UCC- Universidad Católica de Córdoba

Facultad de Ingeniería

INGENIERÍA DE SOFTWARE III

Trabajo Práctico VI

Integrantes:

- Cuozzo, Sofia
- Hernandez, Simon

Docente:

- Schwindt, Ariel
- Bono, Fernando

Trabajo Práctico 06 – Pruebas Unitarias

---

1. Introducción

En el TP06 el objetivo fue agregar una **suite de pruebas unitarias** y conectarla al **pipeline de CI/CD** construido en el TP05.

En concreto:

- Implementar tests para el **backend** (API de tareas).
- Implementar tests para el **frontend** (servidor que sirve la UI).
- Utilizar el patrón **AAA (Arrange – Act – Assert)** en todos los casos.
- Aislar dependencias externas: en particular, la **base de datos** se reemplaza por un **MongoDB en memoria**.
- Integrar la ejecución de `npm test` de backend y frontend al **pipeline de Azure DevOps** para que:
    - **No se construya ni despliegue nada** si los tests fallan.
    - Se publiquen **resultados y cobertura**.

---

2. Frameworks y estrategia de testing

2.1. Stack de backend

Para el backend se utilizaron:

- **Jest** como framework de testing de JavaScript.
- **Supertest** para simular requests HTTP contra la API Express.
- **mongodb-memory-server** para levantar un **MongoDB en memoria** durante los tests.
- **Mongoose** igual que en la aplicación real, pero apuntando a la base en memoria.

2.2. Stack de frontend

Para el frontend se utilizaron:

- **Jest** también como runner.
- **Supertest** para probar el servidor Express del frontend:
    - Respuesta de `/`.
    - Health check `/health`.
    - Servido de archivos estáticos.
    - Manejo de 404.

2.3. Patrón AAA (Arrange – Act – Assert)

Todos los tests siguen la estructura:

1. **Arrange:** preparar el escenario
   (crear tareas en la base en memoria, definir request body, generar IDs, etc.).
2. **Act:** ejecutar la acción
   (hacer `request(app).get(...)`, `post(...)`, etc.).
3. **Assert:** verificar el resultado
   (status code, body, longitud de lista, mensajes de error).

Ejemplo simple de patrón AAA (backend):

```
it('should create a new task successfully', async () => {
  // ARRANGE
  const newTask = { title: 'Nueva tarea', description: 'Descripción
de prueba', completed: false };

  // ACT
  const response = await
request(app).post('/api/tasks').send(newTask);

  // ASSERT
  expect(response.status).toBe(201);
  expect(response.body.title).toBe(newTask.title);
});
```

---

3. Tests del backend

El archivo de tests del backend importa:

```
const request = require('supertest');
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');
const { app, Task } = require('../app');
```

- app: instancia de Express con todas las rutas (`/api/tasks`, `/healthz`, etc.).
- `Task`: modelo de Mongoose para la colección de tareas.

3.1. Setup global: MongoDB en memoria

Bloque de configuración:

```
let mongoServer;

// ARRANGE: Configurar MongoDB en memoria antes de todos los tests
beforeAll(async () => {
  mongoServer = await MongoMemoryServer.create();
  const mongoUri = mongoServer.getUri();
  await mongoose.connect(mongoUri, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  });
});

// Limpiar la BD entre tests
beforeEach(async () => {
  await Task.deleteMany({});
});

// Cerrar conexiones después de todos los tests
afterAll(async () => {
  await mongoose.disconnect();
  await mongoServer.stop();
```

```
});
```

Qué hace:

- `beforeAll`:
    - Crea una instancia de **Mongo en memoria** (`MongoMemoryServer.create()`).
    - Obtiene el `mongoUri` y conecta Mongoose a esa base.
- `beforeEach`:
    - Limpia la colección `Task` antes de cada test para que **cada test sea independiente** (no comparten datos).
- `afterAll`:
    - Cierra la conexión de Mongoose.
    - Apaga el servidor en memoria.

Por qué:

- Aísla totalmente los tests del backend de cualquier base real (Atlas/local).
- Los tests son reproducibles, rápidos y no necesitan red.

3.2. Suite: `API Tasks - CRUD Operations`

```
describe('API Tasks - CRUD Operations', () => {
  ...
});
```

Dentro de este `describe` están todas las pruebas sobre la API `/api/tasks` y el health check.

