

PEAK MATE



Que es Peak Mate	1
Características del proyecto	2
Internacionalización	2
Autenticación del usuario	3
Página SignUp (registro)	3
Página Login	5
Verificación de usuarios	5
Reseteo de contraseña	7
Base de datos	7
Tratamiento de la información	7
Agregar, eliminar o modificar la información	7
Consultar información y cachearla	8
Mostrar la información	9
Skeletons	10
Promises y Suspenses	10
Almacenamiento de ficheros	11
Conexión al Bucket R2 Cloudflare	11
Subida de ficheros	12
Implementación de AWS	14
Formulario y validaciones	15
Aregar un Workout	15
Información consistente	15
Validar la información	16
Insertar el workout en la base de datos	16
Mostrar el workout creado	17
Página workouts	18
Implementar sistema de filtrado	18
Sistema de paginación	19
Implementación de pagos con Stripe	20
Sesiones	20
Webhook	22
Suscripciones	24
Modificación del precio de la suscripción	24
Gestión de las suscripciones	25
Sistema de notificaciones	25
Implementación del SEO	26

Que es Peak Mate

Peak Mate es una aplicación atractiva y completa, pensada para quienes desean entrenar, entrenadores personales y propietarios de gimnasios. Ofrece una plataforma dinámica que permite a cada usuario crear y acceder al contenido que más se ajuste a sus necesidades.

Para los entrenadores personales, **Peak Mate** representa una oportunidad única para mostrar sus metodologías y entrenamientos a los usuarios que buscan planes de ejercicio, pudiendo optar por ofrecerlos de manera gratuita o a través de un sistema de pago.

Los **miembros** de la aplicación son aquellos que buscan información sobre entrenamientos, gimnasios, entrenadores personales o rutinas específicas. Tienen acceso a todo el contenido creado tanto por entrenadores como por propietarios de gimnasios, lo que les permite encontrar opciones altamente personalizadas y adaptadas a sus objetivos.

Los **propietarios de gimnasios** pueden usar la plataforma para dar visibilidad a su establecimiento, permitiendo que los usuarios encuentren el lugar perfecto donde realizar sus entrenamientos, sin complicaciones.

Gracias a estas características, en **Peak Mate** todos los usuarios salen beneficiados. Los miembros tienen acceso a una amplia variedad de contenido actualizado y de calidad, con entrenamientos que serán valorados según el feedback de los usuarios. Los entrenadores, por su parte, se ven motivados a competir por atraer miembros, mejorando continuamente su oferta de entrenamientos, los cuales pueden ser gratuitos o de pago mediante suscripciones. Además, los propietarios de gimnasios cuentan con la herramienta ideal para promocionar su establecimiento, añadiéndolo a la plataforma mediante una suscripción mensual, lo que les permitirá llegar a nuevos clientes potenciales interesados en entrenar en su espacio.

Características del proyecto

Internacionalización

Para tener la página en varios idiomas (por ahora español o inglés, (una vez implementado el sistema añadir o quitar idiomas es muy fácil, como veremos ahora)), en este caso, sin usar librerías externas, vamos a tener la siguiente estructura de carpetas, dentro de la carpeta “App”, tendremos que crear la carpeta [lang] (una carpeta que contendrá todas las páginas del proyecto) y que va entre corchetes porque el valor del idioma no lo conocemos (“es” en el caso del español), ya que depende del idioma en el que se quiera servir la página. Luego por otro lado la carpeta “i18n” (puede tener el nombre que sea, pero se le pone este, ya que es el nombre de lo que estamos implementando), en esta carpeta estarán todos los JSON con los distintos idiomas que queramos implementar.



```
1 {
2   "layout": {
3     "header": {
4       "nav": {
5         "About": "About us",
6         "QA": "FAQs",
7         "Contact": "Contact",
8         "WorkWithUs": "Work with us",
9         "Coach": "Be a coach",
10        "Gym": "Show your gym",
11        "MyAccount": "My account",
12        "Workouts": "Find my workout",
13        "Coaches": "Find my coach"
14      }
15    },
16    "footer": {
17      "descriptionEN": "Where everyone discovers exactly what they seek.",
18      "derechos": "All rights reserved.",
19      "title1": "Help",
20      "linkCol1_1": "Feedback",
21      "linkCol1_2": "Contact",
22      "title2": "Legal",
23      "linkCol2_1": "Terms and conditions",
24      "linkCol2_2": "Privacy policy",
25      "title3": "Products",
26      "linkCol3_1": "Workouts",
27      "linkCol3_2": "Find a Coach",
28      "linkCol3_3": "Find a Gym"
29    }
30 }
```

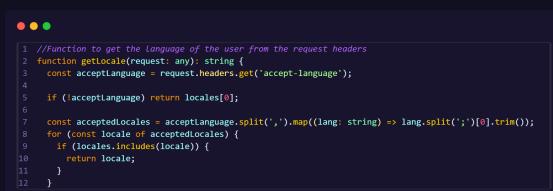
La información que está en verde es la que cambiará en función del idioma, y simplemente cambiando este archivo se aplicará a toda la web.

Por otro lado en esta carpeta estará el archivo “*dictionary.ts*”



```
1 const dictionaries = {
2   en: () => import('../en.json').then((module) => module.default),
3   es: () => import('../es.json').then((module) => module.default),
4 }
5
6 export const getDictionary = async (locale: TypeLanguages) =>
7   dictionaries[locale]()
```

Esto crea un método, con el cual se importa un idioma u otro en función de lo recibido por los parámetros del método. Este método habrá que llamarlo en el “*middleware.ts*”, ya que esto se ejecuta antes de cargar la página, y esta información la necesitamos para cargarla.

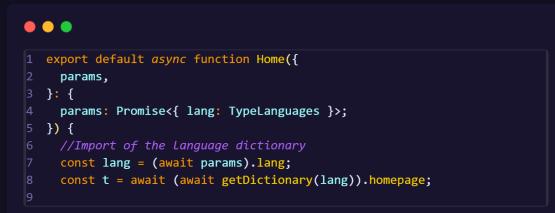


```
1 //Function to get the language of the user from the request headers
2 function getLocale(request: any): string {
3   const acceptLanguage = request.headers.get('accept-language');
4
5   if (!acceptLanguage) return locales[0];
6
7   const acceptedLocales = acceptLanguage.split(',').map((lang: string) => lang.split(';')[0].trim());
8
9   for (const locale of acceptedLocales) {
10     if (locales.includes(locale)) {
11       return locale;
12     }
13   }
14 }
```

Este método, entre otros que ejecuta, coge la petición del usuario y busca entre los lenguajes que prefiere, si no lo tenemos disponible, se pondrá el inglés por defecto.

Y por último agrega a cualquier ruta de la página el idioma (de forma acortada) al inicio de la URL, de manera que todo quedará así: “/en/Workouts”, “/en/Profile”.

Ahora que ya tenemos la URL definida, al montar las páginas extraeremos de la URL la información del idioma.

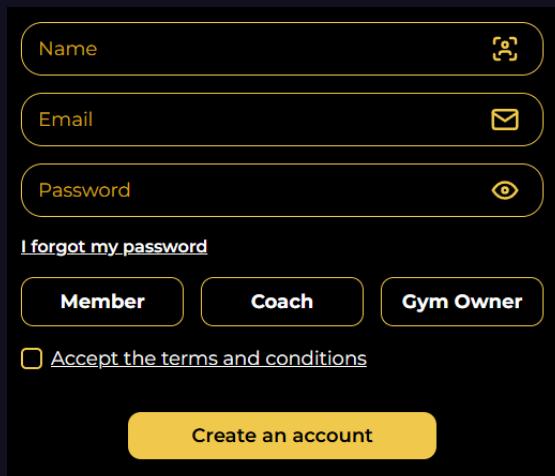


```
1 export default async function Home({  
2   params,  
3 }: {  
4   params: Promise<{ lang: TypeLanguages }>;  
5 }) {  
6   //Import of the Language dictionary  
7   const lang = (await params).lang;  
8   const t = await (await getDictionary(lang)).homepage;  
9 }
```

Esto es una promesa, la cual esperamos y nos da “lang”, que equivaldrá al idioma del usuario “en” o “es” en este caso, y por último con esta información llamará a la función “getDictionary” (vista anteriormente), para cargar el idioma, el .homepage del final es para escoger la parte del JSON (que es un diccionario) que necesitamos para esta página, en vez de cargar todo el JSON al completo.

Autenticación del usuario

La aplicación, para la gestión y organización de todos sus usuarios, permite registrarse rellenando el siguiente form:



Name 

Email 

Password 

[I forgot my password](#)

Accept the terms and conditions

O iniciando sesión con google para mayor comodidad, esto se gestiona en la siguiente página, que mostrará el proceso de manera técnica.

