

User: Context These are the feature folders/screens and backend I'm referring to

What I want you to do first

Analyze the folders/screens I referenced above

Understand their current purpose and how they should logically map to backend concepts (tasks, bids, users, roles, etc.)

The actual problem

Right now, my Flutter screens feel random and disconnected. They don't align well with how the backend is structured or how data flows.

Because of this:

Screens overlap in responsibility

Task-related logic is scattered

Role-based behavior (Poster vs Tasker) is unclear

Switching modes feels hacky instead of intentional

What I want from you

I want high-level architectural guidance, not code.

1. Screen ↔ Backend Alignment

How should Flutter screens be structured so they cleanly reflect backend entities and workflows?

For example:

How should task-related screens be grouped and flow together?

Home

Task list

Task details

Post task

Place bid

My tasks

My posted tasks

What responsibilities should each screen have?

What should NOT be handled by certain screens?

2. Tasks & Bids Flow Design

How should task creation, bidding, assignment, and status changes be represented on the frontend?

Which backend states should directly map to UI states?

How to avoid duplicating logic across multiple screens?

3. Role-Based Screen Management (Poster vs Tasker)

How should the app conceptually handle two roles?

Poster

Tasker

Should screens be:

Shared with conditional behavior?

Or separated by role?

What are the trade-offs of each approach?

4. Role Switching (from Settings)

How should switching between Tasker ↔ Poster be handled from a UX and architecture point of view?

What should happen when the role changes:

Navigation reset?

Screen rebuild?

Cached state invalidation?

What backend data should drive this switch?

5. Backend Changes (if needed)

If some frontend requirements are hard to support cleanly:

What backend changes would make the frontend simpler or more predictable?

Examples:

Role-aware endpoints?

Task status normalization?

Separate “views” of tasks for posters vs taskers?

Important constraints

☒ No implementation or code

☐ Only architecture, structure, and design reasoning

I want minute details, edge cases, and trade-offs

Focus on restructuring, not patching what already exists

GitHub Copilot: I'll analyze the backend and frontend code to provide comprehensive architectural guidance. Let me first understand the current structure and relationships.

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/models.py#1-1), lines 1 to 200]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/accounts/models.py#1-1), lines 1 to 200]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/views.py#1-1), lines 1 to 200]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/views.py#200-200), lines 200 to 400]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/state_machine.py)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/serializers.py#1-1), lines 1 to 200]

Now let me read the Flutter screens to understand the current frontend structure:

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/home/screens/home_screen.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/task_details/screens/task_details_screen.dart#1-1), lines 1 to 200]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/my_tasks/screens/my_tasks_screen.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/my_posted_tasks/screens/my_posted_tasks_screen.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/post_task/screens/post_task_screen.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/place_bid/screens/place_bid_screen.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/settings/screens/settings_screen.dart#1-1), lines 1 to 150]

Let me also check the role switching mechanism and additional screens:

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/task_details/screens/task_details_tasker_screen.dart#1-1), lines 1 to 150]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features/settings/screens/widgets/settings_mode_toggle.dart)]

Searched for files matching ****/controllers/**/*.dart**, 15 matches

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/controllers/user_controller.dart)]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Api/tasks/views.py#400-400), lines 400 to 513]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/controllers/features/task_details_controller.dart)]

Searched for files matching ****/models/**/*.dart**, 4 matches

Searched for regex **enum.*Role|TaskRole|UserRole (**/*.dart)**, 17 results

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/authentication/models/user_model.dart#1-1), lines 1 to 100]

Read [(file:///d%3A/Study%20Data/Semester%207/FYP-I/Source%20Code/Actual%20Project/Final%20App/lib/features)]

Now I have a complete understanding of both your backend and frontend architecture. Let me provide comprehensive architectural guidance.

