Índice

Introducción de la web

El proyecto aborda el desarrollo de una plataforma web para Sacoba, una tienda especializada en muebles de cocina. En un mundo donde el diseño y la funcionalidad de los espacios son esenciales, Sacoba se dedica a ofrecer una amplia variedad de sillas de cocina, bancos de cocina y mesas de cocina que combinan calidad, estilo y comodidad. Esta plataforma web tiene como objetivo proporcionar a los clientes una experiencia de compra fácil, agradable y totalmente personalizable, facilitando el acceso a muebles que transformen su cocina en un espacio acogedor y funcional.

La motivación para la realización de este proyecto nace de la necesidad de poner al día a la empresa y no quedar rezagados frente a la competencia, que ya ha adoptado tecnologías digitales para llegar a sus clientes de manera más eficiente. La implementación de esta web permitirá a Sacoba no solo mantener su posición en el mercado, sino también expandir su alcance y mejorar su servicio al cliente.

El objetivo principal es facilitar al usuario el acceso a los productos de Sacoba. La nueva plataforma web ofrecerá una interfaz intuitiva y amigable que permitirá a los clientes explorar y adquirir sillas de cocina, bancos de cocina y mesas de cocina con facilidad. A través de un diseño centrado en el usuario, la web buscará mejorar la experiencia de compra, proporcionando información detallada, imágenes de alta calidad y opciones de personalización, todo con el fin de que cada cliente encuentre exactamente lo que necesita para su cocina de manera rápida y eficiente.

Explicación de las páginas

Página producto “Mesas”

Comienza haciendo tres consultas a la base de datos de la siguiente forma

```
export default async function ProductoMesa() {
  const promiseMesasNovedad = selectsMesasNovedad();
  const promiseMesasTendencias = selectsMesasTendencia();
  const promiseMesasModelos = selectsMesasModelo();

  const [mesasNovedad, mesasTendencias, mesasModelos] = await Promise.all([
    promiseMesasNovedad,
    promiseMesasTendencias,
    promiseMesasModelos,
  ]);
```

Cada promesa es un select que recoge las mesas que corresponden a las siguientes condiciones, estos son los selects

```
//Selects de mesas
export async function selectsMesasTendencia() {
  const todasMesas = await db
    .select()
    .from(mesas)
    .where(eq(mesas.tendencia, true));
  return todasMesas;
}
export async function selectsMesasNovedad() {
  const todasMesas = await db.select().from(mesas).where(eq(mesas.nuevo, true));
  return todasMesas;
}
export async function selectsMesasModelo() {
  const todasMesas = await db
    .selectDistinct()
    .from(mesas)
    .orderBy(mesas.id)
    .groupBy(mesas.modelo);
  return todasMesas;
}
```

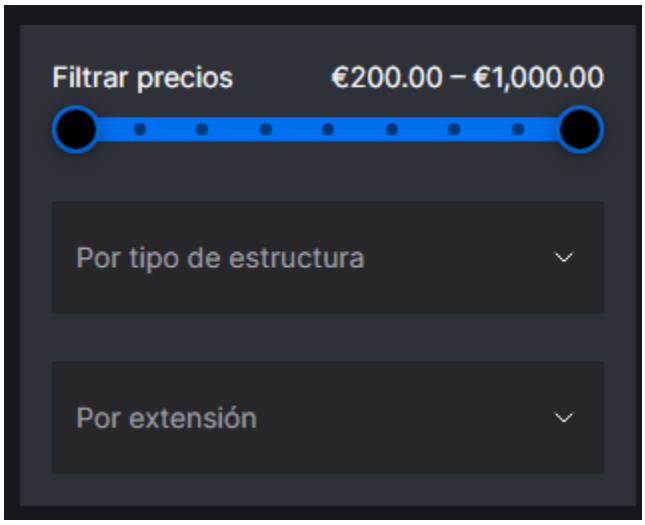
Con la información de las consultas guardadas en variables, pasa a cada componente correspondiente la información requerida

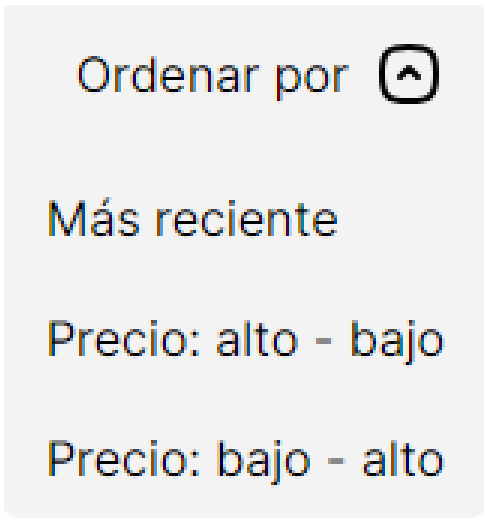
```
{/* Seccion mesas novedades */}
<ObjMesasTendencias mesasTendencias={mesasTendencias}/>

{/* Seccion mesas novedades */}
<ObjMesasNovedades mesasNovedad={mesasNovedad}/>

{/* Apartado todas las mesas */}
<ObjMesasTotales mesasTotales={mesasModelos} />
```

El **"ObjMesasTotales"**, es el más complejo, coge la variable "mesasModelos", que es un array de objetos mesas, y los trata.
En esta página se pueden hacer los siguiente filtros a la información





Gracias a estos filtros, los filtros son combinados y se tienen en cuenta los unos a los otros

```
// Filtrar mesas por tipo de base
const mesasFiltradasPorTiposBase = tiposBaseSeleccionados.length > 0
? mesasFiltradasPorPrecio.filter((mesa) => tiposBaseSeleccionados.includes(mesa.tipoBase))
: mesasFiltradasPorPrecio;

// Filtrar mesas por extensión
const mesasFiltradasPorExtension = tiposExtensionSeleccionados.length > 0
? mesasFiltradasPorTiposBase.filter((mesa) => tiposExtensionSeleccionados.includes(mesa.extension))
: mesasFiltradasPorTiposBase;

// Establecer las mesas ordenadas
setMesasOrdenadas(mesasFiltradasPorExtension);
```

