Refactorizaciones Chicken Docker

Sebastián Ignacio Vega Varela
Cristóbal Alonso González Cifuentes
Braian Alejandro Urra Bastías
Matías Felipe Jener Valdebenito Valenzuela

1 Refactorización: Archivo crud_workout

El archivo `crud_workout` posee muchas funciones, encargadas de realizar las operaciones propias de CRUD, como por ejemplo crear *workouts*, *añadir ejercicios*, *añadir sets*, obtener información del workout actual, *eliminar workouts*, entre otros. Sin embargo algunas de estas funciones eran muy verbosas y no cumplian a cabalidad los principios SOLID. Tal es el caso de las funciones `create_from_template` y `delete_workout` que son las que se refactorizaron en este archivo.

1.1 Método create_from_template

Contexto y Motivo de las Refactorizaciones

`create_from_template` es una función que crea *workouts*, es decir, crea sesiones de entrenamiento con *ejercicios* y sus respectivos *sets*. Esta función cumplía correctamente su propósito, sin embargo, ella sola se encargaba de crear el workout, luego de asignar cada ejercicio a el workout, y para cada ejercicio añadir sus sets con la cantidad de repeticiones, peso, etc.

Solución Aplicada

Es por esto que decidimos separar la responsabilidad de `create_from_template` para hacer el codigo mas modular, ahora esta función simplemente llama a otras tres funciones:

- `create_workout`: crea el workout
- `add_exercises_from_template`: asigna los ejercicios del workout
- `add_sets_for_exercises`: asigna sets a cada ejercicio

Beneficios de la refactorización de create_from_template:

1. **Mejora la legibilidad y el mantenimiento del código**: Antes, la función concentraba toda la lógica de creación del entrenamiento, ejercicios y sets dentro de un único bloque de código extenso. Esto dificultaba entender rápidamente qué hacía cada parte.
2. **Redujo la complejidad ciclomática**: Al separar responsabilidades, cada método tiene menos ramificaciones lógicas (bucles, condiciones, etc.), por lo que se simplifica su análisis, depuración y cobertura en tests unitarios.
3. **Aumentó la reutilización del código**: Las funciones creadas pueden reutilizarse en otros contextos. Por ejemplo, create_workout puede usarse para crear entrenamientos sin plantilla, o add_sets_for_exercises puede reutilizarse para añadir sets personalizados en futuras funcionalidades.
4. **Facilitó las pruebas unitarias**: Cada funcionalidad puede probarse de forma aislada, sin necesidad de recrear todo el flujo completo.

1.2 Método delete_workout

Contexto y Motivo de las Refactorizaciones

La función `delete_workout` es la encargada de eliminar un *workout* junto con todos los ejercicios y series (*sets*) asociados. Anteriormente la función realizaba todo el proceso dentro de un único método: primero eliminaba los *sets*, luego los ejercicios y finalmente el propio *workout*. Esta estructura hacía que la función fuese demasiado extensa y concentrara múltiples responsabilidades, dificultando su comprensión y mantenimiento.

Solución Aplicada

Para mejorar su organización, se aplicó una refactorización que divide el proceso de eliminación en tres funciones más específicas:

- `delete_sets_from_workout`: elimina todos los *sets* asociados a un ejercicio.

- `delete_exercises_from_workout`: elimina los ejercicios asociados a un *workout*, invocando internamente a la función anterior para eliminar sus *sets*.
- `delete_workout`: ahora simplemente coordina la eliminación completa del *workout* llamando a las funciones anteriores.

