IES Virgen del Carmen

JUNTA DE ANDALUCIA
CONSEJERÍA DE EDUCACIÓN

Departamento de Informática

Proyecto(GYM10)

Clase DAM 2A

Eduardo Diaz Soria

Diciembre 2025

UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

INFORMÁTICA - IES Virgen del Carmen - JAÉN

1. Introducción

1.1. Datos del Proyecto

Nombre: Eduardo
Apellidos: Diaz Soria
Título: GYM10

Ciclo: Desarrollo de Aplicaciones Multiplataforma
Año: 2025-2026
Centro Educativo: IES Virgen del Carmen

1.2. Planificación

Esta sería aproximadamente la planificación del proyecto:

- **Semana del 27 al 31 de octubre:**
  - Planteamiento de la idea del proyecto
  - Introduccion a Typescript
  - Análisis de requisitos y diseño de la arquitectura de microservicios
  - Configuración del entorno de desarrollo (Docker, MongoDB)
- **Semana del 3 al 7 de noviembre:**
  - Implementación del microservicio de autenticación (Auth Service)
  - Desarrollo del frontend básico con React (Login, Registro, Dashboard)
  - Integración frontend-backend para gestión de clases
- **Semana del 17 al 21 de noviembre:**
  - Implementación del microservicio de clases y reservas (Class Service)
  - Desarrollo del microservicio de pagos y suscripciones (Payment Service)
  - Generación de códigos QR de acceso
- **Semana del 24 de noviembre al 5 de diciembre:**
  - Desarrollo del microservicio de rutinas y ejercicios (Routine Service)
  - Implementación del sistema de feedback y notificaciones (Feedback Service)
  - Implementación de rutinas personalizadas para usuarios
  - Integración con YouTube para videos de ejercicios
  - Refactorización y optimización del código
  - Eliminación de código duplicado y mejoras de rendimiento

- Pruebas y corrección de errores

- **Semana del 8 al 12 de diciembre:**
  - Documentación del proyecto
  - Preparación de la presentación
  - Revisión final y entrega del proyecto

2. Tecnologías Utilizadas

2.1. Backend

Node.js: Entorno de ejecución JavaScript del lado del servidor, elegido por su alto rendimiento y ecosistema de paquetes npm.

Express.js: Framework minimalista para Node.js que facilita la creación de APIs RESTful y el manejo de rutas y middleware.

TypeScript: Superset de JavaScript con tipado estático que permite detectar errores en tiempo de compilación y mejorar la mantenibilidad del código.

MongoDB: Base de datos NoSQL orientada a documentos, ideal para arquitecturas de microservicios por su flexibilidad y escalabilidad horizontal.

Mongoose: ODM para MongoDB que proporciona modelado de esquemas, validación y métodos de consulta tipados.

JWT (JSON Web Tokens): Sistema de autenticación sin estado que permite validar usuarios entre microservicios sin consultas a base de datos.

Docker: Plataforma de contenedorización utilizada para ejecutar MongoDB de forma consistente y aislada.

bcryptjs: Biblioteca para hashear contraseñas de forma segura antes de almacenarlas en la base de datos.

qrcode: Librería para generar códigos QR únicos con información de autenticación para acceso al gimnasio.

Helmet: Middleware de Express que establece headers HTTP de seguridad para proteger la aplicación.

CORS: Configuración de Cross-Origin Resource Sharing para permitir peticiones desde el frontend.

2.2. Frontend

React 18: Biblioteca de JavaScript para construir interfaces de usuario mediante componentes reutilizables y gestión eficiente del estado.

TypeScript: Mismo lenguaje que en backend para mantener consistencia en todo el stack y aprovechar el tipado estático.

Vite: Build tool moderno que proporciona desarrollo rápido con Hot Module Replacement y builds optimizados para producción.

React Router DOM: Librería para enrutamiento declarativo y protección de rutas según roles de usuario.

Axios: Cliente HTTP para comunicación con las APIs de los microservicios, con interceptores para tokens de autenticación.

react-qr-code: Componente React para renderizar códigos QR en el frontend de forma sencilla.

2.3. Herramientas de Desarrollo

Git: Sistema de control de versiones para gestionar el código fuente y colaborar en el desarrollo.

Docker Compose: Herramienta para definir y ejecutar aplicaciones Docker multi-contenedor, utilizada para MongoDB.

Nodemon: Herramienta que reinicia automáticamente el servidor Node.js cuando se detectan cambios, mejorando la productividad.

Draw.io: Herramienta de diagramación gráfica online utilizada para crear el diagrama de casos de uso de forma visual e intuitiva.

Mermaid: Lenguaje de marcado basado en texto para crear diagramas (clases, flujos, casos de uso) que se integra fácilmente en documentación Markdown.

CodeSnap: Extensión de VS Code que permite capturar código como imágenes de alta calidad para incluir en la presentación del proyecto.

2.4. Arquitectura

Microservicios: Arquitectura que separa el sistema en servicios independientes, cada uno con su propia base de datos, permitiendo escalabilidad y mantenibilidad.

REST API: Estilo arquitectónico para comunicación entre servicios mediante peticiones HTTP estándar (GET, POST, PUT, DELETE).