3.2.1. GET /api/tasks

```
describe('GET /api/tasks', () => {
  it('should return empty array when no tasks exist', async () => {
    // ACT
```

```
    const response = await request(app).get('/api/tasks');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body).toEqual([]);
  });

  it('should return all tasks', async () => {
    // ARRANGE
    await Task.create([
      { title: 'Task 1', description: 'Description 1' },
      { title: 'Task 2', description: 'Description 2' }
    ]);

    // ACT
    const response = await request(app).get('/api/tasks');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body).toHaveLength(2);
    expect(response.body[0].title).toBe('Task 2'); // Orden
descendente por createdAt
  });
});
```

Qué se prueba:

1. **Lista vacía**:
    - Sin datos en la base (gracias al `beforeEach`), un GET a `/api/tasks` debe devolver:
        - `status 200`.
        - `body = []`.
2. **Lista con datos**:
    - Arrange: se crean dos tareas directamente en la DB en memoria.
    - Act: se hace GET `/api/tasks`.
    - Assert:

- `status 200`.
- Longitud 2.
- El primer elemento tiene título `'Task 2'` → verifica que la API ordena por `createdAt` descendente.

3.2.2. POST /api/tasks

```
describe('POST /api/tasks', () => {
  it('should create a new task successfully', async () => {
    // ARRANGE
    const newTask = {
      title: 'Nueva tarea',
      description: 'Descripción de prueba',
      completed: false
    };

    // ACT
    const response = await request(app)
      .post('/api/tasks')
      .send(newTask);

    // ASSERT
    expect(response.status).toBe(201);
    expect(response.body.title).toBe(newTask.title);
    expect(response.body.description).toBe(newTask.description);
    expect(response.body.completed).toBe(false);
    expect(response.body._id).toBeDefined();
  });

  it('should fail when title is missing', async () => {
    // ARRANGE
    const invalidTask = { description: 'Sin título' };

    // ACT
    const response = await request(app)
      .post('/api/tasks')
      .send(invalidTask);

    // ASSERT
```

```
    expect(response.status).toBe(400);
    expect(response.body.error).toBe('El título es requerido');
  });

  it('should fail when title is empty string', async () => {
    // ARRANGE
    const invalidTask = {
      title: '   ',
      description: 'Título vacío'
    };

    // ACT
    const response = await request(app)
      .post('/api/tasks')
      .send(invalidTask);

    // ASSERT
    expect(response.status).toBe(400);
    expect(response.body.error).toBe('El título es requerido');
  });
});
```

Qué se prueba:

- Caso feliz: crear una tarea válida → `201` y body con los datos y `_id`.
- Casos de validación:
    - Sin `title`.
    - `title` vacío/espacios.
- En ambos casos de error:
    - `status 400`.
    - Mensaje `'El título es requerido'`.

Valor: asegura la **lógica de validación** de la API, no solo el CRUD plano.

3.2.3. GET /api/tasks/:id

```
describe('GET /api/tasks/:id', () => {
  it('should return task by id', async () => {
    // ARRANGE
    const task = await Task.create({
      title: 'Test Task',
      description: 'Test Description'
    });

    // ACT
    const response = await
request(app).get(`/api/tasks/${task._id}`);

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body.title).toBe('Test Task');
    expect(response.body._id).toBe(task._id.toString());
  });

  it('should return 404 for non-existent task', async () => {
    // ARRANGE
    const fakeId = new mongoose.Types.ObjectId();

    // ACT
    const response = await request(app).get(`/api/tasks/${fakeId}`);

    // ASSERT
    expect(response.status).toBe(404);
    expect(response.body.error).toBe('Tarea no encontrada');
  });
});
```

Qué se prueba:

- Si existe una tarea con ese `_id`, la API:
    - Devuelve `200`.
    - Devuelve el JSON correcto.
- Si el `_id` no existe, la API:

- Devuelve 404.
- Mensaje 'Tarea no encontrada'.

3.2.4. PUT /api/tasks/:id

```
describe('PUT /api/tasks/:id', () => {
  it('should update task successfully', async () => {
    // ARRANGE
    const task = await Task.create({
      title: 'Original Title',
      description: 'Original Description',
      completed: false
    });

    const updates = {
      title: 'Updated Title',
      description: 'Updated Description',
      completed: true
    };

    // ACT
    const response = await request(app)
      .put(`/api/tasks/${task._id}`)
      .send(updates);

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body.title).toBe('Updated Title');
    expect(response.body.completed).toBe(true);
  });

  it('should return 404 when updating non-existent task', async ()
=> {
    // ARRANGE
    const fakeId = new mongoose.Types.ObjectId();

    // ACT
    const response = await request(app)
      .put(`/api/tasks/${fakeId}`)
```

```
      .send({ title: 'Updated' });

    // ASSERT
    expect(response.status).toBe(404);
  });
});
```

Qué se prueba:

- Caso feliz: una tarea existente se actualiza correctamente.
- Caso de error: intentar actualizar un ID inexistente → `404`.

