CODING NOMADS

17) Revisiting Relationships in the Database Lesson

Flask-SQLAlchemy Joins to Show Followed Users' Posts

20 min to complete · By Brandon Gigous

Contents

Note: This lesson builds on a project that's been gradually worked on throughout this Python + Flask course.

This lesson will show you how to make Flask-SQLAlchemy joins to show a user all the content created by the users they follow, and to show a user the compositions of users that follow them.

Querying For Followed Compositions

The index page currently shows compositions from all users. But wouldn't it be nice to see compositions from a certain group of users, like those one follows? You can have two

lists of compositions, one that shows the followed users' compositions, and one that shows all user compositions.

However, getting compositions created by followed users is not trivial. The obvious way to load them would be to get the list of followed users, then for each user, get their list of compositions. All of those compositions would get funneled and sorted into a single list. This, however, doesn't scale well. As the database grows, the effort to combine the compositions in this huge list won't be efficient at all, and things like pagination will slow to a crawl. Such a problem is known as the N+1 problem.

Using SQLAlchemy Joins

Instead of doing all that work for bad performance, why not just do a database query? You can do this kind of thing *really fast* with a SQLAlchemy **join** operation. This database operation takes two or more tables and jams them together, finding all combinations of rows that satisfy a given condition. The resulting combined rows are inserted into a temporary table.

Let's try an example to help it sink in. Say you have a table of `users`:

id	username
1	paul
2	sven
3	gwen

And these users have a few `compositions`. Here's an imaginary compositions table:

id	artist_id	title
1	1	Turkey Vulture Rap (by paul)
2	3	Soap Opera Sonata (by gwen)
3	2	Tangerine In My Olives (by sven)
4	3	If I Only Had A Bear Trap (by gwen)

And you have an imaginary `follows` table like this:

follower_id	following_id
1	3
2	3
2	1

So, to obtain a list of compositions by users followed by sven, you can do a join on the `compositions` and `follows` table. The first thing to do is filter the `follows` table so that the `follower_id` matches only sven. Then, the temporary join table is created from every possible combination of rows from the `compositions` and `follows` tables where the `artist_id` of the composition is the same as the `following_id` of the follow. Then, you will have all the compositions of the users followed by sven!

id	artist_id	title	follower_id	following_id
1	1	Turkey Vulture Rap (by paul)	2	1
2	3	Soap Opera Sonata (by gwen)	2	3
4	3	If I Only Had A Bear Trap (by gwen)	2	3

You'll notice the `artist_id` and `following_id` are identical. That's because they were the columns used to perform the join, and that's exactly what you'd want to get all compositions by users who sven follows.

- Accountability, feedback, encouragement, and success

Get Mentorship

Flask-SQLAlchemy Join Example

Of course, examples are nice, but what about real code that can perform a database join? That's next, so pay close attention. For the join that occurred above, you could write it as a query shown below. This would be done inside the `User` model:

```
return db.session.query(Composition).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Composition, Follow.following_id == Composition.artist_id)
```

That's a pretty huge mouthful of SQLAlchemy, but taking it apart bit by bit will help it make sense. The starting point for all queries you've seen starts with the `query` attribute of the model you want to query. But this time, it is different. While you ultimately want `compositions`, the first operation that should be done is a filter to the `follows` table. So:

- `db.session.query(Composition)`: You specify that the query will return `Composition` objects because, at the end of the day, that's what you want.
- `select_from(Follow)`: This indicates that the query begins with the `Follow` model.
- `filter_by(follower_id=self.id)`: This filters the `follows` table by the follower user. In sven's case, that's him.

- `join(Composition, Follow.following_id == Composition.artist_id)` : This does the join operation on 1) the result of the filter and 2) the `Composition` objects. It matches whenever the user that's followed matches the artist who made the `Composition

However, this could be even simpler. There are all kinds of ways that database queries can be rearranged to produce the same results, and this query is no different. You can swap the order of the filter and join:

```
return Composition.query.join(Follow, Follow.following_id == Composi
    .filter(Follow.follower_id == self.id)
```

This query starts from the `Composition.query` query object. Using that object, you can perform a SQLAlchemy join and then filter the results.