3. Análisis y Diagramas

3.1. Diagrama de Casos de Uso

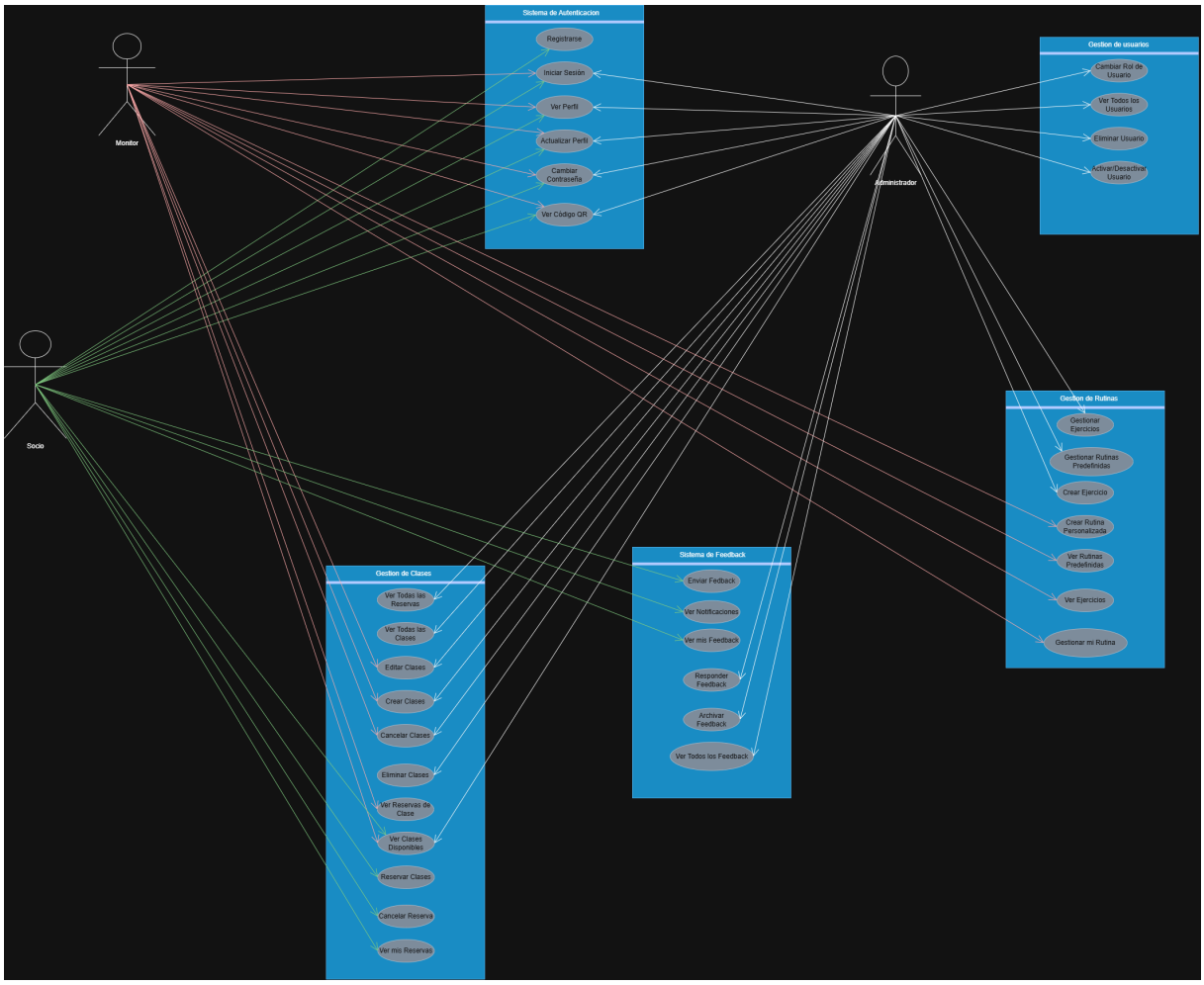

Figura 1: Diagrama de Casos de Uso

3.2. Diagrama de Clases

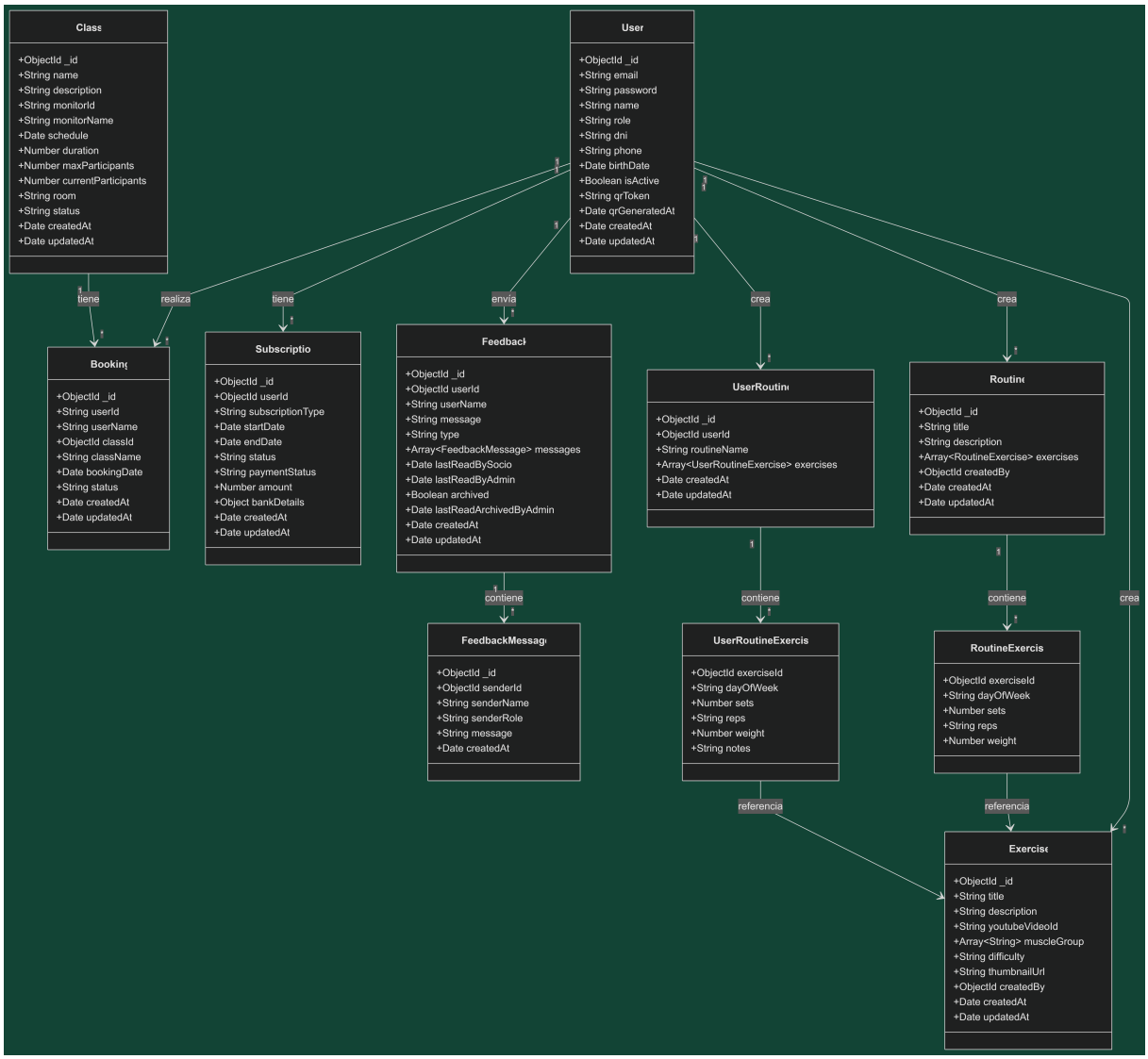

Figura 2: Diagrama de Clases

4. Implementación

Voy a detallar las partes más críticas del código del proyecto, explicando su funcionamiento y la importancia de cada componente en la arquitectura de microservicios.

4.1. 1. Middleware de Autenticación JWT (Auth Service)

El middleware de autenticación JWT es el componente central de seguridad del sistema. Se encuentra en el Auth Service y valida los tokens JWT que se generan durante el login.

```typescript
import { Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { AuthRequest, JWTPayload, UserRole } from '../types';

const JWT_SECRET = process.env.JWT_SECRET;

if (!JWT_SECRET) {
  console.error('Error: JWT_SECRET no está definida en las variables de entorno');
  console.error('Por favor, configura JWT_SECRET en tu archivo .env');
  process.exit(1);
}

const jwtSecret: string = JWT_SECRET;

export const verifyToken = (
  req: AuthRequest,
  res: Response,
  next: NextFunction
): void => {
  try {
    const token = req.headers.authorization?.split(' ')[1]; // Bearer TOKEN

    if (!token) {
      res.status(401).json({
        success: false,
        message: 'No se proporcionó token de autenticación'
      });
      return;
    }

    const decoded = jwt.verify(token, jwtSecret) as JWTPayload;

    // Agregar información del usuario al request
    req.userId = decoded.id;
    req.userRole = decoded.role;

    next();
  } catch (error) {
    res.status(401).json({
      success: false,
      message: 'Token inválido o expirado'
    });
  }
```

```
};
```

Explicación: Este middleware valida los tokens JWT que se envían en el header `Authorization` con formato `Bearer TOKEN`. Extrae el token, lo verifica usando `jwt.verify()` para validar su firma y expiración, y luego inyecta los datos del usuario (`userId` y `userRole`) en el objeto `req` para que los controladores puedan acceder a ellos. Si el token es inválido o no se proporciona, responde con un error 401.

4.2. 2. Comunicación entre Microservicios

En una arquitectura de microservicios, cada servicio debe validar la autenticación consultando al Auth Service. Este ejemplo muestra cómo el Payment Service se comunica con el Auth Service para verificar tokens.

```
import { Response, NextFunction } from 'express';
import axios from 'axios';
import { AuthRequest, UserRole, AuthServiceUserResponse } from '../
