# SQL (Relational) Databases

FastAPI doesn't require you to use a SQL (relational) database.

But you can use any relational database that you want.

Here we'll see an example using SQLAlchemy.

You can easily adapt it to any database supported by SQLAlchemy, like:

- PostgreSQL
- MySQL
- SOLite
- Oracle
- · Microsoft SQL Server, etc.

In this example, we'll use **SQLite**, because it uses a single file and Python has integrated support. So, you can copy this example and run it as is.

Later, for your production application, you might want to use a database server like PostgreSQL.



Tip

There is an official project generator with **FastAPI** and **PostgreSQL**, all based on **Docker**, including a frontend and more tools: https://github.com/tiangolo/full-stack-fastapi-postgresql



Note

Notice that most of the code is the standard SQLAlchemy code you would use with any framework.

The FastAPI specific code is as small as always.

## **ORMs**

FastAPI works with any database and any style of library to talk to the database.

A common pattern is to use an "ORM": an "object-relational mapping" library.

An ORM has tools to convert ("map") between objects in code and database tables ("relations").

With an ORM, you normally create a class that represents a table in a SQL database, each attribute of the class represents a column, with a name and a type.

For example a class Pet could represent a SQL table pets.

And each instance object of that class represents a row in the database.

For example an object orion\_cat (an instance of Pet ) could have an attribute orion\_cat.type, for the column type. And the value of that attribute could be, e.g. "cat".

These ORMs also have tools to make the connections or relations between tables or entities.

This way, you could also have an attribute orion\_cat.owner and the owner would contain the data for this pet's owner, taken from the table owners.

So, orion\_cat.owner.name could be the name (from the name column in the owners table) of this pet's owner.

It could have a value like "Arguilian".

And the ORM will do all the work to get the information from the corresponding table *owners* when you try to access it from your pet object.

Common ORMs are for example: Django-ORM (part of the Django framework), SQLAlchemy ORM (part of SQLAlchemy, independent of framework) and Peewee (independent of framework), among others.

Here we will see how to work with SQLAlchemy ORM.

In a similar way you could use any other ORM.



There's an equivalent article using Peewee here in the docs.

## File structure

For these examples, let's say you have a directory named <code>my\_super\_project</code> that contains a sub-directory called <code>sql\_app</code> with a structure like this:

The file \_\_init\_\_.py is just an empty file, but it tells Python that sql\_app with all its modules (Python files) is a package.

Now let's see what each file/module does.

# Install SQLAlchemy

First you need to install SQLAlchemy:

```
$ pip install sqlalchemy
---> 100%
```

# Create the SQLAlchemy parts

Let's refer to the file sql\_app/database.py.

## Import the SQLAlchemy parts

```
{!../../docs_src/sql_databases/sql_app/database.py!}
```

## Create a database URL for SQLAlchemy

```
{!../../../docs_src/sql_databases/sql_app/database.py!}
```

In this example, we are "connecting" to a SQLite database (opening a file with the SQLite database).

The file will be located at the same directory in the file sql\_app.db.

That's why the last part is ./sql\_app.db.

If you were using a PostgreSQL database instead, you would just have to uncomment the line:

```
SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"
```

...and adapt it with your database data and credentials (equivalently for MySQL, MariaDB or any other).



This is the main line that you would have to modify if you wanted to use a different database.

Create the SQLAlchemy engine

The first step is to create a SQLAlchemy "engine".

We will later use this engine in other places.

```
{!../../../docs_src/sql_databases/sql_app/database.py!}
```

#### Note

The argument:

```
connect_args={"check_same_thread": False}
```

...is needed only for SQLite. It's not needed for other databases.

## 0

#### **Technical Details**

By default SQLite will only allow one thread to communicate with it, assuming that each thread would handle an independent request.

This is to prevent accidentally sharing the same connection for different things (for different requests).

But in FastAPI, using normal functions ( def ) more than one thread could interact with the database for the same request, so we need to make SQLite know that it should allow that with connect\_args= {"check\_same\_thread": False}.

