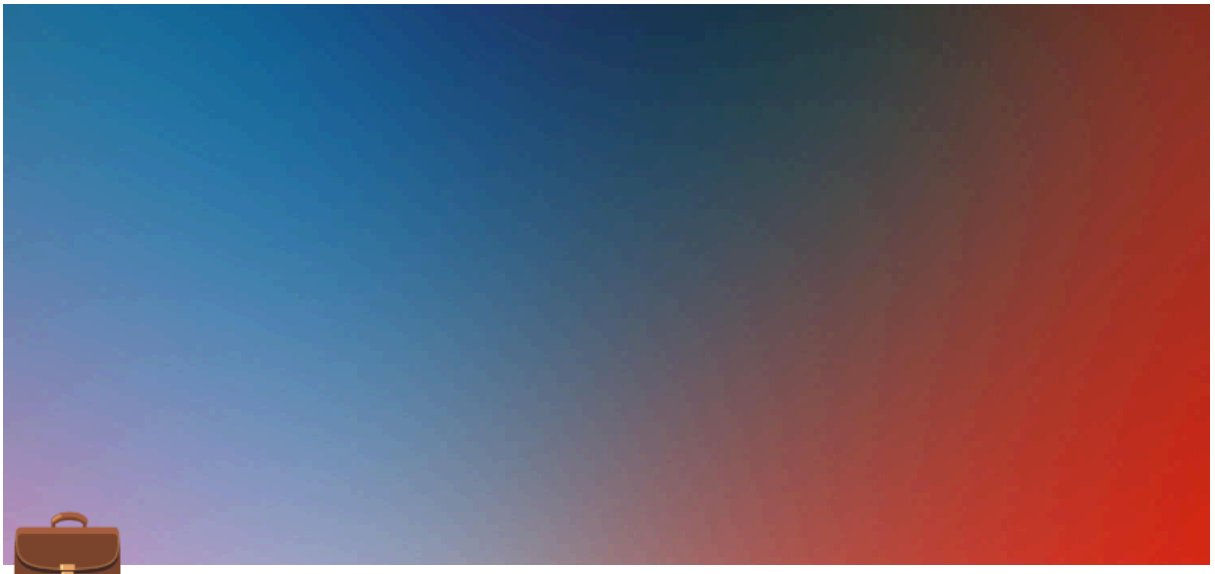

Trello Lite – Task Management API

Backend Technical Specification

1. SYSTEM OVERVIEW

What the System Does

Trello Lite is a RESTful task management API that allows users to organize work into projects and tasks. Users can create projects, manage tasks within those projects, assign tasks to other users, and track progress through task statuses. The system is backend-only and exposes a JSON API consumed by any client.

Core Business Logic

The system revolves around three ownership relationships. A user creates a project and becomes its owner. Within that project, tasks are created and belong to exactly one project. Tasks can be assigned to one or more users through an assignment table. Only project owners and assigned users may interact with tasks in controlled ways. The system enforces these rules at the service layer, not just at the database level.

Task lifecycle follows a fixed status progression: `TODO → IN_PROGRESS → DONE`. While the API does not enforce a strict transition order (any status can be set), the three valid states are enforced through enum validation. Priority levels are `LOW`, `MEDIUM`, and `HIGH`.

High-Level Architecture

The system uses a layered clean architecture with strict separation of concerns:

```
Client → Router (Controller) → Service → Repository → Datab
ase
```

Routers handle HTTP concerns only (parsing requests, returning responses). Services contain all business logic and authorization checks. Repositories handle all database interactions. Models define the data schema. Schemas (Pydantic) handle serialization and validation.

Request → Response Lifecycle

1. Client sends an HTTP request with a JWT in the Authorization header.
2. FastAPI dependency injection extracts and validates the JWT, resolving the current user.
3. The router function receives the validated request body (via Pydantic schema) and the current user.
4. The router calls the appropriate service function, passing the request data and current user.
5. The service performs authorization checks (ownership, membership), applies business rules, and calls the repository.
6. The repository executes the database query via SQLAlchemy and returns an ORM model instance.
7. The service returns the model to the router.
8. The router serializes the model into a Pydantic response schema and returns an HTTP response.
9. On any failure, a custom exception is raised in the service layer, caught by a global exception handler, and returned as a structured JSON error.

