder seguir formándome en este mundo. Ahora que ya tengo el gusanillo dentro.

Como ampliación al proyecto se podrían hacer varias cosas, por ejemplo:

- En el mes del aniversario de un cliente que se le avisase mediante correo electrónico o por teléfono de que tiene un servicio gratuito.
- Crear una página para consultar el tiempo de AEMET mediante una API para que los clientes sepan el tiempo que va a hacer.
- Quería plantear la aplicación de forma que los clientes pudiesen pedir cita directamente online y un trabajador confirmase la cita.

6.- Bibliografía y referencias

- Todas las referencias con respecto al salón de belleza han sido proporcionadas por Mercedes Alonso y Neus Vinagre, dueña y trabajadora respectivamente del salón de belleza Merce Peluqueros de Alcoy.
- https://symfony.com/doc/current/index.html
- https://www.php.net/
- https://dev.mysql.com/doc/
- https://www.w3schools.com/html/
- https://www.w3schools.com/css/
- https://www.w3schools.com/js/default.asp
- https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/dql-doctrine-query-language.html
- https://twig.symfony.com/
- https://getbootstrap.com/
- https://adminlte.io/
- https://getcomposer.org/
- https://www.phpmyadmin.net/docs/

- https://httpd.apache.org/
- https://ubuntu.com
- https://es.wikipedia.org/wiki/CRUD
- https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador
- https://es.wikipedia.org/wiki/Symfony
- https://es.wikipedia.org/wiki/Doctrine
- https://es.wikipedia.org/wiki/MySQL
- https://es.wikipedia.org/wiki/Twig_(motor_de_plantillas)
- https://es.wikipedia.org/wiki/HTML5
- https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada
- https://es.wikipedia.org/wiki/Bootstrap_(framework)
- https://es.wikipedia.org/wiki/JavaScript
- https://es.wikipedia.org/wiki/PHP
- https://github.com/
- https://www.youtube.com/c/dfbastidas
- https://www.pildorasinformaticas.es/
- https://die-antwort.eu/techblog/2017-12-setup-raspberry-pi-for-kiosk-mode/
- https://es.wikipedia.org/wiki/Raspberry_Pi

7.- Anexos

7.1.- Entorno de desarrollo

Parto de una instalación limpia de **Ubuntu** **18.04 LTS**, actualizada con los últimos parches de seguridad de la distribución. También se ha instalado y configurado:

- Servidor **Apache** 2.4.29
- **MySQL** 5.7.30
- **PHP** 7.4.4
- **PhpMyAdmin**
- **MySQL WorkBench**
- **PhpStorm**
- **Composer**

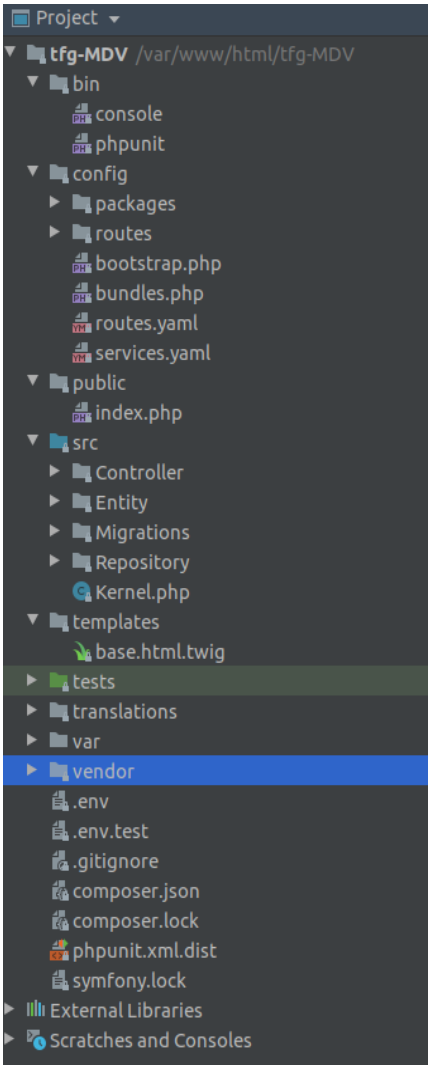
Una vez instalado y configurado todo el entorno de desarrollo, se tiene que instalar el proyecto Symfony mediante la línea de comandos, según las necesidades se puede instalar un proyecto web tradicional, un microservicio, una aplicación de consola o una API. Para ello tenemos estos dos comandos:

La versión 4.4.8. tiene soporte extendido hasta 2023, pero he elegido la última versión en este momento que es la 5 para ver los cambios y mejoras realizados.

Si no se especifica la versión a instalar, por defecto se instala la última versión, que en este caso es la versión 5.

```
composer create-project symfony/website-skeleton

composer create-project symfony/skeleton
```

La estructura de directorios de Symfony varía en función al tipo de proyecto que hayamos creado. En nuestro caso, tenemos la siguiente estructura:



- **config**: Contiene los ficheros de configuración del proyecto (YAML), por ejemplo, dentro de packages tenemos los archivos routing.yaml, security.yaml.

