



Entrega 06 Módulo de Testing - Grupo 7 UNEmployed

Integrantes:

- ANGEL DAVID GÓMEZ PASTRANA
- DIEGO FELIPE CABRERA SALAMANCA
- PABLO LUNA GUZMÁN
- JAURI ESTEBAN CORTÉS CÁRDENAS

1. Para la implementación de las pruebas unitarias se utilizaron las librerías **Mocha** y **Chai**, ambas bastante utilizadas dentro del ecosistema de Node.js y compatibles con TypeScript mediante **ts-node**. **Mocha** actúa como *test runner*, permitiendo estructurar las pruebas en bloques **describe** e **it**, mientras que **Chai** proporciona un conjunto de funciones de aserción, empleando el estilo **expect**, que facilita la validación explícita de los resultados esperados. La ejecución de todas las pruebas del proyecto se realiza mediante el comando configurado en el script de NPM en la sección de Backend, definido como:

TypeScript

```
"test": "node ./node_modules/mocha/bin/mocha.js -r ts-node/register tests/**/*.*.test.ts"
```

Esto dado que si no usamos la instalación local de mocha, Node intenta buscar una instalación global de esta y por tanto generaría problemas de portabilidad.

¿Y por qué no usamos Jest?

Jest es la librería más conocida para hacer test en Java, JavaScript y TypeScript pero Jest al instalarlo, nos traía varias vulnerabilidades de las cuales nos advirtió NPM y aunque no eran críticas, decidimos optar por Mocha con Chai, librerías las cuales no nos traían ninguna vulnerabilidad y además traen una sintaxis sencilla y explícita para los tests que debíamos hacer.

A continuación mostramos los test que se hicieron para algunas de las funciones que se hicieron por integrante:

Test angomezp:

- **Función Distancia de Haversine:**

La función se usa para calcular la distancia entre dos coordenadas sobre una esfera de cierto radio (Asegura mayor precisión para los cálculos de distancia entre puntos del recorrido del entrenamiento)

TypeScript

```
it('should calculate distance between two points correctly', () => {
    const lat1 = 40.7128;
    const lon1 = -74.0060;
    const lat2 = 34.0522;
    const lon2 = -118.2437;

    const distance = haversineDistance(lat1, lon1, lat2, lon2);

    expect(distance).to.be.closeTo(3935.75, 0.1);
});

it('should return 0 for the same coordinates', () => {
    const distance = haversineDistance(40.7128, -74.0060, 40.7128,
-74.0060);
    expect(distance).to.equal(0);
});

it('should calculate short distances correctly', () => {
    const distance = haversineDistance(4.6097, -74.0817, 4.6107,
-74.0827);
    expect(distance).to.be.closeTo(0.157, 0.01);
});

it('should handle very short distances (less than 20 meters)', () => {
    const distance = haversineDistance(4.6097, -74.0817, 4.60972,
-74.08172);
    expect(distance).to.be.closeTo(0.00314, 0.0001);
});

it('should handle large distances', () => {
    const distance = haversineDistance(40.7128, -74.0060, -33.4489,
-70.6693);
    expect(distance).to.be.closeTo(8253.50, 0.1);
});

it('should return NaN for latitude out of valid range (> 90)', () => {
    const distance = haversineDistance(100, 0, 40, 0);
    expect(distance).to.be.NaN;
});

it('should return NaN for latitude out of valid range (< -90)', () => {
```

```

    const distance = haversineDistance(-100, 0, 40, 0);
    expect(distance).to.be.NaN;
});

it('should return NaN for longitude out of valid range (> 180)', () => {
    const distance = haversineDistance(40, 200, 40, 0);
    expect(distance).to.be.NaN;
});

it('should return NaN for longitude out of valid range (< -180)', () => {
    const distance = haversineDistance(40, -200, 40, 0);
    expect(distance).to.be.NaN;
});

it('should accept coordinates at valid boundaries', () => {
    const distance1 = haversineDistance(90, 0, -90, 0);
    const distance2 = haversineDistance(0, 180, 0, -180);
    expect(distance1).to.not.be.NaN;
    expect(distance2).to.not.be.NaN;
});

```

- Los tests de la función “**haversineDistance**” son importantes porque nos permiten comprobar que realmente está calculando bien las distancias y que no falla en situaciones clave. Se revisa que funcione con distancias grandes entre ciudades conocidas, que devuelve 0 cuando los puntos son exactamente los mismos y que siga siendo preciso cuando las distancias son muy cortas, donde normalmente aparecen errores numéricos. También se valida que la función sepa detectar coordenadas que no existen (como latitudes mayores a 90°) y devuelva **NaN** en esos casos, para evitar resultados falsos. Por último, se comprueba que los valores límite válidos sí los acepte sin problema. Todo esto hace que las pruebas no sean triviales, porque revisan tanto la precisión de los cálculos como el manejo de errores y de casos reales que podrían aparecer en el uso de la aplicación.

- **Función Calcular Calorías:**

TypeScript

```

it('should calculate calories for low speed (< 8 km/h)', () => {
    const calories = calculateCalories(5, 70, 1);
    expect(calories).to.equal(560);
});

it('should calculate calories for medium speed (8-12 km/h)', () => {
    const calories = calculateCalories(10, 70, 1);
    expect(calories).to.equal(700);
});

```

```
it('should calculate calories for high speed (> 12 km/h)', () => {
    const calories = calculateCalories(15, 70, 1);
    expect(calories).to.equal(840);
});

it('should calculate calories with different weight', () => {
    const calories1 = calculateCalories(10, 60, 1);
    const calories2 = calculateCalories(10, 80, 1);

    expect(calories1).to.equal(600);
    expect(calories2).to.equal(800);
});

it('should calculate calories with different time', () => {
    const calories1 = calculateCalories(10, 70, 0.5);
    const calories2 = calculateCalories(10, 70, 2);

    expect(calories1).to.equal(350);
    expect(calories2).to.equal(1400);
});

it('should return 0 for zero time', () => {
    const calories = calculateCalories(10, 70, 0);
    expect(calories).to.equal(0);
});

it('should handle exact boundary of 8 km/h', () => {
    const calories = calculateCalories(8, 70, 1);
    expect(calories).to.equal(700);
});

it('should handle exact boundary of 12 km/h', () => {
    const calories = calculateCalories(12, 70, 1);
    expect(calories).to.equal(840);
});

it('should return NaN for negative speed', () => {
    const calories = calculateCalories(-5, 70, 1);
    expect(calories).to.be.NaN;
});

it('should return NaN for zero weight', () => {
    const calories = calculateCalories(10, 0, 1);
    expect(calories).to.be.NaN;
});

it('should return NaN for negative weight', () => {
    const calories = calculateCalories(10, -70, 1);
```

```

        expect(calories).to.be.NaN;
    });

it('should return NaN for negative time', () => {
    const calories = calculateCalories(10, 70, -1);
    expect(calories).to.be.NaN;
});