Beneficios de la refactorización de `delete_workout`:

1. **Mejor organización del código:** Separar las operaciones de eliminación en funciones individuales permitió que cada una tenga un propósito claro, reduciendo la cantidad de lógica dentro de `delete_workout`.
2. **Facilidad para realizar pruebas unitarias:** Ahora es posible testear de forma independiente la eliminación de *sets*, ejercicios y *workouts*, verificando que cada paso se ejecute correctamente sin necesidad de probar todo el flujo completo.
3. **Mayor reutilización del código:** Las funciones `delete_sets_from_workout` y `delete_exercises_from_workout` pueden reutilizarse en otros contextos donde sea necesario eliminar parte del entrenamiento sin borrarlo por completo (por ejemplo, al editar un *workout*).
4. **Reducción de la complejidad ciclomática:** El flujo de eliminación es ahora más simple y lineal, lo que facilita su comprensión, depuración y modificación futura.
5. **Cumplimiento del principio de responsabilidad única (SRP):** Cada método realiza una única tarea concreta, mejorando la modularidad del código y alineándose con las buenas prácticas de diseño.

2 Refactorización de tipos y anotaciones en el backend

Contexto y Motivo de las Refactorizaciones

Durante la revisión del backend se detectó un uso excesivo de `Any` y la falta de anotaciones de tipo en múltiples funciones y endpoints. Esto generaba código menos seguro y complicaba el análisis estático con herramientas como `Ruff` o `mypy`.

Solución Aplicada

Para mejorar la calidad y mantenibilidad del código, se realizaron las siguientes refactorizaciones:

- Se reemplazó `Any` en parámetros y valores de retorno por tipos concretos cuando correspondía, reflejando explícitamente lo que cada función recibe y devuelve.
- Se añadieron anotaciones de tipo faltantes en funciones de `crud`, servicios y endpoints de FastAPI, incluyendo:
  - Parámetros de funciones que antes no estaban tipados.
  - Retornos de funciones que antes estaban declarados como `Any` o no tenían tipo.
- Se tiparon correctamente listas, diccionarios, objetos de base de datos y modelos de Pydantic, eliminando ambigüedades en la manipulación de datos.

Antes de la refactorización:

```
@router.get("/users", response_model=List[UserSchema])
def read_users(
...
) -> Any:
users = user.get_multi(db, skip=skip, limit=limit)
return users
```

Listing 1: Código antes de la refactorización

Después de la refactorización:

```
@router.get("/users", response_model=List[UserSchema])
def read_users(
...
) -> List[UserSchema]:
users = user.get_multi(db, skip=skip, limit=limit)
return users
```

Listing 2: Código después de la refactorización

Beneficios de la refactorización de tipos:

1. **Claridad y autocompletado:** Los tipos explícitos permiten a los desarrolladores comprender rápidamente qué se espera de cada función, mejorando la productividad y la documentación implícita.
2. **Seguridad y robustez:** Se reducen errores en tiempo de ejecución por paso de datos incorrectos, al hacer visibles los contratos de tipos entre funciones.
3. **Facilidad para pruebas unitarias:** Con tipos claros, es más sencillo diseñar pruebas que validen entradas y salidas de forma precisa.
4. **Mantenibilidad y evolución del código:** Las anotaciones de tipo facilitan refactorizaciones futuras y garantizan coherencia en todo el backend.

3 Refactorización: modulos en el backend

Contexto y Motivo de la Refactorización

Durante la revisión del backend, se inició un proceso de refactorización con el objetivo principal de mejorar la mantenibilidad, legibilidad y consistencia del código para facilitar futuras implementaciones y reducir la probabilidad de errores. Mediante el uso de `pylint` como herramienta de análisis, se detectaron varios problemas clave que motivaron estos cambios, entre los que destacaban la falta de documentación (*docstrings*) en múltiples módulos y funciones, la excesiva longitud o complejidad de ciertas líneas de código y advertencias sobre variables públicas derivadas de los modelos de base de datos. Para complementar estas mejoras, se planea utilizar `Black` para formalizar y estandarizar el formato del código en todo el proyecto.

Soluciones Aplicada

- **Documentación de módulos y funciones**: Se añadieron docstrings de módulo y metodos en la mayoria del apartado app.
- **Refactorización de funciones largas**: Se reordenaron bloques de código, se estandarizó la estructura de funciones y se redujo la complejidad percibida por `pylint`.
- **Omisión de ciertas advertencias sobre variables públicas**: Se decidió no evaluar estrictamente las advertencias relacionadas con variables públicas en la creación de tablas, ya que son necesarias para SQLAlchemy.

