2023

GyMedia

Hilario José Bandrés Sasal

I.E.S Romero Vargas

11-6-2023

GYMEDIA

GyMedia es una innovadora aplicación de red social desarrollada en React que ha revolucionado la forma en que los entusiastas del fitness y la nutrición interactúan en línea. Esta plataforma ofrece a los usuarios una experiencia completa y diversa, donde pueden publicar, compartir y descubrir contenido relacionado con su pasión por mantenerse en forma y llevar una vida saludable.

En la sección de posts, los usuarios tienen la libertad de expresar sus ideas, pensamientos y experiencias relacionadas con el mundo del fitness. Pueden compartir no solo texto, sino también fotos inspiradoras, enlaces a artículos interesantes y consejos útiles. Los demás usuarios tienen la posibilidad de interactuar con estas publicaciones mediante comentarios, likes y compartirlas con su propia red.

En la sección de rutinas, GyMedia permite a los usuarios crear y compartir planes de ejercicios y entrenamientos efectivos. Pueden detallar cada ejercicio, establecer repeticiones, series y ofrecer consejos valiosos para maximizar los resultados. Los usuarios pueden descubrir nuevas rutinas, seguir a entrenadores expertos y recibir notificaciones cuando se publiquen nuevas rutinas de su interés. Además, la comunidad puede interactuar, compartir sus experiencias y brindar apoyo mutuo en el camino hacia sus objetivos fitness.

La sección de recetas es un verdadero tesoro culinario para los amantes de la comida saludable. Los usuarios pueden publicar y descubrir recetas deliciosas y nutritivas. Pueden compartir ingredientes, instrucciones de preparación detalladas y consejos nutricionales. La comunidad puede comentar, hacer preguntas y experimentar con las recetas compartidas, fomentando así la creatividad en la cocina y el intercambio de conocimientos.

GyMedia no se limita solo a la interacción y el contenido. La plataforma proporciona estadísticas detalladas sobre la actividad de los usuarios, como el número de seguidores, likes y comentarios recibidos en cada publicación. Estas estadísticas permiten a los usuarios evaluar su impacto y popularidad en la comunidad y ajustar su estrategia en consecuencia.

Para aquellos usuarios que buscan aumentar la visibilidad de sus publicaciones, GyMedia ofrece una opción de suscripción premium. Mediante esta suscripción, los usuarios pueden promocionar sus posts de manera destacada, llegar a un público más amplio y obtener una mayor visibilidad dentro de la comunidad. Esto es especialmente valioso para aquellos que buscan construir una marca personal o promover sus servicios relacionados con el fitness y la nutrición.

En resumen, GyMedia es mucho más que una simple red social en React. Es una plataforma completa y dinámica que permite a los usuarios compartir, descubrir y

promocionar contenido relacionado con posts, rutinas y recetas en un entorno amigable y enfocado en el fitness y la nutrición. Es el lugar perfecto para conectar con personas afines, aprender unos de otros, encontrar inspiración y alcanzar los objetivos personales de bienestar.

Herramientas usadas

1. **React**: React es una biblioteca de JavaScript ampliamente utilizada y respaldada por Facebook. Es conocida por su eficiencia y rendimiento al crear interfaces de usuario interactivas y dinámicas. React es la base del frontend de GyMedia, permitiendo crear componentes reutilizables y una interfaz de usuario moderna y receptiva.
2. **ExpressJS**: ExpressJS es un framework de desarrollo de aplicaciones web para Node.js. Proporciona una capa adicional de abstracción para simplificar la creación de servidores y el manejo de rutas y solicitudes HTTP. ExpressJS se utiliza en el backend de GyMedia para crear una API robusta y manejar las solicitudes de los usuarios.
3. **MongoDB**: MongoDB es una base de datos NoSQL flexible y escalable. Se utiliza en GyMedia para almacenar y gestionar la información del usuario, como perfiles, publicaciones, rutinas, recetas, etc. MongoDB permite un almacenamiento eficiente y una fácil manipulación de datos, lo que resulta ideal para una aplicación como GyMedia.
4. **MongoDB Atlas**: MongoDB Atlas es un servicio en la nube que proporciona una forma fácil y segura de alojar y administrar bases de datos MongoDB. GyMedia utiliza MongoDB Atlas para alojar la base de datos de la aplicación, garantizando la escalabilidad, la disponibilidad y la seguridad de los datos de los usuarios.
5. **FilePond**: FilePond es una biblioteca JavaScript de carga de archivos que facilita la carga y manipulación de imágenes y archivos en la aplicación. Se utiliza en GyMedia para permitir a los usuarios cargar y adjuntar imágenes a sus publicaciones y perfiles de usuario.
6. **Stripe**: Stripe es una plataforma de pago en línea utilizada para procesar transacciones financieras de manera segura. En GyMedia, se integra Stripe para habilitar la funcionalidad de suscripción premium, permitiendo a los usuarios comprar una suscripción y realizar pagos de manera segura.
7. **Moment.js**: Moment.js es una biblioteca de JavaScript utilizada para manipular, formatear y mostrar fechas y horas. En GyMedia, Moment.js se utiliza para gestionar y mostrar las fechas y horarios de las publicaciones, comentarios y otras actividades relacionadas con el tiempo.

8. **Charts**: Charts (o gráficos) son bibliotecas de JavaScript que permiten visualizar datos en forma de gráficos interactivos y atractivos. En GyMedia, se utilizan bibliotecas de gráficos como Chart.js o D3.js para generar visualizaciones y estadísticas detalladas sobre la actividad de los usuarios, como el número de seguidores, likes y comentarios.

Estas herramientas proporcionan las bases tecnológicas necesarias para desarrollar y ofrecer una experiencia de usuario sólida y completa en GyMedia, desde la interfaz de usuario interactiva hasta el manejo eficiente de datos, la seguridad de pagos y la visualización de estadísticas relevantes.

Finalidad

La finalidad de la aplicación GyMedia es proporcionar a los entusiastas de la nutrición y el fitness una plataforma integral que reúna todos los elementos necesarios para satisfacer sus necesidades e intereses. El objetivo principal es crear un espacio virtual en el que las personas interesadas en estos temas puedan conectarse, compartir conocimientos, encontrar inspiración y obtener apoyo mutuo.

GyMedia tiene como propósito facilitar el acceso a información relevante y de calidad sobre nutrición, ejercicios y estilo de vida saludable. La aplicación busca ser una fuente confiable de contenido variado, desde publicaciones inspiradoras y motivadoras hasta rutinas de ejercicios efectivas y recetas saludables y deliciosas.

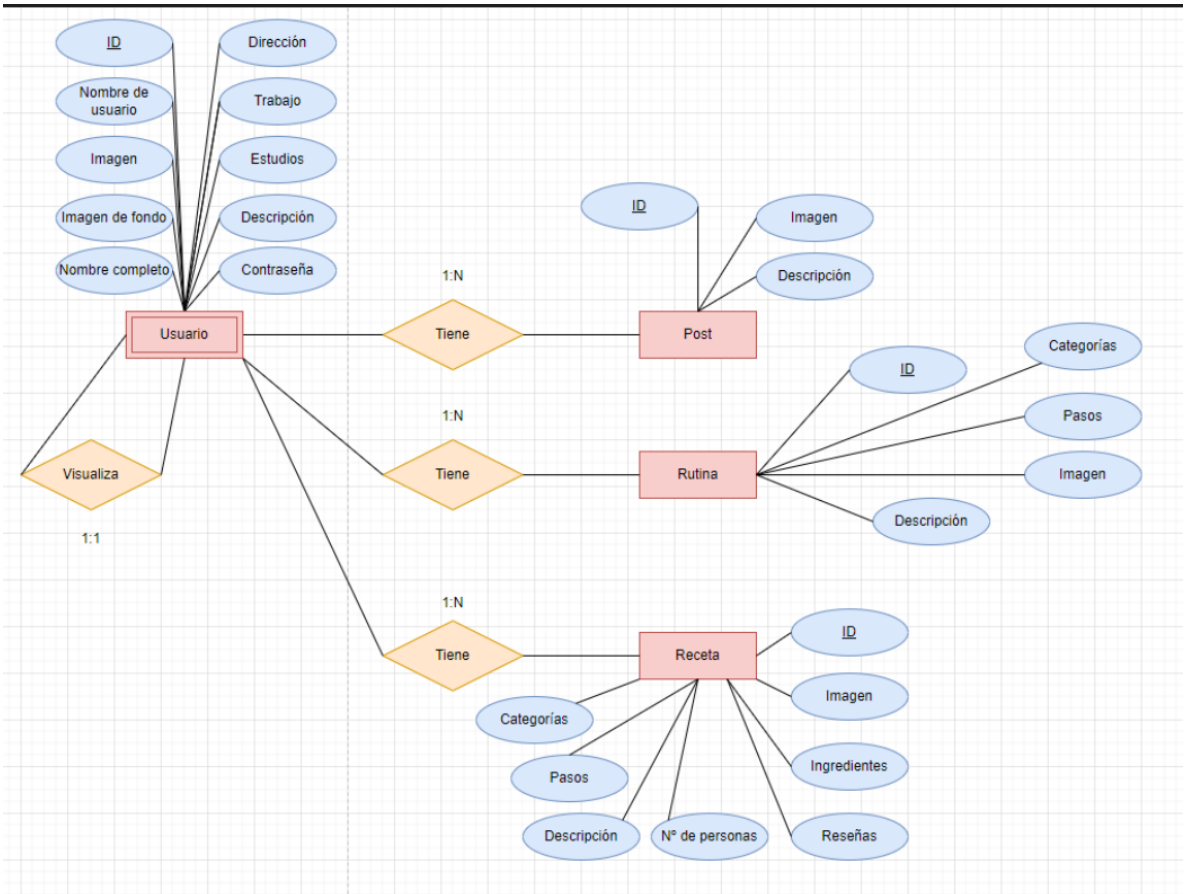
Además, GyMedia tiene un enfoque especial en ayudar a las personas nuevas en el mundo del fitness y la nutrición. La aplicación pretende ser una guía práctica y accesible para aquellos que están comenzando su camino hacia una vida más saludable. Proporciona recursos y consejos útiles, así como la posibilidad de conectarse con personas más experimentadas y recibir orientación personalizada.