Las mesas o el producto oportuno, se muestra por pantalla de manera recursiva y atendiendo al número total de productos existentes en la BBDD, mapeando un array, y pasando los datos de cada mesa al componente **TarjetaDisplayInfo**

```
{mesasOrdenadas.length > 0 ? (
  mesasOrdenadas.map((mesa) => (
    <TarjetaDisplayInfo key={mesa.id} datos={mesa} />
  ))
) : (
  <div>No hay mesas disponibles</div>
)}
```

El componente, se encarga individualmente de mostrar la información que ha recibido de cada objeto. Y gestiona a través de "*onClick*" si el usuario ha seleccionado esa mesa en concreto, si es así, pushea el modelo de la mesa seleccionada a la URL, y nos redirige a la página de **"productoConcretoMesa"**.

```
<Link href={`/ProductoConcretoMesa/${datos.modelo}`}
onClick={handleClick}
>
  <Image
    className="w-full h-60 cursor-pointer"
    src={`/productos/mesas/${datos.imagen}`}
    alt="Imagen mesa"
    width={500}
    height={500}
  />
</Link>
```

```
const handleClick = () => {
  router.push(`/ProductoConcretoMesa/${datos.modelo}`);
};
```

```tsx
export default async function ProductoConcreto({ params }: { params: {modelo: string} }) {
  const promiseMesas = selectsMesaSeleccionada(params.modelo);
  const promiseColores = selectsColoresMesas();
  const [mesas, colores] = await Promise.all([promiseMesas, promiseColores])

  return (
    <main className="flex flex-col items-center">
      <CompClienteMesa mesaSeleccionada={mesas} colores={colores} />
    </main>
  );
}
```

Este será un componente de servidor, que cogerá los parámetros de la ruta (ya que hemos establecido el modelo en ella), y hace dos consultas a la BBDD de la misma forma que hemos hecho las anteriores, pero esta vez seleccionando solo los productos que coincidan con ese modelo, junto a las diferentes opciones de colores que tiene ese producto. Todo ello se lo pasamos al componente de cliente para poder tratar la información adquirida.

El componente para personalizar la mesa tiene bastantes estados, y de diferentes tipos para ir guardando la información que el usuario vaya eligiendo

```tsx
//Estado globales
const { modalVisible, setModalVisible } = useModal();
const { colorElegido, modeloElegido, rutaImagen, grupo } =
  useColorSeleccionado();
const { setMesaFinal } = useMesaFinal();
const { setIndexMesaFinal } = useIndexMesaFinal();

//Estados para coger las variables de la mesa
const [dimensionesMesa, setDimensionesMesa] = useState<string[]>([]);
const [coloresPataMesa, setColoresPataMesa] = useState<string[]>([]);
const [alturasMesa, setAlturasMesa] = useState<string[]>([]);
const [coloresFiltrados, setColoresFiltrados] = useState<TipoColor[]>([]);
const [grosorTapa, setGrosorTapa] = useState(false);

//Estados para ver el indice de la seleccion del usuario
const [indexDimension, setIndexDimension] = useState(0);
const [indexColorPata, setIndexColorPata] = useState(0);
const [indexAltura, setIndexAltura] = useState(0);

//Estados que almacenan la eleccion final del usuario
const [dimension, setDimension] = useState("");
const [grosorElegido, setGrosorElegido] = useState("");
const [alturaElegida, setAlturaElegida] = useState("");
const [colorPataElegido, setColorPataElegido] = useState(coloresPataMesa[0]);
```

Una vez tenemos todos los datos requeridos, los guardamos en este estado global, gracias a la librería de **Zustand**

```ts
// Estado global para guardar los datos finales de la mesa
type MesaStateFinal = {
  mesa: {
    modelo: string;
    dimension: string;
    acabado: string;
    grupo?: string;
    color: string;
    grosor?: string;
    colorPata: string;
    altura: string;
    precio: number;
    cantidad: number;
  };
  setMesaFinal: (
    modelo: string,
    dimension: string,
    acabado: string,
    grupo: string | undefined,
    color: string,
    grosor: string | undefined,
    colorPata: string,
    altura: string
  ) => void;
  setPrecioMesaFinal: (precio: number) => void;
  setCantidadMesas: (cantidad: number) => void;
};
```

Y por último en este proceso, pasamos a la sección del cálculo del precio, el cual tiene en cuenta varios factores para proceder al cálculo del precio total

```
export const calculoGrupo = (
  grupo: string,
  acabado: string,
  grosor: string
) => {
  switch (acabado) {
    case "silestone g1":
      if (grosor === "12mm") {
        switch (grupo) {
          case "g1":
            return 1;
          case "g2":
            return 1.05;
          case "g3":
            return 1.1;
          case "g4":
            return 1.15;
          case "g5":
            return 1.2;
          case "g6":
            return 1.3;
        }
```

```
    } else if (grosor === "20mm") {
      switch (grupo) {
        case "g1":
          return 1.05;
        case "g2":
          return 1.1;
        case "g3":
          return 1.15;
        case "g4":
          return 1.35;
        case "g5":
          return 1.40;
        case "g6":
          return 1.50;
      }
    }
    break;
```

```
//Calcular el incremento del grupo (si aplica)
if (index.grupo && index.grosor) {
  const precioGrupo = calculoGrupo(index.grupo, material, index.grosor);
  if (precioGrupo !== undefined) {
    setPrecioGrupo(precioGrupo);
  }
}
```