```

- Los tests de la función “**calculateCalories**” son útiles porque nos permiten verificar que el cálculo realmente cambia según la velocidad, el peso y el tiempo, que son las variables principales del modelo. Primero se revisan los tres rangos de velocidad (baja, media y alta) para asegurarnos de que la fórmula esté aplicando el factor correcto en cada caso. Luego se prueba cómo cambia el resultado cuando varía el peso y el tiempo, lo cual es clave porque estos valores afectan directamente las calorías quemadas. También se revisan casos límite como cuando el tiempo es cero, para que la función no devuelva un valor absurdo. Además, se prueban las velocidades exactas en los límites de 8 km/h y 12 km/h para confirmar que la lógica de las condiciones funciona bien justo en esos puntos. Finalmente, se validan entradas inválidas como velocidades negativas, pesos no válidos o tiempo negativo, esperando que la función devuelva **NaN** en lugar de cálculos incorrectos. Todo esto hace que las pruebas no sean triviales, porque evalúan diferentes escenarios reales y comprueban que la función sea robusta ante datos erróneos.

- **Funcion calcular Pace:**

```

TypeScript
it('should calculate pace correctly', () => {
    const pace = calculatePace(10);
    expect(pace).to.equal('6:00');
});

it('should handle very slow speeds', () => {
    const pace = calculatePace(2);
    expect(pace).to.equal('30:00');
});

it('should return "0:00" for zero speed', () => {
    const pace = calculatePace(0);
    expect(pace).to.equal('0:00');
});

it('should format seconds with leading zero', () => {
    const pace = calculatePace(12);
    expect(pace).to.equal('5:00');
});

```

```
});

it('should handle very high speeds', () => {
    const pace = calculatePace(20);
    expect(pace).to.equal('3:00');
});

it('should calculate pace with decimals correctly', () => {
    const pace = calculatePace(8.5);
    expect(pace).to.equal('7:03');
});

it('should handle speed very close to zero', () => {
    const pace = calculatePace(0.1);
    expect(pace).to.equal('600:00');
});

it('should handle extremely high speed', () => {
    const pace = calculatePace(100);
    expect(pace).to.equal('0:36');
});

it('should return "0:00" for negative speed', () => {
    const pace = calculatePace(-10);
    expect(pace).to.equal('0:00');
});

it('should handle speed that produces seconds close to 60', () => {
    const pace = calculatePace(11.11);
    expect(pace).to.equal('5:24');
});
```

- Estos tests buscan asegurar que la función “**calculatePace**” no solo haga la fórmula básica, sino que responda bien a casos que suelen romper este tipo de cálculos. Probamos una velocidad normal para verificar que lo básico funcione, y luego velocidades muy bajas o muy altas porque ahí es donde suelen salir errores con divisiones y formatos raros. También revisamos que los segundos queden con un cero adelante cuando toca, que los decimales no dañen el cálculo real del ritmo, y que valores extremos como casi cero, extremadamente altos o negativos no generen errores raros sino un resultado coherente. Incluso probamos una velocidad que produce segundos casi llegando a 60, para verificar que no se desborde el formato. En conjunto, estos tests ayudan a que la función sea confiable en situaciones reales y en casos límite.

Test dicabreras:

- Función lógica de entrenamiento: “calculateAvgSpeed”. Esta función hace parte de las estadísticas que se muestran en la grabación de un entrenamiento. Su implementación se encuentre en el directorio /Backend/src/services/trainingServices.ts.

Esta función se encarga de retornar la velocidad promedio del entrenamiento grabado por un usuario.

Se pusieron a prueba los siguientes casos:

1. Calcular correctamente la velocidad promedio para un entrenamiento que registra 10 km en 1 hora.
2. Calcular correctamente la velocidad promedio para un entrenamiento que registra 5 km en 30 minutos.
3. Calcular correctamente la velocidad promedio para un entrenamiento que registra 0 segundos de actividad.

TypeScript

```
describe('calculateAvgSpeed', () => {

    it('should calculate average speed correctly for 10 km in 1 hour', () => {
        const avg = calculateAvgSpeed(10, 3600); // 10 km, 3600s
        expect(avg).to.be.closeTo(10, 0.0001);
    });

    it('should calculate average speed correctly for 5 km in 30 minutes', () => {
        const avg = calculateAvgSpeed(5, 1800); // 5 km, 1800s => 10 km/h
        expect(avg).to.be.closeTo(10, 0.0001);
    });

    it('should return 0 when totalSeconds is 0', () => {
        const avg = calculateAvgSpeed(5, 0);
        expect(avg).to.equal(0);
    });
});
```

```
});
```

- Función lógica de entrenamiento: “calculateMaxSpeed”. Esta función hace parte de las estadísticas que se muestran en la grabación de un entrenamiento. Su implementación se encuentre en el directorio [/Backend/src/services/trainingServices.ts](#).

Esta función se encarga de retornar la velocidad máxima en cualquier instante del entrenamiento grabado por un usuario.

Se pusieron a prueba los siguientes casos:

1. retorna correctamente la velocidad máxima de un array de velocidades tomada en distintos momentos del entrenamiento.
2. Retornar 0 cuando el array esta vacío, es decir, no hubo ningún registro de distancia y por tanto tampoco velocidad.
3. Determinar correctamente la velocidad para un array con un solo elemento. Retorna este único elemento.

TypeScript

```
describe('calculateMaxSpeed', () => {  
  it('should return the maximum speed from array', () => {  
    const speeds = [3.2, 7.5, 12.1, 9.9];  
  
    const max = calculateMaxSpeed(speeds);  
  
    expect(max).to.equal(12.1);  
  });  
  
  it('should return 0 for empty speeds array', () => {  
    const max = calculateMaxSpeed([]);  
  
    expect(max).to.equal(0);  
  });  
  
  it('should handle single-element array', () => {
```

```
    const max = calculateMaxSpeed([5.5]);

    expect(max).to.equal(5.5);

});

});
```

- Función lógica de gestión de entrenamientos: “saveTraining”. Esta función hace parte de las operaciones que se realizan con los entrenamientos una vez la grabación de este ha sido completada exitósamente. Su implementación se encuentra en el directorio /Backend/src/controller/trainingController.ts.

Esta función permite guardar los entrenamientos verificando siempre que el usuario exista en el sistema y sus credenciales coincidan con las guardadas en la base de datos.

