ChatGPT

# Cursor Setup Instructions: Vertical-Specific UI Expansion

*This instruction list is intended for the Cursor tool (front-end scaffolding, UI layout, and styling tasks).*

## Overview

To support industry-specific interfaces (Hospitality, Real Estate, Professional Services) in DONNA, we will add a vertical selection step to the UI and adjust the dashboard layout accordingly. DONNA's front-end is built with Next.js 14 and Tailwind CSS [1], so we will follow those conventions. The Phase 5 expansion plan highlights new industry-specific modules for **Real Estate Management**, **Restaurant & Hospitality**, and **Professional Services** [2], and our goal is to implement the UI that allows users to select and use these verticals.

## Implementation Steps

1. **Create Vertical Selection Page:** Add a new onboarding page where users choose their vertical (industry) during first-time setup. For example, create a Next.js page at `pages/app/onboarding.tsx` (under the protected `/app` route). This page should render a selection interface for the three vertical options. Use a simple, responsive layout (e.g., a centered container or grid) to display the options. Ensure this page is not behind the main navigation (it might use a minimal layout without the sidebar, since the user hasn't completed setup).

2. **Define Vertical Options (Constants):** In the new page component (or a separate config file if preferred), define a list or array of the vertical options with their identifiers and display names. For example:

```
// In pages/app/onboarding.tsx or a constants file
const VERTICALS = [
  { key: 'hospitality', label: 'Hospitality',
    description: 'Front desk automation, concierge interactions, reservations,
and guest handling.' },
  { key: 'real_estate', label: 'Real Estate',
    description: 'Lead qualification, follow-ups, showing scheduling, and
document handling.' },
  { key: 'professional_services', label: 'Professional Services',
    description:
'Email triage, voice receptionist, document automation, meeting notes, and CRM
updates.' }
];
```

These descriptions are based on the capabilities outlined for each industry 【26†】. The `key` values (such as `'hospitality'`, `'real_estate'`, etc.) will be used in the backend and should exactly match what the backend expects for the vertical identifier.

1. **Build Selection UI:** For each option in `VERTICALS`, render a selectable card or button. Each card should display the vertical's name and description. Use Tailwind CSS (as TailAdmin v2 likely uses Tailwind components [1]) to style these cards for a clear, modern look. For example, you can create a React component `VerticalOptionCard` (in `components/VerticalOptionCard.tsx`) to encapsulate the styling. Each card could have a distinct icon or illustration (if available), the vertical name (e.g., "Hospitality"), and a brief description. Arrange the cards in a responsive grid (e.g., 3 columns on desktop, single column on mobile). Add hover and selected states using Tailwind classes (e.g., border highlighting or background change on selection). Ensure the design is consistent with the existing UI styling (fonts, colors, spacing) so it feels integrated.

2. **Handle User Selection (State Management):** In the onboarding page component, manage the selected vertical using React state. For example, use `useState<string | null>` to track the `selectedVerticalKey`. When a user clicks on one of the VerticalOptionCard components, update the state to that vertical's key, and visually indicate the selection (e.g., add a Tailwind class like `border-blue-500` or a checkmark on the selected card). Provide a **Continue** or **Confirm** button that is enabled only after a selection is made. This button will finalize the choice.

3. **Call Backend API to Save Vertical:** On clicking the confirm button, call the backend API to save the user's vertical selection. Use the appropriate method to make an authenticated request (e.g., `fetch` or Axios). Assuming the backend will expose an endpoint (as per the Augment tasks) like `POST /api/user/vertical` to set the current user's vertical, make a POST request to `'/api/user/vertical'` with a JSON body `{"vertical": "<selected_key>"}`. Include the user's auth token (e.g., via cookies or Authorization header, depending on how JWT is handled in the front-end). For example:

```
await fetch('/api/user/vertical', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json', 'Authorization': 'Bearer
<token>' },
  body: JSON.stringify({ vertical: selectedVerticalKey })
});
```

Handle the response: if successful, proceed to the next step; if there is an error (e.g., invalid value or auth issue), show an error message to the user. *Note:* The exact API path should match what is implemented in the backend. (If the backend uses a different path like `/api/onboarding/vertical`, adjust accordingly in the fetch call.)