Esto por ejemplo, es una función que se encarga de establecer el precio en función del material, grosor y grupo. Cuando ha terminado de calcular todos los precios, lo que tendríamos sería un estado con la siguiente información de la mesa.

```
mesa:
{
  mesa: {
    producto: 'Mesa',
    modelo: 'CORA',
    dimension: ' 122x70(172x70)',
    acabado: 'Silestone g5',
    grupo: 'g5',
    color: 'Calypso',
    grosor: '12mm',
    colorPata: 'aluminio',
    colorExtensible: 'Chocolate F2200 Mate',
    altura: '74',
    precio: 1109,
    cantidad: 2
  }
```

```
silla:
{
  silla: {
    producto: 'Silla',
    modelo: 'BARI',
    formato: 'Taburete alto con respaldo',
    acabado: 'Tapizado premium',
    color: 'Gemini Blanco',
    colorPata: 'aluminio',
    precio: 765,
    cantidad: 5
  }
```

Tratamiento de la información

Generación de Cookies

Todo comienza en la página registro, que es un componente de servidor y tiene un componente "FormRegistro" que es de cliente, este es el que efectuará toda la lógica necesaria

En el componente **FormRegistro**, se recoge toda la información recibida del form y con un *"action={clientAction}"* en el propio form, se llama a la función que va a ejecutar la lógica

```
const clientAction = async (formData: FormData) => {
  const newForm = {
    correoElectronico: formData.get("correoElectronico"),
    nombre: formData.get("nombre"),
    apellidos: formData.get("apellidos"),
    contraseña: formData.get("contraseña"),
  };

  //Validacion del lado del cliente
  const result = FormRegistroValidation.safeParse(newForm);
  if (!result.success) {
    toast.error(result.error.issues[0].message)
    return;
  }

  //Validacion del lado del servidor
  const response = await InsertarRegistro(result.data);
  if (response?.error) {
    //Manejar el error
    toast.error(response.error)
  } else {
    //Seteamos la cookie con el token que contiene la informacion del usuario
    setCookie("client-Token", response.token)

    //Si se ha insertado correctamente redirige al perfil
    redirect("/Perfil");
  }
};
```

Recoge toda la info en un objeto “newForm”, que procede a validar (en el lado del cliente), con la librería **Zod**, de esta forma:

```
export const FormRegistroValidation = z.object({
  correoElectronico: z.string().email({
    message: "Email no válido",
  }),
  nombre: z.string().trim().regex(caracteresPermitidos, {
    message: "El nombre solo puede contener letras",
  }),
  apellidos: z.string().trim().regex(caracteresPermitidos, {
    message: "Los apellidos solo pueden contener letras",
  }),
  contraseña: z.string().min(7, {
    message:
      "La contraseña tiene que tener una longitud mínima de 7 caracteres",
  }),
});
```

En función de esta validación arrojaremos los errores y cortaremos la ejecución de la función, o seguiremos con ella, mandando la información validada al lado del servidor, con un “*await InsertarRegistro(info)”*, donde estaremos llamando a una función asíncrona, que volverá a validar el form, esta vez desde el lado del servidor, luego procederá a ***hashear*** la contraseña y lo volverá a envolver todo en el objeto *usuario* para proceder a pasarlo a la función que lo insertará en la base de datos.

```
const constraseñaHasheada = await bcrypt.hash(result.data.contraseña, 10)
const usuario = {
  usuario: {
    correoElectronico: result.data.correoElectronico,
    nombre: result.data.nombre,
    apellidos: result.data.apellidos,
    contraseña: constraseñaHasheada,
  },
};
const insercionExitosa = await registrarUsuario(usuario);
```

Esta función inserta los datos en la tabla correspondiente, y genera un ***jsonWebToken*** con la información de expiración de este, la información del usuario y la clave de ***hasheo*** que proviene del archivo *".env"*

```
const tokenGenerado = jwt.sign(
    {
      // El token expira en 7 días
      exp: Math.floor(Date.now() / 1000) + 60 * 60 * 24 * 7,
      //Contiene la informacion del usuario
      usuario: {
        correoElectronico: usuario.correoElectronico,
        nombre: usuario.nombre,
        apellidos: usuario.apellidos,
      },
    },
    process.env.AUTH_USER_TOKEN!
  );
```

Por último, volviendo al componente inicial, **FormularioRegistro**, éste recibe la respuesta de las consultas y muestra un error si procede, o setea en las cookies el nuevo token generado, y nos redirige a perfil

```
//Validacion del lado del servidor
const response = await InsertarRegistro(result.data);      You, 2 weeks ago •
if (response?.error) {
  //Manejar el error
  toast.error(response.error)
} else {
  //Seteamos la cookie con el token que contiene la informacion del usuario
  setCookie("client-Token", response.token)

  //Si se ha insertado correctamente redirige al perfil
  redirect("/Perfil");
}
```

Esta función ***LeerDatosCookie,*** es usada por el resto de componentes que necesitan la información que está contenida en las Cookies. Su función es coger el token, y verificar que tenga el formato deseado, por medio de la función *verify,* la cual comprueba gracias al token local, que el guardado en la cookie sea correcto comparando ambos, esa información la pasa al objeto user, y si todo es correcto devolverá un objeto usuario, con toda la información deseada

```
export const LeerDatosCookie = async () => {
  const clientToken = cookies().get("client-Token");

  if (clientToken !== undefined && clientToken.value !== undefined) {
    try {
      const user = verify(clientToken.value, process.env.AUTH_USER_TOKEN!);
      // Convertir el objeto user a JSON
      const usuarioJSON = JSON.stringify(user);
      const usuarioObjeto = JSON.parse(usuarioJSON);
      return {
        status: true,
        usuario: usuarioObjeto.usuario,
      };
    } catch (error) {
      return {
        status: false,
      };
    }
  }
  return {
    status: false,
  };
};
```