En resumen, la finalidad de GyMedia es crear una comunidad en línea que reúna a personas interesadas en la nutrición y el fitness, ofreciéndoles una plataforma completa que incluya información, inspiración, apoyo y la posibilidad de interactuar entre sí. La aplicación busca simplificar y enriquecer la experiencia de aquellos que desean introducirse o profundizar en el mundo del fitness y la nutrición, brindándoles un lugar donde puedan encontrar todo lo que necesitan de manera accesible y confiable.

Planificación

- **Planificación**: 2 semanas
- **Desarrollo de la página de descarga y panel de administración (web):** 8 semanas
- **Pruebas y depuración:** 2 semanas

Esquema relacional



Mapa del sitio web

Página principal:

Sección de posts: Los usuarios pueden ver los posts publicados por otros usuarios, incluso sin iniciar sesión.

Vista sin iniciar sesión:

Página principal: Los usuarios pueden explorar y ver los posts publicados por otros usuarios.

Inicio de sesión: Los usuarios tienen la opción de iniciar sesión en su cuenta existente o registrarse para crear una cuenta nueva.

Página de registro: Los nuevos usuarios pueden crear una cuenta proporcionando su información personal y estableciendo credenciales de inicio de sesión.

Vista iniciando sesión:

Página principal: Los usuarios pueden ver los posts publicados por otros usuarios.

Sección de posts: Los usuarios pueden publicar y compartir contenido relacionado con el fitness y la nutrición.

Sección de rutinas: Los usuarios pueden encontrar y compartir planes de ejercicios y entrenamientos efectivos.

Sección de recetas: Los usuarios pueden descubrir y publicar recetas saludables y sabrosas.

Perfil de usuario:

Información de perfil: Los usuarios pueden ver y editar su información personal, como nombre, foto de perfil, biografía, enlaces sociales, etc.

Publicaciones del usuario: Se muestran las publicaciones realizadas por el usuario en un feed personalizado.

Seguidores y seguidos: Los usuarios pueden ver quiénes los siguen y a quiénes siguen.

Actividad reciente: Se muestra un resumen de las acciones recientes del usuario, como nuevos seguidores, comentarios y likes.

Dashboard:

Estadísticas: Los usuarios pueden ver estadísticas detalladas sobre su actividad en la plataforma, incluyendo el número de seguidores, likes recibidos, comentarios, etc.

Edición de perfil: Los usuarios pueden editar su información de perfil, actualizar su foto de perfil y realizar cambios en su biografía.

Lista de la compra:

Creación de la lista: Los usuarios pueden crear una lista de compras personalizada para organizar los ingredientes necesarios para las recetas que deseen preparar.

Gestión de la lista: Los usuarios pueden agregar, eliminar o marcar elementos de la lista como comprados.

Guía de estilos

Los colores que se han utilizado en toda la página son el naranja y el verde por los siguientes criterios.

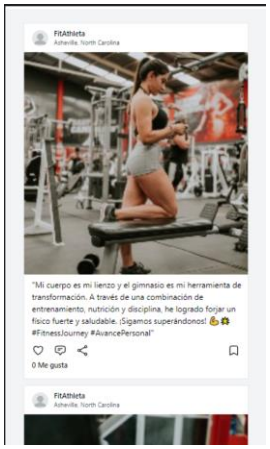
1. Naranja (orange-500): El color naranja es un color cálido y enérgico que puede evocar sensaciones de vitalidad, entusiasmo y acción. En el contexto de un gimnasio, el naranja puede simbolizar:
    - Energía y vitalidad: El naranja es un color dinámico que puede transmitir una sensación de energía y vitalidad, lo cual es relevante en un entorno de ejercicio y actividad física.
    - Motivación y entusiasmo: El naranja puede ayudar a crear un ambiente enérgico y estimulante, que motive a las personas a realizar su entrenamiento con entusiasmo y compromiso.
    - Creatividad y emoción: El naranja también puede despertar la creatividad y la emoción, lo cual puede ser beneficioso en actividades como clases de baile, entrenamientos grupales o actividades lúdicas.
2. Verde (green-700): El color verde es un color asociado con la naturaleza, la salud y el equilibrio. En el contexto de un gimnasio, el verde puede simbolizar:
    - Frescura y revitalización: El verde es un color refrescante que puede evocar una sensación de bienestar y revitalización. Puede ayudar a crear un ambiente natural y relajante, proporcionando un contraste agradable con la energía y el dinamismo del ejercicio.
    - Salud y equilibrio: El verde está asociado con la naturaleza y puede representar la salud y el equilibrio. Puede transmitir una sensación de bienestar físico y mental, lo cual es esencial en un gimnasio.
    - Renovación y crecimiento: El verde también puede simbolizar la renovación y el crecimiento, tanto a nivel físico como personal. Puede transmitir la idea de progreso, desarrollo y superación en el ámbito del fitness y la actividad física.

Al combinar el naranja y el verde en la guía de estilos de un gimnasio, se busca crear un ambiente enérgico y estimulante, con un toque de frescura y equilibrio. Estos colores pueden ayudar a transmitir la idea de un lugar donde las personas pueden encontrar motivación, vitalidad y bienestar físico y mental.

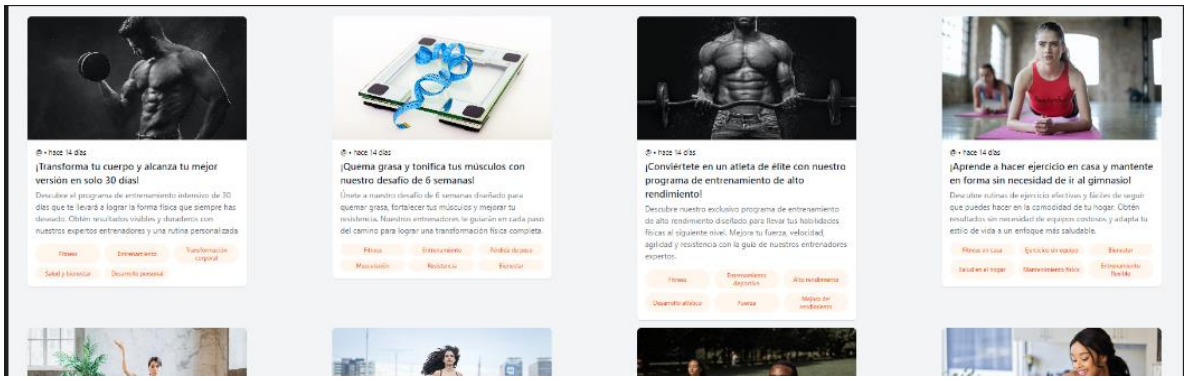
Guía de usuario

Lo primero que nos vamos a encontrar va a ser que nuestra página web puede ser usada de forma anónima, aunque solo visualizar, no se puede crear ningún post ni ningún tipo de publicación. Nos encontramos con una barra de navegación al principio de todo que nos permite seleccionar entre las diferentes opciones.

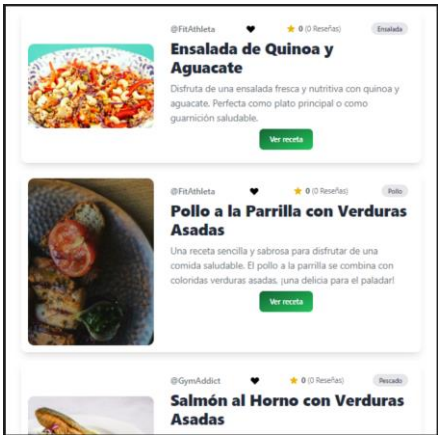
- Inicio



- Rutinas



- Recetas



También nos da las opciones de Iniciar sesión si tenemos una cuenta y Registrarse si no disponemos de ella.

- Iniciar sesión

- Registrarse

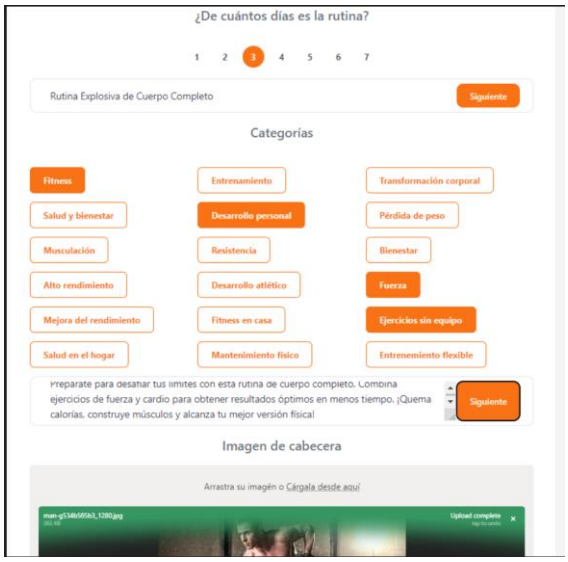
Una vez registrado, nos devolverá un token de inicio de sesión que es el que nos permitirá seguir manejándonos por la página sin tener que volver a iniciar sesión.

