Departamento de Ingeniería de Sistemas
Desarrollo de aplicaciones Backend
Proyecto Final

Plataforma para evaluar algoritmos

Introducción

Construir una **plataforma backend** que funcione como un **juez online** (similar a HackerRank, LeetCode o Codeforces).
El rol de administrador puede publicar **retos algorítmicos**, que los estudiantes envíen sus soluciones en distintos lenguajes (Python, Node.js, C++ y Java), y que esas soluciones sean **ejecutadas en contenedores aislados** para verificar si cumplen los casos de prueba.

Objetivo general

Diseñar e implementar el **MVP de un juez online** que siga la organización de **Clean Architecture**, permita soporte de **múltiples lenguajes** (Java, Python, NodeJs, C++), ejecute soluciones en sandbox y las califique automáticamente.

Reglas

Modalidad: Trabajo en equipos de máximo **4** estudiantes
Duración: 5 semanas (2 entregas parciales)
Arquitectura requerida: Clean Architecture (Use Cases / Entities / Interface Adapters / Drivers)
Stack sugerido: Node.js + NestJS (API), Java (Spring Boot), PostgreSQL, MongoDB, Redis, Docker Compose (obligatorio), Kubernetes (opcional con bonificación), JWT
Base de datos: Debe estar en un ambiente local controlado.

Competencias a desarrollar

- Diseño de dominio usando **Clean Architecture**.
- Modelado de APIs REST y separación de capas.
- Procesamiento asíncrono con colas.
- Pruebas automatizadas y observabilidad.

- Despliegue con **Docker Compose** (mínimo). **Kubernetes** como opcional para bonificación.

Modulos a desarrollas

Modulo 1 - Autenticación/Autorización con JWT. Roles: STUDENT, ADMIN.

Cuando un usuario inicia sesión (con su correo y contraseña), el sistema le entrega un **token JWT** que dice quién es y qué puede hacer. Cada vez que hace una petición, envía ese JWT y el backend verifica si está permitido:

- Un **ADMIN** puede crear retos y subir casos de prueba.
- Un **STUDENT** solo puede verlos y enviar soluciones.

Es la forma de asegurarse de que nadie haga algo que no le corresponde.

Modulo 2 -Gestión de Retos: CRUD + carga de casos de prueba.

Módulo para crear, editar, publicar y eliminar retos algorítmicos, y adjuntarles sus casos de prueba (entradas*/salidas*).

Roles y permisos

- **ADMIN/PROFESOR:** crear/editar/eliminar retos, subir y gestionar casos.
- **STUDENT:** solo lectura de retos publicados (sin ver casos ocultos).

Campos clave del reto

- title, description, difficulty (Easy/Medium/Hard), tags[]
- Límites: timeLimit, memoryLimit
- Estado: draft | published | archived

Ejemplo entidad JSON

{

"title": "Two Sum",

"difficulty": "Easy",

"tags": ["arrays", "hashmap"],

"timeLimit": 1500,

"memoryLimit": 256,

```
  "description": "Dado un arreglo de enteros y un target..."
}
```

Modulo 3 - Submissions: envío de código, ejecución aislada, estados

Para este módulo se debe tener en cuenta:

- Quién envió el código (el estudiante).
- Qué reto estaba intentando resolver.
- En qué **lenguaje de programación** (Python, C++, Java, Node.js, etc.).
- Cuándo lo envió.
- Y cuál fue el **resultado** (si pasó o no los casos de prueba).

Ejemplo del estado inicial

```
{
  "id": "subm-101",
  "user": "juanperez",
  "challengeId": "two-sum",
  "language": "python",
  "status": "RUNNING",
  "createdAt": "2025-10-05T15:30:00Z"
}
```

¿Qué pasa cuando se envía el código?

1. El sistema **recibe el código** y crea un registro del submission.
2. Lo **envía a una cola de procesamiento** (Redis) para que un **worker** lo tome.
3. El worker lanza un **contenedor aislado** (llamado *runner*) que ejecuta el programa.
4. El runner **compila** (si es necesario) y ejecuta el código con cada caso de prueba.
5. Se **compara la salida del estudiante** con la salida esperada (.out).
6. Según el resultado, se actualiza el estado del submission.