Por último, el usuario tiene la opción en el perfil de ***CerrarSesion****,* con tan solo clickar un botón, el cual borrará la *cookie* que permitía al usuario estar logueado

```
export default function CerrarSesion() {
  const router = useRouter();

  const handleBoton = async () => {
    const response = await LogOut();
    if (response) {
      router.push('/');
    } else {
      console.log('Ha sucedido un error');
    }
  };
  return (
      <button className="bg-red-300 p-2 text-black cursor-pointer" onClick={handleBoton}>
        Cerrar sesion
      </button>
  );
}
```

Reseteo de contraseña

En la página de inicio de sesión, está implementada la opción de “**Olvidé mi contraseña**”, esta opción permite recuperar nuestra cuenta en el caso de que hayamos olvidado la contraseña.

Lo hace de la siguiente forma, tras recibir el correo electrónico del usuario, se ejecuta la función “**SendEmail**”, la cual a través de un fetch de datos manda una petición POST, en la petición también adjunta en la parte del body, la información que quiere enviar

Que lo recibe la siguiente función, la cual se encarga de mandar un correo con un *token*, el cual permitirá al usuario restablecer su contraseña

Funcionamiento del carrito

En todos los productos está la opción de agregar la configuración deseada de este al carrito, usando el botón “Añadir al carro”, el cual ejecutará la siguiente función “**HandleCarrito**”, que comprobará si el usuario está logueado o no.

```
useEffect(() => {
  //Funciones para agregar lo elegido a la BBDD
  const handleCarrito = async () => {
    const result = await InsertarCarrito(mesa);
    // Si la consulta da error
    if (!result.success) {
      toast.error(result.message);
    } else {
      const resultLocal = await InsertarCarritoLocal(mesa);
      if (resultLocal.success) {
        let carritoIds = localStorage.getItem("carrito");

        if (carritoIds !== null && resultLocal.idGenerado !== undefined) {
          let idsArray: string[] = JSON.parse(carritoIds); // Convertir a array
          idsArray.push(resultLocal.idGenerado); // Agregar el nuevo ID al array
          localStorage.setItem("carrito", JSON.stringify(idsArray));
        } else {
          localStorage.setItem("carrito", JSON.stringify([resultLocal.idGenerado]));
        }
      } else {
        toast.error("Hubo un error al procesar la solicitud.");
        return;
      }
    }
    toast.success(result.message);
  };
  //Solo lanzamos la funcion si hemos clickado en Añadir Carro
  if (guardarCarro === true) {
    handleCarrito();
  }
}, [mesa.precio, guardarCarro]);
```

Usuario logueado

Si un usuario ya se ha registrado en la página y actualmente está logueado, por medio de las cookies tendremos su *CorreoElectronico*, que actuará como clave primaria de la tabla **“Carrito”**, para saber a qué cliente corresponde la información guardada.

Aquí vemos el camino que sigue el código cuando el usuario está logueado.
Se procede a llamar a la función “**InsertarCarrito”**

```
export const InsertarCarrito = async (producto: any) => {
  // Comprobar si la sesion esta iniciada
  try {
    const cookie = await LeerDatosCookie();
    //La sesion esta iniciada
    if (cookie.status) {
      console.log(producto)
      const insercion = await registrarCarrito({
        producto: producto,
        correo: cookie.usuario.correoElectronico,
      });
      if (insercion) {
        return {
          success: true,
          message: "Producto guardado en el carrito con éxito",
        };
      }else{
        return {
            success: false,
            message: "Hubo un error al procesar la solicitud.",
          };
      }
```

Esta función a su vez llama a la consulta **“RegistrarCarrito”** para guardar la información recibida

```
export async function registrarCarrito({
  producto,
  correo,
}: {
  producto: any;
  correo: string;
}) {
  try {
    await db.insert(carrito).values({
      cliente: correo,
      tipoProducto: producto.producto,
      modelo: producto.modelo,
      detallesProducto: JSON.stringify(producto),
      precioTotal: producto.precio * producto.cantidad,
    });
    // Si la inserción se realiza sin errores, devolvemos true
    return true;
  } catch (error) {
    // Si ocurre algún error, devolvemos false
    return false;
  }
}
```

Usuario sin loguear

En esta opción no tenemos la suerte de tener una clave primaria que identifique al usuario que ha guardado el producto en el carrito, por lo tanto es un poco más complejo el procedimiento.

```
export async function registrarCarritoLocal({ producto }: { producto: any }) {
  //Lo que se guarde aqui solo durará una semana
  try {
    const insert = await db.insert(carritoLocal).values({
      tipoProducto: producto.producto,
      modelo: producto.modelo,
      detallesProducto: JSON.stringify(producto),
      precioTotal: producto.precio * producto.cantidad,
      fecha: new Date().toISOString(),
    });
    // Si la inserción se realiza sin errores, devolvemos true
    return {
      success: true,
      idGenerado: insert.lastInsertRowid,
    };
  } catch (error) {
    // Si ocurre algún error, devolvemos false
    console.log(error);
    return {
      success: false,
    };
  }
}
```

Ejecutamos esta consulta, la cual guarda los datos del producto en una tabla nueva auxiliar llamada **“CarritoLocal”**, las peculiaridades de esta tabla son que solo almacenará datos con una antigüedad máxima de 1 semana, ya que actúa como soporte de una forma temporal.