Una vez que hemos iniciado sesión, podemos observar que las páginas son ligeramente diferentes, puesto que ahora hay una sección de "Tus posts", "Tus Rutinas" o "Tus recetas" en las diferentes secciones.

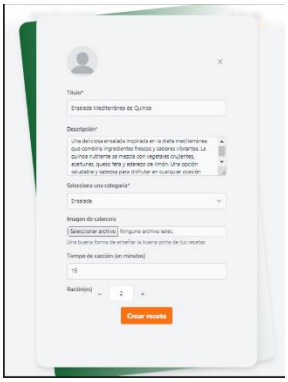
Una vez que estamos ahí, podemos seleccionar la pestaña de "Tus Posts" y eso hará que encontremos además de nuestros posts, un nuevo botón que nos permitirá crear las publicaciones pertinentes.

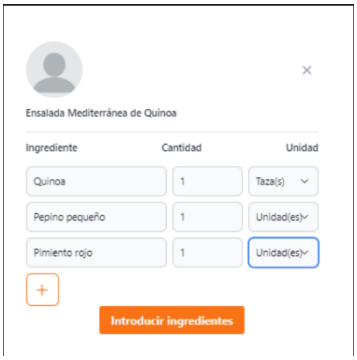
- Creación de Posts

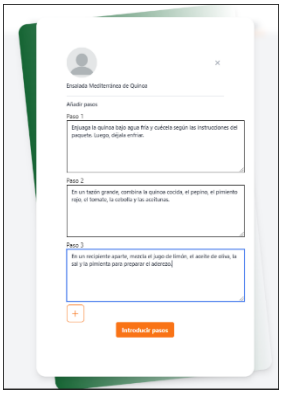
- Creación de Rutinas



- Creación de recetas

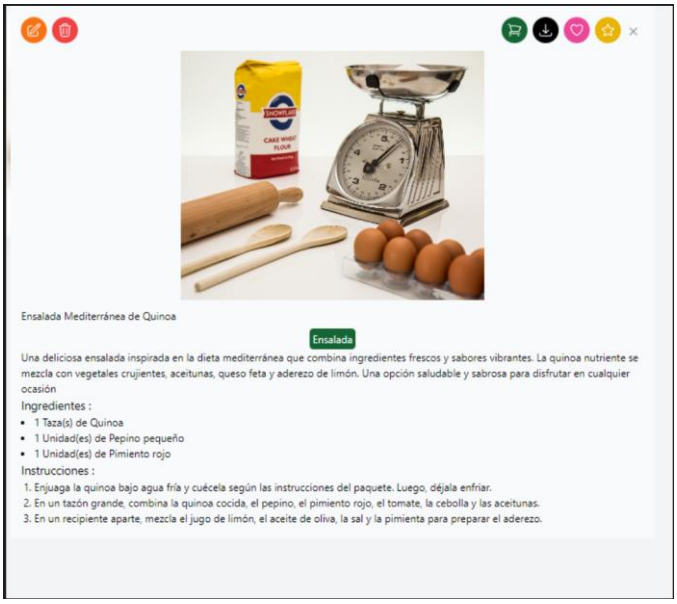






En este último si no ponemos una imagen de cabecera, nos pondrá automáticamente una imagen predeterminada.

También podemos en esto último ahora que tenemos nuestra receta, podemos ver la descripción de esta más detalladamente.

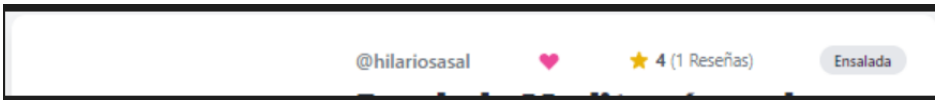


Y si nos fijamos en los botones superiores, podemos editar, borrar, añadir a la compra, descargar, darle me gusta y valorar respectivamente.

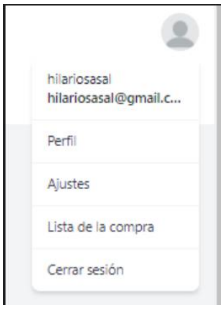
Si la descargamos se mostraría algo como esto.

También podemos valorarlo.

Y guardarlo tanto del botón del corazón exterior como del interior de la descripción de la receta.



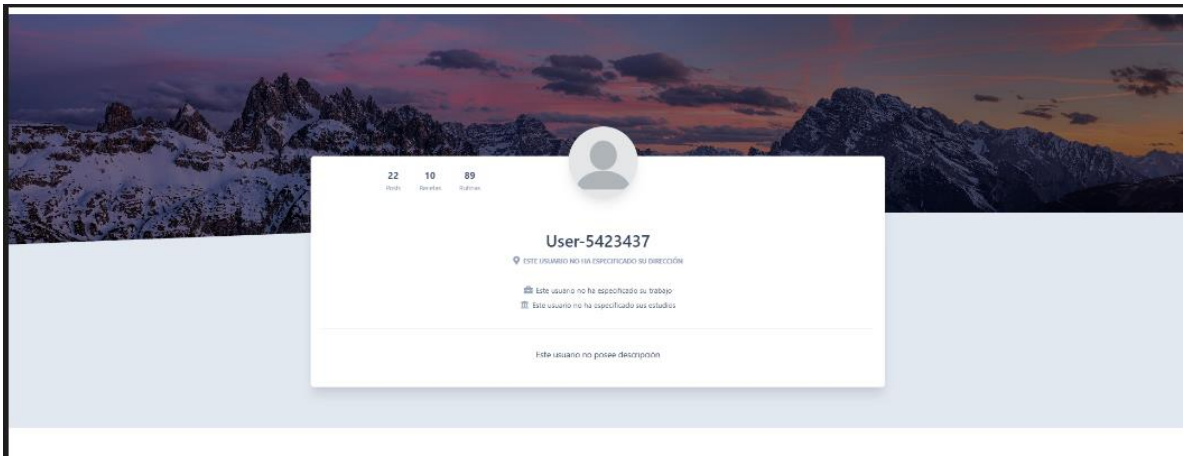
Una vez cubierto esto podemos percatarnos también que donde antes estaba "Iniciar sesión" ahora esta un avatar vacío con un desplegable.



La lista de la compra por ejemplo es la lista de los ingredientes que hayamos introducido en esta anteriormente en los botones antes mencionados.

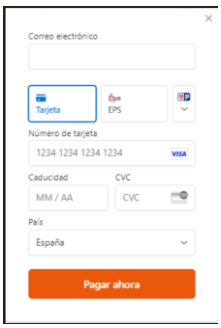
De los cuales se pueden borrar de la lista, o borrar del todo de la lista.

También tenemos nuestro perfil de usuario, que ahora mismo esta por defecto.

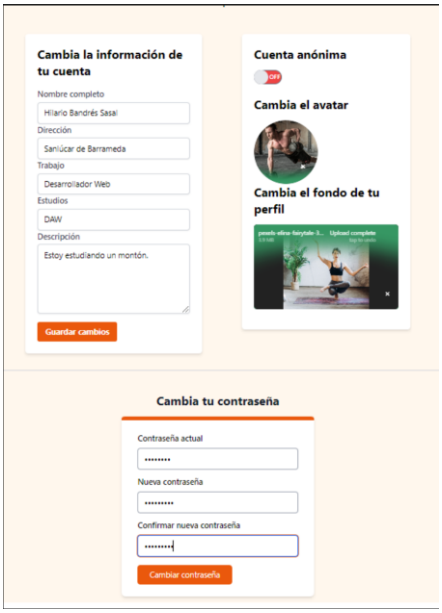


Y ajustes que nos llevaría a una nueva ventana parecida a un dashboard de administración donde tenemos nuestras estadísticas generales.

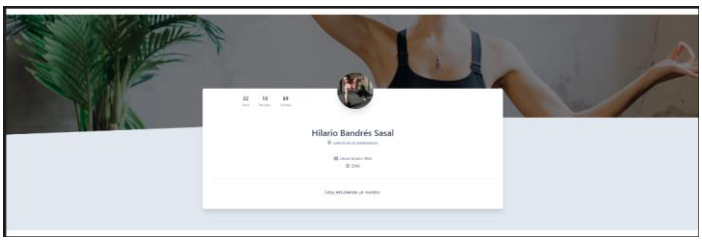
Y una pasarela de pago para poder promocionar nuestra cuenta.



Y por último tendríamos la opción de "Editar perfil" lo que nos permitiría cambiar nuestros datos a unos menos genéricos, además de cambiar nuestra contraseña, o avatar o fondo de perfil.



Que nos haría los cambios correspondientes.



Explicación del código usado

Uso de la API

La API consta de estas rutas principales.

```
app.use("/api/recipes", router);
app.use("/api/auth", authRoutes);
app.use("/api/user", userRoutes);
app.use("/api/post", postRoutes);
app.use("/api/stripe", stripeRoutes);
app.use("/api/exercices", exerciceRoutes);
app.use("/api/postExercices", PostExercicesRoutes);
```

1. `/api/recipes`: Este campo se refiere a la ruta para las operaciones relacionadas con las recetas en la API. Aquí se gestionarán las funciones de crear, leer, actualizar y eliminar

recetas, así como otras operaciones relacionadas con las recetas, como la búsqueda y filtrado.