Durante el proceso, cada submission pasa por uno de estos estados:

Estado	Significado
QUEUED	En espera de ser ejecutado.
RUNNING	Se está ejecutando.
ACCEPTED	Pasó todos los casos correctamente.

WRONG_ANSWER (WA)	El programa dio una salida incorrecta.
TIME_LIMIT_EXCEEDED (TLE)	Se demoró más del tiempo permitido.
RUNTIME_ERROR (RE)	Falló durante la ejecución (ej: división por cero).
COMPILATION_ERROR (CE)	No compiló correctamente (en C++, Java).

Ejemplo de resultado final


```
{
 "id": "subm-101",
 "status": "ACCEPTED",
 "score": 100,
 "timeMsTotal": 720,
 "cases": [
  { "caseId": 1, "status": "OK", "timeMs": 40 },
  { "caseId": 2, "status": "OK", "timeMs": 55 }
 ]
}
```


Modulo 4 - Runner por lenguaje (mínimo: Python, Node.js, C++, Java):

Un runner es **un entorno de ejecución** controlado en el lenguaje correspondiente.

Cada Runner se ejecuta en un **contenedor independiente**:

- Sin acceso a internet (--network none).
- Con límite de **CPU** (--cpus 0.5).
- Con límite de **memoria** (--memory 512m).
- Solo lectura del código (--read-only).
- Se destruye cuando termina (--rm).

Cómo se conectan los Runners

- El **worker** detecta el lenguaje del submission.
- Llama al **Runner correspondiente** para ejecutar el código.
- Recoge los resultados y los envía al backend.
- Luego destruye el contenedor para liberar recursos.

Modulo 5 - Cursos

La plataforma no solo evalúa retos de programación, también organiza a los **estudiantes y profesores en cursos**, igual que una materia universitaria. Cada curso funciona como un **espacio independiente** donde se gestionan retos, envíos y resultados. Donde un **curso** representa una asignatura o grupo académico, por ejemplo:

- Lenguaje de Programación Backend
- NRC 12345
- Periodo 2025-1 (año-semestre)
- Grupo: 1

Cada curso tiene:

- Un **profesor responsable** (o varios).
- Un conjunto de **estudiantes inscritos**.
- Una lista de **retos** asignados solo a ese curso.

Rol del profesor

- Crea y administra su curso.
- Publica retos para sus estudiantes.
- Carga los casos de prueba de cada reto.
- Supervisa los **submissions** y resultados.
- Consulta el **leaderboard** (ranking) por curso o por reto.

Rol del estudiante

- Está inscrito en uno o más cursos.
- Solo puede ver y resolver los **retos de su curso**.
- Realiza submissions y ve su progreso.
- Aparece en el leaderboard del curso según su desempeño.

Modulo 6 - Cómo se califica en la plataforma

El sistema también se puede usar para hacer **evaluaciones formales o parciales**, donde los estudiantes resuelven retos bajo condiciones controladas (tiempo, intentos, etc.), igual que un examen práctico de programación.

Este módulo le permitirá que el profesor cree un **parcial o evaluación automática**, donde:

- Cada estudiante recibe uno o varios **retos asignados**.
- El sistema califica **automáticamente** sus envíos.
- El profesor puede ver los resultados y el desempeño del grupo.

Estructura de un parcial

Un **parcial** o **evaluación** es un conjunto de retos agrupados y con reglas específicas.
Ejemplo:

Evaluación: Parcial 1 - Estructuras de Datos
├— Reto 1: Suma de N números
├— Reto 2: Cola circular
└— Reto 3: Árbol binario equilibrado
Duración: 90 minutos
Fecha: 15 de octubre, 10:00 a.m.

