Knowledge Goal Number & Name: [KG1] Endpoint Definition

File Reference: `app/main.py:20-24 and app/routers/bowls_api.py:109-110 and 125-126`

Code Fragment:

```
# From app/main.py - Router registration with URL prefixes
app.include_router(
    bowls_api.router,
    prefix="/api/bowls",
    tags=["bowls-api"],
)

# From app/routers/bowls_api.py - Specific endpoint paths
@router.get("/{bowl_id}", status_code=status.HTTP_200_OK)
def get_bowl(bowl_id: int, ...):
    ...

@router.post("/create_bowl", status_code=status.HTTP_201_CREATED)
def create_bowl(...):
    ...
```

Justification: This code fragment shows two locations where the server's unique URL paths for handling incoming client requests are defined. The router prefixes in main.py are combined with route decorators in the router files to create complete endpoint URLs, such as /api/bowls/123 for a specific bowl_id and /api/bowls/create_bowl to create a new bowl. These unique URL paths define where resources can be accessed on the server.

Knowledge Goal Number & Name: [KG2] HTTP Methods & Status Codes

File Reference: `app/routers/bowls_api.py:164-178`

Code Fragment:

```
@router.post("/save_bowl", response_model=SaveBowlResponse, status_code=status.HTTP_200_OK)
def save_bowl(
    request: SaveBowlRequest,
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
) -> SaveBowlResponse:
    bowl = get_and_verify_bowl(request.bowl_id, current_user, session,
unauthorized_detail="Not authorized to save this bowl")

    # Mark bowl as saved
    bowl.saved = True
    session.add(bowl)
    session.commit()

    # Return response indicating bowl is saved
    return SaveBowlResponse(bowl_id=bowl.id, saved=True)
```

Justification: The route is defined using the @router.post decorator, which enforces the appropriate POST HTTP method. On success, the function returns a 200 OK status code. If the bowl does not exist or the user is not authorized to access it, the helper function raises an HTTPException such as 404 NOT FOUND status code, preventing unauthorized users from learning whether the resource exists.

Knowledge Goal Number & Name: [KG3] Endpoint Validation

File Reference: `app/routers/auth_api.py:12-14`

Code Fragment (Pydantic Schema):

```
from pydantic import BaseModel, Field


class RegisterRequest(BaseModel):
    username: str = Field(min_length=3, max_length=30, pattern=r"^[a-zA-Z0-9_]+$",
description="Username, 3-30 characters, alphanumeric and underscore only")
    password: str = Field(min_length=6, description="Password, must be at least 6
characters")
```

Justification: This Pydantic RegisterRequest schema is used in a FastAPI route to validate incoming JSON data. By defining min_length, max_length, and pattern constraints within the Field objects, FastAPI automatically performs Endpoint Validation. If a client submits a registration request with a username less than 3 characters, more than 30 characters, or containing invalid characters, or a password less than 6 characters, the server will reject the request and return a 422 Unprocessable Entity status code before the data even reaches the application logic.

Knowledge Goal Number & Name: [KG4] Dependency Injection

File Reference: `app/routers/bowls_api.py:35-43  and139-145`

Code Fragment:

```
def get_and_verify_bowl(
    bowl_id: int,
    current_user: User,
    session: Session,
    unauthorized_detail: str = "Not authorized to modify this bowl"
) -> Bowl:
    bowl = session.get(Bowl, bowl_id)
    return verify_bowl_access(bowl, current_user, unauthorized_detail=unauthorized_detail)

@router.put("/{bowl_id}", response_model=BowlResponse)
def update_bowl(
    request: CreateBowlRequest,
    bowl_id: int = Path(ge=1),
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    ...
```

Justification: This route demonstrates dependency injection because FastAPI supplies both the database session and the authenticated user through Depends(get_session) and Depends(get_current_user) instead of the route creating those objects itself. The route only declares what dependencies it needs, and FastAPI handles providing them at runtime. This keeps database access and authentication logic separate from request handling, which improves modularity and makes the code easier to test and maintain.

Knowledge Goal Number & Name: [KG5] Data Model

File Reference: `app/models.py:12-21`

Code Fragment:

```
from typing import Optional
from sqlmodel import SQLModel, Field


class Ingredient(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    calories: float
    protein: float
    fiber: float
    sugar: float
    icon_filename: Optional[str] = None  # For ingredient list selection
    bowl_image_filename: Optional[str] = None  # For bowl display
    is_drizzle: bool = Field(default=False)  # For drizzle
```

Justification: This SQLModel class defines the formal structure of the Ingredient data model, specifying how ingredient data is organized and stored in the PostgreSQL database. It defines fields with their types and constraints, including a primary key, optional fields with default values, and various data types (int, str, float, bool, Optional). The `table=True` parameter indicates that this model corresponds to a database table.

This defines the formal structure that organizes nutritional information, file references, and ingredient properties.