Se puso a prueba el siguiente caso:

1. Debe guardar el entrenamiento de un usuario exitósamente
2. Gestionar apropiadamente el error cuando el correo del usuario no está registrado
3. Gestionar debidamente el error cuando el usuario no aparece en la base de datos
4. Rechaza el guardado de un entrenamiento cuando la distancia es 0 o no se registró ninguna ruta.
5. El guardado del entrenamiento falla

TypeScript

```
describe('TrainingController.saveTraining', () => {

    let controller: any;

    let originalGetRepository: any;

    let shouldThrowOnSave = false;

    before(() => {

        // Build fake repositories with minimal behavior required by the
        controller

        let coordId = 1;

        let routeId = 1;

        let trainingCounter = 1;
```

```
const fakeUser = {  
  email: 'test@local',  
  username: 'tester',  
  names: 'Test',  
  lastNames: 'User',  
  age: 30  
};  
  
const fakeRepos = {  
  User: {  
    findOne: async (opts: any) => {  
      const email = opts?.where?.email;  
      if (email === fakeUser.email) return fakeUser;  
      return null;  
    }  
  },  
  Coordinate: {  
    create: (obj: any) => ({ ...obj }),  
    save: async (coord: any) => {  
      coord.id = coordId++;  
      return coord;  
    }  
  },  
  Route: {  
    create: (obj: any) => ({ ...obj }),  
  },  
};
```

```
save: async (route: any) => {

    route.id = routeId++;
    return route;
}

},
Training: {

    create: (obj: any) => ({ ...obj }),

    save: async (t: any) => {

        if (shouldThrowOnSave) throw new Error('Simulated save error');

        t.counter = trainingCounter++;
        return t;
    }
};

// Patch appDataSource.getRepository to return appropriate fake
repository

const ds = require('../src/db_connection/config/dataSource');

// Save original getRepository to restore later

originalGetRepository = ds.appDataSourcegetRepository;

ds.appDataSourcegetRepository = (entity: any) => {

    const name = entity && entity.name ? entity.name : String(entity);

    if (name.includes('User')) return fakeRepos.User;

    if (name.includes('Coordinate')) return fakeRepos.Coordinate;

    if (name.includes('Route')) return fakeRepos.Route;

    if (name.includes('Training')) return fakeRepos.Training;
```

```
// Fallback repository with basic create/save methods

    return { create: (o: any) => ({ ...o }), save: async (o: any) => ({
...o }) };

};

// Now import the controller after the stub is in place

controller =
require(' ../../src/db_connection/controller/TrainingController');

});

// Restore original getRepository to avoid interfering with other tests

after(() => {

try {

const ds = require(' ../../src/db_connection/config(dataSource');

    if (originalGetRepository) ds.appDataSource.getRepository =
originalGetRepository;

} catch (e) {

    // ignore restore errors in test environment

}

});

it('should save a training successfully', async () => {

const req: any = {

body: {

userEmail: 'test@local',

distance: 5000,

duration: '00:25:00',
```

```
avgSpeed: 12.0,  
maxSpeed: 15.0,  
rithm: 5.0,  
calories: 300,  
elevationGain: 10,  
trainingType: 'Running',  
route: [  
  { latitude: 4.6, longitude: -74.0, altitude: 2550 },  
  { latitude: 4.6001, longitude: -74.0002, altitude: 2552 }  
],  
datetime: '2025-11-16T12:00:00Z'  
}  
};  
  
const res: any = {  
  statusCode: 0,  
  body: null,  
  status(code: number) { this.statusCode = code; return this; },  
  json(obj: any) { this.body = obj; return this; }  
};  
  
await controller.saveTraining(req, res);  
  
expect(res.statusCode).to.equal(201);  
  expect(res.body).to.have.property('message', 'Training saved successfully');
```

```
expect(res.body).to.have.property('training');

const t = res.body.training;

expect(t).to.have.property('distance');

expect(t.distance).to.equal(5000);

expect(t).to.have.property('avgSpeed');

expect(t.avgSpeed).to.equal(12.0);

});

it('should return 400 if userEmail is missing', async () => {

  const req: any = { body: { distance: 100 } };

  const res: any = { statusCode: 0, body: null, status(code: number) {
    this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
    return this; } };

  await controller.saveTraining(req, res);

  expect(res.statusCode).to.equal(400);

});

it('should return 404 when user not found', async () => {

  const req: any = { body: { userEmail: 'notfound@local', route: [ {
    latitude: 4.6, longitude: -74.0, altitude: 2550 }, { latitude: 4.6001,
    longitude: -74.0002, altitude: 2552 } ] } };

  const res: any = { statusCode: 0, body: null, status(code: number) {
    this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
    return this; } };

  await controller.saveTraining(req, res);

  expect(res.statusCode).to.equal(404);

});
```

```
it('should reject training when no route and zero distance', async () => {

    const req: any = { body: { userEmail: 'test@local', distance: undefined
} };

    const res: any = { statusCode: 0, body: null, status(code: number) {
this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
return this; } };

    await controller.saveTraining(req, res);

    // Now controller rejects trainings with no valid route and zero
distance

    expect(res.statusCode).to.equal(400);

});

it('should return 500 when training repository save fails', async () => {

    // enable throwing in fake repo

    shouldThrowOnSave = true;

    const req: any = { body: { userEmail: 'test@local', route: [ { latitude:
4.6, longitude: -74.0, altitude: 2550 }, { latitude: 4.6001, longitude:
-74.0002, altitude: 2552 } ] } };

    const res: any = { statusCode: 0, body: null, status(code: number) {
this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
return this; } };

    await controller.saveTraining(req, res);

    expect(res.statusCode).to.equal(500);

    // restore flag

    shouldThrowOnSave = false;

});

});
```

Test jcortesca:

Para el módulo desarrollado por mi parte, se implementaron pruebas unitarias enfocadas en tres componentes principales del backend: las funciones del controlador de usuario (`registerUser` y `loginUser`) y la función de utilidades `calculateElevation`. El objetivo de estos tests fue garantizar que los controladores gestionaran correctamente todos los casos esperados, tanto positivos como negativos, y que la función de cálculo respondiera adecuadamente ante datos reales y casos límite. A continuación se presentan los desribes correspondientes.

1. Test para `registerUser`

La función `registerUser` es responsable de registrar un nuevo usuario en el sistema. En este describe se probaron los siguientes escenarios:

- Registro exitoso cuando todos los campos están completos.
- Respuesta adecuada cuando faltan campos requeridos.
- Manejo correcto cuando el usuario ya existe en base de datos.
- Comportamiento apropiado ante errores internos inesperados.

```
it("should register user and return 201 with no missing fields", async () => {

    //cuerpo del request falso
    req.body = { ...baseBody};

    findOneStub.resolves(null);
    saveStub.resolves({ id:1 })

    await registerUser(req, res);

    expect(statusStub.calledWith(201)).to.be.true;
}) ;

it("Should register user and return 201 if 'description' is missig", async() => {

    //body base
    req.body = { ...baseBody};
    //quitar description
    delete req.body['description'];

    await registerUser(req, res);
}) ;
```

```
expect(statusStub.calledWith(201)).to.be.true;
})

requiredFields.forEach((field) => {
  it(`should return 400 if '${field}' is missing`, async () => {
    // clonar body base
    req.body = { ...baseBody };

    // borrar SOLO ese campo
    delete req.body[field];

    await registerUser(req, res);

    expect(statusStub.calledWith(400)).to.be.true;

    const response = jsonStub.firstCall.args[0];
    expect(response.missing).to.include(field);
  });
});

}) ;
```

Explicación:

Estas pruebas son necesarias para cubrir los posibles escenarios de registro. Se valida que la función identifique datos faltantes, detecte duplicados en la base de datos y procese correctamente un registro válido. Además, se comprueba que la función devuelva los códigos HTTP correctos y mensajes coherentes.