2. `/api/auth`: Este campo se refiere a la ruta para las operaciones de autenticación en la API. Aquí se gestionarán las funciones de inicio de sesión, registro de nuevos usuarios, recuperación de contraseña y otras funcionalidades relacionadas con la autenticación y la seguridad de la aplicación.
3. `/api/user`: Este campo se refiere a la ruta para las operaciones relacionadas con los usuarios en la API. Aquí se gestionarán las funciones de obtener información de perfil de usuario, actualizar la información de perfil, seguir a otros usuarios, ver seguidores y seguidos, y otras operaciones relacionadas con la gestión de usuarios.
4. `/api/post`: Este campo se refiere a la ruta para las operaciones relacionadas con los posts en la API. Aquí se gestionarán las funciones de crear, leer, actualizar y eliminar posts, así como otras operaciones relacionadas con los posts, como la búsqueda, el filtrado y la interacción con los posts (comentarios, likes, etc.).
5. `/api/stripe`: Este campo se refiere a la ruta para las operaciones relacionadas con Stripe en la API. Aquí se gestionarán las funciones de integración con la pasarela de pago de Stripe, como la creación de una suscripción, el procesamiento de pagos y otras funcionalidades relacionadas con los aspectos de pago de la aplicación.
6. `/api/exercices`: Este campo se refiere a la ruta para las operaciones relacionadas con los ejercicios en la API. Aquí se gestionarán las funciones de crear, leer, actualizar y eliminar ejercicios, así como otras operaciones relacionadas con los ejercicios, como la búsqueda, el filtrado y la gestión de información sobre los ejercicios.
7. `/api/postExercices`: Este campo se refiere a la ruta para las operaciones relacionadas con los posts de rutinas en la API. Aquí se gestionarán las funciones de crear, leer, actualizar y eliminar posts de rutinas específicamente, así como otras operaciones relacionadas con los posts de rutinas, como la búsqueda, el filtrado y la interacción con estos posts.

API/Recipes

A su vez la API con la ruta de recipes nos encuentra con más rutas de acceso para poder interactuar con la API.

```
router.post("/posts", RecipeController.getRecipePosts);
```

La ruta post corresponde con el siguiente código.

```
export const getRecipePosts = async (req, res) => {
  const { token, active } = req.body;
  if (token == null) {
    try {
      const recipe = await Recipe.find();

      res.status(200).json(recipe);
    } catch (err) {
      res.status(400).json({ message: err.message });
    }
  } else {
```

```
    let userFound;
    let tokenid;
    try {
      const decoded = jwt.verify(token, config.SECRET);
      tokenid = decoded.id;

      userFound = await User.findById(tokenid);
    } catch (error) {
      return res.status(404).json({ message: "Token decoding error" });
    }

    if (!userFound) {
      return res.status(404).json({ message: "User not found" });
    }

    if (active) {
      const likedPosts = userFound.liked_posts;
      const createdPosts = userFound.created_posts;
      const likedRecipes = await Recipe.find({ _id: { $in: likedPosts } });
      const createdRecipes = await Recipe.find({ _id: { $in: createdPosts } });
      const createdAndLikedRecipes = [];
      const result = [];

      createdRecipes.forEach((createdRecipe) => {
        const index = likedPosts.indexOf(createdRecipe._id);
        if (index !== -1) {
          const createdAndLikedRecipe = {
            ...createdRecipe.toJSON(),
            liked: true,
            created: true,
            createdAndLiked: true,
          };
          createdAndLikedRecipes.push(createdAndLikedRecipe);
          likedPosts.splice(index, 1);
        } else {
          const createdRecipeObj = {
            ...createdRecipe.toJSON(),
            liked: false,
            created: true,
            createdAndLiked: false,
          };
          result.push(createdRecipeObj);
        }
      });

      likedRecipes.forEach((likedRecipe) => {
        const likedRecipeObj = {
          ...likedRecipe.toJSON(),
          liked: true,
          created: false,
          createdAndLiked: false,
        };
        if (!createdPosts.includes(likedRecipe._id)) {
          result.push(likedRecipeObj);
        }
```

```
      });

      createdAndLikedRecipes.forEach((createdAndLikedRecipe) => {
        if (
          !result.some((recipe) => recipe._id === createdAndLikedRecipe._id)
        ) {
          result.push(createdAndLikedRecipe);
        }
      });

      return res.status(200).json(result);
    } else {
      const likedPosts = userFound.liked_posts;
      const recipes = await Recipe.find();
      const result = recipes.map((recipe) => ({
        ...recipe.toJSON(),
        liked: likedPosts.includes(recipe._id),
        created: false,
        createdAndLiked: false,
      }));
      return res.status(200).json(result.reverse());
    }
  }
};
```

La función `getRecipePosts` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga un token y un indicador de activación.

En primer lugar, se realiza una verificación para determinar si el token está presente en la solicitud. Si no hay un token, se intenta buscar todas las recetas disponibles en la base de datos utilizando el modelo de `Recipe` y se devuelve una respuesta con el estado 200 y las recetas encontradas en formato JSON. Si ocurre algún error durante la búsqueda, se devuelve una respuesta con el estado 400 y un mensaje de error.

Si hay un token presente en la solicitud, se realiza una serie de acciones para obtener las publicaciones de recetas relacionadas con el usuario correspondiente al token.

Se verifica la validez del token decodificándolo con la clave secreta (`config.SECRET`) y se obtiene el ID del usuario (`tokenid`) a partir del token decodificado. Se busca al usuario correspondiente en la base de datos utilizando el modelo de `User` y, si no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que el usuario no se encontró.

Si el indicador de activación es verdadero (`active = true`), se procede a obtener las publicaciones de recetas según la interacción del usuario. Se recuperan las publicaciones que al usuario le han gustado (`likedPosts`) y las publicaciones que el usuario ha creado (`createdPosts`). Luego, se buscan las recetas que coinciden con las ID de las publicaciones que le han gustado y las ID de las publicaciones que ha creado. Se crean dos arreglos adicionales para almacenar las recetas que el usuario ha creado y le han gustado (`createdAndLikedRecipes`) y el resultado final de las publicaciones de recetas (`result`).

Se realiza un bucle para recorrer las recetas creadas por el usuario. Se comprueba si la receta también está en la lista de publicaciones que le han gustado al usuario y, en caso afirmativo, se

crea un objeto que representa la receta con propiedades adicionales para indicar que fue creada y le gustó al usuario (`createdAndLikedRecipe`). Este objeto se agrega al arreglo `createdAndLikedRecipes`. Si la receta no está en la lista de publicaciones que le han gustado, se crea un objeto representando la receta con la propiedad de creada (`created: true`) y se agrega al arreglo `result`.

A continuación, se realiza un bucle similar para las recetas que le han gustado al usuario. Se crea un objeto representando cada receta con la propiedad de le gustó (`liked: true`). Si la receta no está en la lista de publicaciones que el usuario ha creado, se agrega al arreglo `result`.

Finalmente, se realiza un bucle adicional para agregar las recetas creadas y le gustaron al arreglo `result` si no están duplicadas.

Si el indicador de activación es falso (`active = false`), se obtienen todas las recetas de la base de datos y se crea un objeto para cada una de ellas. Se verifica si la receta está en la lista de publicaciones que le han gustado al usuario y se establece la propiedad de le gustó (`liked`) en consecuencia. Todos los objetos de recetas se agregan al arreglo `result` en orden inverso.

Finalmente, se devuelve una respuesta con el estado 200 y el resultado final en formato JSON.

En resumen, esta función de controlador se encarga de obtener las publicaciones de recetas de acuerdo con la interacción del usuario y proporciona una respuesta adecuada en función de los parámetros de la solicitud.

```
router.post("/", RecipeController.createRecipePost);
```

Corresponde con el siguiente código

```
export const createRecipePost = async (req, res) => {
  let {
    title,
    description,
    category,
    cookingTime,
    token,
    img,
    ingredients,
    steps,
    persons,
  } = req.body;
  if (
    title == null ||
    description == null ||
    category == null ||
    cookingTime == null ||
    ingredients == null ||
    steps == null
  ) {
    return res.status(400).json({ message: "Missing required attributes" });
  } else {
    let userFound;
    let tokenid;
    try {
```

```
      const decoded = jwt.verify(token, config.SECRET);
      tokenid = decoded.id;

      userFound = await User.findById(tokenid);
    } catch (error) {
      return res.status(404).json({ message: "Token decoding error" });
    }

    if (!userFound) {
      return res.status(404).json({ message: "User not found" });
    }

    const username = userFound.username;

    const NewRecipe = new Recipe({
      title,
      description,
      username,
      img,
      category,
      stars: 0,
      reviews: 0,
      cookingTime,
      ingredients,
      steps,
      persons,
    });

    const savedRecipe = await NewRecipe.save();
    await User.findByIdAndUpdate(tokenid, {
      $push: { created_posts: savedRecipe.id },
    });
    const imgId = NewRecipe._id;
    return res.json({ imgId });
  }
};
```

La función createRecipePost es una función asíncrona que recibe una solicitud (req) y una respuesta (res) como parámetros. Se espera que la solicitud contenga varios atributos necesarios para crear una publicación de receta, como el título, descripción, categoría, tiempo de cocción, token de usuario, imagen, ingredientes, pasos y número de personas.

En primer lugar, se realiza una verificación para asegurarse de que todos los atributos requeridos estén presentes en la solicitud. Si alguno de los atributos requeridos es nulo, se devuelve una respuesta con el estado 400 y un mensaje indicando que faltan atributos requeridos.