Página *SignUp* (registro)

Una vez el usuario rellena el formulario que hemos visto previamente, se ejecuta la siguiente función asíncrona.

```

1 //Send the data to the server
2 const clientAction = async (formData: FormData) => {
3   // Disabled the button
4   setLoading(true);
5   try {
6     const response = await extractData(formData, text.validations);
7     console.log(response);
8     // If all goes well, the user is redirected to profile.
9     if (response.success) {
10       //Once registered, we authenticate it
11       const email = formData.get("email");
12       const password = formData.get("password");
13       const result = await signIn("credentials", {
14         email,
15         password,
16         redirect: false,
17       });
18       if(!result.error){
19         router.push("/Profile");
20       }else{
21         throw Error("Error in the authentication");
22       }
23     } else {
24       toast.error(response.error);
25     }
26   } catch (error) {
27     console.log("error" + error);
28   } finally {
29     setLoading(false);
30   }
31 };

```

Que se encargará de recoger toda la información del form, y gracias a la función “`extractData`” que validará los datos recibidos y por último hará la inserción en el servidor, su respuesta, en función de si es exitosa o no, permitirá comprobar si el registro ha sido correcto, en tal caso, se llamará a la método “`signIn`” de NextAuth que ejecuta la siguiente: Tendremos que crear un endpoint (route.ts) siguiendo esta ruta específica de carpetas, “api→auth→[...nextauth]→route.ts”, y en ese archivo tendremos lo siguiente:

```

1 export const authOptions = {
2   providers: [
3     GoogleProvider({
4       clientId: process.env.GOOGLE_CLIENT_ID,
5       clientSecret: process.env.GOOGLE_CLIENT_SECRET,
6     }),
7     CredentialsProvider({
8       name: "Credentials",
9       credentials: {
10         email: { label: "Email", type: "email" },
11         password: { label: "Password", type: "password" },
12       },
13     }), //Checks in the database if the user exists
14     async authorize(credentials) {
15       try {
16         await connectToDatabase();
17
18         const { email, password } = credentials;
19         const user = await Member.findOne({ email });
20
21         // Check if the user exists
22         if (user) {
23           // Check if the authMethod is "Credentials"
24           if (user.authMethod != "Credentials") {
25             throw new Error("Auth");
26           }
27           // Check if the password is correct
28           if (bcrypt.compareSync(password, user.password)) {
29             return user; // Authentication succeeded
30           } else {
31             throw new Error("Invalid credentials"); // Authentication failed
32           }
33         }
34       } catch (error) {
35         throw new Error("Something went wrong during authentication");
36       }
37     },
38   }],
39 };

```

En este caso estaríamos llamando a “`CredentialsProviders`”, que crea una nueva sesión para el usuario registrado, con esto tendremos disponible en todo momento y sin necesidad de consultar en la base de datos, su nombre, tipo de miembro que es y el email. Por otro lado si se registra con Google, el proceso es mucho más automático, ya que gran parte de la validación está gestionada por el propio Google, se ejecutará “`GoogleProvider`” y gracias a las claves secretas se realizará la conexión y el proceso.

Por último, al finalizar este proceso, se ejecuta un callback, que es lo que realmente crea la sesión del usuario.

Página Login

De la misma forma que la página de registro, tendremos un form para llenar la información de logueo (email y contraseña), o usando el inicio de sesión de Google, pero esta vez la comprobación se hace directamente con los métodos anteriores de NextAuth, ya que cuando es un registro siempre van a ser exitosos, puesto que la información del usuario y la contraseña se están introduciendo en ese momento y no pueden ser erróneas (el único caso es que se introduzca un correo ya existente, pero eso se gestiona en otra parte).

Verificación de usuarios

Cuando un usuario se registra por primera vez con credenciales manuales (con el formulario de la página) haremos lo siguiente, ya que en el caso de iniciar sesión con Google la cuenta ya estaría verificada automáticamente, por lo tanto cuando el proceso que acabamos de ver de manera exitosa, mandamos un mensaje al correo electrónico de la cuenta con esta función POST.

```
● ● ●
1 const resend = new Resend(process.env.RESEND_API_KEY);
2
3 export async function POST(req: Request) {
4   const body = await req.json();
5
6   //Generate the verification code and insert it into the database
7   const code = generateVerificationCode();
8   const response = await insertValCode(body.email, code);
9   if (!response.success) {
10     if (response.message === "AlreadyVerified") {
11       return Response.json({ error: "AlreadyVerified" });
12     }
13     return Response.json({ error: "Error inserting the validation code" });
14   }
15   // Select the email template and subject based on body.Lang
16   const Template = body.lang === "en" ? PeakMateConfirmEmailEN : PeakMateConfirmEmailES;
17   const subject = body.lang === "en" ? "Confirm your email address" : "Confirma tu dirección de correo";
18
19   const { data, error } = await resend.emails.send({
20     from: 'Peak Mate <no-reply@peakmate.fit>',
21     to: [body.email],
22     subject,
23     react: Template({ validationCode: code }),
24     text: "",
25   });
26
27   if (error) {
28     return Response.json({ error });
29   }
30
31   return Response.json(data);
32 }
```

Esta función genera un código alfanumérico de 6 dígitos, que tendremos que guardar en la tabla de la base de datos para luego consultarla, y con la ayuda de [Resend](#), mandaremos al correo electrónico del usuario toda la información tal que así;

Peak Mate

Confirm your email address

Your confirmation code is provided below – enter it in the open browser window and we will help you sign in.

USU-ISG

If you did not request this email, there is nothing to worry about; you can simply ignore it.

[Confirm email address](#)

[Go to Peak Mate](#)

© Peak Mate - All rights reserved.

El enlace nos llevará a una página donde tendremos que insertar el código, y en el caso de ser válido el estado de la cuenta cambiará a “verified true”.



Insert the code below to verify your email

We send you an email with the code

[Resend code](#)

_____ - _____

[Verify](#)

[Go to profile](#)

Reseteo de contraseña

De la misma forma que la verificación de usuarios, al iniciar sesión los usuarios que hayan podido olvidar su contraseña tendrán la oportunidad de volver a configurar una nueva mediante un correo de recuperación que se enviará a la cuenta de correo con la que inician sesión en Peak Mate, por este hecho, esto solo se podrá hacer si el método de autenticación del usuario es “*Credentials*” (si se ha creado una cuenta en Peak Mate), ya que si ha usado google para iniciar sesión todo esto no aplicará.

De la misma forma que para el “*verification code*” se guardará un código temporal en la base de datos, para posteriormente poder comprobarlo, con la diferencia de que en este caso guardaremos también un tiempo de expiración al código de 15 minutos, después de esos 15 minutos el código dejará de ser válido y se tendrá que generar uno nuevo, esto es debido a que al ser un reseteo de contraseña hay que hacerlo de forma más segura y controlada.

Base de datos

El proyecto está conectado a [MongoDB Atlas](#), usando [Mongoose](#) como ORM para crear los *schemas*, *selects*, *inserts* y *updates*.

Ejemplo de la creación de la tabla en el schema:

```
● ● ●
1 //Member & Coach Schema
2 const memberSchema = new Schema(
3   {
4     email: { type: String, required: true, unique: true },
5     type_of_member: {
6       type: String,
7       required: true,
8       enum: ["Member", "Coach", "GymOwner"],
9     },
10    password: { type: String, required: true, default: "google" },
11    name: { type: String, required: true },
12    nickname: { type: String, default: "" },
13    photo: { type: String, default: "" },
14    location: { type: String, default: "" },
15    contact_number: { type: Number, default: "" },
16    RSSS: { type: [String] }, // Instagram, Facebook, Twitter, etc
17    public: { type: Boolean, required: true, default: false },
18    //Coach has to be public
19    savedworkouts: [
20      {
21        type: Schema.Types.ObjectId,
22        ref: "Workout", //FK Workout
23      },
24    ], //Workouts saved by the member
25    Coach: [
26      {
27        contact_email: { type: String, default: "" },
28        languages: { type: String, default: "" },
29        // Languages in which the coach works
30        description: { type: String, default: "" },
31      },
32    ],
33    GymOwner: [
34      {
35        plan_type: { type: String, required: true, default: "none" },
36        contact_email: { type: String, default: "" },
37        // ONLY FOR "gymOwner" Basic, Premium, etc
38      },
39      authMethod: { type: String, required: true, enum: ["Google", "Credentials"] },
40      { timestamps: true }
41    );
42  };
43
```