---

2. FEATURE LIST (DETAILED)

2.1 Authentication

Registration
Allows any anonymous user to create an account by providing a unique email and username. The password is hashed before storage. Upon success, the API returns the created user profile (no tokens yet — the user must log in separately). This separation keeps the flows clean and testable.

Edge cases: duplicate email must return 409 Conflict; duplicate username must return 409 Conflict; weak passwords (under 8 characters) must return 422 Unprocessable Entity.

Validation rules: email must be a valid RFC 5322 address; username must be 3–30 characters, alphanumeric with underscores only; password must be at least 8 characters.

Login
Accepts email and password. The system looks up the user by email, verifies the password hash, and if valid, returns an access token and a refresh token. Both are JWTs. The access token is short-lived (30 minutes). The refresh token is long-lived (7 days).

Edge cases: unknown email returns 401, not 404 (to avoid user enumeration); wrong password returns 401; deactivated user account returns 403.

Token Refresh
Accepts a valid refresh token and returns a new access token. The refresh token itself is not rotated unless you choose to implement rotation (noted in section 5). If the refresh token is expired or malformed, return 401.

Logout
Accepts the refresh token and invalidates it. Because JWTs are stateless, invalidation is handled by maintaining a server-side denylist stored in a simple database table (`token_blacklist`). On logout, the refresh token's JTI (JWT ID) is stored in this table. All token validation checks this table.

2.2 User Management

Get Own Profile
Any authenticated user can retrieve their own profile. Returns id, username,

email, created_at.

Update Own Profile
Authenticated users can update their username or email. They cannot change their own role. Password change is a separate endpoint requiring the current password for verification.

Change Password
Requires current_password and new_password. Validates the current password against the hash before updating.

Admin: List All Users
Admin users can list all users with pagination. Normal users receive 403.

Admin: Deactivate User
Admin users can set a user's `is_active` flag to false. Deactivated users cannot log in.

Edge cases: a user cannot deactivate themselves; deactivating a user does not delete their data.

2.3 Project Management

Create Project
Any authenticated user can create a project. The creating user is automatically assigned as the project owner. The project has a name, optional description, and a created_at timestamp.

Validation: name is required, 1–100 characters; description is optional, max 500 characters.

List Projects
A user sees only projects they own or are a member of (i.e., have at least one task assigned to them in that project). Admins see all projects. Returns paginated results.

Get Project by ID
Returns full project details including task count. Only accessible to project members or the owner. Returns 403 for non-members, 404 if the project does not exist.

Update Project
Only the project owner can update the name or description.

Delete Project
Only the project owner can delete a project. Deletion cascades to all tasks and assignments within the project. This is a hard delete. The client must receive a clear 200 with a confirmation message (not 204, to allow for a message body).

Edge cases: deleting a project that has assigned tasks notifies the caller via the response message that all associated data was removed.

2.4 Task Management

Create Task
Only the project owner can create tasks within a project. A task belongs to exactly one project. Fields include title, description, status, priority, and due_date.

Validation: title required, 1–200 characters; status must be one of `TODO`, `IN_PROGRESS`, `DONE` (defaults to `TODO`); priority must be one of `LOW`, `MEDIUM`, `HIGH` (defaults to `MEDIUM`); due_date must be a future date if provided.

List Tasks in Project
Project members (owner or any assigned user) can list tasks. Supports filtering by status, priority, assignee, and due_date range. Supports search by title. Returns paginated results.

Get Task by ID
Returns full task details including list of assigned users. Accessible to project members only.

Update Task
The project owner can update any field. An assigned user can update only the status field of tasks assigned to them. No other user can modify the task.

Edge cases: updating a task's project_id is not allowed (tasks are immovable between projects); setting due_date to the past on an update should be allowed (task may have been delayed) but should be flagged with a warning field in the response.

Delete Task
Only the project owner can delete a task. Deletion cascades to all assignments for that task.

2.5 Assignment System

Assign User to Task
Only the project owner can assign users to a task. The user being assigned must exist. You can optionally validate that the assigned user is a registered user in the system (no concept of "project membership" beyond this implicit one).

Edge cases: assigning a user who is already assigned returns 409; assigning a non-existent user returns 404; assigning to a task in a project the owner doesn't own returns 403.

Unassign User from Task
Only the project owner can unassign users. Unassigning the last user from a task is allowed — tasks can have zero assignees.

List Assignees for a Task
Any project member can view the list of users assigned to a task.

