Deliverable 3: Knowledge Goals Demonstration
Project: ReUse UMass
Student: Arav Mehta
SPIRE ID: 34362486

KG1: Endpoint Definitions

File: app/routers/api/items_api.py

```
router = APIRouter(prefix="/items", tags=["items"])

@router.get("/{item_id}", response_model=ItemRead)
def get_item(item_id: int, db: Session = Depends(get_session)):
obj = db.get(Item, item_id)
if not obj:
raise HTTPException(404, "Item not found")
return obj
```

This demonstrates endpoint definitions by showing a unique URL path at /items/{item_id}. The item_id parameter makes the endpoint dynamic, allowing different items to be retrieved by changing the URL. The APIRouter with the items prefix groups all item related endpoints together in a clear structure.

KG2: HTTP Methods and Status Codes

File: app/routers/api/items_api.py

```
@router.post("", response_model=ItemRead, status_code=201)
def create_item(payload: ItemCreate, db: Session = Depends(get_session)):
obj = Item.model_validate(payload)
db.add(obj)
db.commit()
db.refresh(obj)
return obj
```

This demonstrates the use of the HTTP POST method for creating new resources. Returning status code 201 indicates successful creation according to REST conventions. Errors during validation or database operations automatically return appropriate client or server error codes.

KG3: Endpoint Validation

File: app/schemas/item.py

```
from sqlmodel import SQLModel
```

```
class ItemCreate(SQLModel):
title: str
description: str
category: str
condition: str
location: str
photo_url: Optional[str] = None
owner_email: str
```

This demonstrates endpoint validation using Pydantic and SQLModel schemas. FastAPI enforces that required fields exist and match the correct data types. Invalid input results in a 422 response and prevents bad data from being saved.

KG4: Dependency Injection

File: app/routers/api/deps.py

```
from fastapi import Depends
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/api/v1/auth/token")

async def get_current_user(
token: Annotated[str, Depends(oauth2_scheme)],
db: Session = Depends(get_session)
):
user = db.exec(select(User).where(User.email == email)).first()
return user
```

This demonstrates dependency injection by allowing FastAPI to automatically provide authentication tokens and database sessions. This avoids manual setup inside endpoints and improves modularity, reuse, and testability.

KG5: Data Model

File: app/models/item.py

```
from sqlmodel import SQLModel, Field
from datetime import datetime

class Item(SQLModel, table=True):
id: Optional[int] = Field(default=None, primary_key=True)
title: str
description: str
category: str
condition: str
```

location: str
photo_url: Optional[str] = None
owner_email: str
created_at: datetime = Field(default_factory=datetime.utcnow)
is_claimed: bool = False

This demonstrates the data model that defines how item data is structured and stored. Each field has a clear purpose and type, ensuring consistency between the application and the database.

KG6: CRUD Operations and Persistent Data

File: app/routers/api/items_api.py

@router.delete("/{item_id}", status_code=204)
def delete_item(item_id: int, db: Session = Depends(get_session)):
obj = db.get(Item, item_id)
if not obj:
return
db.delete(obj)
db.commit()

This demonstrates the Delete operation in CRUD. Once committed, the deletion persists in the SQLite database file even after the server restarts.

KG7: API Endpoints and JSON

File: app/routers/api/requests_api.py

@router.post("", response_model=RequestRead, status_code=201)
def create_request(payload: RequestCreate, db: Session = Depends(get_session)):
item = db.get(Item, payload.item_id)
if not item:
raise HTTPException(404, "Item not found")
if item.is_claimed:
raise HTTPException(400, "Item is already claimed")

obj = Request.model_validate(payload)
db.add(obj)
db.commit()
db.refresh(obj)
return obj

This demonstrates API endpoints intended for programmatic use. The endpoint accepts JSON input and returns JSON output, making it suitable for frontend or external application consumption.

KG8: UI Endpoints and HTMX

File: app/routers/frontend.py

```
@router.get("/fragments/items/list", response_class=HTMLResponse)
async def list_items_fragment(
request: Request,
category: Optional[str] = None,
q: Optional[str] = None,
db: Session = Depends(get_session)
):
stmt = select(Item).where(Item.is_claimed == False)
if category and category != "All Items":
stmt = stmt.where(Item.category == category)
if q:
stmt = stmt.where((Item.title.ilike(f"%{q}%")) | (Item.description.ilike(f"%{q}%")))
items = db.exec(stmt).all()
return templates.TemplateResponse(
    "fragments/items_list.html",
    {"request": request, "items": items}
)
```

This demonstrates UI endpoints that return rendered HTML fragments for HTMX requests. These endpoints allow parts of the page to update dynamically without a full reload or heavy JavaScript frameworks.

KG9: User Interaction with CRUD

File: app/templates/fragments/create_item_form.html

```
<form hx-post="/fragments/items/create" hx-target="#form-feedback">
  <div class="form-group">
    <label>Item Title *</label>
    <input type="text" name="title" required placeholder="e.g., Mini Fridge">
  </div>

  <div class="form-group">
    <label>Description *</label>
    <textarea name="description" required rows="3"></textarea>
  </div>
```

```
  <button type="submit" class="btn btn-primary" style="flex: 1;">Post Item</button>
</form>
```

This demonstrates user facing CRUD interaction. When the user submits the form, HTMX sends a POST request to /fragments/items/create, triggering the Create operation and persisting data to the database.

KG10: Separation of Concerns

File: app/routers/frontend.py

```
@router.post("/fragments/items/create", response_class=HTMLResponse)
async def create_item(
request: Request,
title: str = Form(...),
db: Session = Depends(get_session)
):
item = Item(title=title, description=description, ...)
db.add(item)
db.commit()
return """
  <div class="bg-green-100 border border-green-400 text-green-700 px-4 py-3 rounded relative
mb-4">
    <strong class="font-bold">Success!</strong>
    <span class="block sm:inline">Item posted successfully.</span>
    <button hx-get="/fragments/items/list" hx-target="#items-container" class="underline
mt-2">Refresh List</button>
  </div>
  """
```

Application Structure

The models directory handles database schemas.
The routers directory handles business logic and request processing.
The templates directory handles presentation and user interface.

This demonstrates separation of concerns by keeping data modeling, application logic, and presentation logic independent from one another.

Summary

All ten knowledge goals are demonstrated through concrete examples from the ReUse UMass project. The application includes RESTful API endpoints with correct HTTP methods and status codes, full CRUD functionality with persistent storage, schema based validation, dependency injection, HTMX powered user interaction, and a clean separation of concerns. The project

represents a complete and deployable web application that clearly demonstrates the required course concepts.