2. Test para `loginUser`

La función `loginUser` valida credenciales del usuario y autentica el acceso. Para este describe se cubrieron casos clave como:

- Inicio de sesión exitoso.
- Email no encontrado.
- Contraseña incorrecta.
- Campos faltantes.
- Manejo de errores inesperados.

Fragmento del test:

```
it("should return 400 if 'email' is missing", async () => {

    const req: any = {

        body: {

            password: "123"

        }

    };

    const jsonStub = sinon.stub();

    const statusStub = sinon.stub().returns({ json: jsonStub });

    const res: any = {

        status: statusStub

    };

    await loginUser(req, res);

    expect(statusStub.calledWith(400)).to.be.true;

    const response = jsonStub.firstCall.args[0];

    expect(response.message).to.equal("Missing required field: email");

}) ;

it("should return 400 if 'password' is missing", async () => {

    const req: any = {

        body: {

            email: "juan@perez.com"

        }

    }
```

```
};

const jsonStub = sinon.stub();

    const statusStub = sinon.stub().returns({ json: jsonStub });

const res: any = {

    status: statusStub

};

await loginUser(req, res);

expect(statusStub.calledWith(400)).to.be.true;

const response = jsonStub.firstCall.args[0];

expect(response.message).to.equal("Missing required field:
password");

});

it("should return 404 if user email not found", async () => {

const req: any = {

    body: {

        email: "pepe@perez.com",

        password: "123"

    }

};

const jsonStub = sinon.stub();

    const statusStub = sinon.stub().returns({ json: jsonStub });

const res: any = {
```

```
    status: statusStub

};

fakeRepo.findOne.resolves(null);

await loginUser(req, res);

expect(statusStub.calledWith(404)).to.be.true;

const response = jsonStub.firstCall.args[0];

expect(response.message).to.equal("Invalid email");

}) ;

it("should return 401 if password is incorrect", async () => {

  const req: any = {

    body: {

      email: "pepe@perez.com",

      password: "123"

    }

  };

  const jsonStub = sinon.stub();

  const statusStub = sinon.stub().returns({ json: jsonStub });

  const res: any = {

    status: statusStub

  };

  await loginUser(req, res);

  expect(res.json).to.be.true;

  expect(statusStub.calledWith(401)).to.be.true;

  expect(res.status).to.equal(401);

  expect(res.body.message).to.equal("Invalid password");

});

});
```

```
    status: statusStub

};

sinon.stub(bcrypt, "compare").resolves(false);

await loginUser(req, res);

expect(statusStub.calledWith(404)).to.be.true;

const response = jsonStub.firstCall.args[0];

expect(response.message).to.equal("Invalid email");

}) ;

it("should return 500 on internal server error", async () => {

const req: any = {

  body: {

    email: "pedro@pecas.com",

    password: "123"

  }

};

const jsonStub = sinon.stub();

const statusStub = sinon.stub().returns({ json: jsonStub });

const res: any = {

  status: statusStub
```

```
};

fakeRepo.findOne.throws(new Error("DB error"));

await loginUser(req, res);

expect(statusStub.calledWith(500)).to.be.true;

const response = jsonStub.firstCall.args[0];

expect(response.message).to.equal("Internal server error");

}) ;
```

Explicación:

Las pruebas de este bloque verifican que el proceso de autenticación funcione de manera robusta ante situaciones reales: credenciales correctas, errores del usuario (correo incorrecto, contraseña errónea) y errores del servidor. Es fundamental asegurar que la función sea segura, devuelva códigos HTTP adecuados y no filtre información sensible.

3. Test de la función `calculateElevation`

Esta función toma una serie de coordenadas y calcula la diferencia máxima de altitud en un recorrido. Se probaron:

- Cálculo correcto con altitudes variadas.
- Casos donde todas las altitudes son iguales.
- Coordenadas sin altitud.
- Altitudes negativas.
- Arreglos con un solo elemento.
- Garantizar que nunca retorne un valor negativo.

```
describe("calculateElevation", () => {

    it("should return elevation difference for normal altitudes",
() => {

    const coords = [
```

```
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 100 },  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 150 },  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 120 }  
    ];  
  
    const result = calculateElevation(coords);  
  
    expect(result).to.equal(50);  
});  
  
  
  
it("should return 0 when all altitudes are equal", () => {  
  
    const coords = [  
  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 80 },  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 80 },  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 80 }  
    ];  
  
    const result = calculateElevation(coords);  
  
    expect(result).to.equal(0);  
});  
  
  
  
it("should ignore undefined altitudes", () => {  
  
    const coords = [  
  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: 100 },  
        { latitude: 0, longitude: 0, timestamp: 0, altitude: undefined }  
    ];  
  
    const result = calculateElevation(coords);  
  
    expect(result).to.equal(100);  
});
```

```
        { latitude: 0, longitude: 0, timestamp: 0 }, // sin
altitud

        { latitude: 0, longitude: 0, timestamp: 0, altitude: 90
}

];

const result = calculateElevation(coords);

expect(result).toEqual(10);

}) ;

it("should work with negative altitudes", () => {

const coords = [

{ latitude: 0, longitude: 0, timestamp: 0, altitude:
-50 },
{ latitude: 0, longitude: 0, timestamp: 0, altitude:
-10 },
{ latitude: 0, longitude: 0, timestamp: 0, altitude:
-30 }

];

const result = calculateElevation(coords);

expect(result).toEqual(40);

}) ;

it("should return 0 when array has only one coordinate", () =>
{

const coords = [

{ latitude: 0, longitude: 0, timestamp: 0, altitude:
123 }

];
```

```
const result = calculateElevation(coords);

expect(result).toEqual(0);

}) ;

it("should never return negative elevation", () => {

  const coords = [

    { latitude: 0, longitude: 0, timestamp: 0, altitude: 50
} ,

    { latitude: 0, longitude: 0, timestamp: 0, altitude: 50
}

] ;

  const result = calculateElevation(coords);

  expect(result).toEqual(0);

}) ;
```

Explicación:

Esta función se emplea para estimar la ganancia de elevación en un entrenamiento. Los tests permiten asegurar que trabaja correctamente incluso con datos incompletos, altitudes negativas, límites y casos triviales. También confirman que el resultado siempre sea lógico y no negativo, reforzando la fiabilidad del cálculo.

Tests palunag:

1. Validación de las Publicaciones

Este test lo podemos encontrar en `tests/entity/publicacion.test.ts` y se encarga de validar que una publicación cuente con todas las propiedades requeridas para ser insertado en nuestra base de datos sin problemas.