Also, we will make sure each request gets its own database connection session in a dependency, so there's no need for that default mechanism.

#### Create a SessionLocal class

Each instance of the SessionLocal class will be a database session. The class itself is not a database session yet.

But once we create an instance of the SessionLocal class, this instance will be the actual database session.

We name it SessionLocal to distinguish it from the Session we are importing from SQLAlchemy.

We will use Session (the one imported from SQLAlchemy) later.

To create the SessionLocal class, use the function sessionmaker:

```
{!../../docs_src/sql_databases/sql_app/database.py!}
```

#### Create a Base class

Now we will use the function declarative\_base() that returns a class.

Later we will inherit from this class to create each of the database models or classes (the ORM models):

```
{!../../docs_src/sql_databases/sql_app/database.py!}
```

## Create the database models

Let's now see the file sql\_app/models.py.

## Create SQLAlchemy models from the Base class

We will use this Base class we created before to create the SQLAlchemy models.



Tip

SQLAlchemy uses the term "model" to refer to these classes and instances that interact with the database.

But Pydantic also uses the term "**model**" to refer to something different, the data validation, conversion, and documentation classes and instances.

Import Base from database (the file database.py from above).

Create classes that inherit from it.

These classes are the SQLAlchemy models.

```
{!../../docs_src/sql_databases/sql_app/models.py!}
```

The \_\_tablename\_\_ attribute tells SQLAlchemy the name of the table to use in the database for each of these models.

### Create model attributes/columns

Now create all the model (class) attributes.

Each of these attributes represents a column in its corresponding database table.

We use Column from SQLAlchemy as the default value.

And we pass a SQLAlchemy class "type", as Integer, String, and Boolean, that defines the type in the database, as an argument.

```
{!../../docs_src/sql_databases/sql_app/models.py!}
```

## Create the relationships

Now create the relationships.

For this, we use relationship provided by SQLAlchemy ORM.

This will become, more or less, a "magic" attribute that will contain the values from other tables related to this one.

```
{!../../docs_src/sql_databases/sql_app/models.py!}
```

When accessing the attribute items in a User, as in my\_user.items, it will have a list of Item SQLAlchemy models (from the items table) that have a foreign key pointing to this record in the users table.

When you access <code>my\_user.items</code>, SQLAlchemy will actually go and fetch the items from the database in the <code>items</code> table and populate them here.

And when accessing the attribute owner in an Item, it will contain a User SQLAlchemy model from the users table. It will use the owner\_id attribute/column with its foreign key to know which record to get from the users table.

# Create the Pydantic models

Now let's check the file sql\_app/schemas.py.



Tip

To avoid confusion between the SQLAlchemy *models* and the Pydantic *models*, we will have the file models.py with the SQLAlchemy models, and the file schemas.py with the Pydantic models.

These Pydantic models define more or less a "schema" (a valid data shape).

So this will help us avoiding confusion while using both.

## Create initial Pydantic models / schemas

Create an ItemBase and UserBase Pydantic *models* (or let's say "schemas") to have common attributes while creating or reading data.

And create an ItemCreate and UserCreate that inherit from them (so they will have the same attributes), plus any additional data (attributes) needed for creation.

So, the user will also have a password when creating it.

But for security, the password won't be in other Pydantic *models*, for example, it won't be sent from the API when reading a user.

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/schemas.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/schemas.py!}
```

Python 3.10 and above

```
{!> ../../docs_src/sql_databases/sql_app_py310/schemas.py!}
```

#### SQLAlchemy style and Pydantic style

Notice that SQLAlchemy *models* define attributes using =, and pass the type as a parameter to Column, like in:

```
name = Column(String)
```

while Pydantic models declare the types using :, the new type annotation syntax/type hints:

```
name: str
```

Have it in mind, so you don't get confused when using = and : with them.

### Create Pydantic models / schemas for reading / returning

Now create Pydantic *models* (schemas) that will be used when reading data, when returning it from the API.

For example, before creating an item, we don't know what will be the ID assigned to it, but when reading it (when returning it from the API) we will already know its ID.

The same way, when reading a user, we can now declare that items will contain the items that belong to this user.

Not only the IDs of those items, but all the data that we defined in the Pydantic *model* for reading items: Item.

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/schemas.py!}
```

#### Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/schemas.py!}
```

#### Python 3.10 and above

```
{!> ../../docs_src/sql_databases/sql_app_py310/schemas.py!}
```



#### Tip

Notice that the User, the Pydantic model that will be used when reading a user (returning it from the API) doesn't include the password.

### Use Pydantic's orm\_mode

Now, in the Pydantic models for reading, Item and User, add an internal Config class.

This Config class is used to provide configurations to Pydantic.

In the Config class, set the attribute orm\_mode = True.

#### Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/schemas.py!}
```

#### Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/schemas.py!}
```

#### Python 3.10 and above

```
{!> ../../docs_src/sql_databases/sql_app_py310/schemas.py!}
```



#### Tip

Notice it's assigning a value with = , like:

```
orm_mode = True
```

It doesn't use : as for the type declarations before.

This is setting a config value, not declaring a type.

Pydantic's orm\_mode will tell the Pydantic model to read the data even if it is not a dict, but an ORM model (or any other arbitrary object with attributes).

This way, instead of only trying to get the id value from a dict, as in:

```
id = data["id"]
```

it will also try to get it from an attribute, as in:

```
id = data.id
```

And with this, the Pydantic *model* is compatible with ORMs, and you can just declare it in the response\_model argument in your *path operations*.

You will be able to return a database model and it will read the data from it.

#### **Technical Details about ORM mode**

SQLAlchemy and many others are by default "lazy loading".

That means, for example, that they don't fetch the data for relationships from the database unless you try to access the attribute that would contain that data.

For example, accessing the attribute items:

```
current_user.items
```

would make SQLAlchemy go to the items table and get the items for this user, but not before.

Without orm\_mode, if you returned a SQLAlchemy model from your path operation, it wouldn't include the relationship data.

Even if you declared those relationships in your Pydantic models.

But with ORM mode, as Pydantic itself will try to access the data it needs from attributes (instead of assuming a dict ), you can declare the specific data you want to return and it will be able to go and get it, even from ORMs.

# **CRUD** utils

Now let's see the file sql\_app/crud.py.

In this file we will have reusable functions to interact with the data in the database.

CRUD comes from: Create, Read, Update, and Delete.

...although in this example we are only creating and reading.

#### Read data

Import Session from sqlalchemy.orm, this will allow you to declare the type of the db parameters and have better type checks and completion in your functions.

Import models (the SQLAlchemy models) and schemas (the Pydantic models / schemas).

Create utility functions to:

- · Read a single user by ID and by email.
- · Read multiple users.
- · Read multiple items.

{!../../docs\_src/sql\_databases/sql\_app/crud.py!}



#### Tip

By creating functions that are only dedicated to interacting with the database (get a user or an item) independent of your path operation function, you can more easily reuse them in multiple parts and also add unit tests for them.

#### Create data

Now create utility functions to create data.

The steps are:

- Create a SQLAlchemy model instance with your data.
- add that instance object to your database session.
- commit the changes to the database (so that they are saved).
- refresh your instance (so that it contains any new data from the database, like the generated ID).

{!../../../docs\_src/sql\_databases/sql\_app/crud.py!}



#### 

The SQLAlchemy model for User contains a hashed\_password that should contain a secure hashed version of the password.

But as what the API client provides is the original password, you need to extract it and generate the hashed password in your application.

And then pass the hashed\_password argument with the value to save.

#### Α

#### Warning

This example is not secure, the password is not hashed.

In a real life application you would need to hash the password and never save them in plaintext.

For more details, go back to the Security section in the tutorial.

Here we are focusing only on the tools and mechanics of databases.



#### Tip

Instead of passing each of the keyword arguments to Item and reading each one of them from the Pydantic model, we are generating a dict with the Pydantic model's data with:

```
item.dict()
```

and then we are passing the dict's key-value pairs as the keyword arguments to the SQLAlchemy Item, with:

```
Item(**item.dict())
```

And then we pass the extra keyword argument owner\_id that is not provided by the Pydantic model, with:

```
Item(**item.dict(), owner_id=user_id)
```

# Main FastAPI app

And now in the file sql\_app/main.py let's integrate and use all the other parts we created before.

#### Create the database tables

In a very simplistic way create the database tables:

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/main.py!}
```

#### **Alembic Note**

Normally you would probably initialize your database (create tables, etc) with Alembic.