Si todos los atributos requeridos están presentes, se procede a realizar algunas acciones relacionadas con el token de usuario. Se decodifica el token utilizando la clave secreta (config.SECRET) y se obtiene el ID de usuario (tokenid) a partir del token decodificado. Luego, se busca al usuario correspondiente en la base de datos utilizando el modelo de User y, si no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que el usuario no se encontró.

Si el usuario se encuentra en la base de datos, se obtiene su nombre de usuario. A continuación, se crea una nueva instancia del modelo de Recipe con los atributos proporcionados en la solicitud, incluyendo el título, descripción, nombre de usuario, imagen, categoría, tiempo de cocción, ingredientes, pasos y número de personas. La cantidad de estrellas y reseñas se inicializan en cero.

La nueva receta se guarda en la base de datos utilizando el método save() y se asigna a la variable savedRecipe. Luego, se actualiza el usuario en la base de datos agregando el ID de la receta creada a la matriz created_posts del usuario utilizando el método $push de User.findByIdAndUpdate().

Finalmente, se devuelve una respuesta en formato JSON con el ID de la imagen de la receta creada.

En resumen, esta función de controlador se encarga de crear una nueva publicación de receta en la base de datos, asociada al usuario correspondiente, y proporciona una respuesta con el ID de la imagen de la receta creada.

```
router.post("/addReview", RecipeController.addReview);
```

Corresponde con el siguiente fragmento de código

```
export const addReview = async (req, res) => {
  const { idRecipe, stars, token } = req.body;

  try {
    const recipe = await Recipe.findById(idRecipe);

    recipe.reviews += 1;
    recipe.stars = ((recipe.stars + stars) / recipe.reviews).toFixed(2);
    await recipe.save();

    return res.status(200).json(recipe);
  } catch (error) {
    return res.status(404).json({ message: "Recipe not found" });
  }
};
```

La función `addReview` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID de la receta (`idRecipe`), la puntuación de la reseña (`stars`) y el token de usuario (`token`).

En primer lugar, se extraen el ID de la receta, la puntuación de la reseña y el token de la solicitud.

A continuación, se intenta buscar la receta en la base de datos utilizando el modelo de `Recipe` y el ID proporcionado. Si se encuentra la receta, se procede a realizar algunas acciones relacionadas con la reseña.

Se incrementa el contador de reseñas (`reviews`) de la receta en 1 y se calcula la nueva puntuación promedio de estrellas (`stars`). La nueva puntuación se calcula sumando la puntuación de la reseña actual (`stars`) a la puntuación total de la receta y dividiendo el resultado por la cantidad total de reseñas (`reviews`). El resultado se redondea a dos decimales utilizando el método `toFixed(2)`.

Luego, se guarda la receta actualizada en la base de datos utilizando el método `save()`.

Finalmente, se devuelve una respuesta con el estado 200 y la receta actualizada en formato JSON.

En caso de que no se encuentre la receta en la base de datos, se devuelve una respuesta con el estado 404 y un mensaje indicando que la receta no se encontró.

En resumen, esta función de controlador se encarga de agregar una reseña a una receta existente, actualizando la puntuación promedio de estrellas y el contador de reseñas de la receta en la base de datos, y proporciona una respuesta con la receta actualizada en caso de éxito o un mensaje de error si la receta no se encuentra.

```
router.put("/", RecipeController.updateRecipePostsById);
```

Que corresponde con el siguiente código:

```
export const updateRecipePostsById = async (req, res) => {
  const { token, recipeID, like } = req.body;

  let userFound;
  let tokenid;
  let recipeFound;
  try {
    const decoded = jwt.verify(token, config.SECRET);
    tokenid = decoded.id;
    userFound = await User.findById(tokenid);
  } catch (error) {
    return res.status(404).json({ message: "Token decoding error" });
  }

  try {
    recipeFound = await Recipe.findById(recipeID);
  } catch (error) {
    return res.status(404).json({ message: "Recipe not found " });
  }

  if (!userFound) {
    return res.status(404).json({ message: "User not found" });
  }

  if (!recipeFound) {
    return res.status(404).json({ message: "Recipe not found" });
  }

  if (like) {
    if (!userFound.liked_posts.includes(recipeID)) {
      await User.findByIdAndUpdate(tokenid, {
        $push: { liked_posts: recipeID },
      });
    }
  } else {
    await User.findByIdAndUpdate(tokenid, { $pull: { liked_posts: recipeID } });
  }

  return res.json({ message: "Changes done succesfully" });
};
```

La función `updateRecipePostsById` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el token de usuario (`token`), el ID de la receta (`recipeID`) y una bandera (`like`) que indica si se debe agregar o eliminar la receta de la lista de publicaciones favoritas del usuario.

En primer lugar, se intenta decodificar el token utilizando la clave secreta (`config.SECRET`) y se obtiene el ID de usuario (`tokenid`) a partir del token decodificado. Luego, se busca al usuario correspondiente en la base de datos utilizando el modelo de `User`. Si el usuario no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que el usuario no se encontró.

A continuación, se busca la receta correspondiente en la base de datos utilizando el modelo de `Recipe` y el ID proporcionado. Si la receta no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que la receta no se encontró.

Si tanto el usuario como la receta se encuentran en la base de datos, se procede a realizar algunas acciones según el valor de la bandera `like`. Si `like` es verdadero, se verifica si la receta ya está incluida en la lista de publicaciones favoritas (`liked_posts`) del usuario. Si no está incluida, se agrega el ID de la receta a la matriz `liked_posts` del usuario utilizando el método `$push` de `User.findByIdAndUpdate()`. Si `like` es falso, se elimina el ID de la receta de la matriz `liked_posts` del usuario utilizando el método `$pull` de `User.findByIdAndUpdate()`.

Finalmente, se devuelve una respuesta en formato JSON con un mensaje indicando que los cambios se realizaron correctamente.

En resumen, esta función de controlador se encarga de actualizar las publicaciones de recetas por su ID, agregándolas o eliminándolas de la lista de publicaciones favoritas del usuario, y proporciona una respuesta con un mensaje de confirmación de que los cambios se realizaron correctamente.

```
router.delete("/", RecipeController.deleteRecipePostById);
```

Que corresponde con el siguiente código:

```
export const deleteRecipePostById = async (req, res) => {
  const { token, recipeID } = req.body;

  let userFound;
  let tokenid;
  let recipeFound;
  try {
    const decoded = jwt.verify(token, config.SECRET);
    tokenid = decoded.id;
    userFound = await User.findById(tokenid);
  } catch (error) {
    return res.status(404).json({ message: "Token decoding error" });
  }

  try {
    recipeFound = await Recipe.findById(recipeID);
  } catch (error) {
    return res.status(404).json({ message: "Recipe not found " });
```

```
  }

  if (!recipeFound) {
    return res.status(404).json({ message: "Recipe not found" });
  }

  const allUsers = await User.find({});
  allUsers.forEach(async (user) => {
    if (user.liked_posts.includes(recipeID)) {
      user.liked_posts.pull(recipeID);
      await user.save();
    }
  });

  await Recipe.findByIdAndDelete(recipeID);
  await User.findByIdAndUpdate(tokenid, { $pull: { created_posts: recipeID } });

  return res.status(200).json({ message: "Changes done succesfully" });
};
```

La función `deleteRecipePostById` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el token de usuario (`token`) y el ID de la receta (`recipeID`).

En primer lugar, se intenta decodificar el token utilizando la clave secreta (`config.SECRET`) y se obtiene el ID de usuario (`tokenid`) a partir del token decodificado. Luego, se busca al usuario correspondiente en la base de datos utilizando el modelo de `User`. Si el usuario no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que el usuario no se encontró.

A continuación, se busca la receta correspondiente en la base de datos utilizando el modelo de `Recipe` y el ID proporcionado. Si la receta no se encuentra, se devuelve una respuesta con el estado 404 y un mensaje indicando que la receta no se encontró.

Si la receta se encuentra en la base de datos, se realiza lo siguiente:

1. Se recorren todos los usuarios en la base de datos utilizando el modelo de `User`.
2. Si un usuario tiene la receta en su lista de publicaciones favoritas (`liked_posts`), se elimina el ID de la receta de la lista utilizando el método `pull()` y se guarda el usuario actualizado en la base de datos.
3. Se elimina la receta de la base de datos utilizando el método `findByIdAndDelete()` del modelo de `Recipe`.
4. Se elimina el ID de la receta de la lista de publicaciones creadas (`created_posts`) del usuario utilizando el método `$pull` de `User.findByIdAndUpdate()`.

Finalmente, se devuelve una respuesta con el estado 200 y un mensaje indicando que los cambios se realizaron correctamente.

En resumen, esta función de controlador se encarga de eliminar una publicación de receta por su ID, eliminando la receta de la base de datos y actualizando las listas de publicaciones favoritas y

creadas de los usuarios correspondientes, y proporciona una respuesta con un mensaje de confirmación de que los cambios se realizaron correctamente.

```
router.put("/img", fileUploaded, RecipeController.putImage);
```

Corresponde a este código:

```
export const putImage = async (req, res) => {
  const { imgId } = req.body;
  let img = "http://127.0.0.1:6001/assets/RecipeHeaders/DefaultPhoto.jpg";
  if (req.file) {
    img = "http://127.0.0.1:6001/assets/RecipeHeaders/" + req.file.filename;
  }

  const recipe = await Recipe.findByIdAndUpdate(imgId, { img: img });

  if (!recipe) {
    return res.status(404).json({ messsage: "Recipe not found" });
  }

  return res.status(200).json({ message: "Changes done succesfully" });
};
```

La función `putImage` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID de la imagen (`imgId`) que corresponde a la receta que se va a actualizar. Además, se espera que la solicitud contenga un archivo de imagen adjunto (`req.file`).