En los tests se llevan a cabo las siguientes verificaciones:

I. Campos obligatorios

- A. Probado: presencia de `counter`, `userEmail`, `trainingCounter` y `routeId`.
- B. Importancia: las claves primarias compuestas deben existir para garantizar integridad y unicidad en la base de datos; evita errores de inserción por claves faltantes.

II. Likes negativos (rechazado):

- A. Probado: guardar una publicación con `likes<0` lanza error.
- B. Importancia: protege la semántica del dato (un contador de “me gusta” no puede ser negativo) y evita cálculos o representaciones erróneas en la UI / estadísticas.

III. `trouteImage` verificación de longitud máxima:

- A. Probado: acepta exactamente 255 caracteres y rechaza `>255`.
- B. Importancia: respeta la restricción de la columna de la BBDD (`likes<=255``), evitando violaciones de esquema y ataques por entrada excesiva.

IV. `datetime` debe ser tipo Date (rechazado si string):

- A. Probado: valores no-`Date` son rechazados.
- B. Importancia: garantiza formatos y operaciones temporales correctas (ordenamiento, diferencias, zonas horarias), evitando errores en lógica que trabaja con fechas.

V. Duplicado de clave primaria compuesta (rechazado):

- A. Probado: intentar insertar dos publicaciones con la misma tupla (`counter, userEmail, trainingCounter, routeId`) falla.
- B. Importancia: evita duplicados lógicos y corrupción de datos; mantiene consistencia referencial esperada por la aplicación.

VI. likes por defecto a 0:

- A. Probado: si `likes` no se proporciona, el sistema lo normaliza a `0`.
- B. Importancia: evita `null` en contadores, facilitando operaciones aritméticas y que la UI no se rompa.

VII. Rango de privacy:

- A. Probado: valores fuera del rango permitido (en este caso `>2`) son rechazados; valores dentro aceptados.
- B. Importancia: asegura que el campo respete el contrato del dominio (los enums de privacidad válidos) y evita estados inválidos en la capa de presentación o lógica de autorización.

TypeScript

```
import { expect } from 'chai';
import { describe, it, before, after } from 'mocha';

describe('Publication entity validation', () => {
    let originalGetRepository: any;
    let pubRepo: any;

    before(async () => {
        const mod = await import('../src/db_connection/config(dataSource)');
        const ds = (mod as any).default ?? mod;
        originalGetRepository = ds.appDataSource.getRepository;

        let savedCounter = 1;
        const savedRecords: any[] = [];

        const fakePublicationRepo = {
            create(obj: any) { return { ...obj }; },
            async save(obj: any) {
                // required composite parts
                if (obj.counter == null) {
                    throw new Error('counter is required');
                }
                if (!obj.userEmail) {
                    throw new Error('userEmail is required');
                }
                if (obj.trainingCounter == null) {
                    throw new Error('trainingCounter is
required');
                }
                if (obj.routeId == null) {
                    throw new Error('routeId is required');
                }

                if (obj.likes != null && (typeof obj.likes !==
'number' || obj.likes < 0)) {
                    throw new Error('likes must be a
non-negative number');
                }

                if (obj.routeImage != null &&
obj.routeImage.length > 255) {
                    throw new Error('routeImage too long');
                }

                if (obj.datetime && !(obj.datetime instanceof
Date)) {
                    throw new Error('datetime must be a Date');
                }
            }
        };
        originalGetRepository = fakePublicationRepo;
    });
})
```

```
        }

            if (obj.privacy != null && (typeof obj.privacy !==
'number' || obj.privacy < 0 || obj.privacy > 2)) {
                throw new Error('privacy out of bounds');
            }

            const exists = savedRecords.find(r => r.counter
=== obj.counter && r.userEmail === obj.userEmail && r.trainingCounter ===
obj.trainingCounter && r.routeId === obj.routeId);
            if (exists) {
                throw new Error('duplicate publication
primary key');
            }

            if (obj.likes == null) {
                obj.likes = 0;
            }

            const toSave = { ...obj, _id: savedCounter++ };
            savedRecords.push(toSave);
            return toSave;
        }
    };

    ds.appDataSource.getRepository = (entity: any) => {
        const name = entity && entity.name ? entity.name :
String(entity);
        if (name && name.includes('Publication')) {
            return fakePublicationRepo;
        }
        return {
            create: (o: any) => ({ ...o }),
            save: async (o: any) => ({ ...o })
        };
    };
}

pubRepo = ds.appDataSource.getRepository({ name: 'Publication'
});
});

after(async () => {
    try {
        const mod = await
import('../src/db_connection/config/dataSource');
        const ds = (mod as any).default ?? mod;
        if (originalGetRepository) {
```

```
        ds.appDataSource.getRepository =
originalGetRepository;
    }
} catch (error) {
    console.error('Error restoring original getRepository:', error);
}
});

it('succeeds when all required Publication fields are present', async () => {
    const publication = pubRepo.create({
        counter: 1,
        likes: 0,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'test@example.com',
        trainingCounter: 1,
        routeId: 1
    });

    const saved = await pubRepo.save(publication);
    expect(saved).to.be.an('object');
    expect(saved).to.have.property('counter');
    expect(saved).to.have.property('_id');
});

it('fails when a required primary column is missing', async () => {
    const incomplete = pubRepo.create({
        counter: 2,
        likes: 0,
        privacy: 0,
        datetime: new Date(),
        trainingCounter: 1,
        routeId: 1
    });

    let threw = false;
    try {
        await pubRepo.save(incomplete);
    } catch (err) {
        threw = true;
        expect(err).to.exist;
    }

    expect(threw).to.equal(true);
});
```

```
it('rejects negative likes', async () => {
    const bad = pubRepo.create({
        counter: 10,
        likes: -5,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'neg@local',
        trainingCounter: 99,
        routeId: 5
    });

    let threw = false;
    try {
        await pubRepo.save(bad);
    } catch (err) {
        threw = true;
        expect(err).to.exist;
    }

    expect(threw).to.equal(true);
});

it('accepts routeImage at max length (255)', async () => {
    const img = 'a'.repeat(255);
    const item = pubRepo.create({
        counter: 11,
        likes: 1,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'img@local',
        trainingCounter: 100,
        routeId: 6,
        routeImage: img
    });

    const saved = await pubRepo.save(item);
    expect(saved).to.have.property('routeImage');
    expect(saved.routeImage.length).to.equal(255);
});

it('rejects routeImage longer than 255', async () => {
    const img = 'b'.repeat(256);
    const item = pubRepo.create({
        counter: 12,
        likes: 1,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'img2@local',
    });
});
```

```
        trainingCounter: 101,
        routeId: 7,
        routeImage: img
    });

    let threw = false;
    try { await pubRepo.save(item); } catch (e) { threw = true;
expect(e).to.exist; }
    expect(threw).to.equal(true);
});

it('rejects non-Date datetime values', async () => {
    const item = pubRepo.create({
        counter: 13,
        likes: 0,
        privacy: 0,
        datetime: '2025-11-17',
        userEmail: 'date@local',
        trainingCounter: 102,
        routeId: 8
    });

    let threw = false;
    try { await pubRepo.save(item); } catch (e) { threw = true;
expect(e).to.exist; }
    expect(threw).to.equal(true);
});

it('enforces unique composite primary key', async () => {
    const base = pubRepo.create({
        counter: 20,
        likes: 2,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'dup@local',
        trainingCounter: 200,
        routeId: 10
    });

    const first = await pubRepo.save(base);
expect(first).to.have.property('_id');

    const dup = pubRepo.create({ ...base });

    let threw = false;
    try { await pubRepo.save(dup); } catch (e) { threw = true;
expect(e).to.exist; }
    expect(threw).to.equal(true);
});
```

```
});

it('defaults missing likes to 0', async () => {
    const item = pubRepo.create({
        counter: 30,
        privacy: 0,
        datetime: new Date(),
        userEmail: 'nolikes@local',
        trainingCounter: 300,
        routeId: 20
    });

    const saved = await pubRepo.save(item);
    expect(saved).to.have.property('likes');
    expect(saved.likes).to.equal(0);
});

it('privacy bounds: rejects out-of-range values, accepts valid', async () => {
    const bad = pubRepo.create({ counter: 40, likes: 0, privacy: 3, datetime: new Date(), userEmail: 'p@local', trainingCounter: 400, routeId: 30 });
    let threw = false;
    try { await pubRepo.save(bad); } catch (e) { threw = true; expect(e).to.exist; }
    expect(threw).to.equal(true);

    const ok = pubRepo.create({ counter: 41, likes: 0, privacy: 1, datetime: new Date(), userEmail: 'p2@local', trainingCounter: 401, routeId: 31 });
    const saved = await pubRepo.save(ok);
    expect(saved).to.have.property('privacy');
    expect(saved.privacy).to.equal(1);
});
});
```

