CODING NOMADS

9) Database Management using Flask-SQLAlchemy Lesson

Create Models and Relationships with Flask-SQLAlchemy

14 min to complete · By Brandon Gigous

Contents

Once you have Flask-SQLAlchemy installed and configured, you're ready to start playing with models and relationships, whether that be Barbie doll drama or SQL, and preferably it's the latter. The basics are described in this lesson.

Defining Models

A **model** is an object that abstracts away much of the technical SQL functionality and that represents a persistent entity. The models are the objects that your webapp interacts with to get the data it needs and update data as needed. For you, that means it's just a Python object that has attributes that match the columns of a table.

Creating a model means defining a new class that represents a database table. Flask SQLAlchemy gives you a way to do that by providing you a base class for models called `Model` . It also provides several helper classes and functions.

Think back to the theme of the app in this course for a second: a music-sharing social media webapp. That involves having users that can share content, so you'll want to create a `User` model that can represent each user's information. Give defining `Model` s a spin by putting this in your `hello.py` file:

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return f"<Role {self.name}>"


class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return f"<User {self.username}>"
```

Great, you've got a couple of nice-looking models now! Roles will be talked about in a bit. For now, notice the `__tablename__` class variable, which defines the name of the table in your database. So, a `Role` class instance represents a row in the `roles` table, and a `User` represents a row in the `users` table.

You won't need to do much with the table name, but it's defined here so that a default name isn't chosen instead. Using plurals for the table name is popular, but unfortunately, it is not what Flask-SQLAlchemy sets as the default.

Flask SQLAlchemy Table Columns

After `__tablename__` are the attributes of the model, which are instances of the `db.Column` helper class.

Every `db.Column` has a type, and Flask SQLAlchemy gives you helper classes for those, too.

The `Integer` and `String` column types have been defined here, but there are many others. Here are the most common:

Type name	Python type	Description
`Boolean`	bool	True or False value
`DateTime`	datetime.datetime	Date and time value
`Float`	float	Floating-point number
`Integer`	int	Regular integer
`LargeBinary`	str	Binary blob
`PickleType`	Any	Python object Automatic Pickle serialization
`String`	str	Variable-length string
`Text`	str	Variable-length string, better for very long strings

After the first argument, other configuration options can be specified for a column. Some of them are:

Option	Description
`default`	Default value for column
`index`	If True, create an index for this column for more efficient queries
`nullable`	If True, allow empty values, otherwise don't allow empty values
`primary_key`	If True, the column is the table's primary key
`unique`	If True, don't allow duplicate values

For the new `User` model, `unique` is specified for the `username` attribute since you'd of course want your users to have something unique and unused from other users.

The `index` option is set to true which makes username lookups much faster. You'll be querying for usernames quite often later in the course.

Mentorship Makes the Difference!

You don't have to learn alone. Join the CodingNomads' 1-on-1 Mentorship Program and get:

- A team of mentors and advisors dedicated to your success
- Weekly 1-on-1 screen share meetings with your personal, professional mentor
- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

Flask SQLAlchemy One to Many Relationships

Since relational databases are *relational*, they gotta have **relationships**. Relationships are connections between rows of different tables. In the next code snippet, you'll establish a **one-to-many** relationship between `Role` and `User`.

Each user of *many* will have *one* role in the application. Examples of roles include a normal user, a moderator to manage comments, or a site admin to delete or edit certain content. This relationship between models can be defined with another column in the user table and a `relationship` in the role table. To establish a one-to-many relationship from `Role` to `User`, you do this:

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

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Foreign Keys

Look at the `User` model. Why would you want to add a column? Since each user has one role, to know which role that corresponds to, you can add a **foreign key** with the `db.ForeignKey()` helper class that references a row in the `roles` table.

Think of a *foreigner*, a tourist. This person brought an ID or passport from their home country to the new country. It's similar in a database with foreign keys. In this case, the `role_id` is like a tourist's ID. It *references* a `Role` in the `roles` table, but `role_id` is an *attribute* of `User`, like the person in the new country. The `db.ForeignKey()` argument is the column in the `roles` table that contains the primary key.

The `users` attribute of `Role` represents all the users that have a particular role. With an instance of `Role`, you can see the "many" side of the relationship with the `users` attribute. It's simply the list of all users with that role! The purpose of the `db.relationship()` is to tell the application what model is on the other side.

Backref

The `backref` keyword argument allows you to specify `role` as an attribute of a `User` instance *in addition to* the `role_id` attribute. Instead of `a_user_instance.role_id` to get the ID of the role, you can use `a_user_instance.role` to get the actual `Role`.

There are a few other relationship options available in SQLAlchemy:

Option name	What It Does
`backref`	Adds a backreference in the other model in the relationship
`lazy`	Specifies how the data for the related items is to be loaded
`order_by`	Specifies the ordering used for the items in the relationship
`primaryjoin`	Specifies the join condition between the two models explicitly
`secondary`	Specify the name of the association table to use in many-to-many relationships
`uselist`	If set to `False` , use a scalar instead of a list
`secondaryjoin`	Specify the secondary join condition for many-to-many relationships

There is a somewhat rare chance that `db.relationship()` won't be able to locate the foreign key by itself. For example, it could happen when you define two `Role` foreign keys in the `User` model. It can't be sure which one you mean, so giving it additional arguments to remove any ambiguity should solve the problem. In particular, both `primaryjoin` and `secondaryjoin` are options you can use to specify join conditions to remove ambiguity.

The `lazy` option is an interesting one because it can change how the data from the `relationship` is loaded from the database to the program. You'll see it in action in a bit.

You just got through the basics of models and relationships! Does it feel like boot camp? Good, that means you're learning. Now drop and gimme 20 pushups, then carry on to the next lesson.

- State of the tech job market – what's the deal with layoffs, AI taking our jobs, and shady coding bootcamps?
- Who IS getting jobs right now – and the skills in high demand
- What it really takes to get a technical job today – it's more than just tech skills
- And finally, is pursuing these paths still worth your time and money?

Register Here

Summary: Defining Models and Relationships in Flask SQLAlchemy

You just got through the basics models and relationships! Does it feel like boot camp? Good, that means you're learning.

You've taken your first steps into modeling with Flask-SQLAlchemy and have a foundational understanding of how to create database models and define relationships in your web applications. In this lesson, you've:

- Realized that **models** act as Python classes mapping to database tables.
- Defined a model by inheriting from `db.Model` and represented table rows with class attributes tied to `db.Column`.
- Noted the importance of `__tablename__`, which overrides the default naming conventions and usually is set to the plural form of the model name.
- Explored the various **data types** and **column options**, such as `Integer`, `String`, and `Boolean`, as well as configuration options like `primary_key` and `unique`.

- Gained insight into **relationships**, especially one-to-many, allowing you to connect users to roles and vice versa through the use of `db.relationship` and `db.ForeignKey`.
- Became familiar with `backref`, which adds a back-reference from one model to another, providing a convenient way to access related data.

By understanding these basics, you've created a strong foundation to build more complex data models. Keep practicing and explore how different configurations and types can affect your database interactions.

Previous Next → Querying the Database with Flask-SQLALchemy and th...