Relational DBs

Kay Ashaolu - Instructor

Aishwarya Sriram - TA

Chapter 6: Store Data in a SQL Database with SQLAlchemy

Why Use SQLAlchemy?

- **Abstraction & Productivity:**
  - Provides an ORM that maps tables to Python classes.
- **Multithreading Support:**
  - Automatically manages table creation, migrations, and relationships.
- **Cleaner Code:**
  - Reduces boilerplate compared to raw SQL.
- **Flexibility:**
  - Easily switch between databases (e.g., SQLite for development, PostgreSQL for production).

Setting Up the Environment

- **Update `requirements.txt`:**
  - Add `SQLAlchemy` and `Flask-SQLAlchemy`.
- **Installation:**

```
pip install -r requirements.txt
```

- **Docker Note:**
  - Changes in `requirements.txt` bust the cache layer, ensuring latest dependencies.

Creating SQLAlchemy Models

- **Objective:** Replace Python lists with persistent storage.
- **Approach:**
  - Define models for items and stores.
  - Use Flask-SQLAlchemy to bind models with your Flask app.
- **Directory Structure:**
  - Place models under a dedicated `models/` folder.

Example: ItemModel

```python
from db import db

class ItemModel(db.Model):
    __tablename__ = 'items'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    price = db.Column(db.Float(precision=2), nullable=False)
    store_id = db.Column(db.Integer, db.ForeignKey('stores.id'), nullable

    # Relationship to store will be added later
```

Maps a row in the `items` table to a Python object.

Example: StoreModel

```python
from db import db

class StoreModel(db.Model):
    __tablename__ = 'stores'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)

    # One-to-many relationship with items
    items = db.relationship('ItemModel', back_populates='store', lazy='dy
```

Defines a store and its one-to-many relationship with items.

Defining Relationships: One-to-Many

- **Item to Store:**
  - Each `ItemModel` has a `store_id` linking to `StoreModel`.
- **SQLAlchemy Relationships:**
  - Use `db.ForeignKey` to enforce the relationship.
  - Use `db.relationship` with `back_populates` for bidirectional access.
- **Lazy Loading:**
  - `lazy='dynamic'` delays fetching related items until explicitly requested.

Updating the Models with Relationships

- In `ItemModel`:

```
1  store = db.relationship('StoreModel', back_populates='items')
```

- In `StoreModel`:

```
1  items = db.relationship('ItemModel', back_populates='store', lazy='dynami
```

- **Benefit:**
  - Access related store via `item.store` and items via `store.items.all()`.

Flask-SQLAlchemy

- **Import & Initialize:**

```
from db import db
import models
```

- **App Factory Pattern:**

```
def create_app(db_url=None):
    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI'] = db_url or \
       os.getenv('DATABASE_URL', 'sqlite:///data.db')
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    db.init_app(app)

    @app.before_first_request
    def create_tables():
        db.create_all()
    return app
```

- **Note:**
  - Import models before initializing SQLAlchemy.

Database Connection Strings

- **SQLite (Development):**
  - `sqlite:///data.db`
- **Production (e.g., PostgreSQL):**
  - Use environment variables for credentials.
- **Why Use Environment Variables?**
  - Keep sensitive data out of code.
  - Easily switch configurations without code changes.

Flask App Factory Pattern

- **Benefits:**
  - Facilitates testing.
  - Promotes clean app initialization.
- **Key Concept:**
  - Create a function (`create_app`) that returns a configured Flask instance.

Initializing the Database

- **After App Creation:**

```
with app.app_context():
    db.create_all()
```

- **Effect:**
  - Automatically creates tables based on your models.
- **Reminder:**
  - Ensure models are imported so SQLAlchemy is aware of them.

Inserting Data into the Database

- **Creating a New Item:**

```
item_data = {'name': 'chair', 'price': 20.5, 'store_id': 1}
item = ItemModel(**item_data)
db.session.add(item)
db.session.commit()
```

- **Key Points:**
  - Use keyword arguments to map dictionary data.
  - `db.session.commit()` writes changes to disk.

Handling SQLAlchemy Errors

- **Error Handling Example:**

```python
from sqlalchemy.exc import SQLAlchemyError

try:
    db.session.add(item)
    db.session.commit()
except SQLAlchemyError as e:
    db.session.rollback()
    abort(500, message="An error occurred while inserting the item.")
```

- **Benefits:**
  - Catches integrity and other SQLAlchemy errors.
  - Uses rollback to maintain session consistency.

Retrieving Models from the Database

- **Using Query Methods:**

```
item = ItemModel.query.get_or_404(item_id)
```

- **What It Does:**
  - Retrieves an item by its primary key.
  - Automatically aborts with a 404 if not found.