    types';

const AUTH_SERVICE_URL = process.env.AUTH_SERVICE_URL;

if (!AUTH_SERVICE_URL) {
  console.error('Error: AUTH_SERVICE_URL no está definida en las
      variables de entorno');
  console.error('Por favor, configura AUTH_SERVICE_URL en tu archivo .
      env');
  process.exit(1);
}

export const verifyToken = async (
  req: AuthRequest,
  res: Response,
  next: NextFunction
): Promise<void> => {
  try {
    const token = req.headers.authorization?.split(' ')[1];

    if (!token) {
      res.status(401).json({
        success: false,
        message: 'No se proporcionó token de autenticación'
      });
      return;
    }

    // Llamar al Auth Service para verificar el token y obtener datos
        del usuario
```

```
30      const response = await axios.get<AuthServiceUserResponse>(
31        `${AUTH_SERVICE_URL}/api/auth/profile`,
32        {
33          headers: {
34            Authorization: `Bearer ${token}`
35          }
36        }
37      );
38
39      if (!response.data.success || !response.data.data) {
40        res.status(401).json({
41          success: false,
42          message: 'Token inválido'
43        });
44        return;
45      }
46
47      // Agregar informacion del usuario al request
48      const user = response.data.data;
49      req.userId = String(user.id || user._id);
50      req.userRole = user.role;
51      req.userEmail = user.email;
52      req.userName = user.name;
53
54      next();
55    } catch (error) {
56      if (axios.isAxiosError(error)) {
57        if (error.response?.status === 401) {
58          res.status(401).json({
59            success: false,
60            message: 'Token inválido o expirado'
61          });
62          return;
63        }
64      }
65
66      console.error('Error al verificar token:', error);
67      res.status(500).json({
68        success: false,
69        message: 'Error al verificar autenticación'
70      });
71    }
72  };
```

Explicación: A diferencia del Auth Service que valida tokens localmente, los demás microservicios hacen una petición HTTP al Auth Service usando Axios. El token recibido se reenvía al endpoint `/api/auth/profile` del Auth Service, que valida el token y retorna los datos del usuario. Si la respuesta es exitosa, se extraen los datos (`id` o `_id`, `role`, `email`, `name`) y se agregan al objeto `req`. El `userId` se convierte explícitamente a string para garantizar consistencia en las comparaciones.

Este patrón centraliza la autenticación en un solo servicio, evitando duplicar lógica y facilitando el mantenimiento.