Tratamiento de la información

Agregar, eliminar o modificar la información

Para realizar estos procesos se coge la información que precise, ya sea con un formulario u otro método y se pasa al [server component](#), el cual se utiliza para ejecutar mutaciones en el servidor aunque este dentro de un “[client component](#)”.

```

1 //Send the data to the server
2 const clientAction = async (formData: FormData) => {
3   // Disabled the button
4   setLoading(true);
5   try {
6     const response = await extractData(formData, text.validations);
7     // If all goes well, the user is redirected to profile.
8     if (response.success) {
9       //Once registered, we authenticate it
10      const email = formData.get("email");
11      const password = formData.get("password");
12      const result = await signin("credentials", {
13        email,
14        password,
15        redirect: false,
16      });
17      if(!result?.error){
18        router.push("/Profile");
19      }else{
20        throw Error("Error in the authentication");
21      }
22    } else {
23      toast.error(response.error);
24    }
25  } catch (error) {
26    console.log("error" + error);
27  } finally {
28    setLoading(false);
29  }
30 };

```

Este componente llama asíncronamente a la función `extractData` la cual realizará las acciones de validar el formulario (para evitar inyecciones de código), hashea la contraseña (en este caso específico, ya que es el registro de un usuario), y por último llama a la función `insertNewUser` para agregarlo a la base de datos

```

1 export const extractData = async (formData: FormData, text:any) => {
2   const newForm = {
3     email: formData.get("email"),
4     name: formData.get("name"),
5     password: formData.get("password"),
6     user_type: formData.get("userType"),
7   };
8   const result = FormSignupValidation({text}).safeParse(newForm);
9   if (!result.success) {
10     let errorMessage = "";
11     result.error.issues.forEach((issue) => {
12       errorMessage = errorMessage + issue.path[0] + ":" + issue.message + ". ";
13     });
14     return {
15       error: errorMessage,
16       success: false,
17     };
18   }
19   try {
20     const hashedPassword = await bcrypt.hash(result.data.password, 10);
21     const member = {
22       email: result.data.email,
23       type_of_member: result.data.user_type,
24       name: result.data.name,
25       password: hashedPassword,
26       authMethod: "Credentials",
27     };
28     const query = await insertNewUser(member);
29
30     if (query.success) {
31       return {
32         success: true,
33       };
34     } else if(query.message === "DPEmail") {
35       return {
36         error: text.errorEmail,
37         success: false,
38       };
39     }else{
40       return {
41         error: text.error,
42         success: false,
43       };
44     }
45   } catch (error) {
46     return {
47       error: "Hubo un error al procesar la solicitud.",
48       success: false,
49     };
50   }
51 };

```

```

1 //Used in the Signup form
2 export async function insertNewUser(data: MemberType) {
3   await connectToDatabase();
4
5   try {
6     const newMember = new Member({
7       email: data.email,
8       type_of_member: data.type_of_member,
9       name: data.name,
10      password: data.password,
11      authMethod: data.authMethod,
12    });
13    await newMember.save();
14
15    return {success: true};
16  } catch (error) {
17    if (error instanceof MongoServerError && error.code === 11000) {
18      return { success: false, message: "DPEmail" };
19    }
20    return { success: false, message: "Invalid input" };
21  }
22 }

```

Este insert agrega a la tabla `Member` de la base de datos el nuevo registro, a su vez verifica si ese usuario ya está registrado, si es así se lanzará el `error.code=11000`.

Consultar información y cachearla

En este caso aunque se podría realizar un select de la misma forma que la mencionada, no es recomendable porque no se cachearía, lo cual es necesario ya que una vez realizada la

primera consulta no queremos que se vuelve a ejecutar porque aparte de volver a esperar a que el servidor responda con la información, estaríamos haciendo consultas extra innecesarias, por ello es importante este punto, para evitar hacer más consultas de las necesarias, que con un gran volumen de usuarios esto puede suponer mucho dinero en coste de servidor.

Para realizar esta labor es necesario usar *fetch*, lo cual, mediante el usuario de peticiones http, en este caso específico con el método *GET* puesto que es el método que cachea las respuestas, ya que está hecho para consultar información y no enviarla al servidor.

Nota: Todo lo realizado antes con los server components, se puede realizar con peticiones fetch y el método POST.

```
● ● ●
1 // Endpoint to get the user data
2 const userDataPromise = fetch(
3   `${process.env.NEXT_PUBLIC_BASE_URL}/api/selectInfoUser?user=${session.user.id}`,
4   { cache: "force-cache", next: { tags: ["user-data"], revalidate: 3600 } }
5 ).then((response) => {
6   if (response.ok) {
7     return response.json();
8   } else {
9     console.log(response);
10    return null;
11  }
12});
```

Es importante que el endpoint esté dentro de la carpeta API, y esta a su vez en la carpeta APP o SRC. Dentro de la carpeta API crearemos otra con el nombre del endpoint, en el caso de la imagen es “*selectInfoUser*” y dentro de esta el archivo *route.ts*. Si necesitamos pasarle algún tipo de información a la solicitud GET ya sea el id del usuario a consultar u otra información deberemos de pasarsela por la URL, ya que este método no tiene *body*, por lo tanto en el archivo *route.ts* tendremos que recuperar esa información.

```
● ● ●
1 export async function GET(req: NextRequest) {
2   // Get the user id from the request URL
3   const { searchParams } = new URL(req.url);
4   const userId = searchParams.get("user") || "";
5
6   // Select the user data from the database
7   const user = await selectUser(userId);
8
9   // Return the user data
10  return NextResponse.json(user);
11}
```

Una vez ejecutada la lógica pertinente en esta función, que en este caso es una consulta para seleccionar un usuario del cual tenemos su id (clave primaria), envía la información como respuesta a la solicitud.

*Nota: Desde Nextjs v.15 el método GET no cachea por defecto, por ello deberemos añadir esta línea para añadirlo, junto al nombre de esa caché, ya que si la queremos actualizar por algún motivo, tendremos que usar esta función *revalidateTag("nombre-caché")*:*

```
● ● ●
1 { cache: "force-cache", next: { tags: ["user-data"], revalidate: 3600 } }
```

Mostrar la información

Para mayor velocidad de la web, todas las páginas que tienen que esperar información de la base de datos, no la esperan directamente, hacen un “*promise*” de la petición de información, esto provoca que cargue de manera instantánea, en vez de esperar a la respuesta, y como no vamos a tener la información en ese momento, se muestra un

“skeleton” que como veremos en el siguiente punto es un componente igual que el que se debería de mostrar pero sin información.

Skeletons

Para simplificar el proceso, se crea un componente *Skeleton* con diferentes estilos

```
1 // To add any class to the component
2 function classNames(
3   ...classes: (string | undefined | null | boolean)[]
4 ): string {
5   return classes.filter(Boolean).join(" ")
6 }
7
8 function Skeleton({
9   className,
10  ...props
11 }: React.HTMLAttributes<HTMLDivElement>) {
12  return (
13    <div
14      className={classNames(
15        "animate-pulse rounded-md background-secondary p-0",
16        className
17      )}
18      {...props}
19    />
20  );
21 }
22
23 export { Skeleton };
```

Para mostrarle al usuario que en ese espacio se está cargando algo, luego para mostrar un resultado lo más parecido al que se va a mostrar, se crea un componente como por ejemplo “workoutCardSkeleton” el que va a reemplazar momentáneamente mientras se carga a “workoutCard”, por lo tanto solo tendremos que sustituir donde haya información por este componente “skeleton” y darle unas medidas aproximadas al resultado final.

```
1 import { Skeleton } from "../assets/skeleton";
2
3 export default function WorkoutCardSkeleton() {
4   return (
5     <section className="border-gray-600 border rounded-lg w-[414px] h-64">
6       <div className="bg-opacity-50 rounded-l-lg h-full flex flex-col justify-between">
7         </div>
8         <div className="flex justify-between w-full p-4">
9           <div>
10             <Skeleton className="w-40 h-6" />
11           </div>
12           <div>
13             <Skeleton className="w-16 self-center h-6" />
14           </div>
15           </div>
16         </div>
17         <div>
18           <Skeleton className="self-center h-6" />
19         </div>
20       </div>
21     </section>
22   );
23 }
```

Promises y Suspenses

Como he mencionado antes, no vamos a realizar el “await” a la consulta, simplemente la llamaremos y quedará guardada en una *promise*.

```
1 // Endpoint to get the user data
2 const workoutDataPromise = fetchSelectedWorkoutData(selected);
```

Luego, mediante al componente propio de Nextjs, *Suspense*, se realiza lo siguiente:

```
1 <Suspense fallback=<WorkoutInfoSkeleton />>
2   <WorkoutInfo workoutDataPromise={workoutDataPromise} text={t} />
3 </Suspense>
```

Esto pasará al componente “*WorkoutInfo*” la promesa por parámetros (entre otros), y mientras la resuelve, realiza un callback en el que se muestra el componente *Skeleton*.

