**1. Backend Architecture and Design**

**Monolithic Architecture**

* **Django as core backend framework, followas a monolithic architecture.**
* **Django’s MVT pattern is well-suited.**
* **React will handle view layer**

****Layered Approach****

* **Presentation Layer: Django will expose RESTful APIs (using Django REST Framework - DRF) to communicate with the React frontend.**
* **Business Logic Layer**: The task management features, such as task creation, updates, deletion, and notifications, will reside in this layer.
* **Data Access Layer**: Handle interactions between the Django models and PostgreSQL database.

**2. **Database Design****

****Core Models****

* **User**
* **Task**
* **Project** (Optional, for grouping tasks)
* **Team** (Optional, if the system includes teams)
* **Comment** (Optional, for task discussions)

****Model Fields and Relationships****

**User Model**

**Fields:**

* username
* email
* password

**Task Model**

**Fields:**

* title (CharField): The title of the task.
* description (TextField): A detailed description of the task.
* status (CharField): The current status of the task (e.g., To-Do, In Progress, Done).
* priority (CharField): Priority level (e.g., Low, Medium, High).
* due\_date (DateField): The deadline for the task.
* created\_at (DateTimeField): The timestamp for when the task was created.
* updated\_at (DateTimeField): The timestamp for when the task was last updated.
* assigned\_to (ForeignKey): A reference to the User who is responsible for the task.
* project (ForeignKey): A reference to a project (if relevant).
* team (ForeignKey): A reference to the team responsible for the task (optional).

**Relationships:**

* A task is **owned** by a user (ForeignKey).
* A task may **belong to a project** (ForeignKey).
* Tasks can have **many-to-many relationships** with teams (if used).

**Project Model (Optional)**

**Fields:**

* name **(CharField): The name of the project.**
* description (TextField): A description of the project.
* created\_at (DateTimeField): When the project was created.
* updated\_at (DateTimeField): When the project was last updated.

**Relationships:**

* A project can have many tasks (ForeignKey in the Task model).
* A project can be managed by a user or a team (ForeignKey for user/team).

**Team Model (Optional)**

**Fields:**

* name (CharField): The name of the team.
* created\_at (DateTimeField): When the team was created.
* leader (ForeignKey): The team leader (User).

**Relationships:**

* A team can have many users (ManyToManyField).
* A team can be responsible for many tasks.

**Comment Model (Optional)**

**Fields:**

* content (TextField): The comment content.
* author (ForeignKey): Reference to the user who made the comment.
* task (ForeignKey): The task the comment is associated with.
* created\_at (DateTimeField): When the comment was made.

**3. API Design (Django REST Framework)**

**RESTful API**

* We’ll design RESTful API endpoints for managing tasks:
  + **GET /api/tasks/**: Retrieve a list of tasks.
  + **POST /api/tasks/**: Create a new task.
  + **GET /api/tasks/**

**/**: Retrieve a specific task by ID.

* + **PUT /api/tasks/**

**/**: Update a task.

* + **DELETE /api/tasks/**

**/**: Delete a task.

### ****1. Key API Endpoints****

For a Task Management System, the following main endpoints will be required:

#### ****User Authentication****

* **Register User**: POST /api/users/register/
* **Login User**: POST /api/users/login/
* **Logout User**: POST /api/users/logout/
* **User Profile**: GET /api/users/me/

#### ****Tasks****

* **List All Tasks**: GET /api/tasks/ (optional filters: by status, project, assigned user)
* **Retrieve Task**: GET /api/tasks/{task\_id}/
* **Create Task**: POST /api/tasks/
* **Update Task**: PUT /api/tasks/{task\_id}/
* **Delete Task**: DELETE /api/tasks/{task\_id}/

#### ****Projects****

* **List All Projects**: GET /api/projects/
* **Retrieve Project**: GET /api/projects/{project\_id}/
* **Create Project**: POST /api/projects/
* **Update Project**: PUT /api/projects/{project\_id}/
* **Delete Project**: DELETE /api/projects/{project\_id}/

#### ****Teams****

* **List All Teams**: GET /api/teams/
* **Retrieve Team**: GET /api/teams/{team\_id}/
* **Create Team**: POST /api/teams/
* **Update Team**: PUT /api/teams/{team\_id}/
* **Delete Team**: DELETE /api/teams/{team\_id}/

#### ****Comments****

* **List All Comments for a Task**: GET /api/tasks/{task\_id}/comments/
* **Create Comment for a Task**: POST /api/tasks/{task\_id}/comments/
* **Delete Comment**: DELETE /api/comments/{comment\_id}/

**2. Models and Serializers**

**3. Views (API Endpoints) - In Django REST Framework, you can create API views using function-based views or class-based views. I will use class-based views with viewsets to make the code more modular and scalable.**

### ****4. URLs and Routers****

We need to map the API endpoints to the views. Django REST Framework provides a router system that makes this easy.

### ****5. Authentication and Permissions****

We’ll use DRF’s built-in authentication and permission classes to secure the API. For authentication, we can use Django’s session-based authentication or token-based authentication (e.g., JWT).

### ****6. Pagination and Filtering****

For large datasets, it’s good to implement pagination and filtering. Django REST Framework supports pagination out of the box.

### ****7. Permissions and Role-Based Access Control****

If different user roles (like admin, team member, or viewer) have different permissions for tasks or projects, you can implement custom permissions.

#### ****Custom Permission Example****

For example, only task owners should be able to edit or delete their tasks:

### ****8. Testing the API****

You can test the API using tools like **Postman** or **cURL** to make sure the endpoints work as expected. Additionally, Django has built-in support for writing unit tests for your API endpoints.