- **public**: Contiene todos los archivos que se pueden hacer públicos, por ejemplo, los css, los js, los plugins, las imágenes, …

- **src**: Esta carpeta contiene el código relativo a los controladores, las entidades, los formularios y los repositorios.

- **templates**: Esta carpeta contiene las plantillas de las diferentes páginas web que se muestran en el proyecto.

- **translation**: Contiene los archivos de traducción de las diferentes plantillas de la aplicación.

- **var**: Dentro de esta ruta encontramos los directorios caché, logs y sessions donde la aplicación guarda los registros, sesiones y archivos caché.

- **vendor**: Aquí se encuentran las dependencias de código de terceros de nuestra aplicación

- **.env:** Contiene la configuración de acceso a la base de datos, entre otros datos.

- **composer.json**: Contiene todas las dependencias de código de terceros instaladas en nuestra aplicación.

Lo primer que tenemos que hacer es realizar la configuración para la conexión a la base de datos desde el archivo .env :

```
DATABASE_URL=mysql://root:[redacted]@127.0.0.1:3306/FloketPeluquerosDB?serverVersion=5.7
```

Configuramos el tipo de base de datos, en este caso es MySQL, pero podría ser MondoDB, o cualquier otra, junto al usuario con privilegios de root, su contraseña, la ip y el puerto del servidor y el nombre que se le va a dar a la base de datos.

Una vez configurado este archivo, para poder crear la base de datos debemos introducir el siguiente comando:

```
php bin/console doctrine:database-create
```

Ahora ya tenemos nuestra base de datos creada.

7.2.-Puesto de trabajo con Raspberry pi

Con este apartado pretendo crear un puesto de trabajo con el mínimo coste económico posible, ya que la única función que realiza el ordenador es navegar por el servidor web para trabajar por la aplicación, por tanto, el ordenador cliente únicamente va a utilizar un navegador web. No se necesita tener un ordenador muy potente para acceder a la aplicación creada.

Este apartado lo he hecho con un Raspberry pi 2b que tenía guardada en casa sin darle uso, actualmente se puede comprar este miniordenador por unos 40€, también nos haría falta un monitor y un teclado más ratón para poder trabajar. El coste de un monitor barato oscila entre los 60 y los 80€ y un conjunto de teclado más ratón entre 10€ y 20€; por tanto, este puesto de trabajo puede costar entre 110€ y 140€.

La Rasberry pi posee 4 conexiones USB 2.0, conexión RJ45 para ethernet, jack de audio y rca, conexión HDMI, alimentación micro USB y conexión para tarjeta micro sd.

Quiero configurar la Rasberry Pi de tal manera que cuando se encienda, automáticamente se abra el navegador web con la página de la aplicación creada y no se pueda navegar por otra página web ni se pueda cerrar el navegador web. Este modo se conoce como **modo** quiosco, **modo** kiosco o **Kiosk** Mode en inglés, y consiste en simplificar al máximo cualquier sistema operativo, para que tenga acceso únicamente a una aplicación, normalmente un navegador web.

En primer lugar, he descargado e instalado en la tarjeta sd el sistema operativo Raspian Lite, este sistema operativo es la versión sin entorno gráfico, a posteriori le he instalado los componentes básicos para que pueda abrir el navegador web.

Una vez instalado el sistema operativo, accedemos a él mediante su usuario y contraseña que por defecto con pi y raspberry y procedemos a realizar las primeras configuraciones antes de instalar nada.

Para ello con el comando `raspi-config` procedemos a configurar la distribución del teclado, el acceso a la wiFi, el acceso automático para que no pida cada vez usuario y contraseña, …

Normalmente, el entorno gráfico para GNU / Linux consta de cuatro partes:

- Servidor X (generalmente X.Org)
- Administrador de ventanas (Openbox, XFWM,…)
- Entorno de escritorio (PIXEL, LXDE, MATE,…)
- Administrador de inicio de sesión (por ejemplo, LightDM)

Lo mínimo que necesitamos es un X server y un administrador de ventanas. Instalemos solo eso:

```
sudo apt-get install --no-install-recommends xserver-xorg x11-xserver-utils xinit openbox
```

Usaremos Chromium porque proporciona un buen modo kiosco:

```
sudo apt-get install --no-install-recommends chromium-browser
```

Ahora, con todo en su lugar, podemos configurar Openbox. Edite `/etc/xdg/openbox/autostart` y reemplace su contenido con lo siguiente:

```
   # Disable any form of screen saver / screen blanking / power
   management
1  xset s off
2  xset s noblank
3  xset -dpms
4
5  # Allow quitting the X server with CTRL-ATL-Backspace
6  setxkbmap -option terminate:ctrl_alt_bksp
7
8  # Start Chromium in kiosk mode
9  sed -i 's/"exited_cleanly":false/"exited_cleanly":true/'
10 ~/.config/chromium/'Local State'
11 sed -i 's/"exited_cleanly":false/"exited_cleanly":true/;
12 s/"exit_type":"[^"]\+"/"exit_type":"Normal"/'
   ~/.config/chromium/Default/Preferences
   chromium-browser --disable-infobars --kiosk 'http://your-url-here'
```

Primero, deshabilitamos la pantalla en blanco y la administración de energía (no queremos que nuestra pantalla se quede en blanco o incluso se apague por completo después de un tiempo). Luego permitimos salir del servidor X presionando Ctrl-Alt-Backspace. (Debido a que no instalamos un entorno de escritorio, no habrá un botón "Cerrar sesión" o similar).