1. **Redirect to Dashboard After Selection:** Upon successful vertical selection, route the user to the main application dashboard. For example, use Next.js routing to push the user to the main app page (e.g., `router.push('/app')`). After this, the user should see the regular dashboard UI. (If the

dashboard UI is at `/app/index.tsx` or similar, ensure that route is accessible.) At this point, the user's vertical is set in the backend, and the UI can adjust what to display based on it.

2. **Adapt Navigation and UI Based on Vertical:** Update the front-end to conditionally show relevant modules and pages for the chosen vertical. For instance, modify the sidebar or menu component (e.g., `components/Sidebar.tsx` or equivalent) to include links relevant to the user's vertical. You might introduce a context or use the existing global state to store the current user's vertical. If there is a user context provider (e.g., a React Context that holds user info loaded after login), extend it to store `user.vertical`. The vertical value can be obtained from the user profile API (for example, if there's an endpoint `/api/users/me` that now includes the vertical field). Use this value to render navigation items:

3. **Real Estate:** Show menu items like "Properties" or "Showings" if those modules exist or are planned.

4. **Hospitality:** Show items like "Reservations" or "Guest Management."
5. **Professional Services:** Show items like "Clients", "Projects" or other relevant features.

For now, if those pages/modules are not fully implemented, you can create placeholder pages or routes for them (e.g., `pages/app/properties.tsx`, `pages/app/reservations.tsx`, etc., with simple text like "Coming Soon: [Module] for [Vertical]"). This scaffolding will be filled in later, but it allows testing the vertical-specific navigation. Use naming conventions consistent with the project (for example, if other pages use PascalCase or kebab-case file names, follow that).

Additionally, if certain common pages should behave differently per vertical (for example, a dashboard overview might show different summary info), implement conditional rendering in those pages based on `user.vertical`. Keep such logic clean by encapsulating vertical-specific content in separate components or using a lookup map.

1. **Apply Vertical-Themed Styling (Optional):** If the UI design calls for any theming or imagery changes per vertical, implement those now. For example, the Hospitality vertical might use a different accent color or icon set than Real Estate. You can define a Tailwind color palette extension or CSS variable mapping for each vertical. For instance, in the Tailwind config or a CSS file, define classes like `.theme-hospitality { --color-primary: #...; }` and apply a parent class on the `<body>` or app container according to the user's vertical. This way, you could easily switch theme colors or images (like a banner image on the dashboard) based on vertical. If no distinct theming is required, ensure the UI text (titles, descriptions) clearly reflect the chosen vertical to reinforce context (e.g., "Welcome to DONNA - Hospitality Edition" on the dashboard header for a hospitality user, etc.).

2. **Testing and UX Polishing:** After implementing, test the flow end-to-end in the front-end:

3. Create a new account (or use a test account with no vertical set) and log in. You should be taken to the new vertical selection page. Verify that the three options display correctly and the descriptions match the expected content for each vertical.
4. Select each vertical in turn and ensure the selection can be made and the continue button works. For each, confirm that the API call returns success and the app redirects to the dashboard.

5. Verify that upon reaching the dashboard, the sidebar/navigation now includes the appropriate sections for the chosen vertical, and that no irrelevant sections are shown. (For example, a Hospitality user should not see Real Estate-specific menu items.)
6. If the user already has a vertical chosen (simulate by updating the database or via the backend), ensure that the app does **not** show the onboarding page again. Instead, it should go straight to `/app`. You may implement this check by retrieving the user profile on app load: if `user.vertical` is not null, skip onboarding. If it's null, redirect to `/app/onboarding`.

Finally, refine the UI based on feedback: make sure the vertical selection interface is intuitive (e.g., consider adding a heading like "Select Your Industry" on that page), and the overall experience matches DONNA's product style.

*By completing these steps, the front-end will have a new onboarding flow for vertical selection and a dynamically adjusted UI that lays the groundwork for vertical-specific modules.* [2]

---

# Augment Setup Instructions: Vertical Module Backend

*This instruction list is intended for the Augment tool (backend logic, configuration, and routing tasks).*

## Overview

We will enhance DONNA's backend to support the new vertical-specific interfaces. This involves adding a **vertical** attribute to the user's profile (or organization account) in the database, providing an API endpoint to set this vertical during onboarding, and setting up the structure for vertical-specific modules with appropriate access control. The system uses FastAPI for the backend and a role-based access control (RBAC) mechanism [1] [3], so we will integrate the vertical selection into that framework. According to the expansion plan, DONNA will cater to verticals like Real Estate, Hospitality, and Professional Services [2] – we'll implement support for these as first-class modules in the backend.