Architectural Analysis & Recommendations

Current State Summary

Backend Structure

Your backend has a well-defined structure:

- **User.role:** **POSTER**, **TASKER**, or **BOTH** (capability/permission)
- **Task States:** **POSTED** → **BIDDING** → **ASSIGNED** → **IN_PROGRESS** → **COMPLETED** → **CONFIRMED** (with **DISPUTED/CANCELLED** branches)
- **Bid States:** **ACTIVE**, **ACCEPTED**, **REJECTED**
- **Task Filters:** **my_tasks** (poster's tasks), **my_bids** (tasker's bids), **assigned_to_me** (tasker's assigned work)

Frontend Problems Identified

- 1. **Two conflicting concepts of "role"**: Backend role (capability) vs. active viewing mode (intent)
- 2. **Duplicate task details screens**: `TaskDetailsScreen` + `TaskDetailsTaskerScreen`
- 3. **Fragmented task screens**: `MyTasksScreen` (tasker view) + `MyPostedTasksScreen` (poster view) with unclear boundaries
- 4. **Disconnected mode toggle**: Settings toggle UI exists but isn't wired to state
- 5. **Local TaskRole enum** defined ad-hoc in controller, not centralized
- 6. **Home screen** doesn't differentiate what it shows based on mode

1. Screen ↔ Backend Alignment

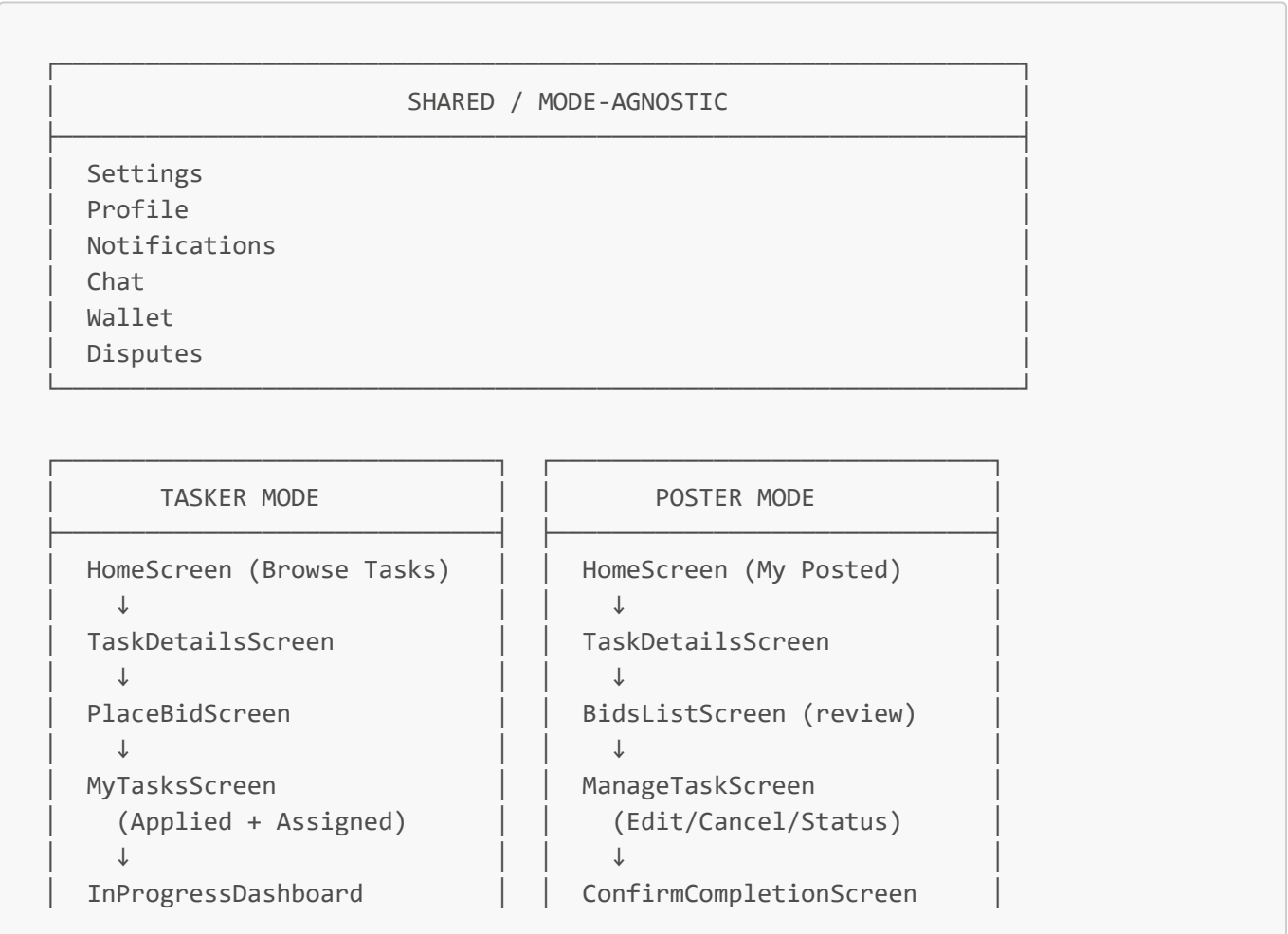
Conceptual Model: "Viewing Mode" vs "User Role"