3.2.5. DELETE /api/tasks/:id

```
describe('DELETE /api/tasks/:id', () => {
  it('should delete task successfully', async () => {
    // ARRANGE
    const task = await Task.create({
      title: 'Task to delete',
      description: 'Will be deleted'
    });

    // ACT
    const response = await
request(app).delete(`/api/tasks/${task._id}`);

    // ASSERT
    expect(response.status).toBe(204);

    // Verificar que realmente se eliminó
    const deletedTask = await Task.findById(task._id);
    expect(deletedTask).toBeNull();
  });

  it('should return 404 when deleting non-existent task', async ()
=> {
    // ARRANGE
    const fakeId = new mongoose.Types.ObjectId();
```

```
    // ACT
    const response = await
request(app).delete(`/api/tasks/${fakeId}`);

    // ASSERT
    expect(response.status).toBe(404);
  });
});
```

Qué se prueba:

- El delete exitoso devuelve `204` y la tarea realmente desaparece de la DB.
- El delete sobre un `_id` inexistente devuelve `404`.

3.2.6. GET /healthz

```
describe('GET /healthz', () => {
  it('should return health status', async () => {
    // ACT
    const response = await request(app).get('/healthz');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body.status).toBe('OK');
    expect(response.body.db).toBe('connected');
  });
});
```

Qué se prueba:

- Que la ruta `/healthz` existe y responde.
- Que el JSON indica:
    - `status: 'OK'`.
    - `db: 'connected'` (Mongoose conectado al Mongo en memoria).

Esto engancha con el **health check del TP05**.

---

4. Tests del frontend

Archivo:

```
const request = require('supertest');
const app = require('../app');
```

El app acá es el servidor Express del frontend (sirve estáticos y un pequeño health check).

4.1. GET /

```
describe('GET /', () => {
  it('should return HTML page (status 200)', async () => {
    // ACT
    const response = await request(app).get('/');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.headers['content-type']).toMatch(/html/);
  });
});
```

Qué se prueba:

- Que la ruta raíz responde `200`.
- Que el `content-type` es HTML.
  Es decir, efectivamente sirve la página principal.

4.2. GET /health

```
describe('GET /health', () => {
  it('should return health status', async () => {
    // ACT
    const response = await request(app).get('/health');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.body).toEqual({
      status: 'OK',
```

```
      service: 'frontend'
    });
  });
});
```

Qué se prueba:

- Que el servidor de frontend expone un health check simple:
    - `status: 200`.
    - Body `{ status: "OK", service: "frontend" }`.

4.3. Archivos estáticos

```
describe('Static files', () => {
  it('should serve index.html', async () => {
    // ACT
    const response = await request(app).get('/index.html');

    // ASSERT
    expect(response.status).toBe(200);
    expect(response.headers['content-type']).toMatch(/html/);
  });
});
```

Qué se prueba:

- Que `index.html` se sirve correctamente desde la carpeta `public`.

4.4. Manejo de 404

```
describe('404 handling', () => {
  it('should return 404 for non-existent routes', async () => {
    // ACT
    const response = await request(app).get('/non-existent-route');

    // ASSERT
    expect(response.status).toBe(404);
  });
});
```

Qué se prueba:

- Que rutas inexistentes devuelven `404` y no explota el servidor.

---

5. Pipeline de CI/CD con tests integrados

El pipeline del TP06 es la evolución del TP05. Ahora tiene **4 stages**:

1. `Tests` → ejecutar tests de backend y frontend.
2. `Build` → empaquetar artefactos (solo si los tests pasan).
3. `DeployQA` → desplegar en QA.
4. `DeployPROD` → desplegar en PROD con aprobación manual.

5.1. Encabezado y variables (igual que TP05)

```
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

variables:
  - group: 'VG-TP05'   # MONGODB_URI (secret)

  - name: nodeVersion
    value: '20.x'
  - name: azureSubscription
    value: 'azure-tp05-connection'
  - name: resourceGroup
    value: 'rg-tp05-ingsoft3-2025'

  # NOMBRES EXACTOS DE LAS WEB APPS
  - name: webappBackQA
    value: 'AppTP5-BackendQA'
  - name: webappFrontQA
    value: 'AppTP05-FrontendQA'
  - name: webappBackPRD
```

```
    value: 'AppTP05-BackendPROD'
  - name: webappFrontPRD
    value: 'AppTP05-FrontendPROD'

  - name: healthPath
    value: '/healthz'
```

Mismas variables que en TP05: conexión a Azure, nombres de Web Apps, health path, etc.