## Implementation Steps

1. **Define Allowed Verticals in Backend Config:** Introduce a canonical list of supported verticals in the backend code to avoid mismatches between front-end and back-end. For example, create an `Enum` or constant in a config module for the vertical categories. In a Python module (e.g., `app/core/config.py` or `app/models/vertical.py`), define something like:

```python
from enum import Enum
class Vertical(Enum):
    HOSPITALITY = "hospitality"
    REAL_ESTATE = "real_estate"
    PROFESSIONAL_SERVICES = "professional_services"
```

This provides a single source of truth for vertical identifiers. These string values should match the keys used on the front-end (e.g., `'hospitality'`, `'real_estate'`, etc.) [2]. If an `Enum` is used, you can easily reference `Vertical.HOSPITALITY.value` to get the string. Alternatively, a simple list `ALLOWED_VERTICALS = ["hospitality", "real_estate", "professional_services"]` could be used. The goal is to have a defined set of verticals that the system recognizes.

1. **Add Vertical Field to Data Model:** Extend the database model to store each user's chosen vertical. If the system is multi-tenant with an Organization or Company model (where a user belongs to an org), it makes sense to store the vertical on the Organization model (so that all users of a company share the same industry) [4]. In that case, add a column `vertical` to the Organization (or Tenant) table. If there is no such abstraction and each user individually has an industry, then add the field to the User model instead.

2. **Model Update:** In the ORM model definition (e.g., `app/models/organization.py` or `app/models/user.py`), add a new field for `vertical`. This could be a `String`/`VARCHAR` field. If using an Enum type supported by the DB and SQLAlchemy/Pydantic, you might tie it to the `Vertical` Enum defined earlier for validation. For example, in SQLAlchemy: `vertical = Column(String, nullable=True)` (or an `Enum(Vertical)` if using SQLAlchemy Enum). Set `nullable=True` initially so existing records are not immediately required to have a value. Optionally, set a default (like `'general'` or `None`) for users who haven't selected one.

3. **Migration:** Create a database migration to add this new column. For instance, if using Alembic, generate a revision script that alters the table to include the `vertical` column. Include this migration as part of the deployment so the database is ready to store the vertical values.

4. **Pydantic Schemas:** Update any Pydantic models (data schemas) that represent the User or Organization. For example, if there is a `UserOut` schema for API responses, add a `vertical: Optional[str]` field to it. If there's a `UserCreate` or `UserUpdate` schema and you plan to allow setting vertical via those, include `vertical` there as well (optional). This ensures that when the backend serializes user info, the vertical is included, and it can accept vertical in incoming data if needed.

5. **Expose Vertical in User APIs:** Ensure that the client can retrieve the user's vertical selection. If an endpoint like `GET /api/users/me` (current user profile) or a similar JWT-authenticated endpoint exists, modify it to include the `vertical` field in the returned data. This way, the front-end can know if a user already has a vertical assigned. For example, after login, the front-end might call `/api/users/me` – the response should now contain something like `"vertical": "hospitality"` (or null if not set yet). This is critical for allowing the front-end to decide whether to prompt for onboarding.

6. **Implement Vertical Selection Endpoint:** Create a new API endpoint to allow the user to set their vertical (industry) as part of onboarding. This will correspond to the front-end's selection action. For instance, define a route in FastAPI:

```
from fastapi import APIRouter, Depends
from app.models import User  # or Organization
from app.schemas import VerticalSelectRequest  # Pydantic model with a
'vertical' field
from app.dependencies import get_current_user

router = APIRouter()

@router.post("/user/vertical")
def set_user_vertical(payload: VerticalSelectRequest, current_user: User =
Depends(get_current_user)):
    # Validate the input vertical
    vert = payload.vertical
    if vert not in ALLOWED_VERTICALS:  # or use Vertical Enum for validation
        raise HTTPException(status_code=400, detail="Invalid vertical
selection")
    # Update the user's vertical
    if current_user.organization:
        current_user.organization.vertical = vert
        # (If using an ORM, add commit)
    else:
        current_user.vertical = vert
    db_session.commit()  # commit the change in the database
    return {"detail": "Vertical updated successfully"}
```