2.6 Filtering & Search

All list endpoints support filtering via query parameters. Filters are additive (AND logic). Search uses a case-insensitive partial match on the title field. Filtering on date ranges requires both `due_date_from` and `due_date_to` parameters; if only one is provided, it acts as a single bound.

Supported filters for tasks: `status`, `priority`, `assignee_id`, `due_date_from`, `due_date_to`.
Supported filters for projects: `name` (search), `owner_id` (admin only).

2.7 Pagination

All list endpoints are paginated using limit/offset. Default limit is 20, maximum is 100. Response always includes `total`, `limit`, `offset`, and `items`.

2.8 Authorization Rules

Detailed rule table is provided in Section 6.

---

3. DATABASE DESIGN

Tables and Fields

users

Column	Type	Constraints
id	UUID	PRIMARY KEY, default gen_random_uuid()
username	VARCHAR(30)	UNIQUE, NOT NULL
email	VARCHAR(255)	UNIQUE, NOT NULL
hashed_password	VARCHAR(255)	NOT NULL
role	VARCHAR(20)	NOT NULL, DEFAULT 'user', CHECK IN ('user','admin')
is_active	BOOLEAN	NOT NULL, DEFAULT true
created_at	TIMESTAMP	NOT NULL, DEFAULT now()
updated_at	TIMESTAMP	NOT NULL, DEFAULT now()

Indexes: unique index on email; unique index on username.

---

projects

Column	Type	Constraints
id	UUID	PRIMARY KEY
name	VARCHAR(100)	NOT NULL
description	VARCHAR(500)	NULLABLE
owner_id	UUID	FK → users.id, NOT NULL
created_at	TIMESTAMP	NOT NULL, DEFAULT now()
updated_at	TIMESTAMP	NOT NULL, DEFAULT now()

Relationships: many projects → one user (owner). Cascade: if owner is deleted, projects are deleted (or owner_id is set to a system user — your choice; hard delete is simpler for this scope).

Indexes: index on owner_id for fast lookup by owner.

---

tasks

Column	Type	Constraints
id	UUID	PRIMARY KEY
title	VARCHAR(200)	NOT NULL
description	TEXT	NULLABLE

Column	Type	Constraints
status	VARCHAR(20)	NOT NULL, CHECK IN ('TODO','IN_PROGRESS','DONE'), DEFAULT 'TODO'
priority	VARCHAR(10)	NOT NULL, CHECK IN ('LOW','MEDIUM','HIGH'), DEFAULT 'MEDIUM'
due_date	DATE	NULLABLE
project_id	UUID	FK → projects.id, NOT NULL
created_by	UUID	FK → users.id, NOT NULL
created_at	TIMESTAMP	NOT NULL, DEFAULT now()
updated_at	TIMESTAMP	NOT NULL, DEFAULT now()

Relationships: many tasks → one project. Cascade: if project is deleted, all tasks are deleted (ON DELETE CASCADE).

Indexes: index on project_id; index on status; index on priority; composite index on (project_id, status) for common filtered queries.

---

task_assignments

Column	Type	Constraints
id	UUID	PRIMARY KEY
task_id	UUID	FK → tasks.id, NOT NULL
user_id	UUID	FK → users.id, NOT NULL
assigned_at	TIMESTAMP	NOT NULL, DEFAULT now()

Relationships: many-to-many between tasks and users through this table. Cascade: if task is deleted, all its assignments are deleted. If user is deleted, their assignments are deleted.

Constraints: UNIQUE(task_id, user_id) to prevent duplicate assignments.

Indexes: index on task_id; index on user_id; composite unique index on (task_id, user_id).

---

token_blacklist

Column	Type	Constraints
id	UUID	PRIMARY KEY
jti	VARCHAR(255)	UNIQUE, NOT NULL

Column	Type	Constraints
user_id	UUID	FK → users.id
expires_at	TIMESTAMP	NOT NULL
created_at	TIMESTAMP	NOT NULL, DEFAULT now()

Purpose: stores invalidated refresh token JTIs. A background cleanup job (or a simple check on insert) can prune entries where expires_at is in the past.

Relationship Summary

- users → projects: one-to-many (one user owns many projects)
- projects → tasks: one-to-many (one project has many tasks)
- tasks ↔ users: many-to-many via task_assignments
- users → task_assignments: one-to-many
- tasks → task_assignments: one-to-many (CASCADE DELETE)

---

4. API ENDPOINT DESIGN

Base URL: `/api/v1`

4.1 Authentication Routes

POST /auth/register
Description: Register a new user account.
Auth required: No.