4.3. 3. Helpers y Utilidades (Refactorización)

Durante el desarrollo se identificó código duplicado en múltiples controladores. Se crearon funciones helper reutilizables para eliminar esta duplicación y mejorar la mantenibilidad.

```
import { Response } from 'express';
import Exercise from '../models/Exercise';
import { ApiResponse, MuscleGroup } from '../types';

// se especifican solo los campos necesarios para mostrar el ejercicio (excluye campos internos como createdBy, createdAt, updatedAt, __v)
// esto optimiza el tamaño de la respuesta y mejora el rendimiento al evitar cargar datos innecesarios
export const EXERCISE_POPULATE_FIELDS = 'title description youtubeVideoId muscleGroup difficulty thumbnailUrl';

// manejar errores de validación de mongoose
export const handleValidationError = (error: any, res: Response<ApiResponse>): boolean => {
  if (error.name === 'ValidationError' && error.errors) {
    const firstError = Object.values(error.errors)[0] as { message?: string };
    res.status(400).json({
      success: false,
      message: firstError?.message || 'Error de validación'
    });
    return true;
  }
  return false;
};

// validar que los grupos musculares sean validos
export const validateMuscleGroups = (muscleGroup: any): { valid: boolean; groups: MuscleGroup[] } => {
  if (!Array.isArray(muscleGroup) || muscleGroup.length === 0) {
    return { valid: false, groups: [] };
  }

  const validGroups = muscleGroup.filter((group: string) =>
    Object.values(MuscleGroup).includes(group as MuscleGroup)
  ) as MuscleGroup[];

  return { valid: validGroups.length > 0, groups: validGroups };
};

// verificar que todos los ejercicios existan
```

```
36 export const validateExercisesExist = async (exerciseIds: string[]):
       Promise<boolean> => {
37   const existingExercises = await Exercise.find({ _id: { $in:
       exerciseIds } });
38   return existingExercises.length === exerciseIds.length;
39 };
40
41 // validar indice de ejercicio en array
42 export const validateExerciseIndex = (index: number, arrayLength:
       number): boolean => {
43   return !isNaN(index) && index >= 0 && index < arrayLength;
44 };
```

Explicación: Estas funciones helper eliminan código duplicado y garantizan consistencia en las validaciones. `EXERCISE_POPULATE_FIELDS` optimiza las consultas definiendo qué campos incluir al hacer `populate()`. `handleValidationError()` maneja errores de validación de Mongoose de forma centralizada. `validateMuscleGroups()` valida y filtra grupos musculares según el enum. `validateExercisesExist()` verifica que todos los IDs existan usando `$in` de MongoDB. `validateExerciseIndex()` previene errores de acceso fuera de rango en arrays.

4.4. 4. Modelo con Validaciones (Exercise)

Los modelos de Mongoose definen la estructura de los datos y las validaciones que se aplican automáticamente. Este ejemplo muestra el modelo `Exercise` con validaciones complejas.

```
1  import mongoose, { Schema, Document } from 'mongoose';
2  import { MuscleGroup, Difficulty } from '../types';
3
4  // Interfaz que define la estructura de un ejercicio
5  export interface IExerciseDocument extends Document {
6    title: string; // nombre del ejercicio
7    description: string; // descripcion detallada del ejercicio
8    youtubeVideoId: string; // ID del video de YouTube
9    muscleGroup: MuscleGroup[]; // Array de grupos musculares
10   difficulty: Difficulty; // Nivel de dificultad del ejercicio
11   thumbnailUrl?: string; // URL opcional de la miniatura del video (se
       genera automáticamente si no se proporciona)
12   createdBy: mongoose.Types.ObjectId; // ID del usuario administrador
       que creó el ejercicio
13   createdAt: Date;
14   updatedAt: Date;
15 }
16
17 // Esquema de Mongoose para ejercicios
18 const ExerciseSchema = new Schema<IExerciseDocument>(
19   {
20     title: {
```

```
21        type: String,
22        required: true,
23        trim: true,
24        maxlength: [200, 'El título no puede exceder 200 caracteres']
25      },
26      description: {
27        type: String,
28        required: true,
29        trim: true,
30        maxlength: [2000, 'La descripción no puede exceder 2000
             caracteres']
31      },
32      youtubeVideoId: {
33        type: String,
34        required: true,
35        trim: true,
36        validate: {
37          validator: function(v: string) {
38            // Validar formato básico de ID de YouTube (11 caracteres
                 alfanuméricos)
39            return /^[a-zA-Z0-9_-]{11}$/.test(v);
40          },
41          message: 'El ID del video de YouTube no es válido'
42        }
43      },
44      muscleGroup: {
45        type: [String],
46        enum: Object.values(MuscleGroup),
47        required: true,
48        validate: {
49          validator: function(v: string[]) {
50            return Array.isArray(v) && v.length > 0;
51          },
52          message: 'Debe seleccionar al menos un grupo muscular'
53        }
54      },
55      difficulty: {
56        type: String,
57        enum: Object.values(Difficulty),
58        required: true
59      },
60      thumbnailUrl: {
61        type: String,
62        default: function(this: IExerciseDocument) {
63          // Generar URL de miniatura de YouTube automáticamente
64          return `https://img.youtube.com/vi/${this.youtubeVideoId}/
               mqdefault.jpg`;
65        }
66      },
67      createdBy: {
68        type: Schema.Types.ObjectId,
```

```
      required: true,
      ref: 'User'
    }
  },
  {
    timestamps: true
  }
);

// Indices para optimizar búsquedas
ExerciseSchema.index({ muscleGroup: 1, difficulty: 1 }); // Índice para búsquedas por grupo muscular
ExerciseSchema.index({ createdBy: 1, createdAt: -1 });
ExerciseSchema.index({ title: 'text', description: 'text' }); // Búsqueda por texto

// Modelo de Mongoose
const Exercise = mongoose.model<IExerciseDocument>('Exercise', ExerciseSchema);

export default Exercise;
```

Explicación: Este modelo define validaciones robustas que se ejecutan automáticamente antes de guardar. Los campos `title` y `description` tienen límites de caracteres con `maxlength`. `youtubeVideoId` valida el formato de 11 caracteres alfanuméricos. `muscleGroup` y `difficulty` usan `enum` para restringir valores válidos. `thumbnailUrl` se genera automáticamente desde el ID de YouTube. Los índices optimizan búsquedas por grupo muscular, dificultad, creador y texto completo. `createdBy` es una referencia a `User` que permite usar `populate()`.