Gracias a **“Cron”**, podemos programar sentencias de manera automática y autónoma

```
import { deletePeriodicoCarritoLocal } from "./deletes";

const cron = require('node-cron');

// Función para eliminar carritos viejos
 async function eliminarCarritoLocalAntiguo() {
  const fechaLimite = new Date();
  fechaLimite.setDate(fechaLimite.getDate() - 7);
  const fecha = fechaLimite.toISOString();

  await deletePeriodicoCarritoLocal({fecha})
}

// Configura un cron job para ejecutar la función cada día a medianoche
cron.schedule('0 0 * * *', () => {
  console.log('Ejecutando limpieza de carritos viejos');
  eliminarCarritoLocalAntiguo();
});
```

Por otro lado, una vez hecha la inserción de datos en la tabla auxiliar, guardamos los ID que se han generado en el **localStorage**, y de esta la forma vinculamos al cliente con el producto, ya que el usuario no logueado tendrá localmente los IDs de los productos que ha guardado en la tabla.

```
const resultLocal = await InsertarCarritoLocal(mesa);
if (resultLocal.success) {
  let carritoIds = localStorage.getItem("carrito");

  if (carritoIds !== null && resultLocal.idGenerado !== undefined) {
    let idsArray: string[] = JSON.parse(carritoIds); // Convertir a array
    idsArray.push(resultLocal.idGenerado); // Agregar el nuevo ID al array
    localStorage.setItem("carrito", JSON.stringify(idsArray));
  } else {
    localStorage.setItem("carrito", JSON.stringify([resultLocal.idGenerado]));
  }
} else {
  toast.error("Hubo un error al procesar la solicitud.");
  return;
}
```

Mostrar el carrito

Una vez ya hemos guardado los productos de la forma correspondiente. A la hora de mostrarlos en la página carrito o haciendo *hover* en él, haremos lo siguiente.

Gracias a esta función podemos comprobar si el usuario está logueado o no, como he mencionado anteriormente

```
export const LeerDatosCookie = async () => {
  const clientToken = cookies().get("client-Token");

  if (clientToken !== undefined && clientToken.value !== undefined) {
    try {
      const user = verify(clientToken.value, process.env.AUTH_USER_TOKEN!);
      // Convertir el objeto user a JSON
      const usuarioJSON = JSON.stringify(user);
      const usuarioObjeto = JSON.parse(usuarioJSON);
      return {
        status: true,
        usuario: usuarioObjeto.usuario,
      };
    } catch (error) {
      return {
        status: false,
      };
    }
  }
  return {
    status: false,
  };
};
```

Este por ejemplo, es en el caso de que no esté logueado, por lo cual cogeremos los IDs de los productos del **LocalStorage**

```
//Si la sesion no esta iniciada
let carritoString = localStorage.getItem("carrito");

if (carritoString !== null) {
  const consultaLocal = await RecogerDatosCarritoLocal(
    JSON.parse(carritoString)
  );
  //Si la consulta sale bn
  if (consultaLocal.success && consultaLocal.carrito !== undefined) {
    const detallesProductos = consultaLocal.carrito.map((producto) => {
      const detalles = JSON.parse(producto.detallesProducto);
      return {
        ...detalles,
        id: producto.id,
      };
    });
    setObjetosCarro(detallesProductos.reverse());
  }
} else {
  setCarritoVacio(true);
}
```

De esta forma haremos la siguiente consulta y los resultados los mostramos con el estado **“setObjetosCarro”**, y con el *.reverse*, para que salga el más reciente el primero

```
export async function selectCarritoUsuarioLocal(id: []) {
  try {
    const carritoUsuario = await db
      .select()
      .from(carritoLocal)
      .where(inArray(carritoLocal.id, id));
    //Comprobamos si hay registros
    if (carritoUsuario.length > 0) {
      return {
        success: true,
        carrito: carritoUsuario,
      };
    } else {
      return {
        success: false,
        message: "El carrito está vacio",
      };
    }
  } catch (error: any) {
    //Si salta un error en la consulta
    return {
      success: false,
    };
  }
}
```

Modificar el carrito

Estando en la página de carrito, el cliente podrá modificar la cantidad de unidades de un mismo producto y eliminar el producto del carrito.

Para ello se usan estas dos funciones, las cuales, como en todo lo demás discriminan si el usuario está logueado o no, para acceder a una tabla u otra

```
const eliminarElemento = async () => {
  //Comprobamos si el usuario esta logueado
  const user = await LeerDatosCookie();
  if (user.status) {
    //Si esta logueado quitamos el producto de la tabla CARRITO
    const response = await EliminarProductoCarrito(producto, "");
    if (response) {
      toast.success("Producto eliminado del carrito");
      setSumaTotal(clave, 0);
      await onDelete();
      return;
    }
  } else {
    //Si NO esta logueado quitamos el producto de la tabla CARRITOLOCAL
    const responseLocal = await EliminarProductoCarrito(producto, "local");
    if (responseLocal) {
      toast.success("Producto eliminado del carrito");
      setSumaTotal(clave, 0);
      await onDelete();
      return;
    }
  }
};
```

```
const modificarCantidadBBDD = async (cantidadCalculada: number) => {
  const productoConNuevaCantidad = {
    ...producto,
    cantidad: cantidadCalculada,
  };
  //Comprobamos si el usuario esta logueado
  const user = await LeerDatosCookie();
  if (user.status) {
    //Si esta logueamos ejecutamos esta consulta
    await ModificarCantidadProducto(productoConNuevaCantidad, "");
  }else{
    await ModificarCantidadProducto(productoConNuevaCantidad, "local");
  }
};
```

Convergencia de carritos

Por último, si un usuario que ha configurado su carrito sin estar logueado, se registra o loguea, se tendrán que mover los elementos de la tabla **“carritoLocal”** a **“carrito**”, ya que al tener la clave primaria del cliente podremos almacenar su carrito sin restricciones.

```
  const response = await InsertarRegistro(result.data);
  if (response?.error) {
    //Manejar el error
    toast.error(response.error);
    setLoading(false);
  } else {
    //Chequeamos si el cliente tiene productos en el carrito
    const carritoString = localStorage.getItem("carrito");
    if (carritoString !== null) {
      const carritoIds = JSON.parse(carritoString);
      await InsertarCarritoExistente(
        carritoIds,
        result.data.correoElectronico
      );
      //Limpiamos el localStorage
      localStorage.clear();
    }
    //Seteamos la cookie con el token que contiene la informacion del usuario
    setCookie("client-Token", response.token);

    //Si se ha insertado correctamente redirige al perfil
    router.push("/Perfil");
  }
  setIsLoading(false);
};
```

De esta forma tras completar el registro del nuevo usuario, se procede a ejecutar la función **“InsertarCarritoExistente”**.