Request body:

```
{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "securepassword123"
}
```

Response (201):

```
{
  "id": "uuid",
  "username": "john_doe",
  "email": "john@example.com",
  "created_at": "2024-01-01T00:00:00Z"
}
```

Errors: 409 if email or username taken; 422 if validation fails.

POST /auth/login
Description: Authenticate and receive tokens.
Auth required: No.

Request body:

```
{
  "email": "john@example.com",
  "password": "securepassword123"
}
```

Response (200):

```
{
  "access_token": "eyJ...",
  "refresh_token": "eyJ...",
  "token_type": "bearer"
}
```

Errors: 401 if credentials invalid; 403 if account deactivated.

POST /auth/refresh
Description: Get a new access token using a refresh token.
Auth required: No (refresh token in body).

Request body:

```
{
  "refresh_token": "eyJ..."
}
```

Response (200):

```
{
  "access_token": "eyJ...",
  "token_type": "bearer"
}
```

Errors: 401 if refresh token expired or blacklisted.

POST /auth/logout
Description: Invalidate the refresh token.
Auth required: Yes (access token).

Request body:

```
{
  "refresh_token": "eyJ..."
}
```

Response (200):

```
{
  "message": "Successfully logged out."
}
```

4.2 User Routes

GET /users/me
Auth required: Yes.
Response (200): Current user profile object.

PATCH /users/me
Auth required: Yes.
Request body: `{ "username": "new_name" }` (partial update allowed)
Response (200): Updated user profile.
Errors: 409 if new username/email conflicts.

POST /users/me/change-password
Auth required: Yes.
Request body: `{ "current_password": "...", "new_password": "..." }`

Response (200): `{ "message": "Password updated successfully." }`
Errors: 400 if current password is wrong.

GET /users *(Admin only)*
Auth required: Yes (admin role).
Query params: `?limit=20&offset=0`
Response (200): Paginated list of users.

PATCH /users/{user_id}/deactivate *(Admin only)*
Auth required: Yes (admin role).
Response (200): `{ "message": "User deactivated." }`
Errors: 403 if non-admin; 400 if user tries to deactivate themselves.

4.3 Project Routes

POST /projects
Auth required: Yes.
Request body:

```
{
  "name": "My Project",
  "description": "Optional description"
}
```

Response (201): Full project object including owner info.
Errors: 422 if name missing.

GET /projects
Auth required: Yes.
Query params: `?limit=20&offset=0&name=search_term`
Response (200):

```
{
  "total": 45,
  "limit": 20,
  "offset": 0,
  "items": [ {...}, {...} ]
}
```

GET /projects/{project_id}
Auth required: Yes.
Response (200): Full project object with task_count.
Errors: 403 if not a member; 404 if not found.

PATCH /projects/{project_id}
Auth required: Yes (project owner only).
Request body: `{ "name": "Updated Name", "description": "..." }`
Response (200): Updated project object.
Errors: 403 if not owner; 404 if not found.

DELETE /projects/{project_id}
Auth required: Yes (project owner only).
Response (200): `{ "message": "Project and all associated data deleted." }`
Errors: 403 if not owner; 404 if not found.

4.4 Task Routes

POST /projects/{project_id}/tasks
Auth required: Yes (project owner only).
Request body:

```
{
  "title": "Design database schema",
  "description": "Create ERD and define all tables",
  "status": "TODO",
  "priority": "HIGH",
  "due_date": "2024-02-15"
}
```

Response (201): Full task object.
Errors: 403 if not owner; 404 if project not found; 422 if validation fails.

GET /projects/{project_id}/tasks
Auth required: Yes (project member).
Query params: `?limit=20&offset=0&status=TODO&priority=HIGH&assignee_id=uuid&due_date_from=2024-01-01&due_date_to=2024-03-01&search=database&sort_by=due_date&sort_order=asc`
Response (200): Paginated task list with assignee summaries.

GET /projects/{project_id}/tasks/{task_id}
Auth required: Yes (project member).

Response (200): Full task with list of assigned users.
Errors: 403, 404.

