CODING NOMADS

9) Database Management using Flask-SQLAlchemy Lesson

Querying the Database with Flask-SQLALchemy and the Flask Shell

24 min to complete · By Brandon Gigous

Contents

You might be wondering how to create and query the models in a database. For that, there's the convenient `flask shell` command, database session, and Flask SQLAlchemy's `query` object.

Flask SQLAlchemy Flask Shell

Flask Shell is the power of your Flask application in a Python interpreter! To begin, make sure you have the `FLASK_APP` environment variable set to `hello.py` and type in `flask shell` into your terminal. (Reminder: the one in VS Code works pretty well!)

```
(env) $ flask shell
>>> # I'm the Python interpreter with a Flask application context! :D
```

Once you're in a Flask shell session, you can keep issuing commands line by line. When you're done, just use `exit()` like you would in a normal Python interpreter session.

Creating Tables with Flask SQLAlchemy

First you'll command Flask-SQLAlchemy to create your database with the models you have defined. It will then look for all your subclasses of `db.Model` and create all the tables for you automagically.

Create a new `flask shell` session if you haven't already and keep it open for the lesson:

```
(env) $ flask shell
>>> from hello import db
>>> db.create_all()
```

You should now have a new file called `data-dev.sqlite`. That's your database! As it might imply, `create_all()` creates all of your new tables.

If you accidentally call the `create_all()` function twice, no worries, it won't re-create any tables that were already created. The bad news is, if you made changes to your models like adding columns and you *already* have a database, again, `create_all()` won't update those.

Dropping Tables

To update them, you have to nuke the current tables. That's the brute force way, there is a thing called "migration" that you'll learn about really soon. To destroy all the data in your tables and update them, use the `drop_all()` function, then the `create_all()` function again:

```
>>> db.drop_all()
>>> db.create_all()
```

It's fine to do this now since you don't really have much or any data in the first place.

Inserting Rows Into Your Tables

To insert users into your database with sample usernames and give them a role, you use the `Role` and `User` constructors. These constructors take keyword arguments that match the attributes in your models. Give it a swirl:

```
(env) $ flask shell
>>> from hello import Role, User
>>> admin_role = Role(name='Administrator')
>>> user_role = Role(name='User')
>>> user_paul = User(username='paul', role=admin_role)
>>> user_sven = User(username='sven', role=user_role)
>>> user_jan = User(username='jan', role=user_role)
>>> user_gwen = User(username='gwen', role=user_role)
```

You can use `role` even though it's not explicitly an attribute in the `User` model. Flask-SQLAlchemy lets you "pretend" that `role` is a column in your users table, but in reality it's a high-level representation of the "one" side of the one-to-many relationship. You can make your life easier by just passing in a `Role` instance.

Isn't it nice not to have to plug in IDs to your objects? The id attribute of the new objects haven't been explicitly set in any of these examples because they are set automatically. Check it out:

```
>>> print(admin_role.id)
None
>>> print(user_sven.id)
None
```


Flask SQLAlchemy Database Session

"Wait a second, did this course just lie to me? Have I been hoodwinked?!" No, of course not! I did say they are set automatically, but I didn't say when.

Now I will: your new objects have to be *committed* to the database before they get their `id` attributes assigned. Before you can commit them, you have to add them to the

database session. The database session is given to you as `db.session`. We can add objects to the session one by one, like so:

```
>>> db.session.add(admin_role)
>>> db.session.add(user_role)
```

Or, if you're slightly impatient like me, you can add them all at once. You can add the rest of the objects this way:

```
>>> db.session.add([user_paul, user_sven, user_jan, user_gwen])
```

Oh, Python, how far you've come... Er, enough daydreaming! At this point you can **commit** your database session:

```
>>> db.session.commit()
```

You've officially added data to your database! How's it feel? Let's make the moment especially meaningful by checking those `id` attributes again:

```
>>> print(admin_role.id)
0
>>> print(user_jan.id)
2
```

Something important to remember about database sessions and committing: if an error occurs when the session is being written to the database, the whole session gets discarded. That just means the database does an "undo" operation on whatever it added to the database from the session you committed. Commit related changes together to avoid any errors.