4.5. 5. Controlador con Validaciones y Helpers (Payment Controller)

Este ejemplo muestra cómo se utiliza un controlador para crear suscripciones, combinando validaciones manuales, helpers y lógica de negocio.

```
export const createSubscription = async (
  req: AuthRequest,
  res: Response<ApiResponse>
): Promise<void> => {
  try {
    if (!checkAuth(req, res)) return;

    const { subscriptionType, bankDetails }: CreateSubscriptionDTO = req.body;

    // Validar tipo de suscripcion
    if (!Object.values(SubscriptionType).includes(subscriptionType)) {
      res.status(400).json({
```

```
13          success: false,
14          message: 'Tipo de suscripcion inválido'
15        });
16        return;
17      }
18
19      // Validar datos bancarios
20      if (!bankDetails || !bankDetails.cardNumber || !bankDetails.
            cardHolder || !bankDetails.expiryDate || !bankDetails.cvv) {
21        res.status(400).json({
22          success: false,
23          message: 'Datos bancarios incompletos'
24        });
25        return;
26      }
27
28      // Validar formato de tarjeta (solo numeros, 16 digitos)
29      const cardNumber = bankDetails.cardNumber.replace(/\s/g, '');
30      if (!/^\d{16}$/.test(cardNumber)) {
31        res.status(400).json({
32          success: false,
33          message: 'Número de tarjeta inválido (debe tener 16 dígitos)'
34        });
35        return;
36      }
37
38      // Validar formato de fecha de expiracion (MM/YY)
39      if (!/^\d{2}\/\d{2}$/.test(bankDetails.expiryDate)) {
40        res.status(400).json({
41          success: false,
42          message: 'Fecha de expiración inválida (formato: MM/YY)'
43        });
44        return;
45      }
46
47      // Validar CVV (3 o 4 digitos)
48      if (!/^\d{3,4}$/.test(bankDetails.cvv)) {
49        res.status(400).json({
50          success: false,
51          message: 'CVV inválido (debe tener 3 o 4 dígitos)'
52        });
53        return;
54      }
55
56      // Verificar si el usuario ya tiene una suscripcion activa
57      const activeSubscription = await Subscription.findOne({
58        userId: req.userId,
59        status: SubscriptionStatus.ACTIVE
60      });
61
62      if (activeSubscription) {
```

```
63        res.status(400).json({
64          success: false,
65          message: 'Ya tienes una suscripcion activa. Cancela la actual
                antes de crear una nueva.'
66        });
67        return;
68      }
69
70      // Calcular fechas y precio
71      const startDate = new Date();
72      const endDate = calculateEndDate(startDate, subscriptionType);
73      const amount = getSubscriptionPrice(subscriptionType);
74
75      // Simular procesamiento de pago (en produccion esto se haria con
            una pasarela real)
76      const paymentStatus = PaymentStatus.COMPLETED;
77      const subscriptionStatus = SubscriptionStatus.ACTIVE;
78
79      // Crear suscripcion (guardar solo ultimos 4 digitos de la tarjeta
            por seguridad)
80      const userIdObjectId = mongoose.Types.ObjectId.isValid(req.userId!)
81        ? new mongoose.Types.ObjectId(req.userId!)
82        : req.userId!;
83
84      const subscription = new Subscription({
85        userId: userIdObjectId,
86        subscriptionType,
87        startDate,
88        endDate,
89        status: subscriptionStatus,
90        paymentStatus,
91        amount,
92        bankDetails: {
93          cardNumber: cardNumber.slice(-4), // Solo guardar ultimos 4
                digitos
94          cardHolder: bankDetails.cardHolder,
95          expiryDate: bankDetails.expiryDate
96        }
97      });
98
99      await subscription.save();
100
101     const safeSubscription = formatSafeSubscription(subscription.
            toObject());
102
103     res.status(201).json({
104       success: true,
105       message: 'Suscripcion creada exitosamente',
106       data: {
107         ...safeSubscription,
108         hasActiveSubscription: true
```

```
109         }
110       });
111     } catch (error) {
112       console.error('Error al crear suscripcion:', error);
113       res.status(500).json({
114         success: false,
115         message: 'Error al crear suscripcion'
116       });
117     }
118 };
```

Explicación: Este controlador combina validaciones, lógica de negocio y seguridad. Utiliza `checkAuth()` para verificar autenticación y valida el tipo de suscripción con el enum. Valida datos bancarios con expresiones regulares (tarjeta de 16 dígitos, fecha MM/YY, CVV de 3-4 dígitos). Verifica que el usuario no tenga una suscripción activa para prevenir duplicados, con mensaje que indica cancelar la actual primero. Usa `calculateEndDate()` y `getSubscriptionPrice()` del modelo para calcular fechas y precios. Simula el procesamiento de pago (en producción se usaría una pasarela real). Por seguridad, solo guarda los últimos 4 dígitos de la tarjeta y formatea la respuesta de forma segura incluyendo `hasActiveSubscription: true` en la respuesta.

4.6. 6. Función de Cálculo de Fechas (Subscription Model)

Las funciones auxiliares en los modelos encapsulan lógica de negocio reutilizable. Esta función calcula la fecha de finalización de una suscripción según su tipo.

```
1  // Funcion auxiliar para calcular la fecha de finalizacion segun el
       tipo de suscripcion
2  // Recibe la fecha de inicio y el tipo de suscripcion, retorna la fecha
        de finalizacion
3  export const calculateEndDate = (startDate: Date, type:
       SubscriptionType): Date => {
4    const endDate = new Date(startDate);
5
6    switch (type) {
7      case SubscriptionType.MONTHLY:
8        // Suscripcion mensual
9        endDate.setMonth(endDate.getMonth() + 1);
10       break;
11     case SubscriptionType.QUARTERLY:
12       // Suscripcion trimestral
13       endDate.setMonth(endDate.getMonth() + 3);
14       break;
15     case SubscriptionType.YEARLY:
16       // Suscripcion anual
17       endDate.setFullYear(endDate.getFullYear() + 1);
18       break;
```

```
  }

  return endDate;
};
```

Explicación: Esta función calcula la fecha de finalización de una suscripción según su tipo. Crea una nueva instancia de `Date` para no modificar la fecha original (los objetos `Date` son mutables). Usa un **`switch`** para añadir el período correspondiente: `setMonth()` para mensuales y trimestrales (maneja cambios de año automáticamente) y `setFullYear()` para anuales (respeta años bisiestos). Se exporta para uso en modelos y controladores, garantizando consistencia en el cálculo de fechas en toda la aplicación.