PATCH /projects/{project_id}/tasks/{task_id}
Auth required: Yes.
Business rule: project owner can update all fields; assigned users can only update `status`.
Request body: `{ "status": "IN_PROGRESS" }` (partial update)
Response (200): Updated task object.
Errors: 403 if insufficient permission; 422 if invalid field for role.

DELETE /projects/{project_id}/tasks/{task_id}
Auth required: Yes (project owner only).
Response (200): `{ "message": "Task deleted." }`
Errors: 403, 404.

4.5 Assignment Routes

POST /projects/{project_id}/tasks/{task_id}/assignments
Auth required: Yes (project owner only).
Request body: `{ "user_id": "uuid" }`
Response (201): `{ "task_id": "...", "user_id": "...", "assigned_at": "..." }`
Errors: 404 if user not found; 409 if already assigned; 403 if not owner.

DELETE /projects/{project_id}/tasks/{task_id}/assignments/{user_id}
Auth required: Yes (project owner only).
Response (200): `{ "message": "User unassigned from task." }`
Errors: 404 if assignment not found; 403 if not owner.

GET /projects/{project_id}/tasks/{task_id}/assignments
Auth required: Yes (project member).
Response (200): List of assigned user objects.

---

5. AUTHENTICATION FLOW (JWT)

Registration Flow

1. Client sends POST /auth/register with username, email, password.
2. Router passes data to AuthService.register().

3. Service checks for existing email and username in the database via UserRepository.
4. If conflict, raise HTTP 409.
5. Service hashes the password using bcrypt with a cost factor of 12.
6. Service calls UserRepository.create() with hashed password and role='user'.
7. Router returns the new user object (no tokens issued).

Login Flow

1. Client sends POST /auth/login with email and password.
2. AuthService.login() retrieves user by email via UserRepository.get_by_email().
3. If user not found, return 401 (do not reveal why).
4. If user is_active is false, return 403.
5. Service verifies the plain password against hashed_password using bcrypt.verify().
6. If mismatch, return 401.
7. Service generates an access token (30 min expiry) and a refresh token (7 days expiry).
8. Both tokens are signed with HS256 using a secret key loaded from environment variables.
9. The access token payload contains: `sub` (user_id), `role`, `exp`, `iat`.
10. The refresh token payload contains: `sub` (user_id), `jti` (unique UUID), `exp`, `iat`, `type: "refresh"`.
11. Return both tokens.

Access Token Validation

1. Every protected route uses a FastAPI dependency: `get_current_user`.
2. Dependency extracts the Bearer token from the Authorization header.
3. PyJWT decodes the token using the server secret. If expired or invalid signature, raise 401.

4. Dependency fetches the user from the database to confirm they still exist and are active.
5. Returns the user object, which is injected into the route handler.

Refresh Token Strategy

The refresh token is validated similarly to the access token but additionally: the `jti` is checked against the token_blacklist table. If found, the token is rejected with 401. The refresh endpoint only returns a new access token — it does not rotate the refresh token by default. If you want to implement rotation (recommended for production), generate a new refresh token, blacklist the old JTI, and return both tokens.

Password Hashing

Use the `passlib` library with the `bcrypt` scheme. Set rounds to 12. Never store plain-text passwords. Never log passwords. Use `CryptContext` from passlib for a clean interface. Hash on registration and on password change. Verify on login and on change-password endpoint.

---

6. AUTHORIZATION LOGIC

Rule Table

Action	Project Owner	Assigned User	Other Auth User	Admin
Create project	✅	✅	✅	✅
View own projects	✅	✅	✅	✅ (all)
Update project	✅	❌	❌	✅
Delete project	✅	❌	❌	✅
Create task	✅	❌	❌	✅
View tasks	✅	✅	❌	✅
Update task (all fields)	✅	❌	❌	✅
Update task (status only)	✅	✅	❌	✅

Action	Project Owner	Assigned User	Other Auth User	Admin
Delete task	✅	❌	❌	✅
Assign user to task	✅	❌	❌	✅
Unassign user from task	✅	❌	❌	✅
View task assignees	✅	✅	❌	✅
List all users	❌	❌	❌	✅
Deactivate user	❌	❌	❌	✅

Authorization Implementation Notes

Authorization checks must live in the service layer, not in the router and not in the repository. This keeps the logic testable and centralized. The pattern for each mutating operation is: (1) fetch the resource, (2) if not found raise 404, (3) check ownership/role, (4) if unauthorized raise 403, (5) proceed.