Note: A `flask shell` session and a database session are two very different things! The `flask shell` session is how you can interact with the database through the Python interpreter, and gives you a Flask application context. The database session is for the database only, and allows you to queue up data to commit to your database.

Modifying and Deleting Rows

Did you happen to spell "Administrator" wrong back when you created the `admin_role` object? This might be a good time to go ahead and rename it to "Admin" to avoid having to type that painfully long word again.

All you need is an instance of the row you want to change, make your changes, then you use the `db.session.add()` command to update the model, and finally a `db.session.commit()`:

```
>>> admin_role.name = 'Admin'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

To delete a row, you use the `db.session.delete()` method. The phone rings, you hear from Jan that she wants her `User` to be removed from the database. That's convenient, since you're already in the `flask shell` session! With just a few keystrokes, you can make her wish a reality:

```
>>> db.session.delete(user_jan)
>>> db.session.commit()
```

Basic Queries and Filtering

You've created roles and users, added them to a database session, then committed them to write them to the database. You took existing roles and users and modified or deleted them, and wrote it back to the database again.

You worked hard to put that data in there, now how to get it *out*? This is where **querying** comes in, and to help out, Flask-SQLAlchemy gives you a `query` object with all of your models.

If you left your `flask shell` session at any time, it's fine to reopen it. Just know that your Python objects you created earlier won't exist anymore and you'll have to recreate them. That's okay, because you can always find them again now that they live in the database!

Query All Data

Getting all data from one of your tables is the most basic query, and you can do it with the `all()` method:

```
(env) $ flask shell
>>> Role.query.all()
[<Role 'Admin'>, <Role 'User'>]
>>> User.query.all()
[<User 'paul'>, <User 'sven'>, <User 'gwen'>]
```

Query with Filters

However, that's not exactly useful if you only need *some* of the data. You can get more specific with **filters**. One such filter is `filter_by()`, and you use the `filter_by()` method on the `query` object:

```
>>> User.query.filter_by(role=user_role).all()
[<User 'sven'>, <User 'gwen'>]
>>> admin_role = Role.query.filter_by(name="Admin").first()
```

While `all()` is definitely useful for getting literally *all* data in a table, you can still use it after applying a filter to get *all* the filtered data.

In the first command, `User` is filtered by the `user_role`. Meaning, the `users` table is being filtered to return *all* users with the "User" role.

In the second command, `first()` returns the *first* result found from the issued query, or if there are no results, it returns `None`. The `admin_role` gets the `Role` with the name of `Admin`. It's basically the same object as the one you originally made.

Wouldn't it be nice if you could actually *see* what the query is? Y'know, like the `SELECT` and `FROM` statements and such. All that has to be done is to convert the query to a string, before `all()` or `first()` is called.

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id \nFROM users \nWHERE :param_1 = users.
```

So if you're ever curious what SQLAlchemy is actually doing to filter your data, this is a great way to see. There are a bunch of other query filters you can apply to your tables. Here are the most common ones used in the wild:

Query Filters

Filter Method	What It Does
`filter()`	Adds an additional filter to original query
`filter_by()`	Adds an additional equality filter to original query
`group_by()`	Groups the results of original query according to the given criteria
`limit()`	Limits number of results of original query to the given number
`offset()`	Applies an offset into the list of results of original query
`order_by()`	Sorts the results of original query according to the given criteria

Keep in mind that these filters all *return* another query object. That means you can keep stringing filters together to get *really, really* specific. As in, something like this:

```
>>> Users.query.filter_by(role=user_role).limit(1).all()
```

```
[<User 'sven'>]
```

Query Executor

That uses two filters side by side, a filter to get only users with the "User" role, and to limit the results to only one. The `all()` query **executor** is used here to give a list of the limited one result. While `all()` and `first()` are great, there are other executors. Gathered here are the most useful:

Executor Method	What It Does
`all()`	Returns all results of a query as a list
`first()`	Returns the first result of a query, or None if there are no results
`first_or_404()`	Returns the first result of a query, otherwise sends a 404 error as the response
`get()`	Returns the row that matches the given primary key, otherwise None
`get_or_404()`	Returns the row that matches the given primary key, otherwise sends a 404 error as the response
`count()`	Returns the result count of the query
`paginate()`	Returns a `Pagination` object that contains the specified range of results