5.2. Stage Tests: correr unit tests de back y front

```
# ============== TESTS ==============
- stage: Tests
  displayName: 'Ejecutar Tests Unitarios'
  jobs:
```

5.2.1. Job TestBackend

```
 - job: TestBackend
    displayName: 'Backend Tests'
    steps:
    - task: NodeTool@0
      inputs:
        versionSpec: '$(nodeVersion)'
```

- Instala Node 20.x para correr los tests de backend.

```
   - script: |
        set -e
        BACK_PKG=$(find "$(Build.SourcesDirectory)" -type f -path
"*/backend/package.json" -print -quit || true)
        [ -z "$BACK_PKG" ] && { echo "❌ No se encontró
backend/package.json"; exit 1; }
        BACKEND_DIR=$(dirname "$BACK_PKG")
        echo "✅ Backend detectado: $BACKEND_DIR"
        echo "##vso[task.setvariable
variable=BACKEND_DIR]$BACKEND_DIR"
      displayName: 'Detectar backend'
```

- Detecta dinámicamente dónde está el `backend/package.json` y setea `BACKEND_DIR` como variable.

```
- script: |
    cd "$(BACKEND_DIR)"
    npm ci
    npm test
  displayName: 'Backend: npm test'
```

- Ejecuta:
  - `npm ci` → instala dependencias de desarrollo y producción.
  - `npm test` → corre Jest, que ejecuta todos los tests de backend (los de Mongo en memoria).

```
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'JUnit'
    testResultsFiles: '$(BACKEND_DIR)/coverage/junit.xml'
    failTaskOnFailedTests: true
  condition: always()

- task: PublishCodeCoverageResults@1
  inputs:
    codeCoverageTool: 'Cobertura'
    summaryFileLocation:
'$(BACKEND_DIR)/coverage/cobertura-coverage.xml'
  condition: always()
```

- Publica:
  - Resultados de tests en formato **JUnit**.
  - Cobertura de código en formato **Cobertura**.
- `condition: always()` → incluso si fallan tests, Azure DevOps sube los reportes para inspeccionar.

Para esto, Jest se configuró para generar `coverage/junit.xml` y `coverage/cobertura-coverage.xml` al correr `npm test`.

5.2.2. Job TestFrontend

```
- job: TestFrontend
  displayName: 'Frontend Tests'
  steps:
  - task: NodeTool@0
    inputs:
      versionSpec: '$(nodeVersion)'
```

- De nuevo `Node 20.x` pero ahora para tests de frontend.

```
  - script: |
      set -e
      FRONT_PKG=$(find "$(Build.SourcesDirectory)" -type f -path
"*/frontend/package.json" -print -quit || true)
      [ -z "$FRONT_PKG" ] && { echo "❌ No se encontró
frontend/package.json"; exit 1; }
      FRONTEND_DIR=$(dirname "$FRONT_PKG")
      echo "✅ Frontend detectado: $FRONTEND_DIR"
      echo "##vso[task.setvariable
variable=FRONTEND_DIR]$FRONTEND_DIR"
    displayName: 'Detectar frontend'
```

- Detecta dónde está el frontend.

```
  - script: |
      cd "$(FRONTEND_DIR)"
      npm ci
      npm test
    displayName: 'Frontend: npm test'
```

- Instala dependencias y ejecuta `npm test` sobre el frontend (tests con Supertest sobre el servidor Express).

```
  - task: PublishTestResults@2
    inputs:
```

```
      testResultsFormat: 'JUnit'
      testResultsFiles: '$(FRONTEND_DIR)/coverage/junit.xml'
      failTaskOnFailedTests: true
  condition: always()
```

- Publica los resultados de tests de frontend.

Conclusión del stage `Tests`:

> Si cualquier test de backend o frontend falla, el stage `Tests` falla, y por configuración el stage `Build` **no se ejecuta**. Es decir, no se construyen artefactos ni se despliega a QA/PROD si la suite de tests no está verde.

5.3. Stage Build (igual que TP05 pero condicionado a Tests)

```
- stage: Build
  displayName: 'Build + Empaquetar backend y frontend (Node 20)'
  dependsOn: Tests
  condition: succeeded()
```

- Solo corre si el stage `Tests` terminó con éxito (`succeeded()`).
- A partir de acá, el contenido es el mismo Build del TP05: detectar rutas, validar backend Node, `npm ci --omit=dev`, empaquetar ZIPs y publicar artefactos `drop`.

5.4. Stages DeployQA y DeployPROD

Los stages de Deploy QA y Deploy PROD son los mismos del TP05:

- **DeployQA**:
  - Depende de `Build`.
  - Obtiene URLs reales de las Web Apps QA.
  - Configura App Settings (Mongo URI, DB_NAME, ALLOWED_ORIGINS, etc.).
  - Despliega `backend.zip` y `frontend.zip` en QA.

- **DeployPROD**:
    - Depende de `DeployQA`.
    - Está asociado al Environment `PROD` con **aprobación manual**.
    - Reutiliza los mismos artefactos (`backend.zip`, `frontend.zip`).
    - Configura App Settings para producción y hace el despliegue.

---

6. Estrategia de mocking y aislamiento

- **Base de datos**:
    - Se mockea usando `mongodb-memory-server`.
      No se usa MongoDB Atlas en los tests → no hay dependencias de red ni secretos.