The service should never raise 404 before checking auth to avoid leaking the existence of resources to unauthorized users in sensitive contexts. For this project scope, 404 before 403 is acceptable and simpler — just be consistent.

"Project member" is defined as: the project owner, OR any user who has at least one task assignment within that project. This check is made dynamically at query time, not via a separate membership table.

---

7. PAGINATION, FILTERING & SEARCH

Limit/Offset Pattern

All list endpoints accept `limit` and `offset` as query parameters. Limit defaults to 20, minimum is 1, maximum is 100. Offset defaults to 0. The response always wraps results in an envelope object containing `total` (total matching records before pagination), `limit`, `offset`, and `items` (the array of results for this page).

The repository layer accepts limit and offset and applies them to the SQLAlchemy query using `.limit()` and `.offset()`. The total is obtained via a separate `.count()` query on the same filter set (before applying limit/offset).

Query Parameter Structure

```
GET /api/v1/projects/{project_id}/tasks
  ?limit=10
  &offset=20
  &status=TODO
  &priority=HIGH
  &assignee_id=550e8400-e29b-41d4-a716-446655440000
  &due_date_from=2024-01-01
  &due_date_to=2024-06-30
  &search=database
  &sort_by=due_date
  &sort_order=asc
```

All parameters are optional. When no parameters are provided, all accessible records are returned with default pagination.

Sorting Logic

Supported `sort_by` values for tasks: `created_at`, `updated_at`, `due_date`, `priority`, `status`. Default sort is `created_at` descending. `sort_order` accepts `asc` or `desc`. Invalid `sort_by` values return 422. Priority sorting follows the logical order LOW < MEDIUM < HIGH, which requires mapping priority strings to integers in the query or using a CASE expression in SQL.

Search Logic

The `search` parameter performs a case-insensitive partial match on the `title` field. Implemented using SQLAlchemy's `ilike()` method with a `%search_term%` pattern. Search is combined with all other filters using AND logic. There is no full-text search engine required for this scope — ILIKE is sufficient.

Example Requests

Fetch the second page of high-priority TODO tasks assigned to a specific user, sorted by due date:

```
GET /api/v1/projects/abc-123/tasks?status=TODO&priority=HIG
H&assignee_id=user-uuid&sort_by=due_date&sort_order=asc&lim
```

```
it=10&offset=10
```

Search for tasks containing "auth" in their title:

```
GET /api/v1/projects/abc-123/tasks?search=auth&limit=5&offs
et=0
```

---

8. FOLDER STRUCTURE

```
trello-lite/
├── app/
│   ├── main.py                  # FastAPI app instance, m
iddleware, router registration
│   ├── core/
│   │   ├── config.py            # Environment settings vi
a pydantic-settings
│   │   ├── security.py          # JWT creation, decoding,
password hashing utilities
│   │   ├── dependencies.py      # FastAPI dependencies: g
et_db, get_current_user, require_admin
│   │   └── exceptions.py        # Custom exception classe
s and global exception handlers
│   ├── models/
│   │   ├── base.py              # SQLAlchemy declarative
base
│   │   ├── user.py              # User ORM model
│   │   ├── project.py           # Project ORM model
│   │   ├── task.py              # Task ORM model
│   │   ├── task_assignment.py   # TaskAssignment ORM mode
l
│   │   └── token_blacklist.py   # TokenBlacklist ORM mode
l
│   ├── schemas/
│   │   ├── common.py            # Shared schemas: Paginat
edResponse, ErrorResponse
│   │   ├── user.py              # UserCreate, UserUpdate,
UserResponse
```

```
│   │   ├── project.py           # ProjectCreate, ProjectU
pdate, ProjectResponse
│   │   ├── task.py              # TaskCreate, TaskUpdate,
TaskResponse, TaskFilter
│   │   └── auth.py              # LoginRequest, TokenResp
onse, RefreshRequest
│   ├── repositories/
│   │   ├── base.py              # Generic base repository
with common CRUD methods
│   │   ├── user_repository.py   # User-specific queries
│   │   ├── project_repository.py # Project-specific querie
s
│   │   ├── task_repository.py   # Task-specific queries w
ith filter/search logic
│   │   └── assignment_repository.py
│   ├── services/
│   │   ├── auth_service.py      # Registration, login, to
ken refresh, logout logic
│   │   ├── user_service.py      # Profile management, pas
sword change, admin operations
│   │   ├── project_service.py   # Project CRUD with autho
rization checks
│   │   ├── task_service.py      # Task CRUD with role-bas
ed field restrictions
│   │   └── assignment_service.py # Assign/unassign with du
plicate prevention
│   ├── routers/
│   │   ├── auth.py              # /auth routes
│   │   ├── users.py             # /users routes
│   │   ├── projects.py          # /projects routes
│   │   ├── tasks.py             # /projects/{id}/tasks ro
utes
│   │   └── assignments.py       # /projects/{id}/tasks/{i
d}/assignments routes
│   └── db/
│       ├── session.py           # SQLAlchemy engine and s
ession factory
│       └── migrations/          # Alembic migration files
```

```
│           └── versions/
├── tests/
│   ├── conftest.py              # Shared fixtures: test D
B, test client, test users
│   ├── unit/
│   │   ├── services/
│   │   │   ├── test_auth_service.py
│   │   │   ├── test_project_service.py
│   │   │   └── test_task_service.py
│   │   └── core/
│   │       └── test_security.py
│   └── integration/
│       ├── test_auth_routes.py
│       ├── test_project_routes.py
│       ├── test_task_routes.py
│       └── test_assignment_routes.py
├── alembic.ini
├── .env                         # Environment variables
(not committed to git)
├── .env.example                 # Template for environmen
t variables
├── requirements.txt
├── requirements-dev.txt         # Testing and dev depende
ncies
└── README.md
```

