1. **Project Setup**:
    - The project was initialized using Django, a high-level Python web framework, and Django REST Framework (DRF), which provides powerful tools for building RESTful APIs.
    - A new Django project (`task_manager`) and a corresponding app (`api`) were created to encapsulate the core functionalities.
2. **Database Configuration**:
    - The project is configured to use a MySQL relational database to store user and task data. This choice was driven by the requirement to use a relational database and ensures robust data management capabilities with support for complex queries and transactions.
3. **User Registration and Authentication**:
    - Django's built-in `User` model was leveraged for user management to simplify the setup of user registration and authentication.
    - Authentication was implemented using JWT (JSON Web Tokens) via the `djangorestframework-simplejwt` package. This ensures stateless, secure user sessions by issuing tokens that users must present for subsequent requests.
4. **Task Management**:
    - A `Task` model was created with fields such as `id`, `title`, `description`, `status`, `priority`, `due date`, `created_at`, `updated_at`, and `user_id`. This model supports CRUD operations for task management.
    - DRF's `ModelViewSet` was utilized to handle these CRUD operations, ensuring that only authenticated users can manage tasks and that users can only see their tasks.
5. **Filtering and Searching**:
    - DRF's filtering and searching capabilities were integrated into the task API, allowing users to filter tasks by status, priority, and due date and search by title or description. This enhances user experience by providing customizable task management options.
6. **Dockerization**:
    - Docker was used to containerize the application, creating a reproducible environment for deployment and development.
    - A `Dockerfile` was created to define the environment for the Django application, while a `docker-compose.yml` file was set up to manage multi-container Docker applications, including the Django web server and a MySQL database service.
7. **Additional Features**:
    - Potential extensions like role-based access control, pagination, and CI/CD pipeline setups were considered. These features provide added security, improve performance, and automate development workflows.

Assumptions

1. **User Authentication**:
    - It was assumed that JWT authentication is suitable for this project due to its stateless nature and the security advantages of using tokens.
2. **Task Visibility**:
    - The system assumes that users should only have access to their tasks, hence each task is tied to a specific user and queries are filtered accordingly.

3. **Basic User Roles**:
    - Initially, all users are treated equally without distinguishing roles (like Admin vs. User). Role-based access control was suggested as an additional feature but not implemented in the base setup.
4. **Docker Setup**:
    - The Docker setup assumes the host environment can run Docker and Docker Compose without additional configuration. It also assumes that the MySQL container's environment variables are set correctly and securely.
5. **Environment Variables**:
    - It is assumed that sensitive data like database credentials and JWT secret keys will be managed using environment variables in a production environment, although they were hard-coded in the example for simplicity.
6. **Development and Production Parity**:
    - The development environment closely mirrors the production setup using Docker, but additional configurations would be needed for production readiness, such as database backups, error logging, and secure secret management.

This approach ensures a scalable, secure, and maintainable application architecture that meets the project requirements