Ejemplos de Refactoring

Clase/Función: `create_user` en `app.api.endpoints.admin.py` **Problema:** Al crear un usuario, se realizaban comprobaciones separadas para verificar si el email o el username ya existían en el sistema. Estas comprobaciones estaban duplicadas y cada una lanzaba su propio error HTTP, lo que generaba código repetitivo y riesgo de inconsistencias si se cambiaba la lógica en un endpoint y no en otro.

Solución aplicada: Se unificó la lógica de validación de manera que primero se verifica la existencia del email y luego del username de forma directa, usando la misma estructura de excepción HTTP. Esto eliminó duplicaciones y estandarizó los mensajes de error.

Clase/Función: `add_exercise_to_template` en `app.api.endpoints.admin.py`

Problema: Al agregar un ejercicio a una plantilla, la función verificaba la existencia de la plantilla y del ejercicio de manera dispersa y repetitiva en varios puntos del código, generando duplicación y aumentando la complejidad de mantenimiento.

Solución aplicada: Se centralizó la validación: primero se comprueba la existencia de la plantilla, luego se procesan los datos del ejercicio, y finalmente se llama al método CRUD para agregarlo. Esto eliminó la duplicación y dejó un flujo de validación consistente y fácil de seguir.

Beneficios

Gracias a estas acciones, se logró:

- Documentar todos los módulos y funciones críticas, eliminando la mayoría de advertencias de docstrings.
- Mejorar la legibilidad y mantenibilidad de funciones y endpoints, reduciendo la complejidad percibida.
- Mantener el funcionamiento completo de la API sin alterar la lógica de negocio.

4 Refactorización: Eliminación de Parámetros no Utilizados en Bloques `catch` en el frontend

4.1 Contexto y Motivo de la Refactorización

Durante el análisis del código con la herramienta **ESLint**, se detectó la presencia de múltiples bloques `try/catch` en los que la variable declarada en la cláusula `catch` no era utilizada dentro del bloque correspondiente. Esta práctica generaba advertencias del tipo `@typescript-eslint/no-unused-vars`, lo que indicaba la existencia de parámetros innecesarios en el manejo de excepciones.

El problema principal radicaba en que estos parámetros, al no tener una función práctica, introducían **ruido visual** y reducían la claridad del código. Además, la presencia de variables no utilizadas es una mala práctica que puede confundir a otros desarrolladores, haciendo pensar que la variable tiene un propósito cuando realmente no lo tiene.

4.2 Solución Aplicada

Para corregir esta situación, se aplicó una refactorización simple pero significativa: la **eliminación del parámetro innecesario** en los bloques `catch`. De esta manera, se pasó de una estructura redundante a una más limpia y concisa, como se muestra a continuación:

```
try {
} catch (error) {
  console.error('Ocurrio un error durante la operacion');
}
```

Listing 3: Código antes de la refactorización

```
try {
} catch {
  console.error('Ocurrio un error durante la operacion');
}
```

Listing 4: Código después de la refactorización

Esta modificación fue aplicada de forma consistente en todos los módulos donde se detectaron advertencias de ESLint relacionadas con variables no utilizadas.

4.3 Ventajas de la Solución

- **Claridad y simplicidad:** El código resultante es más legible y directo, eliminando elementos innecesarios que no aportan funcionalidad.
- **Cumplimiento de buenas prácticas:** La corrección se alinea con las recomendaciones de ESLint y los estándares de TypeScript para evitar parámetros sin uso.
- **Reducción de advertencias:** Se eliminaron las alertas generadas por la regla `no-unused-vars`, contribuyendo a un entorno de desarrollo más limpio.
- **Mantenibilidad mejorada:** Los bloques de manejo de errores ahora reflejan con mayor precisión la intención del desarrollador, facilitando futuras modificaciones.