Esta función llama a la siguiente consulta, que tendrá varias acciones, la primera consultar los productos que corresponden con los IDs guardados en el **localStorage**, en la tabla **"CarritoLocal"**, una vez tenemos esos datos, procedemos a insertarlos uno por uno, en la tabla **"Carrito"**, recorriendo el array que hemos generado previamente con la consulta, y después de cada insert, se llama a la función **"deleteProductoCarritoLocal2"**, que eliminará definitivamente el elemento de la tabla **"CarritoLocal"**.

```
export async function juntarAmbosCarritos({
  carritoIds,
  correo,
}: {
  carritoIds: [];
  correo: string;
}) {
  try {
    const productosEnLocal = await db
      .select({
        id: carritoLocal.id,
        producto: carritoLocal.detallesProducto,
        modelo: carritoLocal.modelo,
        tipo: carritoLocal.tipoProducto,
      })
      .from(carritoLocal)
      .where(inArray(carritoLocal.id, carritoIds));

    if (productosEnLocal.length > 1) {
      //Insertamos los productos 1 por 1
      for (const producto of productosEnLocal) {
        const productoParseado = JSON.parse(producto.producto);
        await db.insert(carrito).values({
          cliente: correo,
          tipoProducto: producto.tipo,
          modelo: producto.modelo,
          detallesProducto: producto.producto,
          precioTotal: productoParseado.precio * productoParseado.cantidad,
        });

        await deleteProductoCarritoLocal2({ id: producto.id });
      }
    } else {
      //Si solo hay un producto que guardar
      const productoParseado = JSON.parse(productosEnLocal[0].producto);
      await db.insert(carrito).values({
        cliente: correo,
        tipoProducto: productosEnLocal[0].tipo,
        modelo: productosEnLocal[0].modelo,
        detallesProducto: productosEnLocal[0].producto,
        precioTotal: productoParseado.precio * productoParseado.cantidad,
      });
      await deleteProductoCarritoLocal2({id: productosEnLocal[0].id});
    }

    return true;
  } catch (error) {
    console.log(error);
    // Si ocurre algún error, devolvemos false
    return false;
  }
}
```

Una vez todo este proceso finaliza, el usuario que se acaba de loguear tendrá de nuevo su carrito sin notar ninguna diferencia salvo que este sí perdurará en el tiempo y podrá ser consultado con cualquier dispositivo que tenga su cuenta.

Proceso de pago

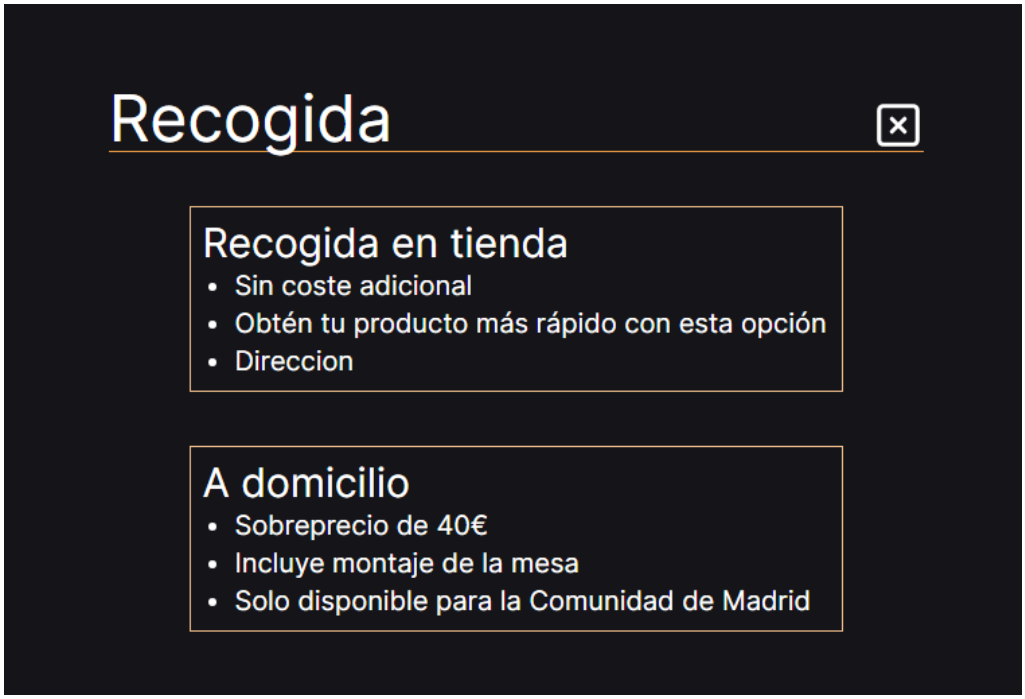
Checkout

Hay dos opciones que un usuario puede seguir para realizar la compra de un producto, la primera sería comprar un producto individualmente, y la segunda consta de almacenar varios en el carrito, de ambas formas llegaríamos a un botón *Comprar,* que está hecho de la siguiente forma

```
return (
  <div className="w-full flex justify-end">
    <button
      onClick={handleOpenModal}
      className=" w-full p-2 bg-fondoSecundari
    >
      Proceder al pago
    </button>
    <ModalCheckoutEnvio
      productos={productos}
      displayModal={displayModal}
      onClose={handleCloseModal}
      procedencia="Carrito"
    />
  </div>
);
}
```

```
return (
  <div className="w-1/2 border border-
    <button
      onClick={handleOpenModal}
      className="bg-colorBase p-2 lg:w-
    >
      Comprar
    </button>
    <ModalCheckoutEnvio
      productos={producto}
      displayModal={displayModal}
      onClose={handleCloseModal}
      procedencia="producto"
    />
  </div>
);
}
```