En primer lugar, se inicializa una variable `img` con la URL predeterminada de una imagen de receta (`DefaultPhoto.jpg`). Luego, se verifica si la solicitud contiene un archivo adjunto de imagen (`req.file`). Si es así, se actualiza la variable `img` con la URL de la ubicación del archivo de imagen en el servidor.

A continuación, se utiliza el modelo de `Recipe` para buscar y actualizar la receta correspondiente en la base de datos. Se utiliza el método `findByIdAndUpdate()` para buscar la receta por su ID (`imgId`) y se actualiza el campo `img` con la nueva URL de la imagen.

Si no se encuentra ninguna receta con el ID proporcionado, se devuelve una respuesta con el estado 404 y un mensaje indicando que la receta no se encontró.

Por último, se devuelve una respuesta con el estado 200 y un mensaje indicando que los cambios se realizaron correctamente.

En resumen, esta función de controlador se utiliza para actualizar la imagen de una receta existente en la base de datos, permitiendo que se adjunte un nuevo archivo de imagen y se actualice la URL de la imagen en la receta correspondiente.

/api/auth

```
router.post("/singUp", checkDuplicateUsernameOrEmail, AuthCtrl.signUp);
```

Corresponde a este código:

```
export const signUp = async (req, res) => {
  const { username, email, password } = req.body;
  const encryptedPassword = await User.encryptPassword(password);

  const newUser = new User({
    username,
    email,
    password: encryptedPassword,
    image: "http://127.0.0.1:6001/assets/UsersImage/defaultProfile.png",
    background_image: "",
    fullname: "User-" + Math.floor(Math.random() * 10000000),
    address: "Este usuario no ha especificado su dirección",
    job: "Este usuario no ha especificado su trabajo",
    studies: "Este usuario no ha especificado sus estudios",
    description: "Este usuario no posee descripción",
    list: [],
  });

  const savedUser = await newUser.save();

  const token = jwt.sign({ id: savedUser._id }, config.SECRET, {
    expiresIn: 86400, //24 hours
  });
  res.set("Authorization", `Bearer ${token}`);
  res.json({ token: token });
};
```

La función `signUp` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el nombre de usuario (`username`), el correo electrónico (`email`) y la contraseña (`password`) del nuevo usuario.

En primer lugar, se utiliza la clase `User` para encriptar la contraseña proporcionada utilizando el método `encryptPassword()`. Esto asegura que la contraseña del usuario se almacene de forma segura en la base de datos.

A continuación, se crea un nuevo objeto `User` con los datos proporcionados, incluyendo el nombre de usuario, el correo electrónico, la contraseña encriptada y otros campos por defecto como la imagen de perfil, la imagen de fondo, el nombre completo, la dirección, el trabajo, los estudios y la descripción.

Se guarda el nuevo usuario en la base de datos utilizando el método `save()` y se devuelve el usuario guardado.

A continuación, se genera un token de autenticación utilizando la biblioteca `jsonwebtoken`. Este token se firma con el ID del usuario (`savedUser._id`) y se configura para que expire en 24 horas.

El token se agrega en la cabecera de la respuesta utilizando el método `set()` y se envía como respuesta en formato JSON junto con un mensaje indicando que el registro se realizó correctamente.

En resumen, esta función de controlador se utiliza para registrar a un nuevo usuario en la aplicación. Crea un nuevo objeto de usuario, encripta la contraseña, lo guarda en la base de datos y genera un token de autenticación para el nuevo usuario.

```
router.post("/signIn", AuthCtrl.signIn);
```

Corresponde al siguiente código:

```
export const signIn = async (req, res) => {
  const userFound = await User.findOne({ email: req.body.email });
  const expired = req.body.remember;
  if (!userFound)
    return res.status(401).json({
      token: null,
      message: "El usuario o la contraseña no son correctos",
    });

  const matchPassword = await User.comparePassword(
    req.body.password,
    userFound.password
  );

  if (!matchPassword)
    return res.status(401).json({
      token: null,
      message: "El usuario o la contraseña no son correctos",
    });

  let token;
  if (expired) {
    token = jwt.sign({ id: userFound._id }, config.SECRET);
  } else {
    token = jwt.sign({ id: userFound._id }, config.SECRET, {
      expiresIn: 86400,
    });
  }

  res.set("Authorization", `Bearer ${token}`);
  res.json({ token: token });
};
```

La función `signIn` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el correo electrónico (`email`) y la contraseña (`password`) del usuario que desea iniciar sesión.

En primer lugar, se busca en la base de datos un usuario que coincida con el correo electrónico proporcionado utilizando el método `findOne()` de la clase `User`. Si no se encuentra ningún usuario, se devuelve una respuesta de error indicando que el usuario o la contraseña no son correctos.

A continuación, se compara la contraseña proporcionada con la contraseña almacenada en la base de datos utilizando el método `comparePassword()` de la clase `User`. Si las contraseñas no coinciden, se devuelve una respuesta de error indicando que el usuario o la contraseña no son correctos.

Luego, se genera un token de autenticación utilizando la biblioteca `jsonwebtoken`. El token se firma con el ID del usuario (`userFound._id`). Si el usuario ha seleccionado la opción de

"recordarme" al iniciar sesión, el token no tiene una fecha de vencimiento específica. De lo contrario, se configura para que expire en 24 horas.

El token se agrega en la cabecera de la respuesta utilizando el método `set()` y se envía como respuesta en formato JSON junto con un mensaje indicando que la autenticación se realizó correctamente.

En resumen, esta función de controlador se utiliza para autenticar a un usuario en la aplicación. Verifica el correo electrónico y la contraseña proporcionados, genera un token de autenticación y lo devuelve en la respuesta.

```
router.post("/token", AuthCtrl.getUserByToken);
```

Corresponde a este código:

```
export const getUserByToken = async (req, res) => {
  const { token } = req.body;
  try {
    if (token == null) {
      console.log("Se buggeo esto wey");
    } else {
      const decoded = jwt.verify(token, config.SECRET);
      const userFound = await User.findById(decoded.id);
      return res.status(200).json({ userFound });
    }
  } catch (error) {
    return res.status(404).json({ message: "Token can not be decoded" });
  }
};
```

La función `getUserByToken` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el token de autenticación (`token`) del usuario.

En primer lugar, se verifica si el token es nulo. Si es nulo, se imprime un mensaje de error en la consola. Esto indica que se produjo un error en el flujo de la aplicación.

Si el token no es nulo, se intenta decodificar el token utilizando la función `verify()` de la biblioteca `jsonwebtoken` y el secreto (`config.SECRET`) utilizado para firmar el token. Si el token no puede ser decodificado correctamente, se devuelve una respuesta de error indicando que el token no puede ser decodificado.

Si el token se decodifica correctamente, se obtiene el ID del usuario (`decoded.id`) del token decodificado y se utiliza para buscar el usuario correspondiente en la base de datos utilizando el método `findById()` de la clase `User`.

Si se encuentra el usuario, se devuelve una respuesta exitosa con el usuario encontrado en formato JSON. La respuesta incluye el objeto `userFound` que contiene la información del usuario.

En resumen, esta función de controlador se utiliza para obtener la información del usuario mediante el token de autenticación proporcionado. Decodifica el token, busca al usuario en la base de datos y devuelve la información del usuario en la respuesta.

/api/post

```
router.get("/", PostController.getAllPosts);
```

Corresponde a este código:

```
export const getAllPosts = async (req, res) => {
  const posts = await Post.find();

  res.status(200).json(posts);
};
```

La función `getAllPosts` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. No se requiere ningún dato adicional en la solicitud para esta función.

Dentro de la función, se utiliza el método `find()` de la clase `Post` para buscar todos los posts en la base de datos. Este método devuelve una lista de todos los posts encontrados.

Luego, se devuelve una respuesta exitosa con el estado HTTP 200 y los posts encontrados en formato JSON. La respuesta incluye el objeto `posts`, que contiene la lista de posts.

En resumen, esta función de controlador se utiliza para obtener todos los posts almacenados en la base de datos y devolverlos en la respuesta.

```
router.post("/create", PostController.createPost);
```

Corresponde con el siguiente código:

```
export const createPost = async (req, res) => {
  const { description, userInfo } = req.body;
  const NewPost = new Post({
    image_path: image,
    description: description,
    creator: userInfo,
    liked_by: [],
    comments: [],
  });
  image = "";

  const savedPost = await NewPost.save();

  return res.status(200);
};
```

La función `createPost` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. La solicitud debe contener la descripción del post (`description`) y la información del usuario (`userInfo`).

Dentro de la función, se crea una nueva instancia de la clase `Post` utilizando los datos proporcionados. Se establece el atributo `image_path` en una variable llamada `image`, pero en el código que proporcionaste, la variable `image` se inicializa después de la creación del post, por lo que su valor será `undefined`.

Luego, se guarda el nuevo post en la base de datos utilizando el método `save()` de la instancia de `Post`. El post guardado se almacena en la variable `savedPost`.

Finalmente, se devuelve una respuesta exitosa con el estado HTTP 200. Sin embargo, ten en cuenta que el código proporcionado no incluye el envío de ninguna respuesta de datos en formato JSON. Si deseas enviar algún dato en la respuesta, puedes modificar el código para incluirlo.