This is the **critical distinction** your architecture is missing:

| Concept | What It Is | Where It Lives | Example |
|-------------|---|---|--------------------------------------|
| User Role | What can the user do? (Permission) | Backend <code>User.role</code> field | <code>BOTH</code> = can post and bid |
| Active Mode | What does the user want to do right now? (Intent) | Frontend-only state | "I'm currently looking for work" |

A user with `role=BOTH` should be able to **switch between modes** without changing their backend role.

Recommended Screen Architecture





Screen Responsibilities

HomeScreen

Responsibility: Discovery entry point based on active mode

| Mode | Shows | Backend Endpoint |
|--------|---------------------------------------|-----------------------------|
| Tasker | Nearby tasks available for bidding | GET /tasks/nearby/ |
| Poster | User's posted tasks + CTA to post new | GET /tasks/?filter=my_tasks |

What NOT to do: Do not show both perspectives on one Home screen with tabs. This confuses the mental model.

TaskDetailsScreen (SHARED, not duplicated)

Responsibility: Single source of truth for viewing any task

The **same screen** should render differently based on:

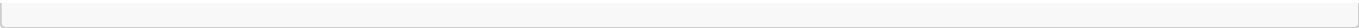
- 1. **Relationship to task:** Are you the poster, a bidder, the assigned tasker, or a stranger?
- 2. **Task status:** Determines which actions are available

Decision matrix for content:

| User Relationship | Task Status | Content Shown |
|---------------------|----------------------|--|
| Stranger | BIDDING | Task details + "Place Bid" CTA |
| Stranger | Any other | "Task no longer available" |
| Bidder (unassigned) | BIDDING | Task details + "Bid Placed" status + Edit/Withdraw |
| Bidder (unassigned) | ASSIGNED | "You were not selected" |
| Assigned Tasker | ASSIGNED/IN_PROGRESS | Work dashboard, chat, mark complete |
| Assigned Tasker | COMPLETED | "Awaiting confirmation" |
| Poster | BIDDING | Task details + Bids list + Accept bid CTA |
| Poster | ASSIGNED | Assigned tasker info + Chat |
| Poster | COMPLETED | Confirm completion CTA |

Key insight: You should **never** pass `TaskRole` as a parameter. Compute it from:

```
context = determineContext(task, currentUser)
// Returns: 'poster' | 'assigned_tasker' | 'bidder' | 'stranger'
```



PlaceBidScreen

Responsibility: Bid creation only

- **Precondition:** User is in Tasker mode, task is in BIDDING status
- **Actions:** Submit bid amount + message
- **Navigation:** Returns to TaskDetailsScreen after success

What NOT to handle: Do not show bid editing here. That belongs in TaskDetailsScreen when user is already a bidder.

MyTasksScreen (Tasker Mode)

Responsibility: Track tasker's involvement across the lifecycle

Tab Structure:

| Tab | Contents | Backend Filter |
|-----------|--|--|
| Applied | Tasks where user has an active bid | ?filter=my_bids&bid_status=ACTIVE |
| Active | Tasks assigned to user that are not complete | ?filter=assigned_to_me&status__in=ASSIGNED,IN_PROGRESS |
| Completed | Tasks user completed | ?filter=assigned_to_me&status__in=COMPLETED,CONFIRMED |

What NOT to include: Posted tasks. Those belong in Poster mode.