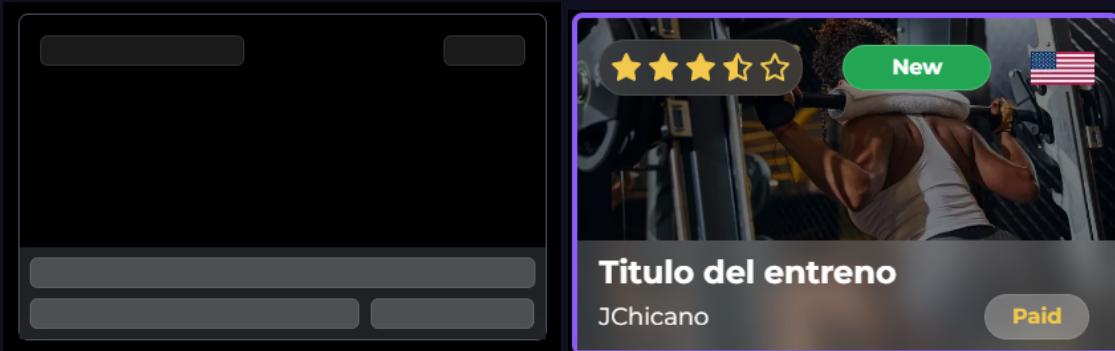
```

1 export default async function WorkoutInfo({
2   workoutDataPromise,
3   text,
4 }: {
5   workoutDataPromise: Promise<OneWorkoutDataTypeSelect>;
6   text: any;
7 }) {
8   // Extract the data from the promise
9   const data = await workoutDataPromise;

```

Esto lo recibe el componente y se encarga de esperar para trabajar con la información.

Este es el resultado final:



En el que el *Skeleton*, muestra un resultado similar al final (sin tener todo en cuenta)

Almacenamiento de ficheros

Continuando con el tema de Base de datos, para almacenar ficheros tendremos que usar otro sistema, ya que en MongoDB y no se pueden almacenar ficheros como tal sino que almacenaremos la URL al fichero (para saber a quién pertenece y luego recuperarlo), esta URL guardará la ruta del fichero.

Conexión al Bucket R2 Cloudflare

Tras investigar la mejor opción, el servicio de R2 de Cloudflare es el que más se ajusta a lo que se necesita en la página web, ya que para almacenar videos como tal ofrecen otro servicio llamado *Stream*, pero es mucho más caro, como se puede ver en la siguiente tabla.

Característica	Cloudflare Stream	Cloudflare R2
Uso principal	Almacenamiento y streaming de video optimizado	Almacenamiento de objetos tipo S3 (archivos en general)
Casos de uso	Streaming de videos en web/apps sin necesidad de servidores adicionales	Almacenamiento de imágenes, backups, logs, archivos estáticos, etc.
Optimización de videos	Sí, convierte videos automáticamente a múltiples calidades para streaming adaptativo	No, los archivos se almacenan tal cual sin procesar
Compatibilidad con reproductores	Ofrece un reproductor nativo y compatibilidad con HLS/DASH	No ofrece reproductor, solo almacenamiento
Entrega a través de CDN	Sí, optimizado para streaming global con baja latencia	Sí, pero no está optimizado específicamente para streaming
Subida de archivos	Soporta subida directa desde usuarios sin servidores intermedios	Usa URLs firmadas o SDKs para manejar subidas seguras
API y SDKs	API para controlar calidad de video, subtítulos, miniaturas, etc.	Compatible con APIs tipo S3 (AWS SDK, Boto3, etc.)
Costo de almacenamiento	Basado en minutos de video almacenados y reproducidos	Basado en GB almacenados, sin costos de salida de datos
Costo de salida de datos	Gratis dentro de Cloudflare, pero tiene costos de reproducción	Gratis (sin egress fees, a diferencia de AWS S3)
Autenticación	Permite restringir acceso con tokens firmados	Gestiona permisos con políticas y CORS
Ideal para	Plataformas de video (cursos, streaming, contenido on-demand)	Aplicaciones que necesitan almacenamiento escalable sin costos de salida

Tendremos dos formas de tratar con los archivos, la primera, para imágenes (archivos que pesan poco) pasándolos al backend a través del body (el cual por defecto permite un peso de 1Mb) y por lo tanto es el servidor el que sube el archivo al bucket, o (para archivos como videos, que tienen un peso elevado), haremos una petición al bucket para que nos devuelva una *signedUrl*, una url temporal firmada con las claves para así poder subir los archivos directamente desde el cliente, al tener él el “acceso”.

```

1 // Validar variable de entorno
2 const R2_BUCKET_NAME = process.env.R2_BUCKET_NAME;
3 if (!R2_BUCKET_NAME) {
4   throw new Error("R2_BUCKET_NAME environment variable is not set.");
5 }
6
7 // Initialize the S3 client
8 const s3 = new S3Client({
9   region: "auto",
10  endpoint: process.env.R2_CDN_URL!,
11  credentials: {
12    accessKeyId: process.env.R2_ACCESS_KEY!,
13    secretAccessKey: process.env.R2_SECRET_KEY!,
14  },
15 });
16
17 // Generate a signed URL for uploading files from the frontend.
18 export async function getUploadSignedUrl(objectName: string,
19   contentType: string) {
20   const command = new PutObjectCommand({
21     Bucket: R2_BUCKET_NAME,
22     Key: objectName,
23     ContentType: contentType,
24   });
25   return getSignedUrl(s3, command, { expiresIn: 300 }); // 5 minutes
26 }
```

Subida de ficheros

Veremos como ejemplo la subida de los ficheros del formulario “*CreateWorkout*” ya que el resto de partes del formulario se explicarán más adelante, y como ya he mencionado para ello en este caso (al ser videos) y usaremos un *signedUrl* ya que lo vamos a tener que subir desde el cliente.

Para permitir al usuario seleccionar el archivo deseado, se crea un input de tipo “file”, en el que en este caso concreto aceptaremos como fichero tanto un “image/” como un “video/” este input por la naturaleza del formulario tendrá un id dinámico en función del día y del ejercicio al que pertenezca, pero para la subida y tratamiento de los ficheros no nos afecta. Una vez que el usuario selecciona el archivo deseado por medio de un “*useEffect*” actualizaremos un estado el cual guarda la ruta de la imagen o vídeo para poder mostrarlo en un componente aparte, ya que los input de tipo “file” no acepta un value el cual podramos definir por motivos de seguridad.

```

1 / Update mediaType and previewUrl when file changes
2 useEffect(() => {
3   if (file) {
4     setMediaType(file.type);
5     const url = URL.createObjectURL(file);
6     setPreviewUrl(url);
7     return () => URL.revokeObjectURL(url);
8   } else {
9     setPreviewUrl(null);
10    setMediaType(null);
11  }
12 }, [file]);
```

Una vez se envía el formulario (el usuario clica el botón “submit”), primero se valida el formulario y luego se procederá a validar cada uno de los archivos que haya seleccionado.

```

1 //For avatars (files of the infoUserProfile Form) IMG
2 const validateFile = (file: File | null, text: any): string | null => {
3     if (!file) {
4         return "No file provided.";
5     }
6
7     const imageTypes = ["image/jpeg", "image/png", "image/heic", "image/webp"];
8     const videoTypes = ["video/mp4", "video/webm", "video/ogg", "video/mov"];
9     const validNamePattern = /^[a-zA-Z0-9_.\s-]+$/;
10
11    if (imageTypes.includes(file.type)) {
12        // Validate image size (less than 2MB)
13        const maxSizeInMB = 2;
14        if (file.size > maxSizeInMB * 1024 * 1024) {
15            return text.sizePhoto + " (2MB Max)";
16        }
17        // Validate image file name
18        if (!validNamePattern.test(file.name)) {
19            return text.namePhoto;
20        }
21    } else if (videoTypes.includes(file.type)) {
22        // Validate video size (less than 200MB)
23        const maxSizeInMB = 200;
24        if (file.size > maxSizeInMB * 1024 * 1024) {
25            return text.sizeVideo + " (200MB Max)";
26        }
27        // Validate video file name
28        if (!validNamePattern.test(file.name)) {
29            return text.nameVideo;
30        }
31    } else {
32        // File type is neither a supported image nor video
33        return text.typePhoto || text.typeVideo;
34    }
35    return null; // No errors, file is valid
36 };
37
38 export default validateFile;

```

Esta función verifica que tipo de archivo es (video/imagen), y comprueba su peso y nombre, ya que tienen que seguir ciertas reglas.