En resumen, esta función de controlador se utiliza para crear un nuevo post en la base de datos con la descripción y la información del usuario proporcionadas en la solicitud.

```
router.post("/image", fileUploadMiddleware, PostController.saveImage);
```

Corresponde con el siguiente código:

```
export const saveImage = async (req, res) => {
  image = "http://127.0.0.1:6001/assets/PostImages/" + req.file.filename;
  return res.status(200).json({ message: "Changes done succesfully" });
};
```

La función `saveImage` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga un archivo adjunto de imagen (`req.file`) que se haya subido al servidor.

Dentro de la función, se construye la URL de la imagen utilizando la ruta base "http://127.0.0.1:6001/assets/PostImages/" y el nombre del archivo de imagen (`req.file.filename`). Esta URL se asigna a la variable `image` que se utiliza en la función `createPost`.

Luego, se devuelve una respuesta exitosa con el estado HTTP 200 y un mensaje indicando que los cambios se realizaron correctamente. Sin embargo, ten en cuenta que el código proporcionado no incluye el envío de ningún dato adicional en la respuesta JSON. Si deseas enviar más información en la respuesta, puedes modificar el código para incluirla.

En resumen, esta función de controlador se utiliza para guardar la URL de una imagen asociada a un post. La URL se construye utilizando la ruta base y el nombre del archivo de imagen proporcionados en la solicitud.

```
router.post("/byToken", PostController.getAllPostsByToken);
```

Corresponde con el siguiente código:

```
export const getAllPostsByToken = async (req, res) => {
  const { token } = req.body;
  console.log(token);
  const decoded = jwt.verify(token, config.SECRET);
  const tokenid = decoded.id;

  const posts = await Post.find({ creator: tokenid });
  res.status(200).json(posts);
};
```

La función `getAllPostsByToken` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga un token de autenticación en el campo `token` del cuerpo.

Dentro de la función, se verifica y decodifica el token utilizando el secreto configurado (`config.SECRET`) para obtener el ID de usuario asociado al token. Luego, se realiza una consulta a la base de datos utilizando el ID de usuario para encontrar todos los posts creados por ese usuario.

Los posts encontrados se almacenan en la variable `posts`, y luego se devuelve una respuesta con el estado HTTP 200 y los posts encontrados en formato JSON.

Ten en cuenta que en el código proporcionado, se imprime el token en la consola utilizando `console.log(token)`, lo cual puede ser útil para fines de depuración, pero puede ser eliminado en producción.

En resumen, esta función de controlador se utiliza para obtener todos los posts creados por un usuario específico, utilizando el token de autenticación para identificar al usuario.

```
router.put("/", PostController.liked);
```

Corresponde con el siguiente código:

```
export const liked = async (req, res) => {
  const { postId, token } = req.body;

  try {
    const decoded = jwt.verify(token, config.SECRET);
    const tokenid = decoded.id;

    const post = await Post.findById(postId);
    console.log(post);

    if (post.liked_by.includes(tokenid)) {
      await Post.findByIdAndUpdate(postId, { $pull: { liked_by: tokenid } });
    } else {
      await Post.findByIdAndUpdate(postId, {
        $addToSet: { liked_by: tokenid },
      });
    }

    const updatedPost = await Post.findById(postId);

    return res.status(200).json(updatedPost);
  } catch (error) {
    console.error(error);
    return res.json(error);
  }
};
```

La función `liked` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID del post en el campo `postId` y el token de autenticación en el campo `token` del cuerpo.

Dentro de la función, se verifica y decodifica el token utilizando el secreto configurado (`config.SECRET`) para obtener el ID de usuario asociado al token. Luego, se realiza una consulta a la base de datos para encontrar el post con el ID proporcionado.

Si el post contiene el ID de usuario en el campo `liked_by`, significa que el usuario ya ha dado "me gusta" al post. En este caso, se utiliza el método `findByIdAndUpdate` para eliminar el ID de usuario del campo `liked_by` del post, utilizando el operador `$pull`.

Si el post no contiene el ID de usuario en el campo `liked_by`, significa que el usuario aún no ha dado "me gusta" al post. En este caso, se utiliza el método `findByIdAndUpdate` para agregar el ID de usuario al campo `liked_by` del post, utilizando el operador `$addToSet`. El operador `$addToSet` asegura que el ID de usuario no se agregue nuevamente si ya está presente en la matriz `liked_by`.

Después de actualizar el post, se realiza otra consulta para obtener el post actualizado y se devuelve una respuesta con el estado HTTP 200 y el post actualizado en formato JSON.

Si ocurre algún error durante el proceso, se captura el error y se devuelve una respuesta con el error en formato JSON.

Ten en cuenta que en el código proporcionado, se imprime el post en la consola utilizando `console.log(post)`, lo cual puede ser útil para fines de depuración, pero puede ser eliminado en producción.

En resumen, esta función de controlador se utiliza para gestionar la acción de "me gusta" en un post. Si un usuario da "me gusta" a un post, su ID se agrega al campo `liked_by` del post. Si el usuario ya ha dado "me gusta", su ID se elimina del campo `liked_by`.

API/user

```
router.post("/avatar", fileUploaded, UserCtrl.updateAvatarImage);
```

Corresponde con este código:

```
export const updateAvatarImage = async (req, res) => {
  const { userId } = req.body;
  const decoded = jwt.verify(userId, config.SECRET);
  const tokenid = decoded.id;
  const filename = "http://127.0.0.1:6001/assets/Avatar/" + req.file.filename;
  const userFound = await User.findByIdAndUpdate(tokenid, {
    image: filename,
  });
  return res.json({ message: "Changes done succesfully" });
};
```

La función `updateAvatarImage` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID de usuario en el campo `userId` y la imagen de avatar en el campo `file` del cuerpo.

Dentro de la función, se decodifica el ID de usuario utilizando el secreto configurado (`config.SECRET`) para obtener el ID de usuario asociado al token. Luego, se construye la URL de la imagen de avatar utilizando el nombre de archivo proporcionado en la solicitud.

A continuación, se utiliza el método `findByIdAndUpdate` para buscar y actualizar el usuario con el ID de usuario obtenido. Se actualiza el campo `image` del usuario con la URL de la imagen de avatar.

Después de actualizar el usuario, se devuelve una respuesta con el estado HTTP 200 y un mensaje indicando que los cambios se realizaron correctamente en formato JSON.

Ten en cuenta que en el código proporcionado, se asume que la imagen de avatar se carga correctamente y se guarda en una ubicación accesible a través de una URL. Asegúrate de que la configuración del servidor y las rutas de almacenamiento de archivos sean correctas para que esto funcione correctamente en tu aplicación.

En resumen, esta función de controlador se utiliza para actualizar la imagen de avatar de un usuario en la base de datos.

```
router.post("/edit", UserCtrl.updateUserInfo);
```

Corresponde a este código:

```
export const updateUserInfo = async (req, res) => {
  const { userId, fullname, address, job, studies, description } = req.body;
  const decoded = jwt.verify(userId, config.SECRET);
  const tokenid = decoded.id;
  const userFound = await User.findByIdAndUpdate(tokenid, {
    fullname: fullname,
    address: address,
    job: job,
    studies: studies,
    description: description,
  });
  return res.json({ message: "Changes done succesfully" });
};
```

La función `updateUserInfo` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID de usuario en el campo `userId` y los campos de información de usuario (`fullname`, `address`, `job`, `studies` y `description`) en el cuerpo.

Dentro de la función, se decodifica el ID de usuario utilizando el secreto configurado (`config.SECRET`) para obtener el ID de usuario asociado al token.

A continuación, se utiliza el método `findByIdAndUpdate` para buscar y actualizar el usuario con el ID de usuario obtenido. Se actualizan los campos de información del usuario con los valores proporcionados en la solicitud.

Después de actualizar el usuario, se devuelve una respuesta con el estado HTTP 200 y un mensaje indicando que los cambios se realizaron correctamente en formato JSON.

En resumen, esta función de controlador se utiliza para actualizar la información de un usuario en la base de datos, como el nombre completo, la dirección, el trabajo, los estudios y la descripción.

```
router.get("/getInfo/:userId", UserCtrl.getUserInfo);
```

Que corresponde a este código:

```
export const getUserInfo = async (req, res) => {
  const { userId } = req.params;

  try {
```

```
    const userInfo = await User.findById(userId);

    return res.status(200).json(userInfo);
  } catch (error) {
    return res.json(error);
  }
};
```

La función `getUserInfo` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el ID de usuario en los parámetros de la URL (`userId`).

Dentro de la función, se utiliza el método `findById` para buscar el usuario en la base de datos utilizando el ID de usuario proporcionado.

Si se encuentra el usuario, se devuelve una respuesta con el estado HTTP 200 y la información del usuario en formato JSON.

Si ocurre algún error durante la búsqueda del usuario, se devuelve una respuesta con el estado HTTP 404 y el mensaje de error en formato JSON.

En resumen, esta función de controlador se utiliza para obtener la información de un usuario en la base de datos utilizando su ID. La información del usuario se devuelve como respuesta en formato JSON.

```
router.get("/getInfoByUsername/:username", UserCtrl.getUserInfoByUserName);
```

Que corresponde a este código:

```
export const getUserInfoByUserName = async (req, res) => {
  const { username } = req.params;

  try {
    const userInfo = await User.findOne({ username });

    return res.status(200).json(userInfo);
  } catch (error) {
    return res.json(error);
  }
};
```

La función `getUserInfoByUserName` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el nombre de usuario en los parámetros de la URL (`username`).

Dentro de la función, se utiliza el método `findOne` para buscar el usuario en la base de datos utilizando el nombre de usuario proporcionado.

Si se encuentra el usuario, se devuelve una respuesta con el estado HTTP 200 y la información del usuario en formato JSON.