En resumen, esta refactorización, aunque sencilla, tuvo un impacto positivo en la calidad del código, mejorando su legibilidad, reduciendo advertencias innecesarias y fortaleciendo la disciplina de desarrollo dentro del proyecto.

5 Refactorización: Reemplazo del Tipo `any` en frontend

5.1 Contexto y Motivo de la Refactorización

Durante la revisión del código con la herramienta **ESLint**, se detectó el uso extendido del tipo `any` en diferentes secciones del proyecto **TypeScript**. Este tipo, aunque permite gran flexibilidad, desactiva el sistema de tipos estáticos, lo que puede ocultar errores de asignación o uso indebido de valores durante la compilación. El uso de `any` fue especialmente común en dos contextos principales:

- En la definición de estructuras de datos dinámicas y objetos de actualización.
- En bloques `catch` y funciones de manejo de errores, donde el tipo del valor capturado no estaba claramente especificado.

Ambos casos generaban advertencias del tipo `@typescript-eslint/no-explicit-any`, indicando la necesidad de reemplazar el uso de `any` por alternativas más seguras y expresivas.

5.2 Solución Aplicada

La refactorización consistió en reemplazar el tipo `any` por el tipo `unknown` o por tipos definidos explícitamente, dependiendo del contexto de uso. Esta mejora permitió conservar flexibilidad, pero garantizando que las operaciones sobre los datos fueran seguras y verificadas.

5.2.1 Caso 1: Estructuras de datos con tipo definido

En las estructuras donde la forma del objeto era conocida, se reemplazó el tipo genérico `any` por un tipo personalizado. Por ejemplo, antes de la refactorización se utilizaba:

```
1 const updates: any = {};
2 let hasChanges = false;
```

Listing 5: Código antes de la refactorización

Posteriormente, se definió un tipo `structUpdates` que describe explícitamente las propiedades del objeto:

```
1  type structUpdates = {
2    weight?: number;
3    reps?: number;
4    duration?: number;
5    rest_duration?: number;
6    notes?: string;
7  };
8
9  const updates: structUpdates = {};
10 let hasChanges = false;
```

Listing 6: Código después de la refactorización

Esta modificación eliminó la ambigüedad del tipo, facilitó la lectura del código y permitió al compilador detectar posibles errores de asignación.

5.2.2 Caso 2: Manejo de errores con tipo `unknown`

En las funciones y bloques `catch`, donde el tipo del valor capturado no podía determinarse con certeza, se reemplazó `any` por `unknown`. Esto obligó a realizar comprobaciones de tipo antes de acceder a las propiedades del error, garantizando mayor seguridad en la ejecución.

Antes de la refactorización:

```
try {
  // ...
} catch (err: any) {
  console.error(err.message);
}
```

Listing 7: Código antes de la refactorización en bloques `catch`

Después de la refactorización:

```
try {
  // ...
} catch (err: unknown) {
  if (err instanceof Error) {
    console.error(err.message);
  }
}
```

Listing 8: Código después de la refactorización

Este cambio hizo explícita la necesidad de verificar el tipo del valor capturado, evitando errores en tiempo de ejecución y reforzando la integridad del sistema de manejo de excepciones.

—

5.3 Ventajas de la Solución

- **Mayor seguridad de tipos:** El uso de `unknown` y de tipos definidos elimina operaciones inseguras y previene errores de conversión.
- **Cumplimiento de buenas prácticas:** Se eliminaron las advertencias de ESLint (`@typescript-eslint/no-explicit-` alineando el código con las normas recomendadas por TypeScript.
- **Mejor legibilidad y mantenimiento:** Los tipos explícitos clarifican la estructura de los objetos y las expectativas de cada variable, facilitando la comprensión por parte de nuevos desarrolladores.
- **Robustez en el manejo de errores:** La obligación de validar el tipo de las excepciones reduce la posibilidad de errores silenciosos en tiempo de ejecución.

En conjunto, esta refactorización fortaleció la base tipada del proyecto, mejorando la calidad, consistencia y mantenibilidad del código TypeScript.