And you would also use Alembic for "migrations" (that's its main job).

A "migration" is the set of steps needed whenever you change the structure of your SQLAlchemy models, add a new attribute, etc. to replicate those changes in the database, add a new column, a new table, etc.

You can find an example of Alembic in a FastAPI project in the templates from Project Generation - Template. Specifically in the alembic directory in the source code.

## Create a dependency

Now use the SessionLocal class we created in the sql\_app/database.py file to create a dependency.

We need to have an independent database session/connection (SessionLocal) per request, use the same session through all the request and then close it after the request is finished.

And then a new session will be created for the next request.

For that, we will create a new dependency with yield, as explained before in the section about Dependencies with yield.

Our dependency will create a new SQLAlchemy SessionLocal that will be used in a single request, and then close it once the request is finished.

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/main.py!}
```

### 🚺 Info

We put the creation of the SessionLocal() and handling of the requests in a try block.

And then we close it in the finally block.

This way we make sure the database session is always closed after the request. Even if there was an exception while processing the request.

But you can't raise another exception from the exit code (after yield ). See more in Dependencies with yield and HTTPException

And then, when using the dependency in a *path operation function*, we declare it with the type Session we imported directly from SQLAlchemy.

This will then give us better editor support inside the *path operation function*, because the editor will know that the db parameter is of type Session:

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/main.py!}
```

## a

#### **Technical Details**

The parameter db is actually of type SessionLocal, but this class (created with sessionmaker()) is a "proxy" of a SQLAlchemy Session, so, the editor doesn't really know what methods are provided.

But by declaring the type as Session, the editor now can know the available methods (.add(), .query(), .commit(), etc) and can provide better support (like completion). The type declaration doesn't affect the actual object.

### Create your FastAPI path operations

Now, finally, here's the standard **FastAPI** path operations code.

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/main.py!}
```

We are creating the database session before each request in the dependency with yield, and then closing it afterwards.

And then we can create the required dependency in the *path operation function*, to get that session directly.

With that, we can just call <code>crud.get\_user</code> directly from inside of the *path operation function* and use that session.



#### 

Notice that the values you return are SQLAlchemy models, or lists of SQLAlchemy models.

But as all the path operations have a response\_model with Pydantic models / schemas using orm\_mode, the data declared in your Pydantic models will be extracted from them and returned to the client, with all the normal filtering and validation.



#### Tip

Also notice that there are response\_models that have standard Python types like List[schemas.Item].

But as the content/parameter of that List is a Pydantic model with orm\_mode, the data will be retrieved and returned to the client as normally, without problems.

## About def vs async def

Here we are using SQLAlchemy code inside of the path operation function and in the dependency, and, in turn, it will go and communicate with an external database.

That could potentially require some "waiting".

But as SQLAlchemy doesn't have compatibility for using await directly, as would be with something like:

```
user = await db.query(User).first()
```

...and instead we are using:

```
user = db.query(User).first()
```

Then we should declare the path operation functions and the dependency without async def, just with a normal def, as:

```
@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
```



#### Info

If you need to connect to your relational database asynchronously, see Async SQL (Relational) Databases.

## Very Technical Details

If you are curious and have a deep technical knowledge, you can check the very technical details of how this async def vs def is handled in the Async docs.

# Migrations

Because we are using SQLAlchemy directly and we don't require any kind of plug-in for it to work with **FastAPI**, we could integrate database migrations with Alembic directly.

And as the code related to SQLAlchemy and the SQLAlchemy models lives in separate independent files, you would even be able to perform the migrations with Alembic without having to install FastAPI, Pydantic, or anything else.

The same way, you would be able to use the same SQLAlchemy models and utilities in other parts of your code that are not related to **FastAPI**.

For example, in a background task worker with Celery, RQ, or ARQ.

# Review all the files

Remember you should have a directory named <code>my\_super\_project</code> that contains a sub-directory called <code>sql\_app</code>.

sql\_app should have the following files:

- sql\_app/\_\_init\_\_.py : is an empty file.
- sql\_app/database.py:

```
{!../../../docs_src/sql_databases/sql_app/database.py!}
```

• sql\_app/models.py:

```
{!../../docs_src/sql_databases/sql_app/models.py!}
```

• sql\_app/schemas.py:

Python 3.6 and above

```
{!> ../../../docs_src/sql_databases/sql_app/schemas.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/schemas.py!}
```

#### Python 3.10 and above

```
{!> ../../docs_src/sql_databases/sql_app_py310/schemas.py!}
```

• sql\_app/crud.py:

```
{!../../docs_src/sql_databases/sql_app/crud.py!}
```

• sql\_app/main.py:

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/main.py!}
```

## Check it

You can copy this code and use it as is.



Info

In fact, the code shown here is part of the tests. As most of the code in these docs.

Then you can run it with Uvicorn:

```
$ uvicorn sql_app.main:app --reload

<span style="color: green;">INFO</span>: Uvicorn running on http://127.0.0.1:8000
(Press CTRL+C to quit)
```

And then, you can open your browser at http://127.0.0.1:8000/docs.

And you will be able to interact with your **FastAPI** application, reading data from a real database:



# Interact with the database directly

If you want to explore the SQLite database (file) directly, independently of FastAPI, to debug its contents, add tables, columns, records, modify data, etc. you can use DB Browser for SQLite.

It will look like this:



You can also use an online SQLite browser like SQLite Viewer or ExtendsClass.

## Alternative DB session with middleware

If you can't use dependencies with <code>yield</code> -- for example, if you are not using **Python 3.7** and can't install the "backports" mentioned above for **Python 3.6** -- you can set up the session in a "middleware" in a similar way.

A "middleware" is basically a function that is always executed for each request, with some code executed before, and some code executed after the endpoint function.

#### Create a middleware

The middleware we'll add (just a function) will create a new SQLAlchemy SessionLocal for each request, add it to the request and then close it once the request is finished.

Python 3.6 and above

```
{!> ../../docs_src/sql_databases/sql_app/alt_main.py!}
```

Python 3.9 and above

```
{!> ../../docs_src/sql_databases/sql_app_py39/alt_main.py!}
```



#### Info

We put the creation of the SessionLocal() and handling of the requests in a try block.

And then we close it in the finally block.

This way we make sure the database session is always closed after the request. Even if there was an exception while processing the request.

## About request.state

request.state is a property of each Request object. It is there to store arbitrary objects attached to the request itself, like the database session in this case. You can read more about it in Starlette's docs about Request state.

Page: 17 of 18

For us in this case, it helps us ensure a single database session is used through all the request, and then closed afterwards (in the middleware).

## Dependencies with yield or middleware

Adding a **middleware** here is similar to what a dependency with yield does, with some differences:

- It requires more code and is a bit more complex.
- The middleware has to be an async function.
  - If there is code in it that has to "wait" for the network, it could "block" your application there and degrade performance a bit.
  - Although it's probably not very problematic here with the way SQLA1chemy works.
  - But if you added more code to the middleware that had a lot of <a href="L/O">L/O</a> waiting, it could then be problematic.
- A middleware is run for every request.
  - So, a connection will be created for every request.
  - Even when the path operation that handles that request didn't need the DB.



#### Tip

It's probably better to use dependencies with yield when they are enough for the use case.



#### Info

Dependencies with yield were added recently to FastAPI.

A previous version of this tutorial only had the examples with a middleware and there are probably several applications using the middleware for database session management.

Last update: September 11, 2022 Created: September 11, 2022

Page: 18 of 18