MyPostedTasksScreen (Poster Mode)

Responsibility: Manage poster's task portfolio

Tab Structure:

| Tab | Contents | Backend Filter |
|-----------|---------------------------------|--|
| Active | Tasks in BIDDING or ASSIGNED | ?filter=my_tasks&status__in=BIDDING,ASSIGNED,IN_PROGRESS |
| Completed | Tasks in COMPLETED or CONFIRMED | ?filter=my_tasks&status__in=COMPLETED,CONFIRMED |
| Cancelled | Cancelled tasks | ?filter=my_tasks&status=CANCELLED |

PostTaskScreen

Responsibility: Task creation flow (multi-step)

- **Precondition:** User is in Poster mode AND has role != TASKER (can post)

- **Flow:** Info → Media → Review → Payment
- **Status on creation:** Backend creates with **BIDDING** immediately

2. Tasks & Bids Flow Design

State Machine Mapping (Backend → Frontend States)

Your backend has 8 task states. The frontend should map these to **4-5 meaningful UI states**:

| BACKEND TO UI STATE MAPPING | | |
|-----------------------------|---------------|----------------------------|
| Backend Status | UI State | Actions Available |
| POSTED | "Draft" | Edit, Delete (poster only) |
| BIDDING | "Open" | Place bid / View bids |
| ASSIGNED | "Assigned" | Start work / Chat |
| IN_PROGRESS | "In Progress" | Mark complete / Chat |
| COMPLETED | "Pending" | Confirm / Dispute (poster) |
| CONFIRMED | "Done" | Review, Archive |
| DISPUTED | "Disputed" | View dispute, respond |
| CANCELLED | "Cancelled" | None (read-only) |

Bid Flow Visualization



Avoiding Logic Duplication

Create a **TaskStatusHelper** utility (not in any screen):

```
// Pseudo-code structure
class TaskStatusHelper {
```



```
// Determines what the user can do with this task
static getAvailableActions(task, currentUser, activeMode):
    context = getUserContext(task, currentUser) // poster, assigned_tasker,
bidder, stranger

    if context == 'poster':
        return getPosterActions(task.status)
    if context == 'assigned_tasker':
        return getTaskerActions(task.status)
    if context == 'bidder':
        return getBidderActions(task.status, bid.status)
    return getStrangerActions(task.status)

static getPosterActions(status):
    switch status:
        BIDDING: ['view_bids', 'edit', 'cancel']
        ASSIGNED: ['chat', 'cancel']
        IN_PROGRESS: ['chat']
        COMPLETED: ['confirm', 'dispute']
        ...
}
```

This helper should be the **single source of truth** for action availability. Screens just render what the helper says.

3. Role-Based Screen Management

The Core Question: Shared vs Separate Screens?

Recommendation: Conditional rendering within shared screens, NOT separate screens.

Why NOT Separate Screens?

| Problem | Impact |
|-----------------------|---|
| Code duplication | Two screens to maintain, bugs appear in one but not other |
| Navigation complexity | Different routes for same conceptual action |
| Edge cases | What if assigned tasker is also poster on another task? |
| Deep linking | URLs become role-dependent |

Why Conditional Rendering?

| Benefit | Example |
|------------------------|---|
| Single source of truth | One <code>TaskDetailsScreen</code> knows all task details logic |
| Context-aware | Same screen, different actions based on user relationship |

| Benefit | Example |
|--------------------|--|
| Cleaner navigation | Always navigate to <code>/task/{id}</code> , screen figures out what to show |
| Maintainability | Fix once, applies everywhere |

Implementation Pattern

Do NOT do this:

```
// BAD: Passing role as parameter
Navigator.push(TaskDetailsScreen(role: TaskRole.poster))
```

DO this:

```
// GOOD: Screen computes context from data
class TaskDetailsScreen extends StatelessWidget {
  final String taskId;

  Widget build(context) {
    final task = taskController.getTask(taskId);
    final user = userController.currentUser;

    final viewContext = _computeViewContext(task, user);
    // viewContext: 'poster' | 'assigned_tasker' | 'bidder' | 'stranger'

    return switch (viewContext) {
      'poster' => _buildPosterView(task),
      'assigned_tasker' => _buildAssignedTaskerView(task),
      'bidder' => _buildBidderView(task),
      'stranger' => _buildStrangerView(task),
    };
  }
}
```

What About MyTasksScreen vs MyPostedTasksScreen?