Estos botones pasan por parámetro los datos al componente **“ModalCheckoutEnvio”**, el cual mostrará una ventana modal, de la siguiente forma



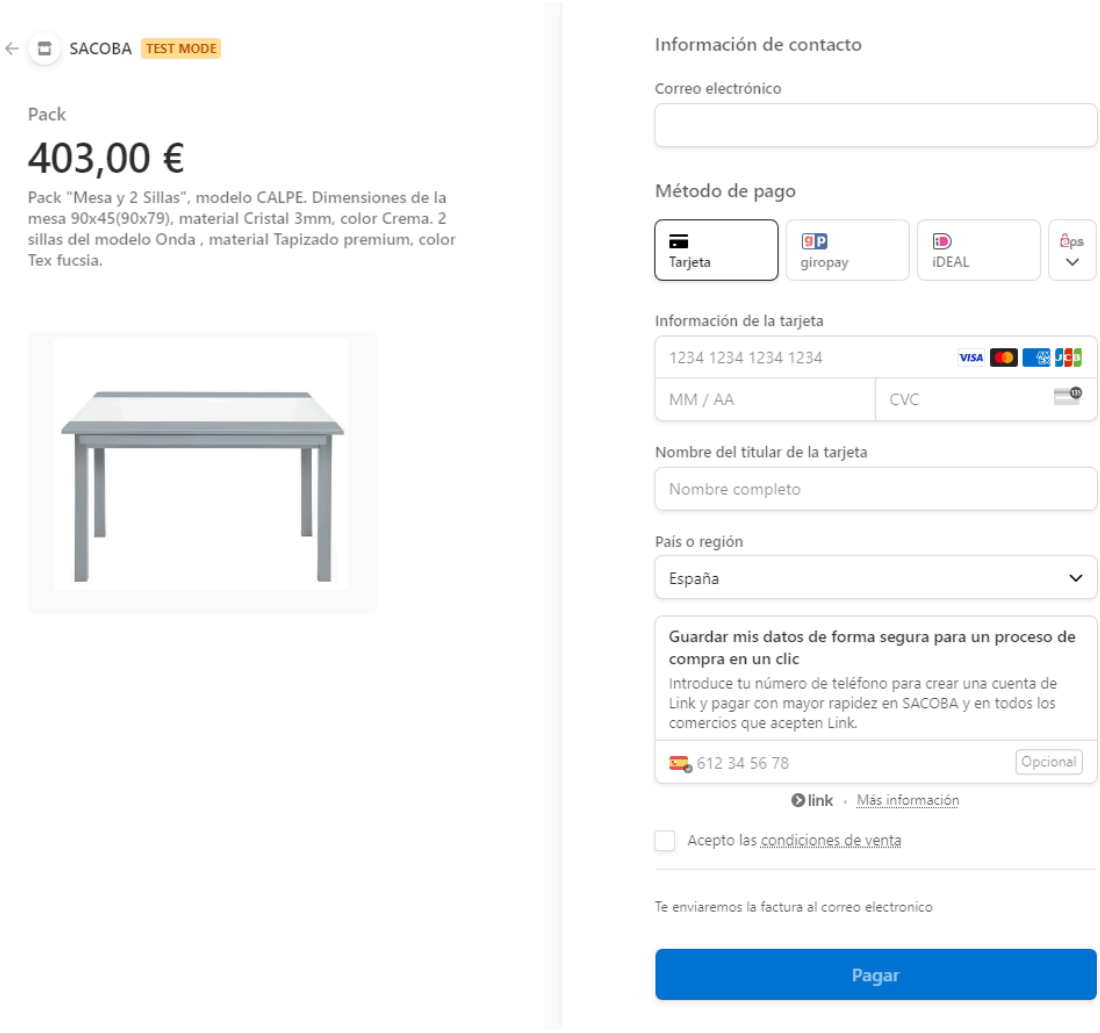
Una vez el cliente ya ha elegido la opción que más le conviene, esta ventana modal detecta la elección deseada y hace una petición *https* a través de un fetching de datos

```
export async function checkoutProductoRecogidaTienda({
  productos,
  onLoad,
  procedencia,
}: {
  productos: any;
  onLoad: () => void;
  procedencia: string;
}) {
  const datos = {
    productos: productos,
    procedencia: procedencia,
  };
  console.log(procedencia)
  try {
    const response = await fetch("../../api/fetching/checkout/", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(datos),
    });
    onLoad();
    if (response.ok) {
      const session = await response.json();
      window.location.href = session.url;
    } else {
      console.log(response);
    }
  } catch (error) {
    console.log(error);
  }
}
```

