# Esercitazione: Gestore di Task con Validazione, Service Layer e Log

## Obiettivo didattico

Realizzare una **mini API Express** che permetta di gestire una lista di attività (*Task*), applicando principi professionali di sviluppo backend:

- Validazione dei dati con **Joi**
- Separazione della logica tramite il modello **Service Layer**
- Gestione del logging e monitoraggio tramite **Morgan** e **Winston**

## Struttura del progetto suggerita

Organizza le cartelle del progetto seguendo questa struttura modulare:

```
task-api/
├── models/
│   └── task.js
├── services/
│   └── taskService.js
├── routes/
│   └── tasks.js
├── middleware/
│   ├── validate.js
│   └── errorHandler.js
├── logger.js
├── server.js
├── package.json
└── .env
```

Ogni cartella ha un ruolo chiaro:

- **models/** → definisce la struttura dei dati nel database
- **services/** → contiene la logica di business
- **routes/** → gestisce le rotte API
- **middleware/** → controlli automatici come validazione o gestione errori
- **logger.js** → sistema di tracciamento eventi ed errori
- **server.js** → punto di ingresso del progetto

## STEP 1 — Modello Task (MongoDB)

Crea un modello per la collezione *tasks* utilizzando Mongoose.
Ogni *Task* deve contenere almeno:

- **title**: testo della task (obbligatorio)
- **completed**: valore booleano (di default `false`)

- **createdAt / updatedAt**: gestiti automaticamente da Mongoose

Questo è il cuore del database, dove vengono salvate tutte le attività.

# STEP 2 — Validazione con Joi

Crea un **middleware di validazione** che utilizzi Joi per verificare che i dati inviati dal client siano corretti prima di salvarli nel database.

Esempio di regole:

- `title`: stringa, minimo 3 caratteri, obbligatoria
- `completed`: boolean (facoltativo)

Se i dati non rispettano lo schema, il middleware deve bloccare la richiesta e restituire un errore chiaro al client (es. `400 Bad Request`).

Obiettivo: rendere il backend più **sicuro e robusto**, evitando dati errati nel database.

# STEP 3 — Service Layer

Crea un file `taskService.js` nella cartella **services/** che contenga le funzioni principali per lavorare sui task.

Le funzioni dovranno eseguire le seguenti operazioni:

- Creare un nuovo task
- Ottenere la lista di tutti i task
- Aggiornare un task esistente
- Eliminare un task

Le rotte non devono contenere logica di database:
devono soltanto **chiamare le funzioni del service** e restituire la risposta.

Obiettivo: separare la logica di business dalla gestione delle rotte per avere codice più ordinato, testabile e leggibile.

# STEP 4 — Rotte e Validazione

Crea un router Express nella cartella **routes/** per gestire tutte le operazioni CRUD:

- `POST /api/tasks` → Crea un nuovo task
- `GET /api/tasks` → Restituisce tutti i task
- `PATCH /api/tasks/:id` → Aggiorna un task esistente
- `DELETE /api/tasks/:id` → Elimina un task

Applica il **middleware di validazione Joi** alle rotte che ricevono dati dal client (POST e PATCH).

Assicurati di gestire gli errori (es. task non trovato, input non valido) con messaggi chiari.

# STEP 5 — Logging e Monitoraggio

Integra un sistema di log per tenere traccia dell'attività del server.

- Usa **Morgan** per stampare in console le richieste HTTP (metodo, rotta, tempo di risposta).
- Usa **Winston** per registrare eventi e errori sia su console che su file.

Ogni errore gestito o richiesta significativa deve essere tracciato nei log.
Puoi creare un file dedicato `logger.js` e importarlo nel server principale.

Obiettivo: imparare a monitorare l'attività del backend e diagnosticare errori in modo professionale.

# STEP 6 — Obiettivi formativi

Al termine dell'esercitazione, dovreste essere in grado di:

Comprendere l'importanza della validazione dei dati in ingresso
Separare correttamente la logica di business dalle rotte
Implementare un logging utile al monitoraggio e debugging
Gestire in modo modulare e scalabile un piccolo progetto backend

# Variante avanzata (per chi termina prima)

Per approfondire, prova ad aggiungere queste funzionalità extra:

- Aggiungi un campo **priority** (bassa, media, alta) e valida il suo valore con Joi.
- Crea una rotta `/api/tasks/completed` che restituisca solo i task completati.
- Aggiungi un log con livello **warn** quando un task viene eliminato.

Obiettivo: imparare ad estendere un progetto mantenendo la struttura pulita.

# Attività di riflessione finale

1. Dove si trova la logica di business e dove la validazione?
2. Come viene gestito un errore di validazione o di database?
3. Cosa si guadagna nel separare **routes**, **services** e **models**?
4. Come potresti espandere questa API (es. aggiungendo utenti e autenticazione)?

# Suggerimento finale

- Testa la tua API con **Postman** o **curl** dopo ogni passaggio.
- Ricontrolla sempre i log per capire il comportamento del server.
- Scrivi messaggi di commit chiari se usi Git:
    - `feat: aggiunge validazione con Joi`
    - `refactor: sposta logica nel service layer`

- fix: corregge gestione errori nei log