Design Document: **Sortable and Performant GET Tasks API**

Issue	Sortable and Performant GET Tasks API
Delivery Date	TBD
Document Status	Approved
Document & Feature Owner	Achintya Chatterjee

Reviewer	Date	Action
Shobhan Sundar Goutam	Jun 26, 2025	Approved
Anuj Chhikara	Jun 29, 2025	Approved
Tejas	July 01, 2025	Approved

Overview

This design document outlines the plan to enhance the **GET /v1/tasks/** API. The current implementation suffers from a critical performance bottleneck: it loads the entire collection of tasks from the database into memory and then performs pagination. This is not scalable. This refactor will address this issue by moving both pagination and new sorting functionality to the database level, ensuring the API is both performant and more feature-rich.

Purpose and Goals

The primary purpose is to provide users with a more powerful and responsive way to view their tasks.

Primary Goal

- Implement robust, user-configurable sorting for the task list based on **priority**, **dueAt**, **createdAt**, and **assignee**.

- Refactor the pagination mechanism to execute at the database level. This will prevent the loading of the entire tasks collection into memory and fix the underlying performance issue.

Secondary Goal

- Ensure the new implementation is well-tested, maintainable, and that sorting and pagination work together seamlessly.

Requirements

1. API Endpoint:

- The existing endpoint **GET /v1/tasks/** will be enhanced.

2. Query Parameters:

- **sort_by**: Specifies the field to sort on.
    - **Allowed Values: priority**, **dueAt**, **createdAt**, **assignee**.

- **order**: Specifies the sort direction.
    - **Allowed Values: asc** (ascending), **desc** (descending).

3. Default Behaviour:

- If **sort_by** is not provided, the API will default to sorting by **createdAt** in **desc** order.
- If **sort_by** is provided but **order** is not, a sensible default order will be applied:
    - **createdAt**: **desc**
    - **dueAt**: **asc**
    - **priority**: **desc** (High to Low)
    - **assignee**: **asc** (Alphabetical)

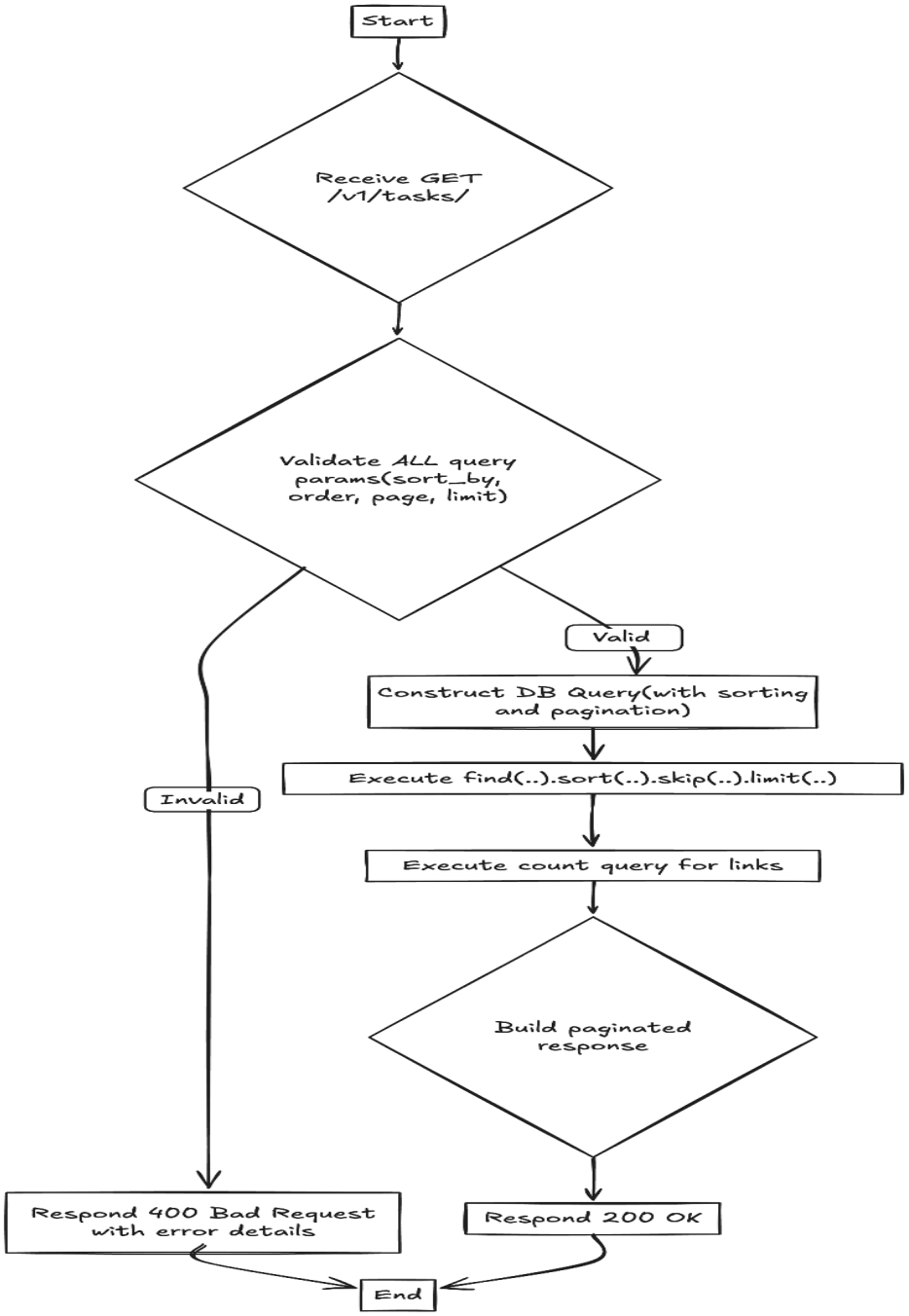
4. Invalid Parameter Handling:

- If an invalid value is provided for **sort_by** or **order**, the API **must return a 400 Bad Request** with a clear error message. (The existing **GetTaskQueryParamsSerializer** already handles this correctly for **page** and **limit**.)

5. Database-Level Operations:

- To ensure optimal performance, sorting and pagination **must** be applied directly within the database query.

Flow Diagram



Proposed Solutions: Service-Oriented Implementation

This solution refactors the existing endpoint to be performant and scalable while adding the new sorting feature.

Files to Create/Modify:

1. Serializer (todo/serializers/get_tasks_serializer.py):

- The **GetTaskQueryParamsSerializer** will be modified to include **sort_by** and **order** fields. **serializers.ChoiceField** will be used to validate against the allowed values, and **default** values will be set to handle the default sorting requirement. This ensures invalid parameters automatically trigger a **400 Bad Request**.

2. View (todo/views/task.py):

- The **get** method in **TaskListView** will be updated to pass the newly validated **sort_by** and **order** parameters from the serializer's **validated_data** to the **TaskService**. No other significant change is needed here, as it already correctly uses **is_valid(raise_exception=True)**.

3. Service (TaskService):

- The **get_tasks** method will be refactored to eliminate the in-memory pagination. It will no longer call **TaskRepository.get_all()**.
- It will now call two repository methods:
    a. **TaskRepository.list(page, limit, sort_by, order)** to get the specific page of sorted data.
    b. **TaskRepository.count()** to get the total number of tasks for building the pagination links.
- The logic to construct the **LinksData** will be based on the total count and the current page/limit.

4. Repository (todo/repositories/task_repository.py):

- The **list** method will be enhanced to accept **sort_by** and **order** as parameters. It will dynamically construct the sort criteria and chain the **.sort()** method to the **find()** query before applying **.skip()** and **.limit()**.
- The **get_all** method will be deprecated for this flow.

Recommended Solution

The **Service-Oriented Implementation** described above is my recommended solution. It directly addresses the critical performance issue while correctly implementing the new sorting feature within the existing architecture. It improves scalability, maintainability, and API correctness by adhering to best practices.

- **Scalability:** The API will now handle large datasets efficiently.
- **Maintainability:** Logic remains separated by layer (View, Service, Repository).
- **Correctness:** Sorting is applied to the entire dataset before pagination, which is the logically correct way to implement this feature.

Implementation Strategy

1. **Serializer:** Update **GetTaskQueryParamsSerializer** to include and validate **sort_by** and **order**.
2. **Repository Layer:** Refactor the **TaskRepository.list** method to apply sorting criteria before pagination. Remove the **get_all** method.
3. **Service Layer:** Refactor the **TaskService.get_tasks** method to use the updated repository methods and remove the in-memory pagination logic.
4. **View Layer:** Update **TaskListView** to pass the new, validated parameters from the serializer to the service.
5. **Database Performance and Indexing Strategy**
   To deliver on the performance goals, an indexing strategy is required. The aim is to prevent slow, in-memory database sorts for the most critical API operations.
   Essential Indexes for Core Functionality:
   We will add two indexes to support the most important sorting operations:

   - **Default Sort Index (createdAt):** The API defaults to sorting by createdAt. To prevent a costly in-memory sort of the entire collection on every default API call, an index on this field is required. Without it, the API will not be scalable.
     Command: db.tasks.createIndex({ "createdAt": -1 })
   - **Assignee Sort Index ($lookup):** To sort by assignee name, a $lookup aggregation is necessary. For this join to be performant, an index must be placed on the assignee foreign key field in the tasks collection.
     Command: db.tasks.createIndex({ "assignee": 1 })
6. **Testing:** Write comprehensive unit and integration tests to verify that sorting and pagination work correctly together for all supported fields and that invalid parameters return a **400 Bad Request**.