Cuando ya se ha comprobado que la validación es correcta se procede a enviar uno por uno al bucket R2 por medio de la `signedUrl`.

```

1  await Promise.all(
2    fileEntries.map(async ([key, fileValue]) => {
3      const parts = key.split("-");
4      const day = parts[1] || "unknown";
5      const exercise = parts[2] || "unknown";
6      if (!(fileValue instanceof File)) return;
7
8      const uniqueName = `workouts/exercises/${Date.now()}-${day}-${exercise}-${
9        fileValue.name
10      }`;
11
12      // Get signed URL for upload
13      const res = await fetch(
14        `${process.env.NEXT_PUBLIC_BASE_URL}/api/uploadBigFiles`,
15        {
16          method: "POST",
17          headers: { "Content-Type": "application/json" },
18          body: JSON.stringify({
19            objectName: uniqueName,
20            contentType: fileValue.type,
21          }),
22        }
23      );
24      if (!res.ok) {
25        errors.push(`Error getting signed URL for ${fileValue.name}`);
26        return;
27      }
28      const { signedURL } = await res.json();
29
30      // Upload the file using the signed URL
31      const uploadRes = await fetch(signedURL, {
32        method: "PUT",
33        body: fileValue,
34        headers: { "Content-Type": fileValue.type },
35        mode: "cors",
36      });
37      if (!uploadRes.ok) {
38        errors.push(`Error uploading file ${fileValue.name}`);
39        return;
40      }
41
42      // Save the mapping: the key (day-exercise) maps to the uniqueName
43      filesMapping[key] = uniqueName;
44    })
45  );

```

Esta función sube los archivos con una ruta concreta, para este caso en la carpeta “exercises” que está dentro de la carpeta “workouts”, para tenerlo todo mejor organizado en función de para qué y de qué sean los archivos y por último en esta ruta se agrega el dia, ejercicio y hora exacta, al nombre del fichero, para que todos los nombres sean únicos, ya que esto será una condición importante. Antes de terminar la función guarda estas rutas en un array.

Por último, a la hora de insertar los datos, en cada ejercicio se insertará en el campo “file” la ruta del fichero al que pertenece.

Implementación de AWS

En principio el resultado final con cloudflare era el adecuado, pero en españa “La Liga” baneaba todas las ips de cloudflare los fines de semana con la excusa de evitar la piratería, por lo tanto durante ese periodo el contenido multimedia dinámico de la web no funcionaba, y es clave para su uso, por lo tanto tuve que migrar el servicio a AWS.

En AWS se usa un Bucket S3 para almacenar los archivos, cuando se hace una petición put (agregar archivos) se ejecuta de la misma forma que antes, directamente se sube el bucket o desde el servidor de la página si es un archivo ligero o desde una URL firmada en el cliente si pesa mucho. Por otro lado a la hora de servir los archivos, el Bucket S3 los manda a otro servicio de AWS llamado Cloudfront y desde ahí se sirve al cliente, este ultimo servicio actúa como intermediario y permite almacenar en caché los archivos servidos para que si se vuelven a pedir no se consulte el Bucket sino este servicio únicamente.

Formulario y validaciones

Agregar un *Workout*

Para esta labor se implementaron varios formularios recursivos dentro de otros, ya que el *Workout* tiene las siguientes características, su información base y los diferentes días de entrenamiento, a su vez, cada día tendrá varios ejercicios, por ello tenemos que tener en cuenta las siguientes situaciones.

Información consistente

Para mantener la información de forma consistente, y que si cambiamos de día la información permanezca, todos los “inputs” reciben como parámetro en el value, la información previa que pueda haber en el formulario, si no existe se pasará un valor vacío para que el usuario pueda insertar su propia información.

```
<input  
  type="number"  
  className="input-field font-semibold rounded-lg w-24"  
  name="durationDay"  
  value={formData.durationDay || ""}  
  max={999}  
  onChange={handleInputChange}  
  onInput={(e) => {  
    if (e.currentTarget.value.length > 3) {  
      e.currentTarget.value = e.currentTarget.value.slice(0, 3);  
    }  
  }}  
/>
```

Cada vez que el input del campo cambia se ejecuta el método “OnChange” el cual guarda la información en el form.

The screenshot shows a user interface for a workout scheduler. At the top, there's a navigation bar with days of the week: L, M, X, J, V, S, D. The day 'X' (Wednesday) is highlighted. Below the navigation, the title 'Weekly schedule' is displayed above a section for 'Wednesday'. This section includes a text input for 'Title for the day', a dropdown for 'Method', and a row of buttons for different exercise types: Weightlifting, CrossFit, Calisthenics, HIIT, Core, and Powerlifting. A large button labeled '+ Add exercise' is at the bottom of this section. At the very bottom of the page, there's a prominent yellow button labeled 'Finish my workout'.

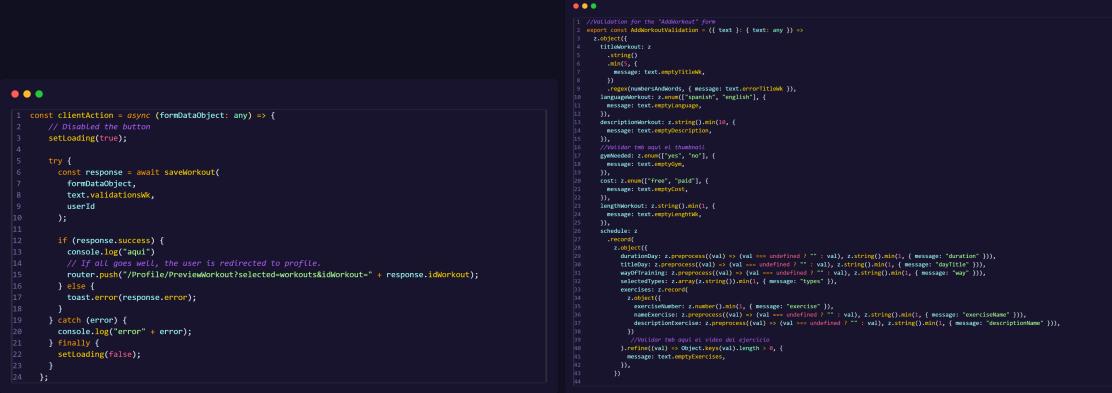
```
1 const handleInputChange = (  
2   e: React.ChangeEvent<  
3     HTMLInputElement | HTMLTextAreaElement | HTMLSelectElement  
4   >  
5 ) => {  
6   const { name, value } = e.target;  
7   setFormData((prev: any) => ({ ...prev, [name]: value, day }));  
8 };
```

Como ejemplo, está imagen de la web representa la parte del formulario de la cual estamos hablando, como se puede apreciar si se cambia de día en la parte superior la información introducida previamente se tiene que guardar automáticamente y también que permita volver al día anterior y poder tanto ver como editar lo que habíamos introducido.

Validar la información

Una vez que el usuario decide que ha terminado pulsará el botón de “Finish My Workout” y se procederá a ejecutar la siguiente función.

El primer paso será pasarle la información a la función “`saveWorkout`”, que hará lo siguiente:



```
1 //Validation for the "Addressworkout" form
2 export const AddressworkoutValidation = ({ text }: { text: any }) =>
3   z.object({
4     titleworkout: z
5       .string()
6       .min(1)
7       .message(text.emptyThumbnail),
8     ... (more validation logic for other fields like duration, exercises, etc.)
9   });
10 
```

Pasar la información a través de esta función, la cual gracias a la biblioteca Zod, validaremos tanto si la información tiene el formato correcto como que el usuario no se haya dejado ningún campo vacío, si se comete alguno de estos hechos se mostrará un error para que el usuario pueda corregirlos, por otro lado si todo es correcto se seguirá ejecutando el método “`saveWorkout`”.

Insertar el workout en la base de datos

Al igual que la dificultad técnica de los formularios por la estructura del workout, a la hora de insertarlo habrá que hacerlo de forma muy concreta.



```
1 // Check if the user type is "Coach"
2 const user = await Member.findById(userId).select("type_of_member");
3 if (user.type_of_member !== "Coach") {
4   return { success: false, message: "NoCoach" };
5 } else {
6   // If the user is a coach, insert the workout
7   const newworkout = new Workout({
8     title: data.titleWorkout,
9     language: data.languageWorkout,
10    coverPhoto: "Not implemented yet", // data.thumbnail
11    difficultyLVL: data.difficulty, // TAKE CARE OF THIS
12    durations_weeks: data.lengthworkout,
13    aim: data.objective,
14    day_week: Object.keys(data.schedule).length, // Number of training days per week
15    // promoted field defined elsewhere
16    free: data.cost === "free", // False or true
17    gym_needed: data.gymNeeded,
18    short_description: data.descriptionWorkout,
19    owner: userId,
20  });
21  const savedWorkout = await newworkout.save();
22
23  // Recorrer Los días del schedule e insertar en WorkoutPerDay
24  for (const [dayKey, dayValue] of Object.entries(data.schedule)) {
25    const newworkoutPerDay = new WorkoutPerDay({
26      workoutID: savedWorkout._id,
27      title: dayValue.titleDay,
28      way_of_train: dayValue.wayOfTraining,
29      type: dayValue.selectedTypes,
30      day_of_Week: parseInt(dayKey),
31      duration_for_day: dayValue.durationDay,
32      //material_needed: dayValue.material_needed,
33    });
34    const savedWorkoutPerDay = await newworkoutPerDay.save();
35
36    // Recorrer los ejercicios de cada día e insertar en Exercise
37    for (const [exerciseKey, exerciseValue] of Object.entries(dayValue.exercises)) {
38      console.log(`Exercise: ${exerciseKey}, ${exerciseValue}`);
39      const newExercise = new Exercise({
40        Workout_per_dayID: savedWorkoutPerDay._id,
41        exerciseNumber: parseInt(exerciseKey),
42        exerciseTitle: exerciseValue.nameExercise,
43        photo: "to be implemented",
44        video: "to be implemented",
45        description: exerciseValue.descriptionExercise,
46      });
47      await newExercise.save();
48    }
49  }
50 }
```