These **should remain separate screens** because they represent fundamentally different modes:

- MyTasksScreen = "Work I'm doing for others" (Tasker mode)
- MyPostedTasksScreen = "Work I've outsourced" (Poster mode)

They are not the same concept with different views—they are **different concepts**.

However, they should NOT be navigable simultaneously. The active mode determines which one appears in the bottom navigation.

4. Role Switching (from Settings)

UX Design Principles

What "Switching Modes" Means

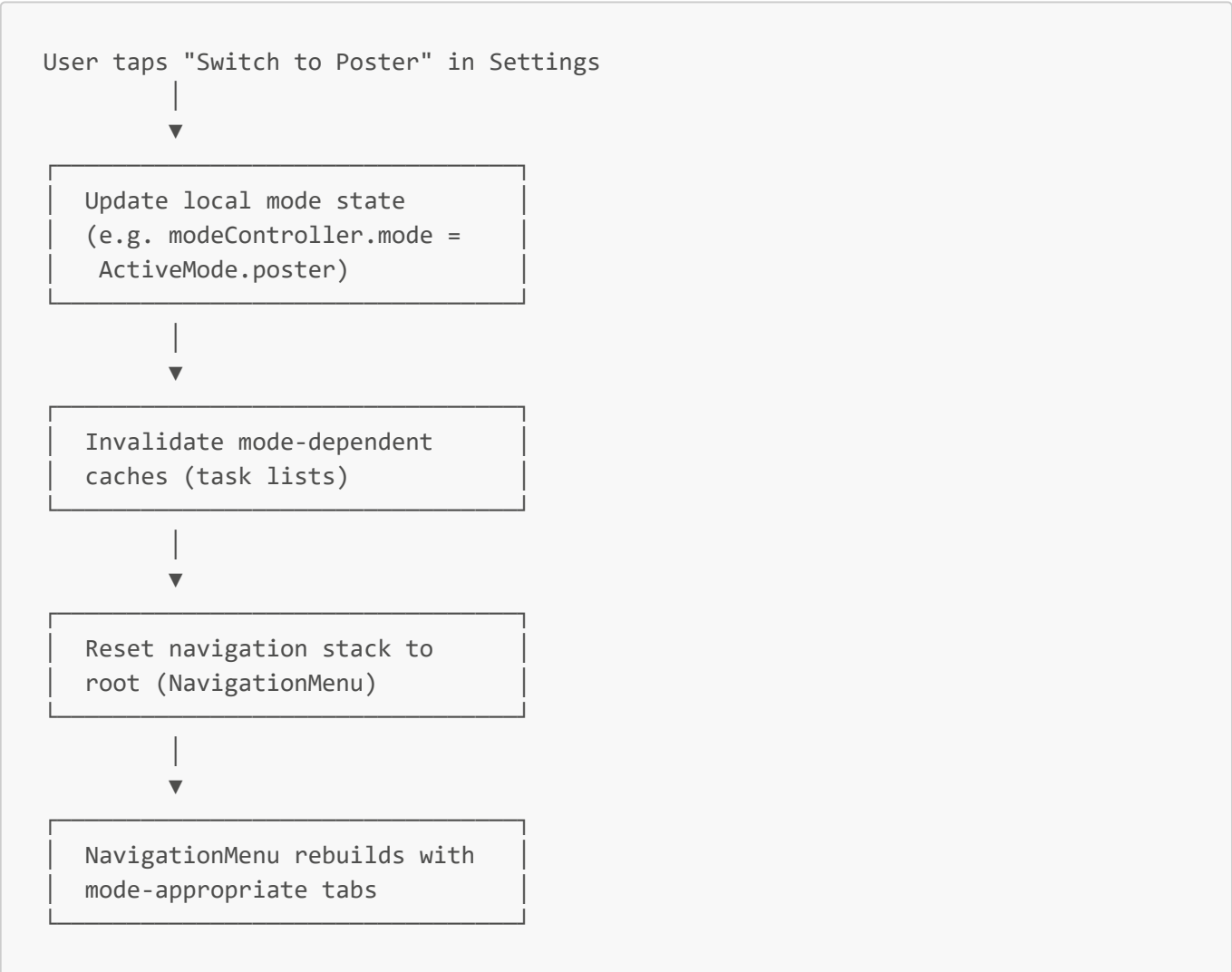
| Mode Switch | Analogy |
|-----------------|--|
| Tasker → Poster | Driver switching Uber app to "Rider" |
| Poster → Tasker | Customer switching to Seller on Amazon |

The switch should feel like **changing which hat you're wearing**, not logging into a different account.