4.7. 7. Función de Precio de Suscripción (Subscription Model)

Esta función centraliza la lógica de precios, facilitando su mantenimiento y permitiendo aplicar descuentos según el tipo de suscripción.

```
// Funcion para obtener el precio segun el tipo de suscripcion
// Retorna el monto a pagar para cada tipo de suscripcion
export const getSubscriptionPrice = (type: SubscriptionType): number => {
  switch (type) {
    case SubscriptionType.MONTHLY:
      return 29.99; // Precio mensual
    case SubscriptionType.QUARTERLY:
      return 79.99;  //ahorro de ~10 euros vs mensual
    case SubscriptionType.YEARLY:
      return 299.99; //(ahorro de ~60 euros vs mensual
    default:
      return 0;
  }
};
```

Explicación: Esta función centraliza todos los precios en un solo lugar, facilitando su mantenimiento. Los precios están diseñados para incentivar suscripciones de mayor duración: mensual (29.99€), trimestral (79.99€, ahorro de ~10€) y anual (299.99€, ahorro de ~60€). Si se proporciona un tipo inválido, retorna 0 en lugar de lanzar un error. Al estar en el modelo, garantiza que todos los controladores y servicios utilicen los mismos precios, evitando inconsistencias.

4.8. 8. Manejo de Errores en Registro (Auth Controller)

El registro de usuarios requiere manejar múltiples tipos de errores: validaciones, duplicados, y errores del servidor. Este ejemplo muestra un manejo robusto de errores.

```
export const register = async (req: Request, res: Response): Promise<void> => {
  try {
    const { email, password, name, dni, role, phone, birthDate }: RegisterDTO = req.body;

    // Verificar si el teléfono ya existe (solo si se proporciona)
    if (phone) {
      const existingPhone = await User.findOne({ phone: phone.trim() });
      if (existingPhone) {
        res.status(400).json({
          success: false,
          message: 'El número de teléfono ya está registrado'
        });
        return;
      }
    }

    // Verificar si el DNI ya existe (solo si se proporciona)
    if (dni) {
      const existingDni = await User.findOne({ dni: dni.toUpperCase().trim() });
      if (existingDni) {
        res.status(400).json({
          success: false,
          message: 'El DNI ya está registrado'
        });
        return;
      }
    }

    // Hash de la contraseña
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Crear nuevo usuario
    // En el registro público, todos los usuarios se crean como SOCIO por defecto
    // Los roles de MONITOR y ADMIN solo pueden ser asignados por administradores
    const newUser = new User({
      email: email.toLowerCase(),
      password: hashedPassword,
      name,
      dni: dni ? dni.toUpperCase().trim() : undefined,
      role: UserRole.SOCIO, // Siempre SOCIO en registro público
      phone: phone ? phone.trim() : undefined,
      birthDate: birthDate ? new Date(birthDate) : undefined,
      isActive: true
```

```
45      });
46
47      await newUser.save();
48
49      // Generar token JWT
50      const payload: JWTPayload = {
51        id: String(newUser._id),
52        email: newUser.email,
53        role: newUser.role
54      };
55
56      const token = jwt.sign(payload, jwtSecret, { expiresIn:
            JWT_EXPIRES_IN });
57
58      const response: AuthResponse = {
59        token,
60        user: {
61          id: String(newUser._id),
62          email: newUser.email,
63          name: newUser.name,
64          role: newUser.role
65        }
66      };
67
68      res.status(201).json({
69        success: true,
70        message: 'Usuario registrado exitosamente',
71        data: response
72      });
73    } catch (error: any) {
74      // Manejar errores de validación de Mongoose (unicidad, etc.)
75      if (error.code === 11000) {
76        const field = Object.keys(error.keyPattern)[0];
77        let message = 'Error de validación';
78        if (field === 'email') {
79          message = 'El email ya está registrado';
80        } else if (field === 'phone') {
81          message = 'El número de teléfono ya está registrado';
82        } else if (field === 'dni') {
83          message = 'El DNI ya está registrado';
84        }
85        res.status(400).json({
86          success: false,
87          message
88        });
89        return;
90      }
91
92      // Manejar errores de validación de Mongoose
93      if (error.name === 'ValidationError') {
94        const messages = Object.values(error.errors).map((err: any) =>
```

```
            err.message);
95        res.status(400).json({
96          success: false,
97          message: messages.join(', ')
98        });
99        return;
100     }
101
102     console.error('Error en registro:', error);
103     res.status(500).json({
104       success: false,
105       message: 'Error al registrar usuario'
106     });
107   }
108 };
```

Explicación: Este controlador maneja el registro de usuarios con validaciones previas (teléfono y DNI), normalización de datos (email a minúsculas, DNI a mayúsculas, `trim()`), y seguridad (contraseñas hasheadas con `bcrypt`). Todos los usuarios se crean como `SOCIO` en registro público para prevenir escalación de privilegios. Después de crear el usuario, se genera un token JWT para login automático. Maneja errores de duplicado con mensajes específicos, combina errores de validación de Mongoose, y captura errores genéricos sin exponer información sensible.