Query Relationships

You've seen how the `users` attribute in the `Role` table acts just like a Python list when you check its value. You can see all users with the "User" role with `user_role.users`:

```
>>> users = user_role.users
>>> users
[<User 'sven'>, <User 'gwen'>]
>>> users[0].role
<Role 'User'>
```

That output looks pretty similar to the result of a query, doesn't it? Almost as if the users table was filtered by the "User" role, then `all()` was called on that hypothetical query object. Hmmmm... Don't worry, you're not crazy, the `user_role.users` expression is actually a query! Turns out there is an *implicit* query that runs when `user_role.users` is evaluated, and then `all()` is automatically called on that query object to give you the list of users.

Flask SQLALchemy Lazy Loading

What if there was a hhhhhhuuuuuuuuggggggggeeeee list of users? What if you wanted to add *more* filters to it? Maybe return them alphabetically or ordered by who won the most arm-wrestling contests? Let's fix that with the `lazy` relationship option.

Go back into `hello.py` and add the `lazy` keyword argument to your `Role` model:

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role', lazy='dynamic')
    # ...
```

Fantastic, now you can do some epic sorting. The `'dynamic'` argument means that SQLAlchemy won't automatically execute the query when you reference the `user` attribute directly:

```
>>> user_role.users.order_by(User.username).all()
[<User 'gwen'>, <User 'sven'>]
```

"Lazy loading," means that data is accessed from the physical device *only* when it's needed. An example is reading from a huge spreadsheet. Loading all the data all at once can take a long time, but what if you only needed one column of data? Or even one cell in the spreadsheet? Lazy loading lets you load only data you need to load and nothing else so you can save that computation for other things. With SQLAlchemy, you can configure how you want data accessed through a relationship loaded, either lazily, loaded right away, or somewhere in between.

Flask SQLALchemy Lazy Values

Here's a list of all the values `lazy` can take on and what they do:

Value	How It Loads The Data
`dynamic`	Rather than loading the items directly the query that can load them is given
`immediate`	Items are loaded when the source object is loaded
`joined`	Items are loaded immediately as a *join*
`noload`	Items are never loaded
`select`	Items are loaded on-demand the first time they are accessed
`subquery`	Items are loaded immediately as a *subquery*

Flask Shell Context Processor

Did you notice that you needed to import your `Role` and `User` before you could use them in your `flask shell` session? The thought of entering them in again and again and again every time you want to interact with your data. Gah, that's painful to think about!

Flask has got you covered. All you have to do is make a function with and decorate it with the `shell_context_processor` decorator. Have the function return a dictionary with all the stuff you want it to have ready for you, and you're all set! Put this in your `hello.py` file:

```
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

Try it out in a new `flask shell` session:

```
(env) $ flask shell
>>> app
```

```
<Flask 'hello'>
>>> User
<class 'hello.User'>
```

And with that, you have now passed Flask-SQLAlchemy 101! Next is Flask-SQLAlchemy 201, where you'll apply what you learned here to your view functions.

Summary

You've done a fantastic job getting to grips with the intermediate aspects of Flask and databases, starting with the creation of models all the way to querying them. You've:

- Discovered the `flask shell` environment, which provides an interactive Python interpreter within the Flask application context, allowing you to interact with your database models directly.
- Learned how to use `db.create_all()` and `db.drop_all()` to manage our database table creation, especially when updating model structures.
- Practiced inserting data into the database using model constructors like `Role(name='Administrator')`, and how these instances don't receive `id` attributes until they are committed to the database.
- Added objects to the **database session** and committed them to the database using `db.session.add()` and `db.session.commit()`.
- Modified and deleted rows in the database, making corrections or complying with deletion requests.
- Got familiar with querying data from database tables, utilizing methods like `all()`, `first()`, and `filter_by()`, and even how to view the actual SQL query strings.
- Introduced to the concept of lazy loading, understanding how and why you might want to defer database queries until you actually need the data.

Previous Next → Video: Interacting with the Database from a Flask Shell...