Task 6: Authorization & Signals

Questions to Explore

1. What is @permission_required, and how does it work for function-based views?
2. What is PermissionRequiredMixin, and how is it used in class-based views?
3. What is the effect of setting raise_exception=True?
4. What is the difference between @login_required and @permission_required?
5. What are Django's default model permissions (add, change, delete, view)?
6. How can we define **custom permissions** using Meta.permissions in a model?
7. How can we use custom permissions using Meta.permissions?
8. How does user.has_perm() work internally?
9. When should we use model-level permissions instead of hard-coded role checks?
9. What is the difference between assigning permissions to a **Group** vs directly to a **User**?
10. Why are group-based permissions more scalable in real projects?
11. In what special cases is assigning permissions directly to users justified?
12. How can we restrict access so that **only the owner of an object can edit it**?
13. What are the limitations of Django's built-in permission system regarding object-level access?
14. How can we combine **ownership checks** with **model permissions**?
16. How can we implement a reusable owner_required decorator?
17. How can we write an OwnerOrPermissionMixin for class-based views?
18. What are the best practices for keeping permission logic clean and testable?

---

19. What are Django signals and when should they be used?
20. What is the post_save signal and how does it work?
21. What is m2m_changed?

22. What is post_migrate signal?

23. Why should we avoid heavy business logic inside signals?

Deliverables

These Deliverables are extend of task 5 project.

1. Permission Decorators & Mixins

- Use @permission_required in at least one function-based view (e.g., create_project).
- Use PermissionRequiredMixin in at least one class-based view (e.g., ProjectUpdateView).
- Implement a custom decorator owner_required to allow only the object owner or a user with a specific permission to access certain views.
- Implement a custom mixin OwnerOrPermissionMixin for CBVs to enforce the same logic.
- Unauthorized access should return **HTTP 403** (PermissionDenied).

2. Model-level Permissions

- Add at least one **custom permission** to your Project model(class Meta)
- Protect at least one view using this permission (@permission_required(custome permission) or user.has_perm()).

3. Group-based Roles

- Create these groups: Admin, Manager, Developer.
- Assign permissions:
    - **Admin**: add/change/delete/view projects, approve projects, edit all profiles
    - **Manager**: add/change/view projects, approve projects (no delete), view all profiles
    - **Developer**: view all profiles, view projects, edit own profile only, edit own projects only
- Try to use mixin and decorator together.
- create these groups automatically using post_migrate signal.
- Ensure signal code is safe to run multiple times(does not create groups multiple times).

5. Signals and Action Logging

- post_save on User:
    - Automatically create a Profile if missing
    - Assign the new user to the Developer group
- pre_save on Project or Profile:
    - just add a prefix to projects name.

---

references:

1. Django official documentation (you should find it yourselves now)
2. All over the internet (just not copy paste from chatbots)