Primero hace un chequeo para comprobar que el usuario que está mandando la información es de tipo “Coach”, si es así, se procede a insertar la información general en la tabla workouts, una vez completado se extrae el id generado y se inserta un nuevo dia en la tabla *WorkoutPerDay*, como es posible que hayan varios días en el workout, esta inserción se tiene que hacer de forma recursiva, en la cual insertaremos como clave foránea el id del workout generado previamente, esto a su vez nos dará como resultado un objeto con los id de los días insertados, por ello, nuevamente recorreremos el array ejercicios (el cual está dentro del objeto dia), y procederemos a insertar individualmente cada ejercicio con la id del dia como referencia.

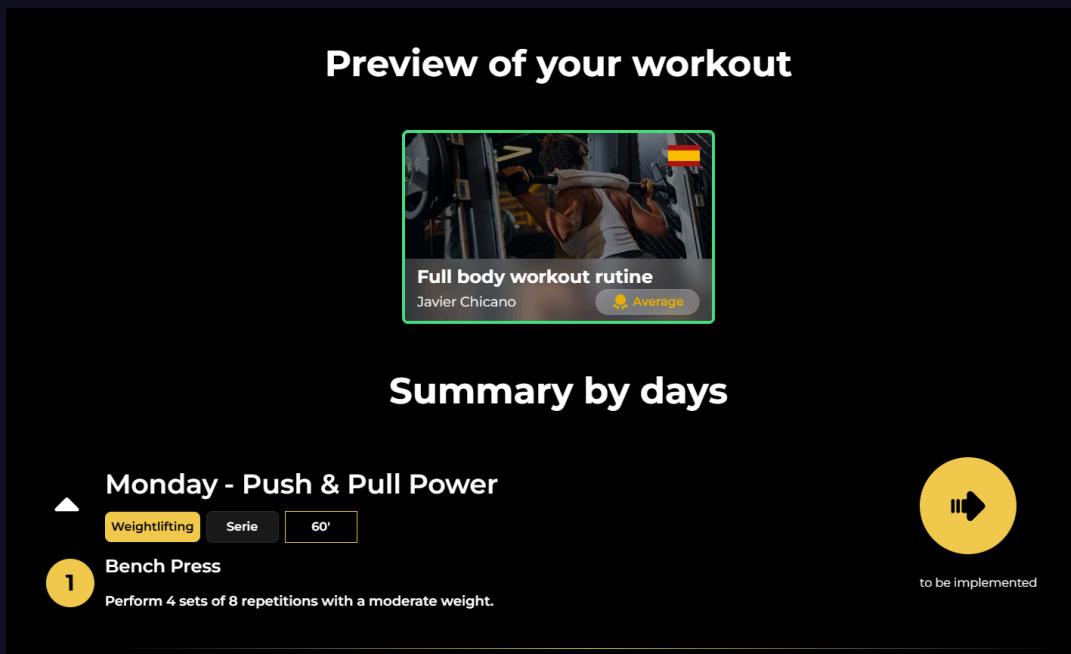
Si todo este proceso se realiza de forma exitosa, se ejecutará lo siguiente:

```
● ● ●
1 router.push("/Profile/PreviewWorkout?selected=workouts&idWorkout=" + response.idWorkout);
```

Esto, manda al usuario a la página “*PreviewWorkout*” e introduce en la URL el id del workout creado, para poder rescatarlo más tarde.

Mostrar el workout creado

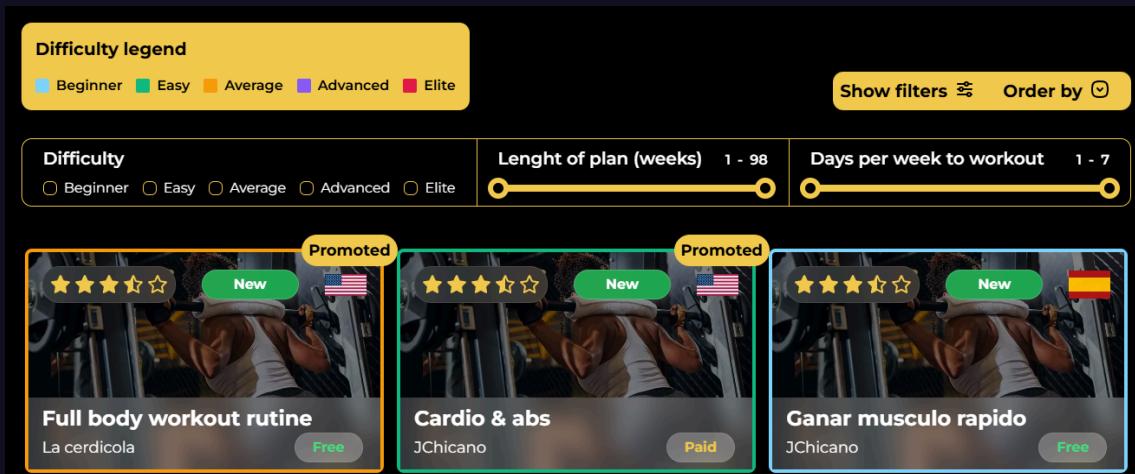
Una vez en la página consultaremos en la base de datos el workout correspondiente al de la id rescatada, y tras hacer los correspondientes procesos (cachear, promises e implementar skeletons), se muestra de la siguiente manera



Lo primero es un preview de como se verá el workout en la página de workouts y el summary se acerca más a cuando un usuario está realizando el workout. Cada día tiene un desplegable con los ejercicios correspondientes.

Página workouts

Vamos a poner el foco en esta página ya que realmente es la más importante y por ello la que más chicha tiene, ya que es la base del producto de la aplicación.



La página por defecto te muestra todos los workouts, ordenados para que los promoted sean los primeros, como opciones tenemos los filtros y ordenar en función de lo que queramos.

El “OrderBy” cogerá el array de workouts de la consulta y separará los promoted del resto, y a estos los ordenará en función de lo seleccionado por el usuario, para que como resultado tengamos de nuevo primero a los promoted y luego ordenados a gusto del usuario.

A su vez esto se puede combinar con los filtros, los cuales funcionan técnicamente de la siguiente manera, Difficulty es un array que almacena los campos seleccionados, pudiendo ser del 1 al 5 y varios a la vez, luego la longitud del plan de entrenamiento en el cual tenemos un Slider que nos dá un valor seleccionado mínimo y otro máximo, con ello seleccionaremos los workouts que estén dentro de ese rango, y por último los días a la semana de entrenamiento que funciona de la misma forma que el anterior. Estos 3 filtros se pueden combinar entre ellos y con el OrderBy a la vez.

Implementar sistema de filtrado

Una vez que el usuario define los filtros (y por lo tanto cambia el valor de “filters”), se ejecuta este useEffect()

```

1 // Build query string from filters state and update url
2 useEffect(() => {
3   const sp = new URLSearchParams();
4   if (filters.weeks) sp.set("weeks", filters.weeks.join(","));
5   if (filters.daysWeek) sp.set("daysWeek", filters.daysWeek.join(","));
6   if (filters.difficulty && filters.difficulty.length > 0)
7     sp.set("difficulty", filters.difficulty.join(","));
8   if (filters.orderBy) sp.set("orderBy", filters.orderBy);
9   router.push(`?${sp.toString()}`);
10 }, [filters]);

```

El cual pushea a la URL de la página los filtros con sus valores y el orderBy, para que la página que muestra los workouts pueda coger los datos correspondientes.

```

1 // Await and extract filters using the new function
2 const resolvedSearchParams = await searchParams;
3 const filters = extractFilters(resolvedSearchParams);
4
5 // If filters is empty, pass undefined; otherwise, pass the filters object
6 const workoutsPromise = fetchWorkoutData(
7   Object.keys(filters).length > 0 ? filters : undefined
8 );

```

Una vez en la página extraemos de la URL los datos, con la función “*extractFilters*”, que no solo los extrae sino que también los valida, para que solo puedan ser ciertos valores, ya que la URL es editable por el usuario y cualquiera podría modificar estos filtros a su gusto. Una vez validados se pasa por parámetro al *fetch*, de manera que si hay filtros se pasa como objeto y si no los hay se pasa “*undefined*” para que se muestren los workouts de manera normal.