2. Validación del AuthController

Este test lo podemos encontrar en `tests/controllers/authController.test.ts` y tiene como responsabilidad revisar y validar las entradas del controlador que desarrollamos con demasiadas lágrimas y sudor para integrar el OAuth de los locos de Google

En los tests se llevan a cabo las siguientes verificaciones:

- I. **`idToken` ausente → 400**
 - A. Probado: si la petición no incluye `idToken`, el controlador responde con HTTP `400` y un mensaje de error.
 - B. Importancia: protege la API de requests inválidos y evita llamadas innecesarias a la verificación del token. Garantiza respuestas claras para clientes.
- II. **`idToken` válido, un nuevo usuario → 200 + user**
 - A. Probado: cuando `verifyIdToken` devuelve una payload con `email` y no hay usuario en la base (repo devuelve `null`), el controlador crea al usuario y responde `200` junto con los datos esperados del usuario.
 - B. Importancia: cubre el flujo de primer acceso via Google OAuth (crear y devolver datos de usuario), crítico para la experiencia de onboarding.
- III. **`idToken` válido, con usuario existente → 200 + user**
 - A. Probado: si el repo devuelve un usuario existente, el controlador devuelve `200` con los datos de ese usuario sin intentar crear uno nuevo.
 - B. Importancia: asegura que el endpoint no crea duplicados y que devuelve datos correctos para usuarios ya registrados.
- IV. **Fallo al guardar nuevo usuario → 500**
 - A. Probado: si `userRepository.save` lanza una excepción, el controlador atrapa el error y responde `500` con un mensaje genérico.
 - B. Importancia: valida manejo de errores en la capa de persistencia y evita exponer detalles internos; asegura que la API responde correctamente en fallos internos.
- V. **Payload sin `email` → 400**
 - A. Probado: si `verifyIdToken` devuelve un `payload` inválido o sin `email`, el controlador responde `400` con `Invalid token payload`.
 - B. Importancia: garantiza que sólo se aceptan tokens con la información mínima requerida (`email`) y evita crear usuarios sin identificador único.
- VI. **`verifyIdToken` lanza (error durante verificación) → 500**
 - A. Probado: si el cliente de verificación de Google lanza (p. ej. error de red o formato), el controlador devuelve `500`.
 - B. Importancia: comprueba la robustez frente a errores externos y asegura que la aplicación no falle de forma inesperada ante problemas en la verificación de tokens.

```
TypeScript
import { expect } from 'chai';
import { describe, it, before, after } from 'mocha';
import sinon from 'sinon';

describe('AuthController (basic)', () => {
    let originalGetRepository: any;
    let verifyStub: sinon.SinonStub;
    let fakeUserRepo: any;

    before(async () => {
        const ga = await import('google-auth-library');
        verifyStub = sinon.stub(ga.OAuth2Client.prototype,
        'verifyIdToken');

        // Patch appDataSource.getRepository to return a fake user repo
before
        // importing the controller so the module-level repository is
set to the fake.

        const dsMod = await
import('../src/db_connection/config/dataSource');
        originalGetRepository = dsMod.appDataSource.getRepository;
        // create a shared fake repo so tests can mutate its behavior
per-case
        fakeUserRepo = {
            findOne: async (opts: any) => null,
            create: (u: any) => ({ ...u }),
            save: async (u: any) => ({ ...u })
        };

        dsMod.appDataSource.getRepository = (entity: any) => {
            const name = entity && entity.name ? entity.name :
String(entity);
            if (name && name.includes('User')) {
                return fakeUserRepo;
            }
            return originalGetRepository(entity);
        };
    });

    after(async () => {
        if (verifyStub && verifyStub.restore) {verifyStub.restore();}
        try {
            const dsMod = await
import('../src/db_connection/config/dataSource');
            if (originalGetRepository)
{dsMod.appDataSource.getRepository = originalGetRepository;}
        } catch (error) {
```

```
        console.error(error);
    }
});

it('returns 400 when idToken missing', async () => {
    const controller = await
import('../src/db_connection/controller/AuthController');
    const req: any = { body: {} };
    const res: any = { statusCode: 0, body: null, status(code:
number) { this.statusCode = code; return this; }, json(obj: any) { this.body
= obj; return this; } };

    await controller.verifyGoogleToken(req, res);

    expect(res.statusCode).to.equal(400);
    expect(res.body).to.have.property('message');
});

it('returns 200 and user when idToken valid (new user)', async () => {
    verifyStub.resolves({ getPayload: () => ({ email: 'new@local',
given_name: 'New', family_name: 'User', picture: 'http://p' }) });

    const controller = await
import('../src/db_connection/controller/AuthController');
    const request: any = { body: { idToken: 'valid-token' } };
    const response: any = { statusCode: 0, body: null, status(code:
number) { this.statusCode = code; return this; }, json(obj: any) { this.body
= obj; return this; } };

    await controller.verifyGoogleToken(request, response);

    expect(response.statusCode).to.equal(200);
    expect(response.body).to.have.property('user');
    expect(response.body.user).to.have.property('email',
'new@local');
});

it('returns 200 and user when idToken valid (existing user)', async () => {
    fakeUserRepo.findOne = async () => ({
        email: 'exist@local',
        username: 'exist',
        names: 'Exist',
        lastNames: 'User',
        profilePhoto: undefined,
        description: undefined,
        gender: undefined
    });
});
```

```
        });

        verifyStub.resolves({ getPayload: () => ({ email:
'exist@local', given_name: 'Exist', family_name: 'User', picture: undefined
}) });

        const controller = await
import('../src/db_connection/controller/AuthController');
        const request: any = { body: { idToken: 'valid-existing' } };
        const response: any = { statusCode: 0, body: null, status(code:
number) { this.statusCode = code; return this; }, json(obj: any) { this.body
= obj; return this; } };

        await controller.verifyGoogleToken(request, response);

        expect(response.statusCode).to.equal(200);
        expect(response.body).to.have.property('user');
        expect(response.body.user).to.have.property('email',
'email');
        expect(response.body.user.email).to.equal('exist@local');
    });

it('returns 500 when saving new user fails', async () => {
    // simulate no existing user and save throws
    fakeUserRepo.findOne = async () => null;
    fakeUserRepo.save = async () => { throw new Error('DB
failure'); };
    verifyStub.resolves({ getPayload: () => ({ email: 'fail@local',
given_name: 'Fail', family_name: 'User', picture: undefined }) });

    const controller = await
import('../src/db_connection/controller/AuthController');
    const request: any = { body: { idToken: 'will-fail' } };
    const response: any = { statusCode: 0, body: null, status(code:
number) { this.statusCode = code; return this; }, json(obj: any) { this.body
= obj; return this; } };

    await controller.verifyGoogleToken(request, response);

    expect(response.statusCode).to.equal(500);
    expect(response.body).to.have.property('message');
});

it('returns 400 when verifyIdToken payload has no email', async () =>
{
    fakeUserRepo.findOne = async () => null;
    fakeUserRepo.save = async (u: any) => ({ ...u });
    // verifyIdToken returns payload without email
});
```

```
    verifyStub.resolves({ getPayload: () => ({ given_name: 'NoEmail' }) });

        const controller = await
import('../src/db_connection/controller/AuthController');
        const request: any = { body: { idToken: 'no-email' } };
        const response: any = { statusCode: 0, body: null, status(code: number) { this.statusCode = code; return this; }, json(obj: any) { this.body = obj; return this; } };

        await controller.verifyGoogleToken(request, response);

        expect(response.statusCode).to.equal(400);
        expect(response.body).to.have.property('message');
    });

it('returns 500 when verifyIdToken throws', async () => {
    // simulate crypto/verification error
    verifyStub.rejects(new Error('verify failure'));

        const controller = await
import('../src/db_connection/controller/AuthController');
        const request: any = { body: { idToken: 'will-throw' } };
        const response: any = { statusCode: 0, body: null, status(code: number) { this.statusCode = code; return this; }, json(obj: any) { this.body = obj; return this; } };

        await controller.verifyGoogleToken(request, response);

        expect(response.statusCode).to.equal(500);
        expect(response.body).to.have.property('message');
    });
});
```