What Should Happen on Switch

| Action | Recommendation | Reason |
|--------------------|--|-------------------------------------|
| Navigation reset | Yes | Bottom tabs should reflect new mode |
| Screen rebuild | Yes, for mode-dependent screens | Home shows different content |
| Cache invalidation | Partial | Clear task lists, keep user profile |
| Backend sync | Optional | If you track "last active mode" |

Recommended Switch Flow



State Architecture for Mode

Create a **dedicated ModeController** (separate from UserController):

```
ModeController (GetXController)
├─ activeMode: Rx<ActiveMode> // TASKER or POSTER
├─ switchMode(ActiveMode)     // Handles the switch logic
├─ canSwitchTo(ActiveMode)    // Check if user's role allows
└─ onModeChanged()            // Callback to reset navigation
```

Why separate from UserController?

- User data (role, profile, etc.) persists across mode switches
- Mode is transient frontend-only state
- Cleaner separation of concerns

Edge Cases to Handle

| Edge Case | Handling |
|---|---|
| User with role=TASKER tries to switch to Poster | Block switch, show "Upgrade to post tasks" |
| User has active bid and switches to Poster | Allow switch, bids still exist (mode is view preference) |
| User is assigned tasker on task, switches to Poster | Allow switch, but that task still shows assignment in MyTasks when they switch back |
| Deep link to task details while in wrong mode | Auto-switch to appropriate mode OR show "You're viewing from wrong mode, switch?" |

UI Feedback on Switch

The Settings toggle should:

1. Show current mode clearly (not just highlight)
2. Confirm switch with brief animation
3. Indicate navigation will reset (subtle warning)
4. Be instant (no loading states needed for local change)

5. Backend Changes

Recommended Additions

1. Add **active_mode** to User Model (Optional)

Purpose: Persist user's last active mode across sessions/devices

```
class User(AbstractUser):
    # Existing...
    active_mode = models.CharField(
        max_length=10,
        choices=[('POSTER', 'Poster'), ('TASKER', 'Tasker')],
        default='POSTER',
        help_text="User's current viewing mode (frontend hint)"
    )
```

Trade-offs:

- Pro: Consistent experience across devices
- Pro: Can be used for analytics ("how often do users switch?")
- Con: Extra API call to sync mode changes
- Con: Adds complexity for minor benefit

Recommendation: Skip unless you need cross-device sync. Frontend-only mode is simpler.

2. Create Role-Aware Task Endpoints

Currently you have `filter=my_tasks|my_bids|assigned_to_me`. Consider consolidating:

New endpoint: `GET /tasks/dashboard/`

```
@action(detail=False, methods=['get'])
def dashboard(self, request):
    """
    Returns tasks relevant to user based on mode parameter.
    Query param: mode = 'tasker' | 'poster'
    """
    mode = request.query_params.get('mode', 'poster')

    if mode == 'poster':
        return {
            'active': Task.objects.filter(poster=user, status__in=[...]),
            'completed': Task.objects.filter(poster=user, status__in=[...]),
            'stats': get_poster_stats(user)
        }
    else: # tasker
        return {
            'applied': Bid.objects.filter(tasker=user, status='ACTIVE'),
            'active': Task.objects.filter(assigned_tasker=user, status__in=[...]),
            'completed': Task.objects.filter(assigned_tasker=user, status__in=[...]),
            'stats': get_tasker_stats(user)
        }
```

Benefits:

- Single call gets all dashboard data
- Reduces N+1 queries on frontend
- Mode-aware response structure

3. Add `user_context` to Task Detail Response

When fetching a single task, include the viewing context:

```
class TaskDetailSerializer(serializers.ModelSerializer):
    # Existing...
    user_context = serializers.SerializerMethodField()
    available_actions = serializers.SerializerMethodField()

    def get_user_context(self, obj):
        user = self.context['request'].user
        if obj.poster == user:
            return 'poster'
        if obj.assigned_tasker == user:
            return 'assigned_tasker'
        if obj.bids.filter(tasker=user).exists():
            return 'bidder'
        return 'stranger'

    def get_available_actions(self, obj):
        context = self.get_user_context(obj)
        return TaskStateMachine.get_available_actions(obj.status, context)
```

