Dove si trova la logica di business e dove la validazione?

Logica di business:

La logica principale del progetto è incapsulata nei **service**.

File principale: services/taskService.js

Esempi di funzioni di business:

```
async function createTask(data) { ... }

async function updateTask(id, update) { ... }

async function getAllTasks(filter = {}) { ... }
```

Qui avviene tutta la gestione della persistenza su MongoDB, il salvataggio, l'aggiornamento, la cancellazione e il recupero dei dati. Il router (routes/tasks.js) si limita a orchestrare le richieste, chiamando i servizi.

Validazione:
La validazione è separata nei **middleware** (middleware/validate.js) usando **Joi**.

Prima che i dati arrivino al service, validateBody(schema) controlla che il body della richiesta sia conforme allo schema.

Esempio:

```
const createSchema = Joi.object({

     title: Joi.string().min(3).required(),

     completed: Joi.boolean().optional(),

     priority: Joi.string().valid(…priorities).optional(),

});

router.post("/", validateBody(createSchema), async (req, res,
next) => { ... });
```

Il router si occupa della orchestration e routing, senza una logica complessa.

Come viene gestito un errore di validazione o di database?

Errore di validazione:

Catturato dal middleware validateBody.

Risposta inviata immediatamente con HTTP 400:

```
{
"message": "Validation error”,
"details": ["title is required", …]
}
```

Errore di database:

Catturato dal try/catch nel router e passato al middleware centralizzato errorHandler.

L'errorHandler logga l’errore con Winston e restituisce:

- 400 per CastError (ID non valido)
- 500 per errori generici

```
{
  "message": "Internal Server Error”
}
```

Tutti gli errori vengono loggati centralmente e le risposte HTTP sono coerenti e dettagliate per il client.

Cosa si guadagna nel separare routes, services e models?

- **Manutenibilità:** ogni parte ha un solo compito (Single Responsibility Principle).
  - models/ → definizione schema DB
  - services/ → logica di business
  - routes/ → gestione endpoint e orchestrazione
- **Testabilità**: si possono testare i servizi senza dover avviare Express.
- **Riutilizzabilità**: i servizi possono essere richiamati da altri router o da altri microservizi senza duplicare logica.
- **Pulizia del codice**: il router rimane leggibile, solo orchestration; il service contiene solo logica DB.
- **Scalabilità**: quando l'app cresce, si possono aggiungere nuove funzionalità senza mescolare livelli.

Come potresti espandere questa API?

Esempi concreti di espansione:

1. **Autenticazione e utenti**:
   - Creare modello User con email/password.
   - Middleware auth per proteggere le route (JWT).
   - Associare i task a un utente (userId).
2. **Ruoli e permessi**:
   - Ruoli tipo admin/user per controllare accesso e modifiche.
   - Esempio: solo l'utente proprietario può modificare o cancellare il proprio task.
3. **Sicurezza**
   - In produzione usare NODE_ENV=production e non loggare dati sensibili.
   - Proteggere la stringa MONGO_URI e non committare .env.
4. **Filtri e paginazione avanzata**:
   - Filtrare per range date, priorità multiple.
   - Paginazione per grandi quantità di task (limit, skip).