In the above pseudocode, `VerticalSelectRequest` would be a Pydantic schema with something like `vertical: Literal["hospitality", "real_estate", "professional_services"]` for automatic validation, or you can manually check as shown. The endpoint uses `get_current_user` dependency to ensure the user is authenticated (likely via JWT). It then updates the relevant model (organization or user) with the chosen vertical and commits the change. It returns a success message (or possibly the updated user profile). Mount this router in your FastAPI app (for example, include it in your main API router or app in `main.py`). This allows the front-end to call `/api/user/vertical` to set the vertical.

*Note:* If the registration process is planned to include vertical selection (instead of a separate call), you could alternatively accept the vertical during sign-up. In that case, ensure the registration logic populates the `vertical` field. However, the separate onboarding endpoint approach as above is more flexible and aligns with having a distinct onboarding flow.

1. **Set Up Vertical-Specific Module Routers:** Create scaffolding for each vertical's specific endpoints. Even if the actual business logic for these modules isn't implemented yet, we should prepare the structure so that the front-end's vertical-specific UI has corresponding backend routes (even if they return placeholder data for now). Organize these by creating a sub-package, e.g., `app/api/verticals/`, and inside it, create separate router modules for each vertical:

2. **Hospitality Router:** Create `app/api/verticals/hospitality.py`. Define `hospitality_router = APIRouter(prefix="/hospitality", tags=["hospitality"])`.

This router will contain endpoints relevant to hospitality (e.g., reservation management, concierge tasks, etc.). For now, add a sample endpoint, for example:

```
@hospitality_router.get("/dashboard")
def hospitality_dashboard(current_user: User = Depends(get_current_user)):
    # This is a placeholder endpoint that could return some summary or
required data.
    return {"message": f"Hospitality dashboard data for
{current_user.name}"}
```

3. **Real Estate Router:** Similarly, create `app/api/verticals/real_estate.py` with `real_estate_router = APIRouter(prefix="/real-estate", tags=["real-estate"])`. Example placeholder endpoint:

```
@real_estate_router.get("/dashboard")
def real_estate_dashboard(current_user: User = Depends(get_current_user)):
    return {"message": f"Real estate dashboard data for
{current_user.name}"}
```

4. **Professional Services Router:** Create `app/api/verticals/professional_services.py` with `professional_router = APIRouter(prefix="/professional-services", tags=["professional-services"])`. Add a similar placeholder endpoint.

The prefix URLs (e.g., `/hospitality/...`) will neatly namespace each vertical's endpoints. **Mount these routers in the main app** (for example, in your main `app.py` or wherever you set up routes, do: `app.include_router(hospitality_router); app.include_router(real_estate_router); app.include_router(professional_router)`). This ensures that requests to these endpoints are routed properly. At this stage, these endpoints might just return dummy data or a message, but we have the framework ready for future vertical-specific logic implementation.

1. **Enforce Vertical-Based Access Control:** Implement logic to ensure that only users belonging to a given vertical can access that vertical's endpoints. This is critical for security and logical separation of modules. There are two potential approaches, and you can even combine them:

2. **Dependency Check:** Write a dependency function that checks the authenticated user's vertical against an expected value. For example:

```
from fastapi import HTTPException, status
def require_vertical(expected: str):
    def verifier(current_user: User = Depends(get_current_user)):
        user_vert = current_user.organization.vertical if
current_user.organization else current_user.vertical
        if user_vert != expected:
            raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="Forbidden: wrong vertical")
        return current_user  # if needed downstream
    return Depends(verifier)
```

Then, attach this dependency to the routers or individual endpoints. For example, when defining the Hospitality router, do: `hospitality_router = APIRouter(prefix="/hospitality", dependencies=[require_vertical("hospitality")], tags=["hospitality"])`. This means every endpoint under `/hospitality` will automatically ensure the current user's vertical is "hospitality". Similarly, add `dependencies=[require_vertical("real_estate")]` for the real estate router, and so on. With this in place, if a user from a different vertical attempts to call an endpoint not meant for them, they will receive a 403 Forbidden. This aligns with the backend-driven RBAC approach mentioned in the architecture [3], extending it to vertical-based rules.