Databases are efficient. You might think that doing the join first and the filtering second would be more costly, but thanks to the smart people who build database engines for a living, these two queries have exactly the same performance. SQLAlchemy first takes a look at all the filters that are to be applied, then rearranges the order of operations of the query to make it as optimized as possible. You can even confirm the queries are the same by doing a `print()` on the query command (remember that from the Database Management section?).

Now you can have access to this conveniently as a property from the `User` model:

```
class User(UserMixin, db.Model):
    # ...
    @property
    def followed_compositions(self):
        return Composition.query.join(
            Follow, Follow.following_id == Composition.artist_id)\
            .filter(Follow.follower_id == self.id)
```

Don't worry too much if this is still difficult to understand. Sometimes, joins are hard to keep track of. Even experienced database developers get turned around in how to do a

complex join, like yours truly.

Yeehaw! You're almost done with getting follower and following and follow-these-compositions-and-those-users working in your web app. The last step is to just show the followed compositions or all compositions in the home page of your app, which this lesson will guide you through.

Viewing Choices

With a quick addition to the `index()` view function, you can allow users to view compositions posted by followed users or all compositions posted by all users. You might remember the discussion on cookies from the sections on User Authentication and Sending Emails. Well, you can use your own stored cookies to allow the user to change what they see:

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', ''
    if show_followed:
        query = current_user.followed_compositions
    else:
        query = Composition.query
    pagination = query.order_by(Composition.timestamp.desc()).pagina
        page,
        per_page=current_app.config['RAGTIME_COMPS_PER_PAGE'],
        error_out=False)
    compositions = pagination.items
    return render_template(
        'index.html',
        form=form,
        compositions=compositions,
        show_followed=show_followed,
        pagination=pagination
    )
```

The user's choice is stored in a cookie called `show_followed`. Whenever the user is logged in, the `show_followed` cookie is loaded. The string value of the cookie is converted to a boolean, and depending on its value, the `query` variable is set to either followed or all compositions. All compositions are shown with the `Composition.query`, and only followed compositions are shown with your newly created property `User.followed_compositions`.

Setting Cookies

But how does this cookie get set? With more view functions, of course! At least, that's an easy and effective way of doing it:

```
@main.route('/all')
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60*60) # 30 d
    return resp