Flujo general del parcial

1. **El profesor crea la evaluación**
    a. Define el nombre, fecha, duración y los retos que la componen.
    b. Asigna la evaluación a uno o varios cursos.
2. **Los estudiantes presentan**
    a. Se autentican y acceden al módulo del parcial.
    b. Solo pueden ver los retos activos y enviar soluciones dentro del tiempo límite.
    c. Pueden hacer varios envíos (según la configuración del profesor).
3. **El sistema evalúa automáticamente**
    a. Cada envío se ejecuta en su runner (Python, C++, etc.).
    b. Se califican los casos de prueba y se calculan los puntos.
    c. Al terminar el tiempo, la evaluación se cierra automáticamente.
4. **El profesor revisa resultados**
    a. Ve el puntaje total de cada estudiante.
    b. Puede descargar reportes o revisar caso por caso.

Ejemplo de calificación automática

Estudiante	Reto	Score	Estado	Tiempo total
Ana Pérez	Suma de N números	100	ACCEPTED	950 ms
Jorge Ruiz	Cola circular	80	WRONG ANSWER	1.2 s

Paula Díaz	Árbol binario	70	TIME LIMIT EXCEEDED	2.0 s

Puntaje total de Ana: 100
Puntaje total de Jorge: 80
Puntaje total de Paula: 70

Modulo 7 - Leaderboard

Es una **tabla de clasificación** donde aparecen los mejores resultados de los estudiantes: quién resolvió el reto, cuántos puntos obtuvo y en cuánto tiempo.

Tipos de leaderboard

1. **Por reto**
    a. Muestra el mejor envío (*submission*) de cada estudiante para ese reto.
    b. Se ordena por **puntaje** y luego por **tiempo total de ejecución**.
2. **Por curso**
    a. Suma el puntaje obtenido por cada estudiante en todos los retos del curso.
    b. Permite al profesor ver el desempeño general de la clase.
3. **Por evaluación o parcial**
    a. Se calcula con base en los retos incluidos en el examen.
    b. Se usa principalmente para exámenes automáticos o competencias internas.

Criterios de ordenamiento

Criterio	Descripción
Score	Puntos obtenidos (100 = todos los casos pasaron).
Tiempo total	Suma del tiempo que tomó en ejecutar todos los casos.
Fecha de envío	En caso de empate, gana quien lo resolvió primero.
Lenguaje (opcional)	Puede filtrarse por lenguaje usado.

Cómo se genera

1. Cada vez que un estudiante envía una solución (**submission**), el sistema la evalúa.
2. Se guarda su **mejor resultado** por reto (mayor score o menor tiempo).
3. El backend recalcula el **ranking** y actualiza la tabla.
4. Los resultados se muestran públicamente dentro del curso o del reto.

Privacidad y control

- En modo **competencia**, todos pueden ver el leaderboard.
- Los **parciales no se tienen en cuenta en los leaderboard**
- El sistema puede filtrar por **curso**, **grupo**, o **fecha**.

Modulo 8 Observabilidad mínima

Se debe implementar

1. **Logs estructurados**
    a. Cada petición y cada ejecución (submission) debe generar un log en formato JSON.
    b. Todos los logs deben incluir un **ID único de seguimiento** (requestId o submissionId)
    para poder rastrear todo el flujo de un envío desde que se recibe hasta que finaliza.
    c. Registrar eventos clave:
        i. creación de submission,
        ii. inicio/fin de ejecución,
        iii. resultado final,
        iv. errores o tiempos excesivos.

    Ejemplo:

```
{
 "level": "info",
 "msg": "Runner finished execution",
 "submissionId": "subm-42",
 "status": "ACCEPTED",
 "durationMs": 730
}
```

2. **Métricas básicas**
    a. Exponer un endpoint /metrics o un servicio equivalente.
    b. Debe registrar al menos:
        i. submissions_total → cantidad total de envíos procesados.
        ii. submissions_failed_total → cantidad de fallos o errores internos.
        iii. average_execution_time_ms → tiempo promedio por ejecución.
        iv. active_runners → cuántos contenedores o procesos están
3. **Trazabilidad mínima**
    a. Todo log o métrica de un mismo submission debe estar **correlacionado** por el mismo ID.
    b. Permitir identificar el recorrido completo:
    API → Cola → Worker → Runner → Resultado.

Modulo 9 - Asistente creativo

El sistema puede incluir un asistente inteligente (basado en IA) que ayude al administrador o profesor a crear nuevos retos. Este asistente no reemplaza al profesor ni evalúa código, sino que apoya la creación de contenido para la plataforma.