Folder Purpose Summary

`core/` contains cross-cutting concerns: settings, security utilities, reusable dependencies, and error handling. Nothing in core should import from routers, services, or repositories.

`models/` contains only SQLAlchemy ORM class definitions. No business logic here.

`schemas/` contains Pydantic models for request validation and response serialization. These are completely separate from ORM models.

`repositories/` contains all database interaction. No business logic, no HTTP concepts. Returns ORM model instances or raises database-level exceptions.

`services/` contains all business logic and authorization. Services call repositories. Services raise HTTP exceptions (this is acceptable in FastAPI's pattern). Services do not know about HTTP requests or response objects.

`routers/` contains only route definitions and HTTP-layer concerns. They call services and return Pydantic responses.

`db/` contains the SQLAlchemy session setup and Alembic migration configuration.

`tests/` is fully separated into unit tests (with mocks) and integration tests (with a real test database).

---

9. TESTING STRATEGY

What to Test

Every service function must have unit tests. Every route must have at least one integration test covering the happy path and the most important error paths. Authorization logic is particularly critical and must be tested exhaustively — every forbidden action must have a test proving it returns 403.

Unit vs Integration

Unit tests target the service layer in isolation. All repository calls are mocked. This makes unit tests fast and deterministic. Unit tests verify: business rules, authorization checks, correct calls to the repository with the right arguments, and correct exceptions raised under error conditions.

Integration tests target the full request/response cycle using FastAPI's `TestClient`. They use a real SQLite or PostgreSQL test database (PostgreSQL is strongly preferred to match production). Integration tests verify: correct HTTP status codes, correct response shapes, correct database state after mutations, and correct behavior of dependent systems like token blacklisting.

Mocking Strategy

Use Python's `unittest.mock.patch` or the `pytest-mock` library's `mocker` fixture. In unit tests, mock at the repository boundary — inject a mock repository into the service rather than patching SQLAlchemy internals. Services should accept repositories as constructor arguments or via dependency injection to make this

clean. For integration tests, use a real test database seeded with fixture data — do not mock anything in integration tests.

Example Test Cases

Authentication service unit tests: test that registration hashes the password and does not store the plain text; test that duplicate email raises 409; test that login with wrong password raises 401; test that login with deactivated account raises 403; test that logout adds the JTI to the blacklist.

Project service unit tests: test that create_project sets owner_id to the current user's id; test that update_project by a non-owner raises 403; test that delete_project cascades (mock verifies repository.delete_tasks_by_project is called).

Task service unit tests: test that an assigned user updating title raises 403; test that an assigned user updating status succeeds; test that creating a task as a non-owner raises 403.

Integration tests: test the full login-then-create-project flow; test pagination returns correct total and items count; test filtering by status returns only matching tasks; test search returns tasks with partial title match.

