KG1: Endpoint Definitions

File: src/routers/restaurants.py (lines 15–20)

```python
@router.get("/api", response_model=list[Restaurant])
def get_restaurants(
    city: Optional[str] = None,
    cuisine: Optional[str] = None,
    db: Session = Depends(get_db)
):
```

This endpoint defines the /restaurants/api URL that clients can send GET requests to. It supports optional query parameters for city and cuisine, allowing users or applications to filter restaurants. This lets endpoints show where resources are accessed and how data can be requested from the server.

KG2: HTTP Methods and Status Codes

File: src/routers/restaurants.py (lines 93–101)

```python
@router.delete("/api/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_restaurant(id: int, db: Session = Depends(get_db)):
    restaurant = db.get(Restaurant, id)
    if not restaurant:
        raise HTTPException(status_code=404, detail="Restaurant not found")

    db.delete(restaurant)
    db.commit()
    return None
```

This route uses the DELETE HTTP method to remove a restaurant. If the restaurant is successfully deleted, the server returns a 204 No Content status code, which indicates success without returning a response body. If the restaurant does not exist, a 404 Not Found error is returned. This demonstrates correct and meaningful use of HTTP methods and status codes.

KG3: Endpoint Validation

File: src/schemas/review.py (lines 4–10)

```python
class ReviewCreate(BaseModel):
    rating: int = Field(ge=1, le=5)
    text: str = Field(min_length=10, max_length=500)
```

This Pydantic schema validates incoming review data. Ratings are restricted to values between 1 and 5, and review text must be between 10 and 500 characters. FastAPI automatically applies this validation before the request reaches the route logic, returning a 422 error if the data is invalid. This helps prevent bad input and improves application reliability.

KG4: Dependency Injection

Files: src/database.py and src/routers/reviews.py

```python
def get_db():
    db = Session(engine)
    try:
        yield db
    finally:
        db.close()
```

```python
@router.post("/api/restaurants/{restaurant_id}/reviews")
def create_review(
    restaurant_id: int,
    review: ReviewCreate,
    db: Session = Depends(get_db)
):
```

The database session is injected into route functions using Depends(get_db) instead of being created inside the function. FastAPI handles creating and closing the session automatically. This keeps the code modular, avoids resource leaks, and makes the application easier to test.

KG5: Data Model

File: src/models/restaurant.py (lines 8–18)

```python
class Restaurant(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    city: str = Field(index=True)
    cuisine: str
    address: Optional[str] = None
    phone: Optional[str] = None
    reviews: list["Review"] = Relationship(back_populates="restaurant")
```

This model defines the structure of the restaurant table in the database. It specifies field types, primary keys, indexes for faster queries, and a one-to-many relationship with reviews. The model directly reflects how restaurant data is stored and accessed.

KG6: CRUD Operations and Persistent Data

File: src/routers/restaurants.py (lines 69–82)

```python
@router.put("/api/{id}")
def update_restaurant(id: int, restaurant: RestaurantUpdate, db: Session =
Depends(get_db)):
    db_restaurant = db.get(Restaurant, id)
    if not db_restaurant:
        raise HTTPException(status_code=404, detail="Restaurant not found")

    update_data = restaurant.model_dump(exclude_unset=True)
    for key, value in update_data.items():
        setattr(db_restaurant, key, value)

    db.commit()
    db.refresh(db_restaurant)
    return db_restaurant
```

This route updates an existing restaurant in the database. Calling db.commit() saves the changes to the SQLite file on disk, ensuring the data remains even after the server restarts. This satisfies both the CRUD update requirement and persistent storage requirement.

KG7: API Endpoints and JSON

File: src/routers/restaurants.py (lines 51–66)

Python

```
@router.get("/api/{id}")
def get_restaurant(id: int, db: Session = Depends(get_db)):
    restaurant = db.get(Restaurant, id)
    if not restaurant:
        raise HTTPException(status_code=404, detail="Restaurant not found")

    return {
        "id": restaurant.id,
        "name": restaurant.name,
        "city": restaurant.city,
        "cuisine": restaurant.cuisine,
        "address": restaurant.address,
        "phone": restaurant.phone
    }
```

This endpoint returns restaurant data in JSON format, which is intended for consumption by other applications or frontend clients. FastAPI automatically serializes the Python dictionary into JSON, making the data easy to use in external systems.

KG8: UI Endpoints and HTMX

Files: src/routers/fragments.py and templates/index.html

Python

```
@router.get("/restaurants/search", response_class=HTMLResponse)
def search_restaurants_fragment(request: Request, city: str = "", cuisine: str = "", db:
Session = Depends(get_db)):
    query = select(Restaurant)
    if city:
        query = query.where(Restaurant.city.contains(city))
    if cuisine:
        query = query.where(Restaurant.cuisine.contains(cuisine))

    restaurants = db.exec(query).all()
    return templates.TemplateResponse("fragments/restaurant_list.html", {"request":
request, "restaurants": restaurants})
```

```HTML
<form hx-get="/fragments/restaurants/search" hx-target="#results" hx-trigger="submit, keyup delay:500ms from:input">
```

This endpoint returns HTML fragments instead of JSON. HTMX allows the browser to request these fragments and update part of the page dynamically without a full reload, simplifying the frontend while still providing interactive behavior.

KG9: User Interaction and CRUD

Files: restaurant_detail.html and fragments.py

Submitting the review form triggers an HTMX POST request that creates a new review in the database. The backend performs a Create operation, and the updated HTML fragment is returned and displayed immediately. This shows how a user action in the UI directly maps to a database operation.

KG10: Separation of Concerns

The project is organized into clear layers: models define database structure, schemas handle validation, routers contain business logic, and templates control presentation. Each part has a single responsibility, making the code easier to maintain, test, and extend.