Qué hace el asistente

- **Generar ideas de retos**
  - El administrador ingresa un tema o categoría, por ejemplo:
    - "Árboles binarios"
    - "Búsqueda binaria"
    - "Algoritmos de ordenamiento"
  - La IA sugiere **1 o más retos** con título, descripción, entradas y salidas esperadas.
- **Proponer ejemplos y casos de prueba iniciales**
  - El asistente puede generar **ejemplos de entrada y salida** para cada reto.
  - Estos casos sirven como base para construir los archivos .in y .out.
  - El equipo humano debe **validar** esos ejemplos antes de publicarlos (los .out deben verificarse con scripts reales).

Arquitectura — Clean Architecture

- **Domain/Entities**: User, Challenge, Submission, Leaderboard, TestCase
- **Use Cases**: CreateChallenge, SubmitSolution, ProcessSubmission.
- **Interfaces/Ports**: IChallengeRepo, ISubmissionRepo, IJobQueue, IStorage.
- **Adapters**: REST Controllers, Repositorios SQL, Adapter Redis.
- **Frameworks**: NestJS, Postgres, Redis.
- **Workers**: consumen jobs, ejecutan runners efímeros.

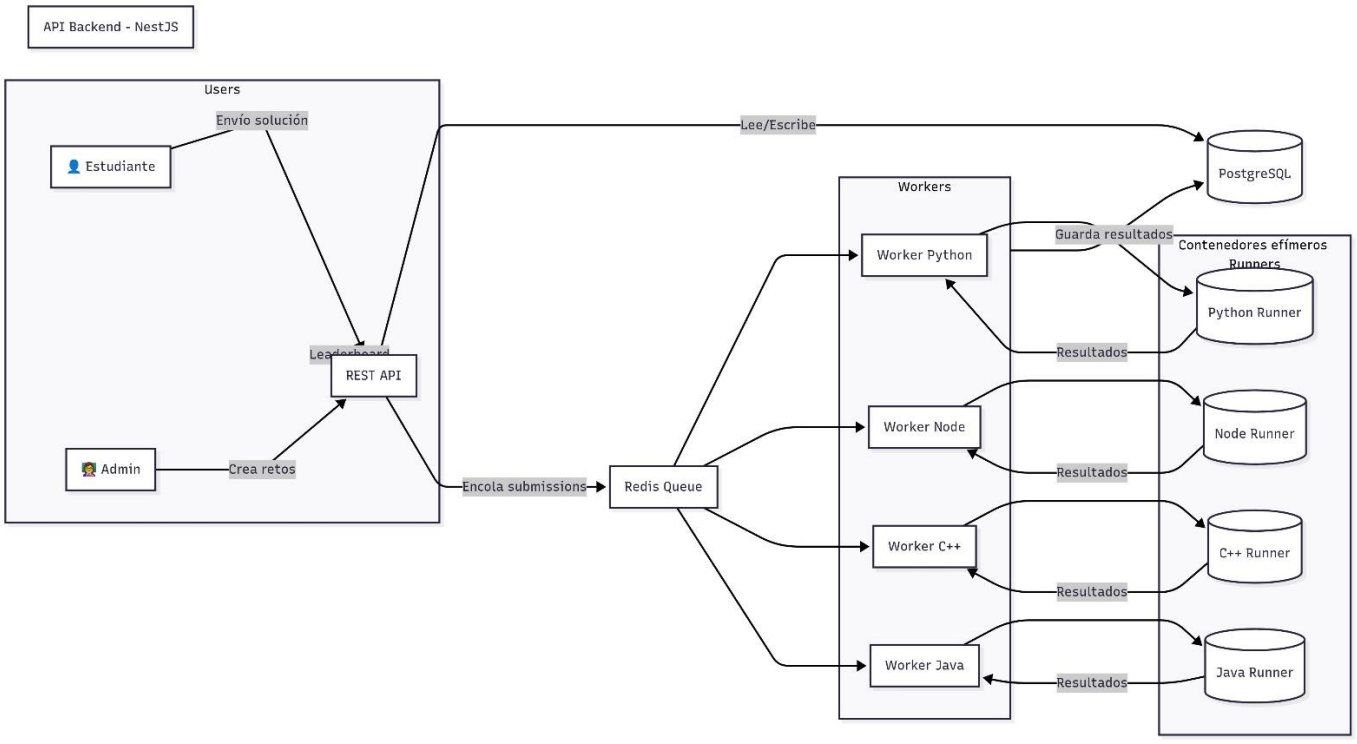
Componentes principales

1. **API Backend (NestJS en Node.js)**
   - **Implementar los 9 módulos con sus métodos: POST, GET, PUT y DELETE, en cada caso correspondiente**
2. **Base de datos (PostgreSQL)**
   - Tablas para usuarios, retos, casos de prueba y submissions.
3. **Cola de trabajos Bull NestJS ó cualquier MQ (Redis)**
   - Recibe los envíos y los distribuye a los workers por lenguaje.

4. **Workers por lenguaje**
    - Procesos que leen de la cola y lanzan **contenedores efímeros (runners)** donde se ejecuta el código del estudiante.
    - Cada worker está especializado en un lenguaje (ej: worker_java, worker_python).
5. **Runners efímeros**
    - Contenedores aislados que compilan y ejecutan el código con límites de tiempo/memoria y sin red.
    - Evalúan el código contra casos de prueba y devuelven resultados al worker.

Flujo del sistema

1. **El** admin **o profesor crea un reto o evaluación, define enunciado y sube los casos de prueba.**
2. **Un** estudiante **se autentica y envía una solución (ej: código en Java).**
3. **La** API **guarda el submission y lo encola en Redis.**
4. **El** worker correspondiente **toma el job y lanza un** runner **con docker run --network none ….**
