Django Exercise: Advanced Project and Task Management System

You will create a system where users can manage **projects** and **tasks** using advanced Django features. These include class-based views, custom managers, signals, caching, session management, and form validation.

Tasks with Hints:

1. Create a Django Project

- **Task**: Set up a new Django project called `advanced_project_manager` and create an app called `projects`.

Hint:

- Use `django-admin startproject advanced_project_manager` and `python manage.py startapp projects`.
- Remember to add `'projects'` to `INSTALLED_APPS` in `settings.py`.

---

2. Define Models with Custom Managers and QuerySets

1. **Project Model**:
    - Fields: `name` (CharField), `description` (TextField), `start_date` (DateField), `end_date` (DateField).
    - Implement a **custom manager** to filter active projects (projects that haven't ended yet).

    Hint:

    - Define a `ProjectManager` class that filters projects where `end_date` is greater than or equal to today's date.
    - Use `QuerySet.as_manager()` to apply this custom manager to your `Project` model.
2. **Task Model**:
    - Fields: `title` (CharField), `description` (TextField), `completed` (BooleanField), `due_date` (DateField), and a foreign key to the `Project` model.
    - Implement a **custom manager** to retrieve tasks based on their completion status.

    Hint:

    - Similar to the project, create a `TaskManager` class with methods like `completed()` and `incomplete()` to filter tasks by their status.

---

3. Class-Based Views for Projects and Tasks

1. **Project List and Task List Views**:
    - Use **ListView** to display the list of projects and tasks.
    - Add a filter using **GET parameters** to sort projects by their start date and tasks by their due date.

    Hint:

    - Use `get_queryset()` method in your `ListView` to apply filtering logic based on the `GET` parameters passed in the URL (e.g., `?sort=due_date`).
2. **Project Create and Task Create Views**:
    - Create and update projects and tasks using **CreateView** and **UpdateView**.

    Hint:

    - Use `CreateView` and `UpdateView` with `Project` and `Task` models and connect them to corresponding templates.
    - Define `form_valid()` to handle form submissions.
3. **Mixin for Project Sorting**:
    - Implement a **mixin** that sorts projects by their start date.
    - Apply this mixin to the project list view.

    Hint:

    - Create a `ProjectSortMixin` that overrides `get_queryset()` to add sorting functionality.
    - Use Python's `sorted()` function or `order_by()` in Django's ORM.
4. **Task Completion View**:
    - Implement a view that toggles the `completed` status of a task.

    Hint:

    - Use a class-based view (like `UpdateView` or `DetailView`) to change the `completed` status and save the task.

---

4. Implement Signals for Automatic Task Updates

- Use **Django signals** to automatically mark a project as "completed" when all associated tasks are marked as completed.

Hint:

- Use Django's `post_save` signal on the `Task` model. When a task is marked as completed, check if all other tasks for the same project are also completed. If yes, update the project's `status` to "completed."

---

5. Caching for Performance Optimization

1. **Cache the Project List**:
    - Implement **caching** on the project list view using Django's caching framework.
    - Cache the list of projects for 30 seconds.

    Hint:

    - In `views.py`, use the `@cache_page(30)` decorator for the project list view.
    - Alternatively, use low-level caching with `cache.set()` and `cache.get()` for finer control.
2. **Cache-Invalidation**:
    - Ensure that the cache is invalidated when a project is created, updated, or deleted.

    Hint:

    - Use signals or override `form_valid()` in the `CreateView` and `UpdateView` to invalidate the cache using `cache.delete()`.

---

6. Add Session-Based Task Management

- Implement session-based task filtering where users can save their preferred sorting/filtering settings for tasks (e.g., sort by due date or filter completed tasks).

Hint:

- Store user preferences in `request.session` (e.g., `request.session['task_filter'] = 'completed'`).
- In `get_queryset()`, apply the saved session filter to the task list.

---

7. Context Processors for Global Access

- Create a **context processor** that provides all ongoing projects globally across all templates (without manually passing them in every view).

Hint:

- In `context_processors.py`, create a function like `ongoing_projects` that retrieves active projects and returns them in a dictionary.
- Add the context processor in `TEMPLATES` settings under `OPTIONS['context_processors']`.

---

8. Django Forms and Validation

1. **Custom Form Validation**:
    - Add validation to the project creation form to ensure that the `end_date` is always after the `start_date`.
    - Add validation to the task form to ensure that tasks aren't marked as completed if their due date is in the future.

    Hint:

    - In `forms.py`, override `clean()` for both the project and task forms to perform custom validation.
    - Use `self.add_error()` to attach specific errors to fields.
2. **ModelForm with Mixins**:
    - Create a **ModelForm** that uses a mixin to handle common form validation logic (e.g., checking if a date is in the past).

    Hint:

    - Create a form mixin with a validation method like `check_future_date()` and apply it to both the project and task forms.

---

9. Testing Models, Views, and Signals

1. **Model Testing**:
    - Write tests for custom managers and querysets to ensure they return the correct data.
    - Test signals to ensure projects are automatically marked as complete when tasks are all completed.

    Hint:

    - Use `assertQuerysetEqual()` to test custom querysets and managers.
    - Trigger the signal by saving tasks in your tests and asserting the project's status.
2. **View Testing**:
    - Write tests for creating, updating, and deleting projects and tasks.

    Hint:

    - Use Django's `client.post()` for testing form submissions and `client.get()` for testing list and detail views.
3. **Signal Testing**:
    - Simulate task completions in the tests to ensure that signals trigger the correct project status updates.

    Hint:

- Use `django.test.signals` to ensure your signals work as expected during tests. Simulate the completion of all tasks and check the project's `completed` status.

---

10. URL Configuration

- Set up URLs for the following views:
  - Project list, create, update, and delete.
  - Task list (within a project), create, update, and delete.
  - A view to toggle the completion status of a task.

Hint:

- Use `path()` to configure URLs, and include primary keys (`<int:pk>`) for project and task update/delete views.

---

11. Templates

- Create templates for each view:
  - **Project List Template**: Display a list of projects, using caching and the mixin for sorting.
  - **Task List Template**: Display tasks within a project, allowing users to filter and sort.
  - **Project and Task Form Templates**: Use form validation and session-based filtering.
  - **Task Completion Toggle Template**: A simple button to mark a task as completed/incomplete.

Hint:

- Use `{% for project in projects %}` and `{% for task in tasks %}` loops in templates.
- Display session-based filtering options using `request.session`.

Bonus Challenges

1. **Task Notifications**:
   - Implement a system that notifies users via email when a project is nearing its end date or tasks are overdue.

   Hint:

   - Use Django's `send_mail()` function and set up a periodic check using **Celery** or Django management commands to send email notifications.
2. **Task Prioritization**:
   - Add a `priority` field to tasks (e.g., low, medium, high) and allow users to filter tasks by priority.

Hint:

- Create a `ChoiceField` for the task `priority` and use `get_queryset()` to apply the priority filter.

Learning Outcomes

By completing this exercise with hints, students will:

- Use **custom managers** and **querysets** to filter and retrieve specific data.
- Implement **Django signals** to automate task updates and project completion.
- Apply **caching** to improve performance for high-traffic views.
- Use **context processors** for global template data access.
- Handle **custom form validation** and manage forms using **mixins**.
- Write comprehensive **unit tests** for models, views, and signals.