How to Use Alembic for Database Migrations in Your FastAPI Application

May 24, 2024 • 5 min read
Tags: FastAPI, SQLAlchemy, Alembic

Introduction

When building a FastAPI application, managing database schema changes can be a daunting task. As your application evolves, your database schema must adapt to accommodate new features, bug fixes, and changing requirements.

Alembic is a lightweight and flexible database migration tool that simplifies the process of managing database schema changes. By using Alembic, you can track changes to your database schema, revert to previous versions, and collaborate with your team more effectively.

In this tutorial, I will show you how to use Alembic for database migrations in a FastAPI application by building the project from scratch. I will be using PostgreSQL as the database server, but you should be able to follow this tutorial by using any database server.

By the end of this tutorial, you will have:

- A complete working CRUD API.
- Knowledge to make changes to your models and apply database migrations.

Setup a new project

We will create a new project from scratch. It is recommended to create new virtual environment to isolate your project's dependencies from the system Python environment.

Create and activate a new virtualenv:

```
python3 -m venv venv
. venv/bin/activate
```

Pro Tip: Feel free to replace the virtualenv above with Pipenv or Poetry.

Install the required packages:

```
pip install fastapi sqlalchemy alembic psycopg2-binary
pip freeze > requirements.txt
```

Note that I installed `psycopg2-binary` for PostgreSQL. If you are using a different database server, you should install the corresponding database driver. For example, install `mysql-connector-python` if you are using MySQL.

Create the *src/* directory and make it a package:

```
mkdir src
touch src/__init__.py
```

Your directory structure should look like this:

```
.
├── requirements.txt
└── src
    └── __init__.py
```

We will put all of our database and FastAPI code in the *src/* directory.

Create the SQLAlchemy model

We will setup the database connection in *src/database.py* file and define our model in *src/models.py* file. By separating the database connection and models definition, we can maintain a clean and modular architecture for our FastAPI application.

Create a new file named *src/database.py* and add the following code:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker


# Replace with your own PostgreSQL instance
DATABASE_URL = 'postgresql://dbuser:dbpassword@localhost/dbname'

engine = create_engine(DATABASE_URL)
SessionLoad = sessionmaker(bind=engine)
```

The code above sets up a connection to a PostgreSQL database using SQLAlchemy. The connection string is hardcoded in the file just to make it simpler. To add more security layer, you can load the connection string from environment variables instead.

Pro Tip: Make sure that you have created the database and the user on your PostgreSQL server.

Now we write the SQLAlchemy model. Create a new file named *src/models.py* and add the following code:

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column


class Base(DeclarativeBase):
    pass


class Item(Base):
    __tablename__ = 'items'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(nullable=False)
    price: Mapped[float] = mapped_column(server_default='0')
```

After we have created the model, we want to automate the database migration using Alembic.

Using Alembic for database migration

We use Alembic to automate database migrations, which involves managing changes to the database schema over time.

Start by initializing Alembic:

```
alembic init alembic
```

The command above will create the *alembic/* directory with the necessary configuration files. After you run this command, your directory structure will look like this:

```
.
├── alembic
│   ├── README
│   ├── env.py
│   ├── script.py.mako
│   └── versions
├── alembic.ini
├── requirements.txt
└── src
    ├── __init__.py
```

```
├── database.py
└── models.py
```

From all of the files generated by Alembic, you only need to modify the *alembic/env.py* file. This file is a Python script that serves as the entry point for Alembic's migration process. It's responsible for setting up the environment, loading the configuration, and executing the migration scripts.

Open *alembic/env.py* and find the line that contains:

```
target_metadata = None
```

and replace it with:

```
from src.models import Base
target_metadata = Base.metadata
```

By importing `Base` from `src.models` and setting `target_metadata` to `Base.metadata`, we are configuring Alembic to work with your SQLAlchemy models. Alembic will generate the proper migration scripts by reading our model's metadata.

Below those lines, add the following code as well:

```
# alembic/env.py
from src.database import DATABASE_URL
config.set_main_option('sqlalchemy.url', DATABASE_URL)
```

By importing the `DATABASE_URL` variable and setting the `sqlalchemy.url` option, we are configuring Alembic to connect to our database using the specified connection string. It allows us to specify the connection string in the environment variables instead of hardcode it in the *alembic.ini* file.