Knowledge Goal Number & Name: [KG6] CRUD Operations & Persistent Data

File Reference: `app/routers/bowls_api.py:214-240`

Code Fragment:

```
@router.post("/remove_ingredient", response_model=RemoveIngredientResponse,
status_code=status.HTTP_200_OK)
def remove_ingredient(
    request: RemoveIngredientRequest,
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
) -> RemoveIngredientResponse:
    bowl = get_and_verify_bowl(request.bowl_id, current_user, session)

    # Check if ingredient exists in bowl
    bowl_ingredient = session.exec(
        select(BowlIngredient).where(
            BowlIngredient.bowl_id == request.bowl_id,
            BowlIngredient.ingredient_id == request.ingredient_id,
        )
    ).first()

    # ... error handling if not found ...

    session.delete(bowl_ingredient)
    session.commit()

    return RemoveIngredientResponse(bowl_id=request.bowl_id)
```

Justification: The remove_ingredient function demonstrates the Delete operation from CRUD by removing a BowlIngredient instance from the database using session.delete() and then committing it with session.commit(). The session.commit() call persists the deletion to PostgreSQL, which ensures the data change remains even after the application server restarts. This demonstrates Persistent Data because the deletion is stored in a database that persists beyond the application's runtime.

Knowledge Goal Number & Name: [KG7] API Endpoints & JSON

File Reference: `app/routers/bowls_api.py:181-211`

Code Fragment:

```
@router.post("/add_ingredient", response_model=AddIngredientResponse,
status_code=status.HTTP_200_OK)
def add_ingredient(
    request: AddIngredientRequest,
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
) -> AddIngredientResponse:
    # ... logic to add ingredient to bowl ...
    return AddIngredientResponse(bowl_id=request.bowl_id)
```

Justification: This endpoint is designed to be consumed by other applications rather than rendered in the browser. It accepts JSON input using the AddIngredientRequest Pydantic model and returns JSON output using the AddIngredientResponse model, which FastAPI automatically serializes. For example, a client can send {"bowl_id": 3, "ingredient_id": 5, "quantity": 1.5} and receive a JSON response like {"bowl_id": 3}, demonstrating the standard text-based JSON format used for API communication.

Knowledge Goal Number & Name: [KG8] UI Endpoints & HTMX

File Reference: `app/routers/bowls_ui.py:528-536 and app/templates/ingredient_list.html:35-50`

Code Fragment:

```
@router.post("/bowl/add_ingredient", response_class=HTMLResponse)
def add_ingredient_to_bowl(
    request: Request,
    ingredient_id: int = Form(...),
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    # ... logic to add ingredient ...
    return templates.TemplateResponse("bowl_section.html", {"request": request, "bowl":
bowl, "bowl_data": bowl_data})
```

```
<form hx-post="/bowl/add_ingredient" hx-target="#bowl-section" hx-swap="innerHTML">
  <input type="hidden" name="ingredient_id" value="{{ ingredient.id }}" />
  <button type="submit">Add to Bowl</button>
</form>
```

Justification: This endpoint returns HTML content for web browsers using response_class=HTMLResponse and templates.TemplateResponse(), demonstrating a UI Endpoint designed for human users rather than other applications. The HTMX attributes in the HTML form (hx-post, hx-target, hx-swap) allow the frontend to make AJAX requests directly from HTML without writing JavaScript code, automatically updating the #bowl-section element with the server's HTML response when the form is submitted.

Knowledge Goal Number & Name: [KG9] User Interaction (CRUD)

File Reference: `app/routers/bowls_ui.py:37-57, 482-488 and app/templates/saved_bowls.html:13-21`

Code Fragment:

```
@router.post("/bowl/delete", response_class=HTMLResponse)
def delete_bowl(
    request: Request,
    bowl_id: int = Form(...),
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    bowl = session.get(Bowl, bowl_id)
    bowl = verify_bowl_access(bowl, current_user)
    session.delete(bowl)
    session.commit()
    # ... returns updated saved bowls list ...
```

```
<form hx-post="/bowl/delete" hx-target="#saved-bowls" hx-swap="innerHTML">
  <input type="hidden" name="bowl_id" value="{{ bowl.id }}" />
  <button type="submit">Remove</button>
</form>
```

Justification: This code demonstrates User Interaction (CRUD) by showing how a user-facing action (clicking the "Remove" button) corresponds to a backend CRUD operation (Delete). When a user clicks the "Remove" button in the template, it triggers a POST request to the delete_bowl endpoint, which performs the Delete operation by removing the bowl from the database using session.delete() and session.commit().

Knowledge Goal Number & Name: [KG10] Separation of Concerns

File Reference: `app/models.py:24-28, app/routers/bowls_ui.py:67-77, and app/templates/empty_bowl.html:3-5`

Code Fragment:

```
# Data Access Layer - app/models.py:24-28
class Bowl(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    user_id: int = Field(foreign_key="user.id")
    saved: bool = Field(default=False)

# Business Logic Layer - app/routers/bowls_ui.py:67-77
def get_or_create_unsaved_bowl(user_id: int, session: Session) -> Bowl:
    bowl = session.exec(select(Bowl).where(Bowl.user_id == user_id, Bowl.saved ==
False)).first()
    if not bowl:
        bowl = Bowl(name="My Bowl", user_id=user_id, saved=False)
        session.add(bowl)
        session.commit()
    return bowl
```

```
<!-- Presentation Layer - app/templates/empty_bowl.html:3-5 -->
<form hx-get="/bowl" hx-target="#bowl-section" hx-swap="innerHTML">
  <button type="submit">Create New Bowl</button>
</form>
```

Justification: Data model lives in app/models.py, request handling lives in app/routers, and HTML/UI lives in app/templates. This keeps presentation, routing/business logic, and persistence concerns separated

AI Assitance:

I used an AI assistant (Cursor/Composer) to help with code formatting (PEP 8 compliance), helper functions post creating the actual website to shorten code, improving clarity of KGD justifications, and CSS styling suggestions. All code logic, database operations, and technical understanding demonstrated in this project is my own work.