Si ocurre algún error durante la búsqueda del usuario, se devuelve una respuesta con el estado HTTP 404 y el mensaje de error en formato JSON.

En resumen, esta función de controlador se utiliza para obtener la información de un usuario en la base de datos utilizando su nombre de usuario. La información del usuario se devuelve como respuesta en formato JSON.

```
router.get("/changeInfo", UserCtrl.updateUserInfo);
```

Que corresponde a este código:

```
export const updateUserInfo = async (req, res) => {
  const { userId, fullname, address, job, studies, description } = req.body;
  const decoded = jwt.verify(userId, config.SECRET);
  const tokenid = decoded.id;
  const userFound = await User.findByIdAndUpdate(tokenid, {
    fullname: fullname,
    address: address,
    job: job,
    studies: studies,
    description: description,
  });
  return res.json({ message: "Changes done succesfully" });
};
```

La función `updateUserInfo` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga los siguientes datos en el cuerpo (`body`):

- `userId`: el ID del usuario obtenido del token.
- `fullname`: el nombre completo actualizado del usuario.
- `address`: la dirección actualizada del usuario.
- `job`: el trabajo actualizado del usuario.
- `studies`: los estudios actualizados del usuario.
- `description`: la descripción actualizada del usuario.

Dentro de la función, se utiliza el método `jwt.verify` para decodificar el `userId` y obtener el ID del usuario. Luego se utiliza el método `findByIdAndUpdate` para buscar y actualizar el usuario en la base de datos utilizando el ID obtenido. Se actualizan los campos de `fullname`, `address`, `job`, `studies` y `description` con los valores proporcionados en la solicitud.

Si la actualización se realiza correctamente, se devuelve una respuesta con un estado HTTP 200 y un mensaje en formato JSON indicando que los cambios se realizaron correctamente.

En resumen, esta función de controlador se utiliza para actualizar la información de un usuario en la base de datos. Se actualizan los campos específicos proporcionados en la solicitud y se devuelve una respuesta indicando el éxito de la operación.

```
router.post("/addShopping", UserCtrl.addToShoppingList);
```

Corresponde con este código:

```
export const addToShoppingList = async (req, res) => {
  const { recipeId, token, list } = req.body;

  try {
    const decoded = jwt.verify(token, config.SECRET);
    const tokenid = decoded.id;

    const shoppingList = await User.findById(tokenid);
    shoppingList.list.push(...list);
    await shoppingList.save();

    return res.status(200).json(shoppingList);
  } catch (error) {
    return res.status(404).json({ message: "User not found" });
  }
};
```

La función `addToShoppingList` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga los siguientes datos en el cuerpo (`body`):

- `recipeId`: el ID de la receta para la cual se agregarán elementos a la lista de compras.
- `token`: el token de autenticación del usuario.
- `list`: una lista de elementos que se agregarán a la lista de compras.

Dentro de la función, se utiliza el método `jwt.verify` para decodificar el token y obtener el ID del usuario. Luego se utiliza el método `findById` para buscar al usuario en la base de datos utilizando el ID obtenido.

Se accede a la propiedad `list` del usuario y se agregan los elementos de la lista proporcionada en la solicitud utilizando el operador spread (`...`). Luego se guarda el usuario actualizado en la base de datos utilizando el método `save`.

Si la operación se realiza correctamente, se devuelve una respuesta con un estado HTTP 200 y la lista de compras actualizada del usuario en formato JSON.

En caso de que no se encuentre el usuario en la base de datos, se devuelve una respuesta con un estado HTTP 404 y un mensaje indicando que el usuario no se encontró.

En resumen, esta función de controlador se utiliza para agregar elementos a la lista de compras de un usuario en la base de datos. Los elementos se agregan a la lista existente del usuario y se guarda la lista actualizada en la base de datos.

```
router.get("/ShoppingList", UserCtrl.getShoppingList);
```

Corresponde con este código:

```
export const getShoppingList = async (req, res) => {
  const { token } = req.body;
  console.log(token);

  try {
```

```
    const decoded = jwt.verify(token, config.SECRET);
    const tokenid = decoded.id;

    const shoppingList = await User.findById(tokenid);

    return res.status(200).json(shoppingList.list);
  } catch (error) {
    return res.status(403).json({ message: "User not found" });
  }
};
```

La función `getShoppingList` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros. Se espera que la solicitud contenga el token de autenticación del usuario en el cuerpo (`body`).

Dentro de la función, se utiliza el método `jwt.verify` para decodificar el token y obtener el ID del usuario. Luego se utiliza el método `findById` para buscar al usuario en la base de datos utilizando el ID obtenido.

Si se encuentra el usuario, se accede a la propiedad `list` del usuario, que contiene la lista de compras. Se devuelve una respuesta con un estado HTTP 200 y la lista de compras en formato JSON.

En caso de que no se encuentre el usuario en la base de datos, se devuelve una respuesta con un estado HTTP 403 (prohibido) y un mensaje indicando que el usuario no se encontró.

En resumen, esta función de controlador se utiliza para obtener la lista de compras de un usuario desde la base de datos. La lista se devuelve como parte de la respuesta en formato JSON.

/api/stripe

```
router.post("/payment", StripeCtrl.payments);
```

Corresponde con este código:

```
export const payments = async (req, res) => {
  const paymentIntent = await stripe.paymentIntents.create({
    amount: 4995,
    currency: "eur",
    automatic_payment_methods: {
      enabled: true,
    },
  });

  res.send({
    clientSecret: paymentIntent.client_secret,
  });
};
```

La función `payments` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros.

Dentro de la función, se utiliza el método `stripe.paymentIntents.create` para crear un nuevo intento de pago. Se proporcionan los siguientes parámetros al método `create`:

- `amount`: el monto del pago en centavos. En este caso, se establece en 4995, lo que representa 49.95 euros.
- `currency`: la moneda en la que se realiza el pago. En este caso, se establece en "eur" para euros.
- `automatic_payment_methods`: un objeto que especifica si se permiten los métodos de pago automáticos. En este caso, se habilita con `{ enabled: true }`.

Después de crear el intento de pago, se obtiene el atributo `client_secret` del objeto `paymentIntent` devuelto por la API de Stripe.

Finalmente, se envía una respuesta al cliente con el `client_secret` como parte de la respuesta en formato JSON.

En resumen, esta función de controlador se utiliza para crear un nuevo intento de pago utilizando la API de Stripe y devolver el `client_secret` necesario para finalizar el proceso de pago en el lado del cliente.

/api/postExercices

```
router.post("/byId", PostExreciceCtrl.getExercicesById);
```

Corresponde al código:

```
export const getExercicesById = async (req, res) => {
  try {
    const { id } = req.body;
    const exercice = await PostExercices.findById(id);
    console.log(exercice);

    return res.status(200).json(exercice);
  } catch {
    return res.status(404).json({ message: "Exercicie not found" });
  }
};
```

La función `getExercicesById` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros.

Dentro de la función, se intenta obtener el ID del ejercicio de la solicitud utilizando `req.body.id`. Luego, se utiliza el método `PostExercices.findById` para buscar el ejercicio por su ID en la base de datos.

Si se encuentra el ejercicio, se devuelve una respuesta al cliente con el estado 200 y el ejercicio encontrado en formato JSON.

Si no se encuentra el ejercicio, se devuelve una respuesta al cliente con el estado 404 y un mensaje indicando que el ejercicio no se encontró.

En resumen, esta función de controlador se utiliza para obtener un ejercicio específico por su ID y devolverlo al cliente.

```
router.post("/byToken", PostExreciceCtrl.getExercicesByToken);
```

Corresponde con este código:

```
export const getExercicesByToken = async (req, res) => {
  const { token } = req.body;
  const decoded = jwt.verify(token, config.SECRET);
  const tokenid = decoded.id;
  const exercices = await PostExercices.find({ creator: tokenid });

  return res.status(200).json(exercices);
};
```

La función `getExercicesByToken` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros.

Dentro de la función, se extrae el token de la solicitud utilizando `req.body.token`. Luego, se decodifica el token utilizando `jwt.verify` y se obtiene el ID del usuario (`tokenid`) desde el token decodificado.

A continuación, se utiliza el método `PostExercices.find` para buscar todos los ejercicios creados por el usuario con el ID obtenido (`tokenid`).

Si se encuentran ejercicios, se devuelve una respuesta al cliente con el estado 200 y la lista de ejercicios encontrados en formato JSON.

Si no se encuentran ejercicios, se devuelve una respuesta al cliente con el estado 200 y una lista vacía.

En resumen, esta función de controlador se utiliza para obtener una lista de ejercicios creados por un usuario específico identificado por su token y devolverla al cliente.

```
router.get("/", PostExreciceCtrl.getExercices);
```

Corresponde con este código:

```
export const getExercices = async (req, res) => {
  const exercices = await PostExercices.find();

  return res.status(200).json(exercices);
};
```

La función `getExercices` es una función asíncrona que recibe una solicitud (`req`) y una respuesta (`res`) como parámetros.

Dentro de la función, se utiliza el método `PostExercices.find` para buscar todos los ejercicios en la base de datos.

Se espera a que la búsqueda se complete y se obtiene la lista de ejercicios encontrados.

A continuación, se devuelve una respuesta al cliente con el estado 200 y la lista de ejercicios encontrados en formato JSON.

En resumen, esta función de controlador se utiliza para obtener una lista de todos los ejercicios disponibles y devolverla al cliente.

Bibliografía usada

Moment.js

Stripe

ExpressJS

React

MongoDB Atlas

MongoDB