3. **Role-Based Method (alternative or additional):** If the system's RBAC is role-oriented (e.g., roles like Admin, User, etc.), you can represent vertical membership as roles too. For instance, define new roles such as `ROLE_HOSPITALITY`, `ROLE_REAL_ESTATE`, `ROLE_PROF_SERVICES`. When a user selects a vertical, the system can assign the corresponding role to that user. Then, protect the vertical routers using role requirements. For example, use FastAPI dependency or custom logic to ensure the user has `ROLE_HOSPITALITY` for the hospitality endpoints. This approach piggybacks on the existing RBAC system. However, since each user will only belong to one vertical at a time, the dependency check method described above might be simpler. It's fine to implement either. The key is that vertical-specific routes must be accessible only to the appropriate users, which is an extension of our RBAC enforcement [3].

4. **Integrate Vertical Selection with Auth Flow:** When the user completes onboarding by selecting a vertical, they should now have the proper access moving forward. Ensure that after setting the vertical, the user's credentials reflect this. If you include the vertical or roles in JWT claims, you might need to issue a new token or adjust token generation. For example, if using JWT and you decide to embed the vertical as a claim (so the front-end can quickly read it), update the token creation function to include `vertical: current_user.vertical` in the payload. This isn't strictly necessary if the front-end can call the profile API and if the backend always checks the database for vertical on protected routes. But adding it to JWT could optimize checks (just be cautious to update it when the vertical is set). If roles were used, add the vertical role to the user's claims.

Additionally, consider if a user without a vertical (i.e., a new user pre-onboarding) should be restricted from accessing any core functionality. It's often a good idea to treat "no vertical chosen" as a state that only allows calling the vertical selection endpoint. You can enforce this by modifying `get_current_user` or other dependencies to check: if `user.vertical` is null, perhaps direct them to complete onboarding (though this might be handled on front-end by redirect logic). At minimum, ensure that any core endpoints that assume a vertical will handle a null or throw a friendly error prompting completion of onboarding.

1. **Testing the Backend Changes:** After implementing, test the following scenarios:

2. Run database migrations and verify the new column is created in the user/organization table.

3. Use the API (via Swagger UI or Postman) to get your user profile (e.g., GET `/api/users/me`), ensure it includes `"vertical": null` (or no field) initially for an existing user.
4. Call the POST `/api/user/vertical` endpoint with a valid vertical value (e.g., `"hospitality"`) while authenticated as a test user. Verify that a 200 OK is returned and the database now reflects the user's vertical as "hospitality". Test that an invalid value (e.g., `"finance"`) returns a 400 Bad Request (your validation should catch it).
5. After setting a vertical, try accessing an endpoint from a different vertical's router. For example, set your user's vertical to "hospitality", then call a Real Estate endpoint like GET `/real-estate/dashboard`. You should receive a 403 Forbidden due to the enforcement dependency. Conversely, calling `/hospitality/dashboard` should succeed (perhaps returning the placeholder message). This confirms the role-based vertical enforcement is working correctly.
6. If JWT claims were updated, test that the token contains the correct claims after selection, and that those claims (or roles) are recognized by the auth logic.

The backend is now prepared to serve different content or functionality based on industry vertical. As new features for each vertical are developed, they can be added to their respective routers. The structure allows for **industry-specific modules** to be encapsulated and protected, aligning with the product vision of targeted vertical solutions [2]. Each vertical module can grow independently (e.g., adding property management APIs under `/real-estate` or reservation handling under `/hospitality`) without affecting users of other verticals. The configuration and enforcement put in place here ensure a scalable and secure expansion for DONNA's multi-vertical support.

Finally, ensure to update documentation (if any) for these new endpoints and configurations. With both the front-end and back-end changes implemented, DONNA's UI and backend will work in parallel to provide a tailored experience for Hospitality, Real Estate, and Professional Services users, as planned in the expansion roadmap. [2]

---

[1] [3] [4] comprehensive_prd_06092025.md
https://github.com/Bizoholic-Digital/bizosaas-platform/blob/927735a65b815da658f554fae0faf1868d31ec8f/comprehensive_prd_06092025.md

[2] index.md
https://github.com/Abigor78/Workspace_AI-BusinessSuite/blob/f7fcda0bfbace5e999b1b257538c5ddebccdd919/docs/prd/phase5-expansion/index.md