4.9. 9. Validación de Reservas con Comunicación entre Servicios (Booking Controller)

Las reservas de clases requieren validar múltiples condiciones y comunicarse con otros microservicios. Este ejemplo muestra cómo se integran estas validaciones.

```
1  export const createBooking = async (req: AuthRequest, res: Response):
       Promise<void> => {
2    try {
3      // Solo los socios pueden reservar clases
4      if (req.userRole !== UserRole.SOCIO) {
5        res.status(403).json({
6          success: false,
7          message: 'Solo los socios pueden reservar clases'
8        });
9        return;
10     }
11
12     const { classId }: CreateBookingDTO = req.body;
13
14     if (!classId) {
15       res.status(400).json({
16         success: false,
17         message: 'El ID de la clase es obligatorio'
18       });
```

```
19        return;
20      }
21
22      // Verificar que la clase existe
23      const classData = await Class.findById(classId);
24
25      if (!classData) {
26        res.status(404).json({
27          success: false,
28          message: 'Clase no encontrada'
29        });
30        return;
31      }
32
33      // Verificar que la clase no esta cancelada o completada
34      if (classData.status === ClassStatus.CANCELLED) {
35        res.status(400).json({
36          success: false,
37          message: 'No se puede reservar una clase cancelada'
38        });
39        return;
40      }
41
42      if (classData.status === ClassStatus.COMPLETED) {
43        res.status(400).json({
44          success: false,
45          message: 'No se puede reservar una clase completada'
46        });
47        return;
48      }
49
50      // Verificar que la clase es futura
51      if (new Date(classData.schedule) <= new Date()) {
52        res.status(400).json({
53          success: false,
54          message: 'No se puede reservar una clase que ya ha pasado'
55        });
56        return;
57      }
58
59      // Verificar que hay cupo disponible
60      if (classData.currentParticipants >= classData.maxParticipants) {
61        res.status(400).json({
62          success: false,
63          message: 'La clase está completa. No hay cupos disponibles.'
64        });
65        return;
66      }
67
68      // Verificar que el usuario esta autenticado y tiene datos
69      if (!req.userId || !req.userName) {
```

```
70        res.status(401).json({
71          success: false,
72          message: 'Usuario no autenticado correctamente'
73        });
74        return;
75      }
76
77      // Verificar que el usuario tiene una suscripción activa
78      try {
79        const token = req.headers.authorization?.split(' ')[1];
80        const subscriptionResponse = await axios.get(
81          `${PAYMENT_SERVICE_URL}/api/payments/me/active`,
82          {
83            headers: {
84              Authorization: `Bearer ${token}`
85            }
86          }
87        );
88
89        if (subscriptionResponse.data.success && subscriptionResponse.
              data.data) {
90          const hasActiveSubscription = subscriptionResponse.data.data.
                hasActiveSubscription;
91          if (!hasActiveSubscription) {
92            res.status(403).json({
93              success: false,
94              message: 'Necesitas una suscripción activa para reservar
                    clases'
95            });
96            return;
97          }
98        }
99      } catch (error: any) {
100       // Si el servicio de pagos no está disponible, denegar la reserva
              por seguridad
101       res.status(503).json({
102         success: false,
103         message: 'No se pudo verificar tu suscripción. Por favor,
                intenta más tarde.'
104       });
105       return;
106     }
107
108     // Verificar que el usuario no tiene ya una reserva activa para
          esta clase
109     const existingBooking = await Booking.findOne({
110       userId: req.userId,
111       classId,
112       status: BookingStatus.CONFIRMED
113     });
114
```

```
115      if (existingBooking) {
116        res.status(400).json({
117          success: false,
118          message: 'Ya tienes una reserva activa para esta clase'
119        });
120        return;
121      }
122
123      // Crear la reserva
124      const newBooking = new Booking({
125        userId: req.userId,
126        userName: req.userName,
127        classId,
128        className: classData.name,
129        bookingDate: new Date(),
130        status: BookingStatus.CONFIRMED
131      });
132
133      await newBooking.save();
134
135      // Incrementar el contador de participantes
136      classData.currentParticipants += 1;
137      await classData.save();
138
139      res.status(201).json({
140        success: true,
141        data: newBooking,
142        message: 'Reserva creada exitosamente'
143      });
144    } catch (error) {
145      console.error('Error al crear reserva:', error);
146      res.status(500).json({
147        success: false,
148        message: 'Error al crear la reserva'
149      });
150    }
151  };
```

Explicación: Este controlador integra múltiples validaciones: solo socios pueden reservar, verifica existencia y estado de la clase, valida que sea futura y haya cupo disponible y verifica que el usuario esté autenticado correctamente.

Se comunica con el Payment Service vía HTTP para verificar que el usuario tenga una suscripción activa, mostrando cómo los microservicios se comunican entre sí. Valida que no exista una reserva duplicada antes de crear. Maneja errores de comunicación apropiadamente: si el servicio de pagos no está disponible, retorna 503 en lugar de permitir la reserva por seguridad. Después de crear la reserva incrementa el contador de participantes de la clase para mantener consistencia de datos.

4.10. 10. Estructura de Rutas con Middlewares (Exercise Routes)

Las rutas definen los endpoints de la API y aplican middlewares para autenticación y autorización. Este ejemplo muestra cómo se organizan las rutas con middlewares agrupados.

```
import { Router } from 'express';
import {
  getAllExercises,
  getExerciseById,
  createExercise,
  updateExercise,
  deleteExercise
} from '../controllers/exerciseController';
import { verifyToken, isAdmin } from '../middleware/authMiddleware';

const router = Router();

// middlewares para rutas protegidas de admin
const adminMiddleware = [verifyToken, isAdmin];

// Rutas públicas (no requieren autenticación para ver)
router.get('/', getAllExercises);
router.get('/:id', getExerciseById);

// Rutas protegidas (requieren autenticación y rol admin)
router.post('/', ...adminMiddleware, createExercise);
router.put('/:id', ...adminMiddleware, updateExercise);
router.delete('/:id', ...adminMiddleware, deleteExercise);

export default router;
```

Explicación: Se agrupan los middlewares `verifyToken` e `isAdmin` en una constante para mejorar la legibilidad. Las rutas GET son públicas (cualquiera puede ver ejercicios), mientras que POST, PUT y DELETE requieren autenticación y rol admin usando el operador spread. El orden de middlewares es importante: primero se verifica el token y luego el rol. Esto separa responsabilidades (lógica de negocio en controladores, autenticación en middlewares) y permite reutilizar el array de middlewares en múltiples rutas, garantizando consistencia y seguridad.