Test Coverage Goals

Aim for 80% overall coverage as a minimum. The service layer should reach 90%+ coverage. The authorization paths specifically must achieve 100% branch coverage — every allowed and forbidden combination must be tested. Use `pytest-cov` to generate coverage reports. Coverage is checked per layer, not just globally.

---

10. 1-MONTH EXECUTION PLAN

Week 1 — Project Setup & Core CRUD

The goal of week 1 is to have a running API with full CRUD operations for all resources, no authentication yet.

Start by setting up the project structure exactly as defined in Section 8. Initialize a virtual environment, install FastAPI, SQLAlchemy, Alembic, psycopg2, and uvicorn. Configure pydantic-settings to load from a `.env` file.

Set up the PostgreSQL database and write the initial Alembic migration for all five tables. Write all ORM models. Write all Pydantic schemas for create, update, and response for all resources. Write the base repository with generic get, create, update, delete methods. Write concrete repositories for User, Project, Task, and Assignment. Write services for Project, Task, and Assignment with no authorization yet. Write all routers and register them on the app. At the end of week 1, every endpoint should be reachable and functional with no auth enforcement.

Deliverables: running server with all CRUD endpoints returning correct data; database schema fully migrated; all schemas validating input correctly.

Week 2 — Authentication & Authorization

The goal of week 2 is to lock down the API with JWT authentication and implement all authorization rules.

Write the security utilities in `core/security.py`: password hashing with bcrypt, JWT creation with PyJWT, JWT decoding. Write the AuthService with register, login, refresh, and logout functions. Write the auth router. Write the `get_current_user` FastAPI dependency and inject it into all protected routes. Write the `require_admin` dependency for admin-only routes. Now go back to every service written in week 1 and add authorization checks per the rule table in Section 6. Implement the token blacklist check in the refresh token validation path. Implement the partial-update restriction for assigned users on tasks (they can only update status). Write the `get_current_user_if_member` dependency that validates project membership for task routes.

Deliverables: all endpoints protected; register/login/refresh/logout working and tested manually; 403 returned for all unauthorized actions; admin role enforced; token blacklisting working.

Week 3 — Advanced Features: Filtering, Search, Pagination, Sorting

The goal of week 3 is to complete the filtering, search, pagination, and sorting features on all list endpoints.

Update the task repository to accept a `TaskFilter` schema that encapsulates all filter parameters. Implement the ILIKE search on title. Implement date range filters. Implement status and priority filters. Implement assignee_id filter using a

JOIN on task_assignments. Implement sorting by mapping `sort_by` and `sort_order` parameters to SQLAlchemy `order_by()` clauses, with special handling for priority ordering. Implement the paginated response wrapper in a generic utility function used by all list endpoints. Apply the same pagination pattern to the project list and user list (admin) endpoints. Validate all query parameters at the router level using Pydantic query parameter models. Return 422 for invalid sort_by values.

Deliverables: all list endpoints support filtering, search, pagination, and sorting; paginated response format is consistent across all endpoints; all edge cases handled (empty results, out-of-range offset, etc.).

Week 4 — Testing, Polishing, and Documentation

The goal of week 4 is to reach test coverage goals, fix bugs, and make the project production-ready.

Write all unit tests for the service layer with mocked repositories. Write all integration tests with a real test database. Focus test effort on authorization paths — write a dedicated test file that exhaustively tests every forbidden action. Run `pytest-cov` and address any gaps below the coverage targets. Review all error messages for clarity and consistency. Ensure all 422 responses include useful validation error details (FastAPI does this automatically from Pydantic). Add global exception handlers in `core/exceptions.py` for clean error responses. Write a `README.md` with setup instructions, environment variable documentation, and example curl requests for every endpoint. Review the folder structure and refactor any service or repository that grew too large. Do a final pass on security: ensure no passwords are logged, ensure JWT secret is not hardcoded, ensure CORS middleware is configured. Run the full test suite and confirm it passes cleanly.

Deliverables: full test suite passing with 80%+ overall coverage and 90%+ service layer coverage; clean README with setup guide; all error responses consistent and informative; no hardcoded secrets.

This specification is complete and self-contained. Every decision documented here is intentional. When in doubt during implementation, refer back to the authorization rule table in Section 6, the lifecycle description in Section 1, and the cascade rules in Section 3. Build in the order of the execution plan — resist the urge to add auth before you have working CRUD, as debugging auth on top of broken CRUD doubles the complexity.