@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60) # 30
    return resp
```

Now, you can add these routes to your index page template. When either of these functions are invoked, the `show_followed` cookie gets set to the proper value. However, only response objects can set cookies. The good news is that these response objects can be created within your view functions with the `make_response()` function. Flask automatically makes response objects out of whatever you pass in the `return` statement, whether it be a string containing HTML or a `render_template()` function call. Or, they can be manually created and passed directly.

The `make_response()` function takes the name of the cookie first, then the value it will take on. The `max_age` argument sets the number of seconds until the cookie expires,

and without a value, the cookie will be deleted when the browser closes. The maximum age of these cookies is set to 30 days, so if followed posts are chosen for the user to view, it will be remembered until 30 days have passed. It gets renewed if the user makes a different choice.

Add Tabs To Home Page

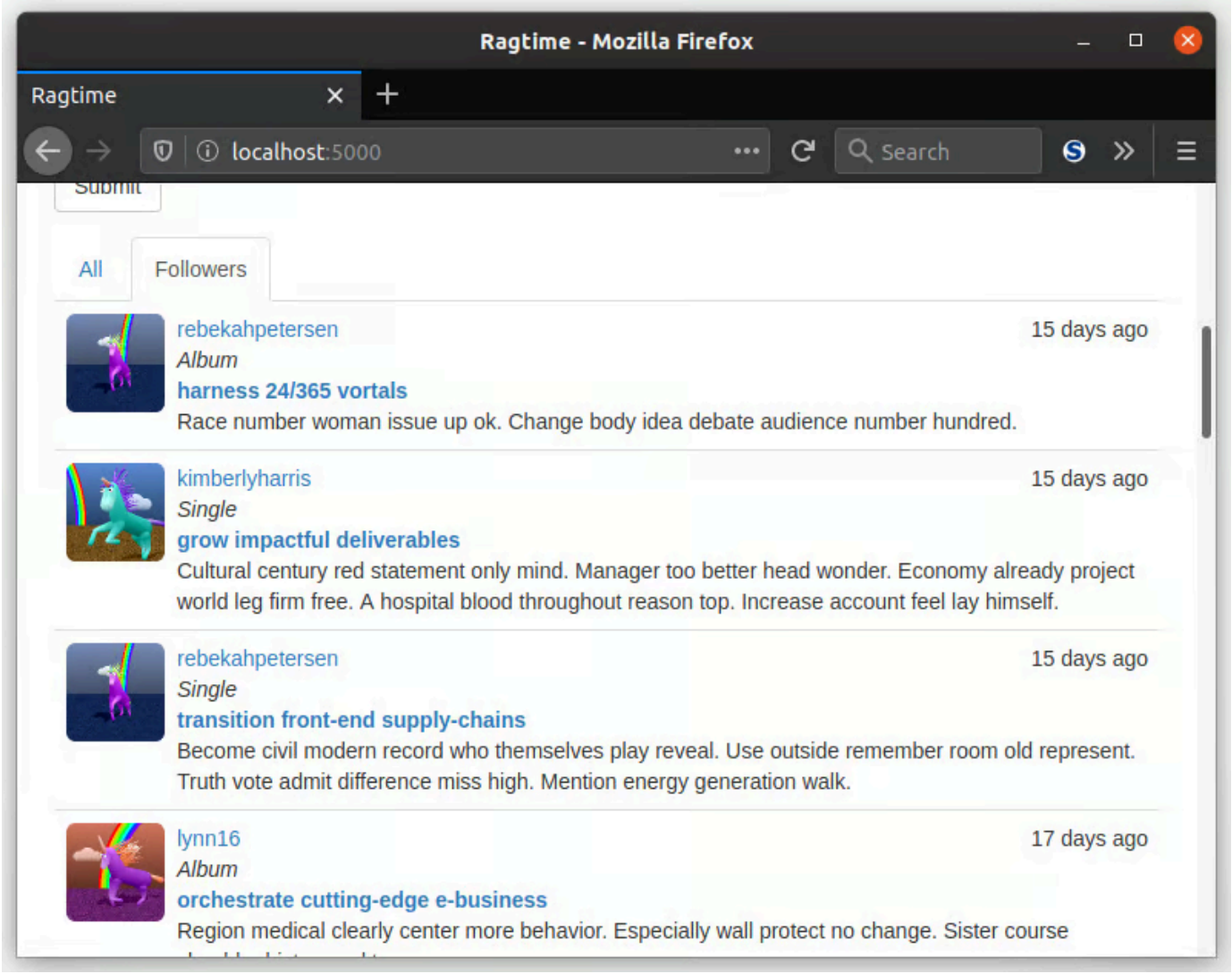
All that's needed for your index page template is some nice-looking tabs. These can be added while making use of your `show_followed` cookie and new routes:

```
{# ... #}
<div class="composition-tabs">
    <ul class="nav nav-tabs">
        <li{% if not show_followed %} class="active"{% endif %}>
```

```
                <a href="{{ url_for('.show_all') }}">All</a>
            </li>
            {% if current_user.is_authenticated %}
            <li{% if show_followed %} class="active"{% endif %}>
                <a href="{{ url_for('.show_followed') }}">Followers</a>
            </li>
            {% endif %}
        </ul>
        {% include '_compositions.html' %}
    </div>
    {# ... #}
```

If you haven't already, you can use this custom style for the tabs:

```
div.composition-tabs {
    margin-top: 16px;
}
```



But what about the user's own compositions? Shouldn't those show up, too? You can address this easily by registering all users as their own followers as they are created.

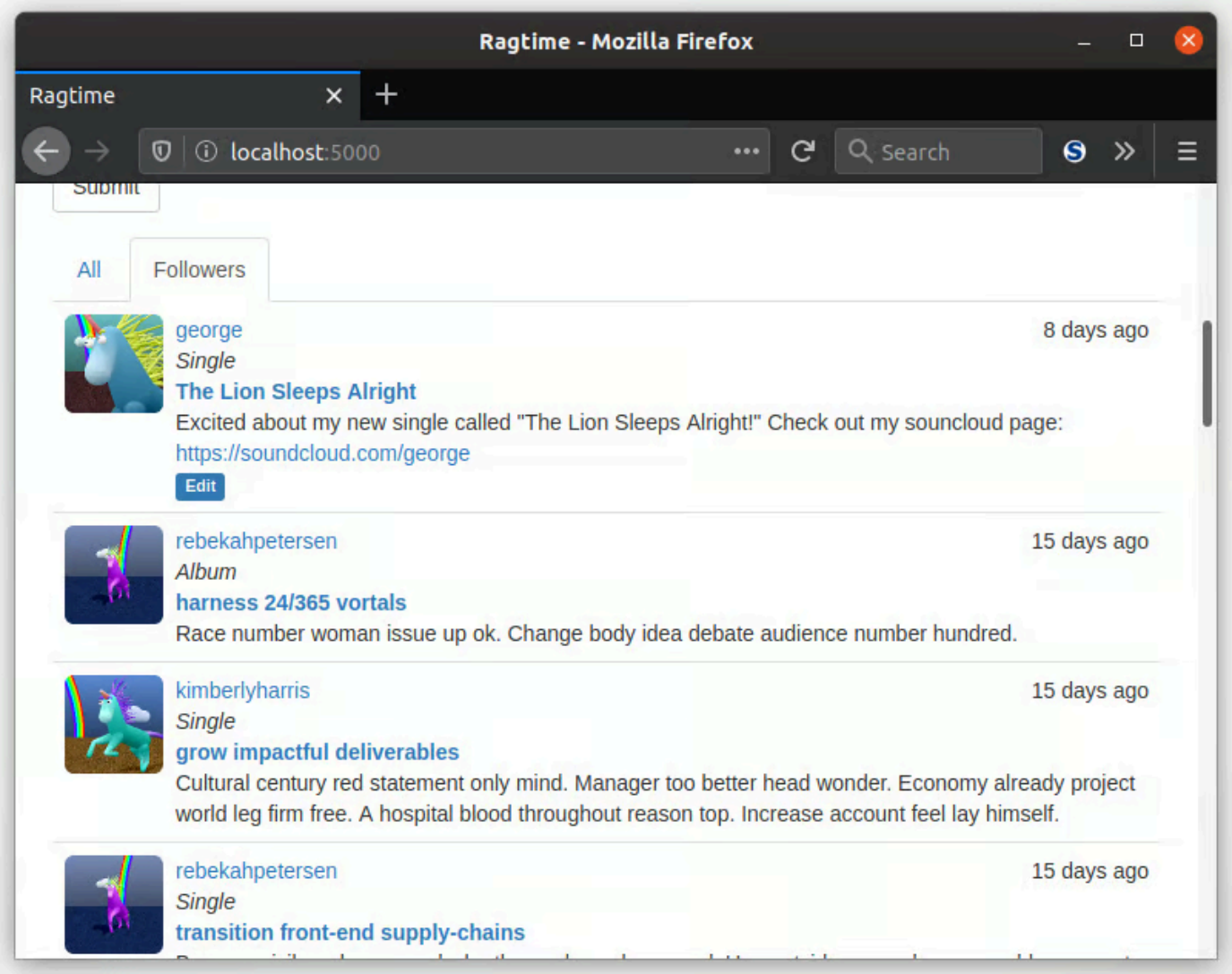
```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

As you might imagine, the users in the database have already been created, and so this won't work for those users. You could, of course, nuke the database and start over, but you don't have to. There's an easy solution, and that is to add a static method, just like for the `insert_roles()` function in the `Role` model. You can make an update function like so:

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
```

Then, you can run it straight from a Flask shell session:

```
(env) $ flask shell
>>> User.add_self_follows()
```



The only problem now? The user count in the user pages doesn't reflect the true count. Sure, the user is following themselves, but that's not what the users would expect. You just made a bit of hack so that they could see their own posts. So, to fix that, make sure you change the follower and following count to be one less in the user page.

Task: In your `user.html` template, change the follower and following count to be one less.

Awesome sauce! You're all done with adding following functionality to your webapp! The next section will introduce you to APIs and how to include one in your very own webapp.

Summary: Using SQLAlchemy Joins to Query your Database

In this lesson, you:

- Discovered the **N+1 problem** and why fetching data in separate queries for each user doesn't scale well.
- Discovered how to use **database joins** to efficiently query compositions created by users followed by a particular user.
- Learned to write **SQLAlchemy join queries** - starting with selecting the model and chaining with filters and join operations - to get the desired data.
- Understood the use of **properties in a model** to reference complex queries, like `followed_compositions` in the `User` model.
- Learned how to **toggle views** using cookies in Flask to either show compositions by all users or only those the current user follows.

Previous Next → Video: Database Joins with SQLAlchemy