1. **[KG1] Endpoint Definitions:**

File path: main.py: 206-214

Code Fragment:

```
@app.get("/wardrobe", response_class=HTMLResponse)
async def wardrobe_page(request: Request, user: User =
Depends(require_user), db: Session = Depends(get_db)):
    items = db.exec(select(ClothingItem).where(
        ClothingItem.user_id == user.id)).all()
    return templates.TemplateResponse("wardrobe.html", {
        "request": request,
        "user": user,
        "items": items
    })
```

Justification: This code defines a unique endpoint at the URL path '/wardrobe' that the server exposes to handle GET requests. The decorator @app.get("/wardrobe") specifies where the wardrobe resource can be accessed and the function handles the request to display the user's wardrobe page.

2. **[KG2] HTTP Methods & Status Codes: Methods**

File path: main.py: 254-263

Code Fragment:

```
@app.delete("/delete-item/{item_id}", response_class=HTMLResponse)
async def delete_item_endpoint(item_id: int, user: User =
Depends(require_user), db: Session = Depends(get_db)):
    item = db.get(ClothingItem, item_id)
    if not item or item.user_id != user.id:
        raise HTTPException(status_code=404, detail="Item not
found")
```

Justification: This endpoint uses the DELETE HTTP method through @app.delete to remove a clothing item and raises a 404 NOT FOUND status code if the item does not exist or if it belongs to another user, and this properly indicates the result of the request.

3. **[KG3] Endpoint Validation:**

File path: schemas.py:7-12

Code Fragment:

```
class ItemCreate(BaseModel):
    name: str = PydanticField(..., min_length=2, max_length=100)
    category: str
    color: str
    season: str
    notes: str = ""
```

Justification: This schema validates data for the /add-item endpoint, adding minimum and maximum length constraints (minimum length of 2 characters, max length of 100 characters) on the name field before processing the added item. This prevents invalid data from being processed and ensures data integrity.

4. **[KG4] Dependency Injection:**

File path: main.py: 217-220

Code Fragment:

```
    @app.post("/add-item", response_class=HTMLResponse)
    async def add_item(item: ItemCreate = Form(...),
                       user: User = Depends(require_user),
                       db: Session = Depends(get_db)):
```

Justification: This shows dependency injection using the Depends() function which injects the database session (db) and the user from external sources rather than creating them inside the function. By separating the creation of these dependencies from when they are called, modularity is improved and testing is easier.

5. **[KG5] Data Model:**

File path: models.py: 17-27

Code Fragment:

```python
class ClothingItem(SQLModel, table=True):
    """Data model for clothing items"""
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    category: str
    color: str
    season: str
    notes: str = ""

    user_id: int = Field(foreign_key="user.id")
    user: User = Relationship(back_populates="items")
```

Justification: This code fragment uses SQLModel to define the structure for clothing items in the database, including field types, relationships like foreign keys, and how the data is organized. It represents the database table schema with columns for id, name, category, color, season, notes, and user_id, establishing the data model for persistent storage.

6. **[KG6] CRUD Operations & Persistent Data:**

File reference: main.py: 218-228

Code Fragment:

```python
async def add_item(item: ItemCreate = Form(...),
                   user: User = Depends(require_user),
                   db: Session = Depends(get_db)):
    item_data = item.dict()
    item_data['category'] = item_data['category'].strip().lower()
    item_data['color'] = item_data['color'].strip().lower()
    item_data['season'] = item_data['season'].strip().lower()
    new_item = ClothingItem(**item_data, user_id=user.id)
    db.add(new_item)
```

```
    db.commit()
    db.refresh(new_item)
```

Justification: This demonstrates the Create operation of CRUD by adding a new item to the database using db.add(), db.commit(), then refreshing it with db.refresh(). The data remains available even after the server restarts because it is stored in database.db, demonstrating persistent data storage beyond the runtime of the application.

7. **[KG7] API Endpoints & JSON:**

File reference: main.py: 296-303

```
@app.get("/api/items")
async def api_get_items(request: Request, db: Session =
Depends(get_db)):
    user = get_current_user(request, db)
    if not user:
        return {"error": "Unauthorized", "items": []}
    items = db.exec(select(ClothingItem).where(
        ClothingItem.user_id == user.id)).all()
    return {"username": user.username, "item_count": len(items),
"items": [item.dict() for item in items]}
```

Justification: This endpoint returns wardrobe data in JSON format rather than HTML. It returns a dictionary that includes username, item count, and the list of items. It is designed for consumption by other applications to fetch wardrobe info.

8. **[KG8] UI Endpoints & HTMX: UI Endpoints** are server paths designed to return full or partial HTML content for a web browser. **HTMX** is a library that allows developers to use HTML attributes to make AJAX requests directly, updating parts of the UI without needing complex JavaScript frameworks.

File Reference: templates/wardrobe.html: 11-13

Code Fragment:

```
            <form hx-post="/add-item" hx-target="#items-grid"
hx-swap="innerHTML" class="add-form">
            <div class="form-row">
                <div class="form-group">
```

Justification: This demonstrates a UI endpoint using HTMX attributes like hx-post, hx-target, and hx-swap to make an AJAX request to /add-item and update the #items-grid element without JavaScript. The server endpoint returns partial HTML that HTMX dynamically inserts into the page to update parts of the UI without a full page reload, ensuring a more seamless user experience.

9. **[KG9] User Interaction (CRUD):** Refers to the **user-facing actions** on the web application that correspond to the backend CRUD operations, such as clicking a "Save" button (**Create/Update**) or viewing a list of items (**Read**).

File Reference: templates/wardrobe.html: 71-77

Code Fragment:

```
    <div class="section-card">
        <h2>✨ Outfit Generator</h2>
        <button hx-get="/generate-outfit" hx-target="#outfit-result"
class="btn btn-primary">
            Generate Random Outfit
        </button>
        <div id="outfit-result"></div>
    </div>
```

Justification: This UI element (an Outfit Generator button) allows users to interact with and **Read** data by retrieving clothing items. Pressing this button triggers a GET request to /generate-outfit which performs a Read operation on the user's wardrobe items. A user-facing action like pressing a button to generate a random outfit corresponds to backend CRUD operations.

10. **[KG10] Separation of Concerns:** The design principle that dictates that an application should be divided into distinct sections, where each section

addresses a separate concern. For a web application, this means clearly separating the **presentation layer** (UI/HTMX), the **business logic** (what the app does), and the **data access layer** (CRUD/Data Model) into different modules or files.

File reference: models.py: 1-27 (entire file)

Code fragment:

```python
from typing import List, Optional

from sqlmodel import Field, Relationship, SQLModel



class User(SQLModel, table=True):
    """Data model for users"""
    id: Optional[int] = Field(default=None, primary_key=True)
    username: str = Field(index=True, unique=True)
    password: str
    name: str
    email: str

    items: List["ClothingItem"] = Relationship(back_populates="user")



class ClothingItem(SQLModel, table=True):
    """Data model for clothing items"""
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    category: str
    color: str
    season: str
    notes: str = ""
```

```
user_id: int = Field(foreign_key="user.id")
user: User = Relationship(back_populates="items")
```

Justification: The data access layer defining the database schema and relationship is completely isolated in a dedicated models.py file separate from application routing and business logic in main.py and presentation templates in the templates directory. This structure clearly separates concerns by file: data models are in one file, endpoints and logic are in another, and UI are in the HTML templates.