El *fetch* funciona igual que los vistos previamente, el cual luego se lo pasa al “*route*” que a su vez llama al método *select*, continuando arrastrando los filtros y pasándolo por los *props*, y es ahí donde en función de los filtros que define el usuario se hace una consulta u otra.

Como información extra, mencionar que el cacheo de las consultas funciona de la siguiente manera, cada vez que el “*queryString*” (la cadena con la información de los filtros definidos), cambia, la consulta se realizará pero se vuelve a repetir con una consulta pasada, esta, al estar ya en la caché se mostrará directamente sin necesidad de consultar a la base de datos. Por ejemplo, si se selecciona como filtros, que el nivel de dificultad sea 1 y 3, se van a mostrar los correspondientes, si luego se agrega otro filtro o se sustrae, la consulta se volverá a hacer ya que sigue siendo diferente, pero si volvemos a poner como filtros únicamente la dificultad a 1 y 3 (como en un inicio), se mostrará directamente de la caché, al haberse hecho anteriormente, esto ahorra recursos de la base de datos, al evitar consultas.

Sistema de paginación

Esto es una función realmente importante, ya que con el volumen de creación de workouts que pueden tener los usuarios, si consultabamos el volumen total para mostrar todos los workouts de golpe cada vez que un usuario entra en la página cada vez iba a tardar mas, por no hablar del coste en peticiones a la base de datos, por ello, se ha implementado un botón al final de los workouts que pone “Cargar más”, al principio se mostrarán los 18 primeros workouts de la consulta (valor que está especificado en una constante y puede ser

cambiado cuando sea) y es cuando el usuario clicka en este botón cuando hacemos nuevamente otra consulta para mostrar los 18 siguientes resultados, de esta forma un usuario puede seguir viendo la información de todos los workouts pero no es necesario dársela nada más entrar a la página ya que puede que no la necesite.

Implementación de pagos con Stripe

Esta podría considerarse la parte más importante de todo el desarrollo, ya que sin ello el proyecto no tendría sentido, a su vez, debido a la complejidad de la aplicación habrá que tener muchas cosas en cuenta, desde que un usuario puede cancelar su suscripción, un coach puede aumentar el precio de esta y por consiguiente las que están activas deberán de cambiar o simplemente que un usuario quiera suscribirse por primera vez a un coach, y junto a todo esto, aparte de implementar el proceso con [Stripe](#), a su vez combinará todos los procesos vistos anteriormente, ya que la información del pago tendrá que guardarse en la base de datos, y de la misma forma, para informar al cliente de sus acciones se tendrán que enviar correos electrónicos automáticos con facturas, recordatorios o información del estado de sus suscripciones.

Sesiones

Cuando un usuario hace click en el botón suscribirse, lo primero que se verifica es que esté logueado, si es así que no solo lo esté sino que su email haya sido verificado, y por último que no este con una suscripción activa a la persona que se está suscribiendo (para no cobrarle dos veces lo mismo).

A continuación se chequea el precio de la suscripción (especificado por el coach), si es de 0, es decir, gratis, se añade a la base de datos y se termina el proceso, ya que no necesitamos acciones extras por parte del usuario. De lo contrario, si el usuario tiene que pagar para suscribirse, se creará una *session de stripe*, en la cual rescataremos el *priceld*, que corresponde al producto creado por el Coach (suscripción la cual el miembro está comprando), y con ello crearemos un *session checkout*, de la siguiente forma:



```
1 // Create a new session
2 const session = await stripe.checkout.sessions.create({
3   customer_email: body.email,
4   payment_method_types: ["card"],
5   line_items: [
6     {
7       price: priceId,
8       quantity: 1,
9     },
10    ],
11   mode: "subscription",
12   success_url: `${url}/Coach/${body.coachId}`,
13   cancel_url: `${url}/`,
14   custom_text: {
15     submit: {
16       // We will send the receipt to the user email
17       message: body.message,
18     },
19     terms_of_service_acceptance: {
20       message: `${body.tos}(${url}/tos)` ,
21     },
22   },
23   consent_collection: {
24     terms_of_service: "required",
25   },
26   // Added subscription_data to ensure metadata is attached to the subscription
27   subscription_data: {
28     metadata: {
29       userId: body.userId,
30       userEmail: body.email,
31       lang: body.lang,
32       coachId: body.coachId,
33     },
34   },
35 });

});
```

[← Entorno de prueba predeterminado](#) TEST MODE

Suscribirse a Personalized Training Plan

32,00 € por mes

Get full access to all workouts created by the coach, including current and future sessions. Enjoy direct, personalized support whenever you need it.

Información de contacto

Correo electrónico	jchicano43@gmail.com
--------------------	----------------------

Método de pago

Información de la tarjeta

1234 1234 1234 1234	
MM / AA	CVC

Nombre del titular de la tarjeta

País o región

Guardar mis datos de forma segura para un proceso de compra en un clic
Paga más rápido en Entorno de prueba predeterminado y en todos los comercios que acepten Link.

Accept the [terms and conditions](#)

We will send you an email with the payment details.

Pagar y suscribirse

Al confirmar tu suscripción, permitirás que Entorno de prueba predeterminado efectúe cargos de futuros pagos conforme a las condiciones estipuladas. Siempre puedes cancelar tu suscripción.

Powered by | [Condiciones](#) [Privacidad](#)

Una vez paga, si toda la información proporcionada es correcta, el proceso será exitoso y el usuario volverá a la página. Mientras tanto, Stripe mandará eventos, los cuales capturaremos en nuestro webhook con la intención de actualizar nuestra base de datos con los nuevos datos.

Webhook

Quiero destacar que para probar esto estando la página en local, es necesario lanzar un puerto público para que Stripe lo pueda ver y mandar ahí la información, ya que sino no podremos interceptar la información de los eventos.

PROBLEMS	OUTPUT	TERMINAL	PORTS 1	GITLENS	DEBUG CONSOLE									
			<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Port</th> <th>Forwarded Address</th> <th>Running Process</th> <th>Visibility</th> <th>Origin</th> </tr> </thead> <tbody> <tr> <td>3000</td> <td>https://r57gr8qw-3000.ualt1.devrun...</td> <td></td> <td></td> <td>User Forwarded</td> </tr> </tbody> </table>	Port	Forwarded Address	Running Process	Visibility	Origin	3000	https://r57gr8qw-3000.ualt1.devrun...			User Forwarded	
Port	Forwarded Address	Running Process	Visibility	Origin										
3000	https://r57gr8qw-3000.ualt1.devrun...			User Forwarded										

De esta forma, nuestra página estará hosteada de forma temporal por esa URL pública, a la cual podrá acceder todo el mundo y entre ellos Stripe (que es lo que buscamos para poder recibir los eventos).

En el Webhook.ts, tendremos la recepción de esos eventos con un switch, que gestionará las acciones de cada tipo de evento, a continuación veremos el más típico, “*invoice.payment_succeeded*” que es el que se lanza cuando un checkout ha sido exitoso.

```

1  switch (event.type) {
2      // Payment succeeded and subscription created
3      case "invoice.payment_succeeded":
4          const invoice = event.data.object;
5          let subscriptionData;
6          if (typeof invoice.subscription === "string") {
7              const subscription = await stripe.subscriptions.retrieve(
8                  invoice.subscription
9              );
10             //Construct the data object
11             subscriptionData = {
12                 idUserSaving: subscription.metadata?.userId, // Get userId from subscription metadata
13                 idCoachSubscribed: subscription.metadata?.coachId, // Get coachId from subscription metadata
14                 language: subscription.metadata?.lang,
15                 customerEmail: invoice.customer_email || "unknow", // Customer email
16
17                 // Stripe Data
18                 stripeSubscriptionId: invoice.subscription, // Subscription ID in Stripe
19                 stripeCustomerId: invoice.customer?.toString(), // Customer ID in Stripe
20                 stripePriceId: invoice.lines.data[0]?.price?.id || null, // Stripe plan ID
21                 amountPaid: invoice.amount_paid / 100, // Converted from cents to euros
22                 currency: invoice.currency, // Payment currency
23
24                 // Dates
25                 startDate: new Date(invoice.created * 1000), // Payment date
26                 endDate: invoice.lines.data[0]?.period?.end
27                     ? new Date(invoice.lines.data[0].period.end * 1000)
28                     : undefined, // Billing period end
29                 renewalDate: invoice.lines.data[0]?.period?.start
30                     ? new Date(invoice.lines.data[0].period.start * 1000)
31                     : undefined, // Next renewal
32
33                 // Subscription status
34                 status: invoice.status === "paid" ? "active" : "unpaid",
35
36                 // Update control
37                 lastUpdated: new Date(),
38             };
39         }
40     if (subscriptionData) {
41         //Check if the subscription already exists (existingSubscription)
42         const { sub, user, coach } = await findSubscription(
43             subscriptionData.idUserSaving,
44             subscriptionData.idCoachSubscribed
45         );
46         if (!sub) {
47             // 1 Case: Insert into database (if not exists)
48             await insertNewSubscription(subscriptionData);
49         } else if (
50             sub.stripeSubscriptionId === subscriptionData.stripeSubscriptionId
51         ) {
52             // 2 Case: User update subscription automatically
53             await updateSubscription(subscriptionData, "auto");
54         } else if (sub.status === "canceled" || sub.status === "expired") {
55             // 3 Case: User resubscribes to a coach they were previously subscribed to
56             await updateSubscription(subscriptionData, "resubscribe");
57         } else if (
58             sub.status === "free" ||
59             sub.stripePriceId !== subscriptionData.stripePriceId
60         ) {
61             // 4 Case: The user previously had a free plan or a cheaper plan, and the coach has switched the price
62             await updateSubscription(subscriptionData, "increasePrice");
63         }
64
65         //Send an email to the user with the invoice
66         sendInvoiceEmail(
67             subscriptionData,
68             user.name,
69             coach.nickname ? coach.nickname : coach.name
70         );
71         console.log("Checkout payment completed:", invoice);
72         revalidateTag("coachSelected-data");
73         revalidateTag("workouts-user");
74     }
75     break;