En el body de la petición, pasa los productos y la procedencia, que como hemos visto puede ser desde un producto o desde el carrito.
Cuando la petición devuelve una respuesta exitosa redirige al usuario a una url propia de **Stripe**, con la cual se realizará el *checkout*.

```
try {
  //Creacion de la sesion de pago
  const session = await stripe.checkout.sessions.create({
    customer_email: correoElectonico,
    line_items: productosDivididos,
    mode: "payment",
    success_url: `https://www.sacoba.es/Success`,
    cancel_url: `https://www.sacoba.es/`,
    shipping_address_collection: {
      allowed_countries: ["ES"],
    },
    custom_text: {
      submit: {
        message: "Te enviaremos la factura al correo electronico",
      },
      terms_of_service_acceptance: {
        message: `Acepto las [condiciones de venta](${baseUrl}/CondicionesVenta)`,
      },
    },
    consent_collection: {
      terms_of_service: "required",
    },
    metadata: {
      tipo: tipoCliente,
      ids: idsProductos,
      tipoEnvio: "Domicilio",
      tipoCompra: body.procedencia,
    },
  });
```

Esta tiene diferentes peculiaridades, la primera y más importante es que si ya contamos con el correo electrónico del usuario porque está logueado este se pondrá como información por defecto del checkout, si no es así lo tendrá que proporcionar el cliente en el momento de la compra, así como la dirección de entrega del producto en el caso de que no haya elegido la opción de recogida en la tienda, la cual no tiene un coste extra.



Una vez el usuario completa todos los datos y paga exitosamente, se le redirige a la siguiente página para indicarle que ha sido procesado correctamente. Mientras que en el backend se realizará un *webhook* para hacer las acciones necesarias, pero todo esto ocurre en paralelo y el usuario solo tiene que saber que la información del pedido se le mostrará en el perfil.

Webhooks

Como ya he mencionado un webhook es un proceso que corre en paralelo a las acciones del usuario, y tiene como fin en este caso, detectar cuando el pago ha sido exitoso, pero se podría hacer de cualquier otro proceso, como si el proceso de pago ha sido comenzado y no terminado, para investigar las causas o simplemente tener información de hasta qué punto llega cada cliente, para poder mejorar los procesos.

Este webhook enfocado al pago exitoso, recibe una solicitud *POST* de **Stripe**, solamente cuando se ha procesado exitosamente, y las acciones que realiza son las siguientes;

```
try {
  event = stripe.webhooks.constructEvent(payload, sigHeader, endpointSecret);
} catch (err: any) {
  console.error(`Webhook Error: ${err.message}`);
  return NextResponse.json(
    { error: "Invalid payload or signature" },
    { status: 400 }
  );
}

switch (event.type) {
  case "checkout.session.completed":
    const session = event.data.object;
    // Handle the checkout.session.completed event
    await registrarPedido({
      datos: {
        cliente: session.customer_details.email,
        fecha: getDate(session),
        idProductos: session.metadata.ids,
        tipoCliente: session.metadata.tipo,
        tipoEnvio: session.metadata.tipoEnvio,
        tipoCompra: session.metadata.tipoCompra,
        precioTotal: session.amount_total,
        direccion: session.customer_details.address,
      },
    });

    console.log("Checkout session completed:", session);
    break;
  // Handle other event types if needed
  default:
    console.log(`Unhandled event type: ${event.type}`);
    break;
}

return NextResponse.json({ received: true }, { status: 200 });
}
```

La primera, construir un evento que por parámetro necesita ciertos códigos de seguridad. Una vez llega al switch tenemos el caso deseado, recibe los datos de la sesión y los proporciona a la función **“resgitrarPedido”**, extrayendo cada datos de donde sea conveniente, la mayoría de ellos proceden del *metadata*, los cuales han sido puestos ahí para proporcionar datos extra, que nos son necesarios para el *webhook* pero no lo son para el *checkout*.

La función **“resgitrarPedido”**, se encarga de recoger todas las casuísticas posibles durante el proceso de compra para ejecutar las consultas que sean necesarias en cada caso.

```
export async function registrarPedido({ datos }: { datos: PedidoParams }) {
  let direccionProporcionada = "no especificada";
  if (datos.tipoEnvio === "Domicilio") {
    direccionProporcionada = JSON.stringify(datos.direccion);
  }
  try {
    let productos;
    const ids = JSON.parse(datos.idProductos);

    //Comprobar si los productos estan guardados en el carrito o no
    if (datos.tipoCompra === "Carrito") {
      const consulta = await selectCarritoParaPedido(ids, datos.tipoCliente);
      productos = consulta.carrito;
      console.log("CONSULTA",productos)
    } else {
      productos = JSON.parse(datos.idProductos);
    }

    await db.insert(pedidos).values({
      cliente: datos.cliente,
      fecha: datos.fecha,
      productos: JSON.stringify(productos),
      importe: datos.precioTotal / 100,
      tipoEnvio: datos.tipoEnvio,
      direccion: direccionProporcionada,
    });

    //Cuando ya tenemos los datos en la tabla pedidos, borramos los productos del carrito
    if (datos.tipoCompra === "Carrito") {
      await deleteCarritoComprado(ids, datos.tipoCliente);
    }

    return true;
  } catch (error) {
    // Si ocurre algún error, devolvemos false
    console.log("Error en la insercion de un pedido", error);
    return false;
  }
}
```

Tiene en cuenta si el pedido ha sido a domicilio porque si es así, insertamos la dirección en la tabla, a su vez discrimina la procedencia del tipo de compra, puesto que si ha sido desde el carrito, los productos estaban almacenados en la tabla, por lo tanto consulta los IDs que han sido comprados y los guarda en la variable *productos*, si proviene directamente de un producto se guarda directamente este en la variable, al no tener que consultar ninguna tabla de la base de datos.

Tras estos procesos ya tiene toda la información suficiente para insertar los datos en la tabla *pedidos*, una vez lo ejecuta, volvemos a comprar la procedencia del pedido y si es del carrito borramos los IDs que previamente hemos consultado para que no aparezcan más en el carrito del cliente.