3. Validación de la entidad de Comentarios

Este test lo podemos encontrar en `tests/entity/comment.test.ts` y tiene como responsabilidad revisar y validar las entidades que se pueden crear con el esquema definido en la base de datos para los comentarios que pueden realizar los usuarios en las rutas compartidas, i.e. las publicaciones de la app.

En los tests se llevan a cabo las siguientes verificaciones:

I. Columnas primarias obligatorias

- A. Probado: Falla si falta cualquiera de las columnas que componen la clave primaria (por ejemplo, `userEmail` o `publicationCounter`).
- B. Importancia: La clave primaria identifica un `Comment` de forma única. Insertar sin todos los valores rompe la integridad relacional y puede provocar inconsistencias o sobrescrituras.

II. Texto obligatorio no vacío (`text` no vacío):

- A. Probado: Rechazo si `text` está vacío o compuesto solo por espacios.
- B. Importancia: Un comentario sin contenido es inválido desde el punto de vista de negocio; además, evita almacenamiento de datos inútiles y UX confusa.

III. Likes no negativos (`likes>=0`):

- A. Probado: Rechazo cuando `likes` es negativo.
- B. Importancia: El número de `likes` debe representar una cuenta no negativa; valores negativos implican corrupción de estado o errores lógicos en la capa de negocio.

IV. Valor por defecto para `likes` (cuando falta):

- A. Probado: Si `likes` no se provee, se setea a `0`.
- B. Importancia: Evita `null` o `undefined` y asegura consistencia en cálculos posteriores (suma/ordenamiento). Refleja el comportamiento esperado por la aplicación.

V. Rechazo de `dateTime` no `Date`:

- A. Probado: Rechazo cuando `dateTime` existe pero no es una instancia de `Date`.
- B. Importancia: `dateTime` debe ser un objeto de fecha para operaciones (ordenación, formateo, comparaciones). Datos mal tipados conducen a errores en runtime.

VI. `dateTime` por defecto cuando falta:

- A. Probado: Si `dateTime` no se proporciona en la inserción, se rellena con `new Date()` (instante actual).
- B. Importancia: Simula el comportamiento de `CreateDateColumn` de TypeORM; asegura que todos los comentarios tengan timestamp, facilitando el orden.

VII. Rechazo de clave primaria duplicada (composite PK):

- A. Probado: Insertar un `Comment` con la misma combinación de claves primarias que uno existente falla.
- B. Importancia: Evita duplicidades y mantiene la integridad de la colección de comentarios.

VIII. userEmail excede longitud máxima (> 100):

- A. Probado: Rechazo cuando `userEmail` supera 100 caracteres.
- B. Importancia: La definición de la entidad fija `length: 100` en la columna primaria `userEmail`. Permitir valores más largos puede provocar errores en la base de datos o truncamientos inesperados; estas pruebas detectan entradas inválidas temprano.