Updating Models (PUT Requests)

- **PUT Request Considerations:**
  - Must be idempotent: same request repeated yields the same state.
- **Updating vs. Creating:**

```
item = ItemModel.query.get(item_id)
if item:
    item.name = item_data['name']
    item.price = item_data['price']
else:
    item = ItemModel(id=item_id, **item_data)
    db.session.add(item)
db.session.commit()
```

- **Note:**
  - Ensure proper handling of auto-generated IDs.

Retrieving a List of Models

- **Example: Get All Items**

```
items = ItemModel.query.all()
return items  # Serialized using a schema with many=True
```

- **Key Idea:**
  - Use the `all()` method to fetch a complete list.
  - Schema serialization handles list conversion.

Deleting Models with SQLAlchemy

- **Simple Delete Example:**

```
item = ItemModel.query.get_or_404(item_id)
db.session.delete(item)
db.session.commit()
return {"message": "Item deleted."}, 200
```

- **Concepts:**
  - Delete the model instance.
  - Commit changes to ensure deletion is persisted.

Deleting Related Models with Cascades

- **Problem:**
  - Deleting a store may orphan its items.
- **Solution: Cascade Deletion**

```
items = db.relationship('ItemModel', back_populates='store',
                        cascade="all, delete", lazy='dynamic')
```

- **Benefits:**
  - Automatically deletes child items when the parent store is deleted.
- **Database Considerations:**
  - SQLite vs. PostgreSQL enforcement of foreign key constraints.

Chapter 6 Summary

- **What We Covered:**
  - Switching from in-memory data to SQLAlchemy models.
  - Defining one-to-many relationships.
  - Configuring Flask-SQLAlchemy with an app factory.
  - CRUD operations: Insert, Retrieve, Update, Delete.
  - Handling cascading deletes and error management.
- **Next Steps:**
  - Transitioning to more complex relationships (many-to-many).

Chapter 7: Many-to-Many Relationships with SQLAlchemy

- **Focus:**
  - Implementing many-to-many relationships using tags.
- **Context:**
  - Online stores use tags to group items (e.g., "office", "furniture", "tech").
- **Outline:**
  - One-to-many between stores and tags.
  - Many-to-many between items and tags via a secondary table.

One-to-Many: Stores and Tags

- **TagModel Example:**

```python
from db import db

class TagModel(db.Model):
    __tablename__ = 'tags'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    store_id = db.Column(db.Integer, db.ForeignKey('stores.id'), nullabl

    # Relationship back to store
    store = db.relationship('StoreModel', back_populates='tags')
```

- **In StoreModel:**

```python
tags = db.relationship('TagModel', back_populates='store', lazy='dynamic'
```

- **Key Point:**
  - Each store can have many unique tags.

Marshmallow Schemas for Tags

- **PlainTagSchema:**

```
class PlainTagSchema(ma.Schema):
    class Meta:
        fields = ("id", "name")
```

- **Extended Tag Schema:**
  - Include nested relationships if needed.
- **In StoreSchema:**

```
tags = fields.Nested(PlainTagSchema, many=True, dump_only=True)
```

- **Purpose:**
  - Avoid recursive nesting when serializing store and tag objects.

Many-to-Many: Items and Tags

- **Concept:**
  - Items can have multiple tags and vice versa.
- **Implementation:**
  - Create a secondary table to map items to tags.

Defining the Secondary Table

- **item_tags.py Example:**

```python
from db import db

class ItemTags(db.Model):
    __tablename__ = 'item_tags'
    id = db.Column(db.Integer, primary_key=True)
    item_id = db.Column(db.Integer, db.ForeignKey('items.id'), nullable=F
    tag_id = db.Column(db.Integer, db.ForeignKey('tags.id'), nullable=Fal
```

- **Integration:**
  - Import this model in `models/__init__.py` for auto-creation.

Wiring Up the Many-to-Many Relationship

- **In TagModel:**

```
items = db.relationship('ItemModel', secondary='item_tags',
                        back_populates='tags')
```

- **In ItemModel:**

```
tags = db.relationship('TagModel', secondary='item_tags',
                       back_populates='items')
```

- **Key Benefit:**
  - SQLAlchemy manages the association table automatically.

API Endpoints for Many-to-Many Operations

- **Linking an Item to a Tag:**

```
item = ItemModel.query.get_or_404(item_id)
tag = TagModel.query.get_or_404(tag_id)
item.tags.append(tag)
db.session.commit()
```

- **Unlinking:**

```
item.tags.remove(tag)
db.session.commit()
```

- **Deleting a Tag:**
  - Ensure no items are linked before deletion.

Chapter 7 Summary and Next Steps

- **Review:**
  - Implemented one-to-many for stores and tags.
  - Set up a secondary table for many-to-many between items and tags.
  - Developed API endpoints for linking, unlinking, and deleting tags.
- **Further Learning:**
  - Experiment with cascade behaviors and alternate relationship configurations.
- **Questions?**
  - Use this as a foundation to expand your API features.

Questions?