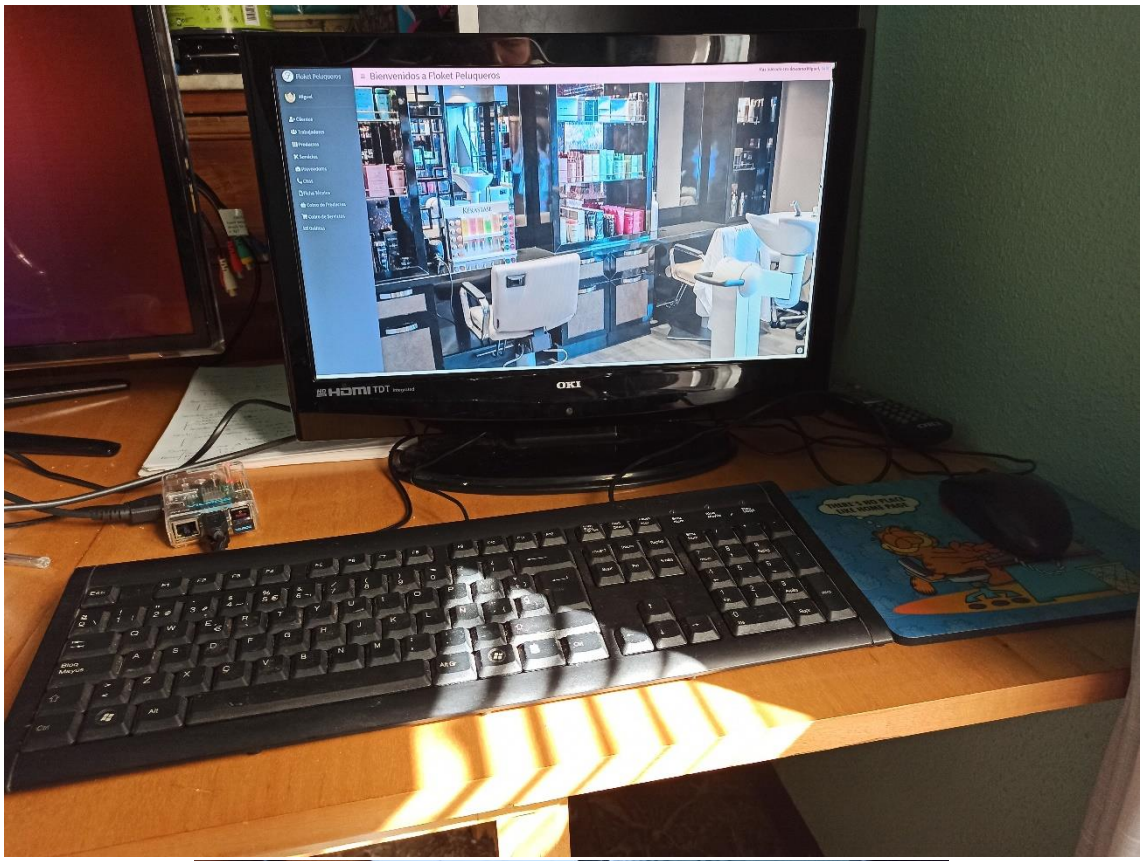
Finalmente le decimos a Openbox que inicie Chromium en modo kiosco. Esto resulta un poco complicado porque a Chromium le encanta mostrar varias burbujas de herramientas para la restauración de la sesión, etc. La forma más sencilla de evitar todo esto parece ser engañar a Chromium para que piense que salió limpiamente la última vez que se ejecutó.

Ahora solo queda una cosa: el servidor X debería iniciarse automáticamente al arrancar.

Debido a que ya configuramos el Pi para que inicie sesión automáticamente el usuario pi, podemos usar su **.bash_profile** para iniciar X. Simplemente se tiene que agregar la siguiente línea:

```
[[-z $ DISPLAY && $ XDG_VTNR -eq 1]] && startx - -nocursor
```

La condición asegura que X solo se inicie en la primera consola (y si aún no se está ejecutando). Debido a que el inicio de sesión automático usa la primera consola, esto tiene el efecto deseado de iniciar automáticamente el servidor X en el arranque.


Bienvenidos a Floket Peluqueros
OKI


VILROS
802.11n