Benefits:

- Frontend doesn't need to compute context
- Backend is single source of truth for business rules
- Actions list prevents UI from showing invalid options

4. Normalize Task Status for UI

Add a computed `ui_status` field:

```
class TaskDetailSerializer(serializers.ModelSerializer):
    ui_status = serializers.SerializerMethodField()

    def get_ui_status(self, obj):
        mapping = {
            'POSTED': 'draft',
            'BIDDING': 'open',
            'ASSIGNED': 'assigned',
            'IN_PROGRESS': 'in_progress',
            'COMPLETED': 'pending_confirmation',
            'CONFIRMED': 'done',
            'DISPUTED': 'disputed',
```

```
        'CANCELLED': 'cancelled',
    }
    return mapping.get(obj.status, 'unknown')
```

Benefits:

- Frontend uses stable, documented UI states
- Backend can refactor internal states without breaking apps
- Clear contract between backend and frontend

5. Add "My Bid" to Nearby Tasks Response

When a Tasker is browsing, include their bid info if it exists:

```
class TaskListSerializer(serializers.ModelSerializer):
    user_bid_status = serializers.SerializerMethodField()

    def get_user_bid_status(self, obj):
        user = self.context['request'].user
        bid = obj.bids.filter(tasker=user).first()
        if bid:
            return {'id': bid.id, 'amount': bid.amount, 'status': bid.status}
        return None
```

Benefits:

- Browse screen shows "You bid \$X" without extra API call
- Prevents re-bidding on already-bid tasks
- Better UX with immediate feedback

Summary of Recommendations

Frontend Architecture Changes

| Current | Recommendation |
|--------------------------------|--|
| TaskRole enum in controller | Global ActiveMode enum in dedicated controller |
| Passing role as parameter | Compute context from task + user data |
| Two TaskDetails screens | One screen with conditional rendering |
| Hardcoded mode toggle | Connected to ModeController |
| Separate nav for tasker/poster | Dynamic nav tabs based on active mode |

Screen Mapping

| Screen | Responsibility | Mode |
|--------|----------------|------|
|--------|----------------|------|

| Screen | Responsibility | Mode |
|---------------------|---|------------------------|
| HomeScreen | Discovery (browse OR my posted) | Both (content differs) |
| TaskDetailsScreen | View any task, actions based on context | Shared |
| PlaceBidScreen | Create bid | Tasker only |
| MyTasksScreen | Applied + Active + Completed (tasker) | Tasker only |
| MyPostedTasksScreen | Manage posted tasks | Poster only |
| PostTaskScreen | Create task | Poster only |
| Settings | Mode switch + account | Shared |

Backend Additions

| Change | Priority | Effort |
|----------------------------------|----------|--------|
| user_context in task detail | High | Low |
| available_actions in task detail | High | Medium |
| ui_status normalization | Medium | Low |
| /tasks/dashboard/ endpoint | Medium | Medium |
| user_bid_status in list | Medium | Low |
| active_mode persistence | Low | Low |

User: Can you tell me where i can find the chat output as a file?

GitHub Copilot: In VS Code, you can save the chat output in a few ways:

- 1. **Copy to clipboard:** Click the copy icon on any response, or select all text and copy manually
- 2. **Export chat session:**
 - Right-click in the chat panel → "Export Chat..."
 - Or use Command Palette (**Ctrl+Shift+P**) → search "Copilot: Export Chat"
- 3. **Insert into file:** Click the "Insert at Cursor" icon on a response to paste it into your active editor
- 4. **Chat history location** (if you need the raw files):
 - Windows: %APPDATA%\Code\User\globalStorage\github.copilot-chat\
 - The files are stored in JSON format, not easily readable

Quickest option for your analysis: Select all the text in my response, copy it, create a new file (e.g., ARCHITECTURE_GUIDE.md), and paste.

Would you like me to create a markdown file with the architectural guidance I provided?

User: yes plz create a markdown file , and dont miss anything , as it is

GitHub Copilot: Using "Create File"