```

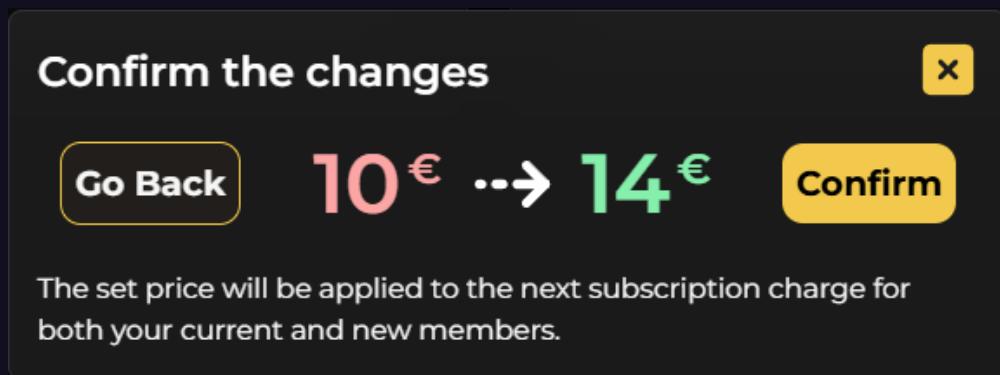
En este evento, crearemos un objeto con parte de la información que nos proporciona Stripe, y según el caso específico con el que nos encontremos añadiremos la información en la base de datos de una forma o de otra.

Es importante, que como veremos más adelante, siempre que hagamos una modificación en las suscripciones Stripe lanzará sus eventos correspondientes y por lo tanto será en este archivo Webhooks.ts donde tendremos que gestionar la inserción, actualización o incluso borrado de las suscripciones en la base de datos.

Suscripciones

Este es el proceso que más trabajo da ya que cada Coach puede modificar el precio de su suscripción de manera unilateral y sin “supervisión”, así que habrá que tener en cuenta todos los casos.

Modificación del precio de la suscripción



Esta es la ventana de confirmación del nuevo precio de la suscripción de un coach, una vez confirme se darán los siguientes casos;

- Que pase de un precio X a uno mayor, en ese caso se pararán de forma automática todas las suscripciones existentes a ese coach, y se notificará del nuevo precio a cada cliente mediante un correo, y serán estos los que tendrán que aceptar pagar el nuevo precio, en el caso de que no quieran, cuando la suscripción llegue a su fecha final se cancelará, (por ello esta acción puede conllevar en la pérdida de suscriptores por parte del Coach).
- Que pase de un precio mayor a uno menor, al ser “beneficioso” para el cliente final, este proceso actualiza todas las suscripciones existentes de forma automática y simplemente notifica a los clientes de la bajada de precio, pero no es necesario ninguna acción por su parte.
- Que pase de ser gratis a de pago, en este caso, como he mencionado anteriormente al haber sido gratis la suscripción del cliente se guardó sin pasar por Stripe, por lo tanto aparte de notificar al cliente y poner en pausa su suscripción, para la reactivación de la misma se tendrá que pasar por el proceso de checkout.

- Que pase de un precio X a gratis, aunque esta situación puede ser la más rara, no es la mejor ya que al volverse gratis las suscripciones existentes cambian de plan y los datos de pago en Stripe se borrarán es decir las suscripciones en Stripe se eliminan, esto provoca que si en un futuro se quiere volver a poner de pago se convierta en el caso 3 (visto anteriormente), que es el que menos favorece a la permanencia de los clientes.

Gestión de las suscripciones

El usuario podrá ver sus suscripciones, tanto activas como canceladas o pendientes de pago (este último caso será por ejemplo, cuando el Coach ha subido el precio), desde la sección de suscripciones en la página del perfil.

Desde ahí será capaz de desuscribirse (en el caso de las gratis), esto provocará la eliminación de la base de datos y por lo tanto no volverá a mostrarse en esta ventana, y en cualquier otro caso (las de pago) se podrá modificar su estado y ver el historial de la misma clicando el botón modify en cada suscripción, eso nos llevará a una “*billing.session*” de stripe

The screenshot shows a user interface for managing a subscription. At the top, it says "SUSCRIPCIÓN ACTUAL". Below that, it displays a "Personalized Training Plan" at a price of "10,00 € por mes". To the right, there is a button labeled "Cancela la suscripción". Underneath the plan details, it states "Tu suscripción se renovará el 12 de mayo de 2025." Further down, it shows a payment method as "Visa **** 4242" with an expiration date of "Caduca el 04/2044". There is also a link "+ Añadir método de pago". At the bottom, under "HISTORIAL DE FACTURAS", it lists an invoice from "12 abr 2025" for "10,00 €" which is marked as "Pagada". The plan name "Personalized Training Plan" is also listed next to the invoice.

Desde aquí podremos ver el historial de las facturas, el método de pago y cambiarlo, y por último cancelar o activar la suscripción.

Sistema de notificaciones

Cuando ocurren los diferentes procesos de la web, es necesario notificar a los clientes de que han ocurrido ciertas acciones, por ello se implementa un sistema de envío de correos electrónicos a los usuarios, cuando se loguea, se verifica su correo (como hemos visto en el apartado de [verificación de la cuenta](#)), o cuando se procesan pagos y se modifican las suscripciones.

Implementación del SEO

Una vez terminada la página, en lo que a funcionamiento se refiere, es necesario implementar el SEO para mejorar el posicionamiento en los motores de búsqueda, por lo tanto este proceso no es estrictamente necesario pero sí altamente recomendado para que la página tenga más exposición.

Se instalará el paquete “*next-sitemap*” que ayuda con la creación de los ficheros como el “*robots.txt*” y el “*sitemap.xml*”, con los que indicaremos a los bots de google la estructura de la página y la importancia de cada indexación de la misma, ambos archivos se generan automáticamente en función del proyecto. Por otro lado, tendremos el “*manifest.json*” para mejorar el SEO en móviles y facilitar la conversión de la página a una APP PWA (Progressive Web App).

Se ha añadido un apartado en el root layout de la aplicación (Componente que envuelve todas las páginas de la web), el cual genera diferentes metadatos.

```
● ● ●
1 export async function generateMetadata({
2   params,
3 }: {
4   params: { lang: TypeLanguages };
5 }): Promise<Metadata> {
6   const { lang } = await params;
7
8   return {
9     title: getTitle[lang] || getTitle.en,
10    description: getDescription[lang] || getDescription.en,
11    openGraph: {
12      title: "Peak Mate",
13      description: "Entrena desde casa con planes personalizados.",
14      url: "https://www.peakmate.fit",
15      siteName: "Peak Mate",
16      images: [
17        {
18          url: "https://www.peakmate.fit/og-image.jpg",
19          width: 800,
20          height: 600,
21          alt: "Peak Mate"
22        }
23      ],
24      type: "website",
25    },
26    alternates: {
27      canonical: "https://www.peakmate.fit",
28    },
29    icons: {
30      icon: "/favicon.ico",
31      shortcut: "/favicon.ico",
32      apple: "/apple-touch-icon.png",
33    },
34    manifest: "/manifest.json",
35  };
36 }
```