TypeScript

```

import { expect } from 'chai';
import { describe, it, before, after } from 'mocha';

describe('Comment entity validation', () => {
    let originalGetRepository: any;
    let commentRepo: any;

    before(async () => {
        const mod = await
import('../src/db_connection/config(dataSource');
        const ds = (mod as any).default ?? mod;
        originalGetRepository = ds.appDataSource.getRepository();

        let savedCounter = 1;
        const savedRecords: any[] = [];

        const fakeCommentRepo = {
            create(obj: any) { return { ...obj }; },
            async save(obj: any) {
                if (obj.publicationCounter == null) {
                    throw new Error('publicationCounter is
required');
                }
                if (!obj.userEmail) {
                    throw new Error('userEmail is required');
                }

                // enforce length limit similar to entity
definition
                if (typeof obj.userEmail === 'string' &&
obj.userEmail.length > 100) {
                    throw new Error('userEmail too long');
                }
                if (obj.trainingCounter == null) {
                    throw new Error('trainingCounter is
required');
                }
                if (obj.routeId == null) {
                    throw new Error('routeId is required');
                }
            }
        }
        commentRepo = fakeCommentRepo;
    });

    it('should validate user email length', async () => {
        const comment = await originalGetRepository.create();
        comment.userEmail = 'a'.repeat(101);
        const result = await originalGetRepository.save(comment);
        expect(result).to.be.undefined;
    });
});

```

```
        if (obj.counter == null) {
            throw new Error('counter is required');
        }

        // text required and not empty
        if (!obj.text || typeof obj.text !== 'string' ||
obj.text.trim().length === 0) {
            throw new Error('text is required');
        }

        // likes must be non-negative integer if provided
        if (obj.likes != null && (typeof obj.likes !==
'number' || obj.likes < 0)) {
            throw new Error('likes must be
non-negative');
        }

        // datetime if provided must be a Date
        if (obj.datetime != null && !(obj.datetime
instanceof Date)) {
            throw new Error('datetime must be a Date');
        }

        // If datetime is missing, default to now (mirrors
CreateDateColumn behavior)
        if (obj.datetime == null) {
            obj.datetime = new Date();
        }

        // TODO: See how to simplify this search callback
lol
        const exists = savedRecords.find(r =>
r.publicationCounter === obj.publicationCounter && r.userEmail ===
obj.userEmail && r.trainingCounter === obj.trainingCounter && r.routeId ===
obj.routeId && r.counter === obj.counter);
        if (exists) {
            throw new Error('duplicate comment primary
key');
        }

        if (obj.likes == null) {
            obj.likes = 0;
        }

        const toSave = { ...obj, _id: savedCounter++ };
        savedRecords.push(toSave);
        return toSave;
    }
}
```

```
};

ds.appDataSource.getRepository = (entity: any) => {
    const name = entity && entity.name ? entity.name :
String(entity);
    if (name && name.includes('Comment')) {
        return fakeCommentRepo;
    }
    return {
        create: (o: any) => ({ ...o }),
        save: async (o: any) => ({ ...o })
    };
};

commentRepo = ds.appDataSource.getRepository({ name: 'Comment' });
});

after(async () => {
    try {
        const mod = await
import('../src/db_connection/config(dataSource');
        const ds = (mod as any).default ?? mod;
        if (originalGetRepository) {
            ds.appDataSource.getRepository =
originalGetRepository;
        }
    } catch (error) {
        console.error(error);
    }
});
it('succeeds when all required Comment fields are present', async () => {
    const comment = commentRepo.create({
        publicationCounter: 1,
        userEmail: 'c@local',
        trainingCounter: 1,
        routeId: 1,
        counter: 1,
        text: 'Nice!',
        likes: 0,
        datetime: new Date()
    });

    const saved = await commentRepo.save(comment);
    expect(saved).to.be.an('object');
    expect(saved).to.have.property('_id');
```

```
expect(saved.text).to.equal('Nice!');");
});

it('fails when a required primary column is missing', async () => {
    const incomplete = commentRepo.create({
        // missing userEmail
        publicationCounter: 2,
        trainingCounter: 2,
        routeId: 2,
        counter: 1,
        text: 'Hello'
    });

    let threw = false;
    try { await commentRepo.save(incomplete); } catch (e) { threw = true; expect(e).to.exist; }
    expect(threw).to.equal(true);
});

it('rejects empty text', async () => {
    const item = commentRepo.create({
        publicationCounter: 3,
        userEmail: 't@local',
        trainingCounter: 3,
        routeId: 3,
        counter: 1,
        text: ' '
    });
    let threw = false;
    try {
        await commentRepo.save(item);
    } catch (error) {
        threw = true; expect(error).to.exist;
    }
    expect(threw).to.equal(true);
});

it('rejects negative likes', async () => {
    const item = commentRepo.create({
        publicationCounter: 4,
        userEmail: 'l@local',
        trainingCounter: 4,
        routeId: 4,
        counter: 1,
        text: 'ok',
        likes: -1
    });
    let threw = false;
```

```
try {
    await commentRepo.save(item);
} catch (error) {
    threw = true; expect(error).to.exist;
}
expect(threw).to.equal(true);
});

it('defaults likes to 0 when missing', async () => {
    const item = commentRepo.create({
        publicationCounter: 5,
        userEmail: 'nl@local',
        trainingCounter: 5,
        routeId: 5,
        counter: 1,
        text: 'no likes'
    });
    const saved = await commentRepo.save(item);
    expect(saved).to.have.property('likes');
    expect(saved.likes).to.equal(0);
});

it('sets datetime to now when missing', async () => {
    const item = commentRepo.create({
        publicationCounter: 50,
        userEmail: 'dt@local',
        trainingCounter: 50,
        routeId: 50,
        counter: 1,
        text: 'auto date'
    });
    const saved = await commentRepo.save(item);
    expect(saved).to.have.property('datetime');
    expect(saved.datetime).to.be.instanceOf(Date);
    // saved.datetime should be recent (within a minute)
    expect(Date.now() - saved.datetime.getTime()).to.be.lessThan(60
* 1000);
});

it('rejects duplicate composite primary key', async () => {
    const base = commentRepo.create({
        publicationCounter: 6,
        userEmail: 'dup@local',
        trainingCounter: 6,
        routeId: 6,
        counter: 1,
        text: 'first'
});
```

```
});

const first = await commentRepo.save(base);
expect(first).to.have.property('_id');

const dup = commentRepo.create({
    publicationCounter: 6,
    userEmail: 'dup@local',
    trainingCounter: 6,
    routeId: 6,
    counter: 1,
    text: 'second'
});
let threw = false;
try {
    await commentRepo.save(dup);
} catch (error) {
    threw = true; expect(error).to.exist;
}
expect(threw).to.equal(true);
});

it('rejects userEmail longer than 100 characters', async () => {
    const longEmail = 'a'.repeat(101) + '@local';
    const item = commentRepo.create({
        publicationCounter: 61,
        userEmail: longEmail,
        trainingCounter: 61,
        routeId: 61,
        counter: 1,
        text: 'too long email'
    });
    let threw = false;
    try {
        await commentRepo.save(item);
    } catch (error) {
        threw = true; expect(error).to.exist;
    }
    expect(threw).to.equal(true);
});

it('rejects non-Date datetime values', async () => {
    const item = commentRepo.create({
        publicationCounter: 7,
        userEmail: 'd@local',
        trainingCounter: 7,
        routeId: 7,
        counter: 1,
        text: 'date',
    });
});
```

```
        datetime: '2025-11-17'  
    });  
    let threw = false;  
    try {  
        await commentRepo.save(item);  
    } catch (error) {  
        threw = true; expect(error).to.exist;  
    }  
    expect(threw).to.equal(true);  
});  
});
```