4.11. 11. Interceptor de Axios (Frontend)

El frontend utiliza interceptores de Axios para manejar automáticamente la autenticación y los errores en todas las peticiones HTTP. Esto centraliza la lógica de autenticación y evita duplicar código en cada componente.

```
import axios, { type AxiosInstance, AxiosError } from 'axios';

```

```
3  const API_URL = import.meta.env.VITE_API_URL;
4
5  if (!API_URL) {
6    console.error('Error: VITE_API_URL no está definida en las variables
       de entorno');
7    console.error('Por favor, configura VITE_API_URL en tu archivo .env')
       ;
8    throw new Error('VITE_API_URL no está configurada. Por favor,
       configura las variables de entorno.');
9  }
10
11 // Crear instancia de axios
12 const api: AxiosInstance = axios.create({
13   baseURL: API_URL,
14   headers: {
15     'Content-Type': 'application/json'
16   }
17 });
18
19 // Interceptor para agregar token a las peticiones
20 api.interceptors.request.use(
21   (config) => {
22     const token = localStorage.getItem('token');
23     if (token) {
24       config.headers.Authorization = `Bearer ${token}`;
25     }
26     return config;
27   },
28   (error) => {
29     return Promise.reject(error);
30   }
31 );
32
33 // Interceptor para manejar errores de respuesta
34 api.interceptors.response.use(
35   (response) => response,
36   (error: AxiosError) => {
37     if (error.response?.status === 401) {
38       const url = error.config?.url || '';
39       // No redirigir si es un error de login o cambio de contraseña
40       if (!url.includes('/login') && !url.includes('/change-password'))
           {
41         // Token expirado o inválido - solo redirigir si no estamos en
             login
42         localStorage.removeItem('token');
43         localStorage.removeItem('user');
44         window.location.href = '/login';
45       }
46     }
47     return Promise.reject(error);
48   }
```

```
);

export default api;
```

Explicación: Se crea una instancia de Axios configurada con la URL base de la API desde variables de entorno. El interceptor de request agrega automáticamente el token JWT almacenado en `localStorage` al header `Authorization` de todas las peticiones, evitando tener que añadirlo manualmente en cada llamada.

El interceptor de response maneja errores 401 (no autorizado): si el token expira o es inválido, limpia el almacenamiento local y redirige al login, excepto en rutas de login o cambio de contraseña donde el error 401 es esperado. Esto garantiza que el usuario siempre esté autenticado y mejora la experiencia de usuario.

4.12. 12. Configuración de Conexión a Base de Datos

Cada microservicio necesita conectarse a MongoDB de forma independiente. Esta función centraliza la configuración de conexión y maneja errores de forma consistente.

```
// configuración de conexión a mongodb para el servicio de autenticación
import mongoose from 'mongoose';
import dotenv from 'dotenv';

dotenv.config();

const connectDB = async (): Promise<void> => {
  try {
    const mongoUri = process.env.MONGODB_URI;

    if (!mongoUri) {
      console.error('Error: MONGODB_URI no está definida en las variables de entorno');
      console.error('Por favor, configura MONGODB_URI en tu archivo .env');
      process.exit(1);
      return;
    }

    const conn = await mongoose.connect(mongoUri);
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error('Error conectando a MongoDB:', error);
    process.exit(1);
  }
};

```

```
26 export default connectDB;
```

Explicación: Esta función configura la conexión a MongoDB usando Mongoose. Primero carga las variables de entorno con `dotenv.config()`. Valida que `MONGODB_URI` esté definida, deteniendo la aplicación si falta para evitar ejecutarse con configuración incorrecta. Si la conexión falla, registra el error y termina el proceso, ya que sin base de datos el microservicio no puede funcionar. Cada microservicio tiene su propia instancia de esta función, permitiendo que cada uno se conecte a su base de datos independiente, siguiendo el principio de microservicios donde cada servicio tiene su propia persistencia.

5. Conclusiones

5.1. Aprendizajes Adquiridos

En proyecto he aprendido sobre arquitectura de microservicios, uno de los tipos de arquitectura más utilizados actualmente en la industria de software como seria Netflix.

He aprendido a crear y diseñar y organizar aquellos servicios independientes, donde cada uno de ellos tiene su propia responsabilidad y base de datos, permitiendo que la escalabilidad y la mantenibilidad son mucho mayores que las que podría proporcionar una aplicación monolítica.

También he aprendido mucho sobre TypeScript, donde he utilizado la ventaja de contar con un tipado estático, ya que sirve para la detección de errores en el tiempo de compilación y es útil para mejorar la calidad del código.

El trabajo con React a su vez me ha hecho aprender por ejemplo cómo gestionar el estado, así como también utilizar hooks y elaborar componentes reutilizables para que nos ayuden a evitar el código duplicado.

Otras de las cosas que aprendido es como hacer comunicaciones entre microservicios utilizando APIs REST como forma de comunicación.

5.2. Reflexión

Si que hacer el proyecto de nuevo, comenzaría con una planificación más detallada de la arquitectura desde el inicio, definiendo claramente las interfaces y contratos entre servicios antes de comenzar la implementación.

Esto habría evitado tener que refactorizar código más adelante. También implementaría tests automatizados desde el principio, tanto unitarios como de integración.

En cuanto al frontend, habría creado un sistema de diseño más estructurado con componentes base reutilizables desde el principio, evitando duplicación de estilos CSS.

5.3. Posibles Mejoras

En el proyecto una mejora sería implementar notificaciones en tiempo real utilizando WebSockets, permitiendo que los usuarios reciban actualizaciones instantáneas sobre sus reservas o mensajes sin necesidad de recargar la página.

Otra idea interesante sería desarrollar una aplicación móvil nativa usando React Native, aprovechando gran parte del código del frontend existente. También se podría integrar un sistema de pagos real con pasarelas como PayPal, en lugar de la simulación actual.

Otra funcionalidad útil sería un sistema de recordatorios por email o SMS para las clases reservadas.

6. Bibliografía

Algunas paginas que he usado para fotos e iconos del proyecto:

https://wallpaperscraft.com/

https://www.freepik.es/fotos/gym

https://www.flaticon.es/

https://htmlcolorcodes.com/

Algunos videos que me han servido:

https://www.youtube.com/watch?v=4W3UWjyyVkQ

https://www.tiktok.com/@dev_flash

Alguna documentacion que he mirado para el proyecto:

https://react.dev/

https://www.typescriptlang.org/

https://mongoosejs.com/