Create the database migration script:

```
alembic revision --autogenerate -m "Initial migration"
```

And run the database migration:

```
alembic upgrade head
```

After you run the command, you should see the "items" table is created in your PostgreSQL database.

Create the API endpoints

To create the API endpoints using FastAPI, we'll start by writing the Pydantic models. These models are used to serialize data into JSON and perform data validation. It is also being used in the responses where the data is converted into JSON format.

Open *src/schemas.py* and add the following code:

```
from pydantic import BaseModel, PositiveFloat


class Item(BaseModel):
    id: int
    name: str
    price: PositiveFloat


class ItemCreate(BaseModel):
    name: str
    price: PositiveFloat


class ItemUpdate(BaseModel):
    name: str | None = None
    price: PositiveFloat | None = None
```

In the code above, we created 3 Pydantic models: `Item` will be used as a response model for API endpoints that return a single item or a list of items, `ItemCreate` and `ItemUpdate` will be used to ensure that data is validated and sanitized when creating or updating items.

Now that the models are ready, we can write the API endpoints to create, read, update, and delete the items. Open *src/main.py* and add the following code:

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy import select
from sqlalchemy.orm import Session

from . import models, schemas
from .database import SessionLocal


app = FastAPI()


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
```

```
        db.close()


@app.get('/items')
async def read_items(
    skip: int = 0,
    limit: int = 15,
    db: Session = Depends(get_db)
) -> list[schemas.Item]:
    """
    List all items
    """
    users = db.execute(
        select(models.Item)
    ).scalars().all()
    return users


@app.post('/items')
async def create_item(
    data: schemas.ItemCreate,
    db: Session = Depends(get_db)
) -> schemas.Item:
    """
    Create new Item
    """
    item = models.Item(**data.model_dump())
    db.add(item)
    db.commit()
    db.refresh(item)

    return item


@app.get('/items/{item_id}')
async def read_item(
    item_id: int,
    db: Session = Depends(get_db)
) -> schemas.Item:
    item = db.get(models.Item, item_id)
    if item is None:
        raise HTTPException(status_code=404, detail='Item not found')
    return item


@app.put('/items/{item_id}')
async def update_item(
    item_id: int,
    data: schemas.ItemUpdate,
    db: Session = Depends(get_db)
) -> schemas.Item:
    """
```

```
    Update item
    """
    item = db.get(models.Item, item_id)
    if item is None:
        raise HTTPException(status_code=404, detail='Item not found')

    for key, val in data.model_dump(exclude_none=True).items():
        setattr(item, key, val)

    db.commit()
    db.refresh(item)

    return item

@app.delete('/items/{item_id}')
async def delete_item(
    item_id: int,
    db: Session = Depends(get_db)
) -> dict[str, str]:
    """
    Delete item
    """
    item = db.get(models.Item, item_id)
    db.delete(item)
    db.commit()
    return {'message': 'Item successfully deleted'}
```

The code above provides the API endpoints to perform CRUD (Create, Read, Update, Delete) operations for the items and is self-explanatory.

Pro Tip: Try the API endpoints above using Curl or HTTPie to make sure that it is working as expected.

Making new changes to the model

Lets say you want to make some changes to the model and you want Alembic to generate the migration script to alter the database. As an example, lets add a new “quantity” column.

Open `src/models.py` and add the column:

```
class Item(Base):
    ...
    quantity: Mapped[int] = mapped_column(server_default='0')
```

The new line tells SQLAlchemy that the quantity column is of type `int`, and set the default value to 0 if no value is provided when inserting a new row. It will also set the value to 0 for existing rows.

Add the same field to the Pydantic model so the API will return the item's quantity as well. Open `src/schemas.py` and add the field:

```
class Item(BaseModel):
    ...
    quantity: PositiveInt
```

Create and run the database migration:

```
alembic revision --autogenerate -m "Added quantity column"
alembic upgrade head
```

After you run the command, the table in your PostgreSQL database should have the "quantity" column. The field will also be returned in the API responses as well.