5. **El runner compila/ejecuta el código contra los casos de prueba.**
6. **El worker guarda resultados en la base de datos,**
7. **El estudiante puede consultar su estado y ver el leaderboard.**



Estructura de datos

Entidad	Descripción
User	Contiene la información del usuario y su rol (STUDENT, PROFESSOR, ADMIN).
Course	Representa una materia o grupo. Tiene un nombre, un código y uno o varios profesores.
CourseStudent	Relación entre cursos y estudiantes (inscripción).
Challenge	Reto asociado a un curso.
Submission	Envío de un estudiante dentro de un curso.

Ejecución en Docker Compose (mínimo)

- Servicios:
  - **API** (NestJS).
  - **DB** (Postgres).
  - **Redis** (cola).
  - **Workers por lenguaje** (java, python, node, cpp) → escalar con `docker compose up --scale worker_java=3`.

Kubernetes (opcional con bonificación)

- Desplegar cada runner como **Deployment** con HPA o **KEDA** para escalar según la cola.
- Alternativa avanzada: un **Job por envío**, cada submission lanza un pod efímero.

Paso a paso (obligatorio en Compose, opcional en K8s)

0. **Semana 2 (23 Octubre)**
   - Diseño de modelos y capas (domain/usecases/interfaces).
   - Implementar auth + CRUD retos.
   - Montar Compose con api + db + redis.
   - Workers stub con Redis.
1. **Semana 5 (11 Noviembre)**
   - Implementar runners efímeros en Compose (`docker run`).
   - Guardar resultados de submissions + leaderboard.
   - Añadir logs y métricas básicas.
   - *Bonus: adaptar manifiestos a Kubernetes funcionales*

Rúbrica de evaluación

- Diseño de dominio y API (20%).
- Sandbox y runner en Compose (20%).
- Procesamiento asíncrono con Redis (20%).
- Pruebas, observabilidad y seguridad (10%).
- Documentación de la API usando swagger, starlight y Video (20%).
- UI simple(10%):

Bonificación: bonificación de 0.5 redimible en cualquier parcial.

Entregables

- Repo con código del API (se revisará commits por cada integrante) y workers.
- docker-compose.yml con todos los servicios.
- Scripts de semilla
- README con instrucciones (docker compose up).
- Evidencia de ejecución con múltiples instancias (--scale).
- Video del proyecto
- (Opcional) Manifiestos de Kubernetes y demo del escalado.