Implementing User Following

30 min to complete · By Brandon Gigous

Contents

- Introduction
- Database Relationships
- Many-To-Many Relationship
- Association Table
- Implementing Followers In The Database
- Optimizing the Association Table
- Follow Model
- Associating The Association
- Python Helper Methods
- (Un)Following Other Users
- View Functions To Display Users
- Add To User Template
- Showing Followers and Following
- Summary: Implementing User Following with Many-to-Many Relationships

What do all social media apps have in common? The ability to follow, friend, buddy up, and/or subscribe to other users. That way, users can more easily keep up with what others are doing. In this section, you'll implement that capability into your app. This lesson, in particular, will introduce you to the many-to-many relationship.

Database Relationships

You've already learned that database relationships establish links between records. The most common type of relationship is the one-to-many relationship, where one record is linked with a bunch of other related records. Technically speaking, that means a foreign key is used on the "many" side to reference the primary key on the "one" side. You've already seen and implemented this kind of relationship as *one* role given to *many* users and *one* user having *many* compositions.

There are two more relationship types that can be seen as variants of the one-to-many relationship. These are the many-to-one relationship and the one-to-one relationship:

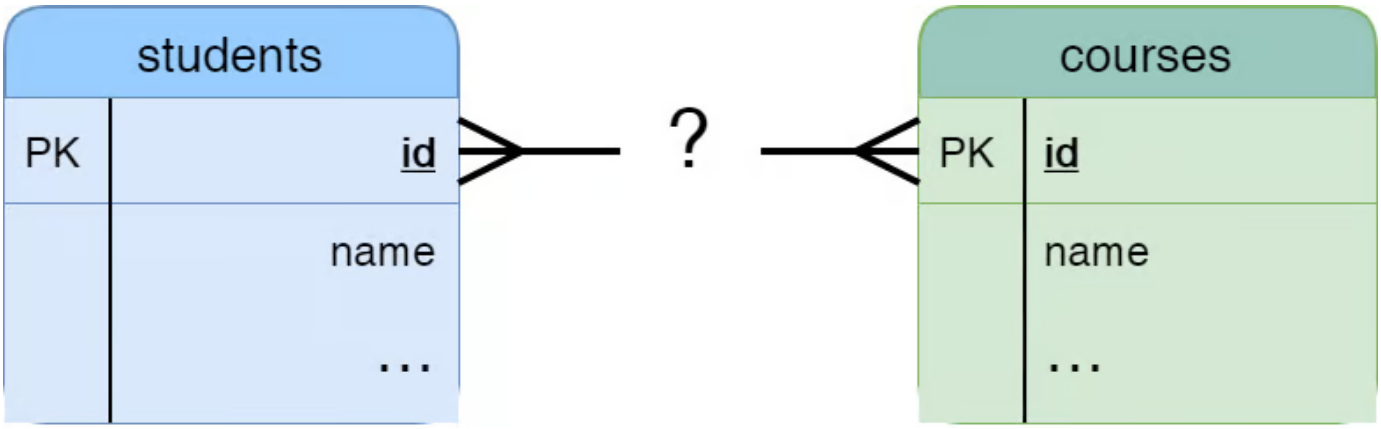
- The many-to-one relationship is simply the reverse of the one-to-many relationship type, or in other words, a one-to-many relationship from the perspective of the "many" side.
- A one-to-one relationship is just like a one-to-many relationship, except the "many" side can have only one element.

Things get murky when you start thinking about many-to-many relationships because they are not as simple as the other three relationship types. A many-to-many relationship means there are *many* elements on one side linked to *many* elements on the other side.

Many-To-Many Relationship

One thing that one-to-many, many-to-one, and one-to-one relationships have in common is that they all have at least *one* side with a single entity. So, to link a "many" to a "one", all you need is a foreign key to reference the "one" element on the "many" side. You might be thinking, "Okay, so I'd need a foreign key for one 'many' side. That just means I need to make another foreign key for another 'many' side, right?" Well, it's not quite that simple.

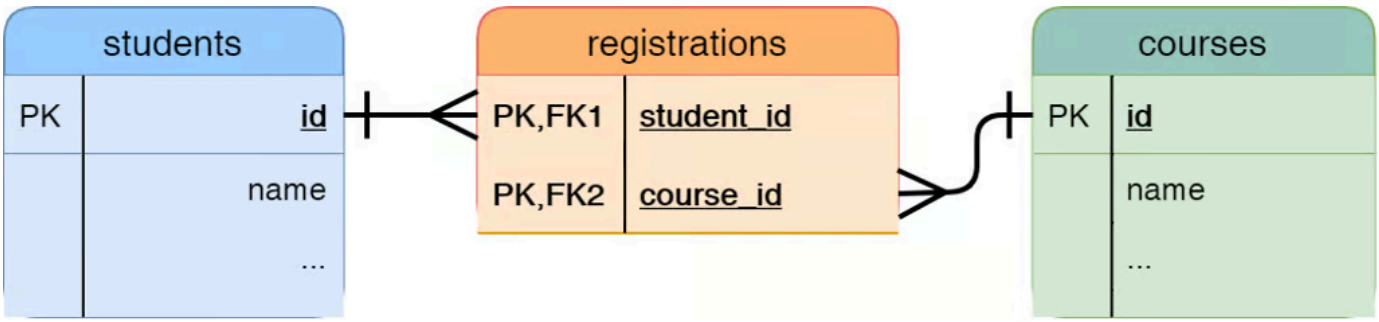
Let's think of an example. You're a student in a CodingNomads course, but there are more students than you taking the same course. That's one "many" side. The other "many" side is that one student can access *many* courses. How would you go about representing this in a database? Adding a foreign key to a course in the `student` table doesn't quite work because one student could take many courses. Adding a foreign key to a student in the `courses` table doesn't work either because multiple students could take one course!



So what's the solution? Well, think of it this way. The `students` table needs a *list* of foreign keys to `courses`, and the `courses` table needs a *list* of foreign keys to the `students` table. Kind of sounds like two one-to-many relationships, doesn't it? The solution is to create a "middle man" of sorts, another table called an **association table**.

Association Table

An association table "decomposes" so to speak the many-to-many relationships into two one-to-many relationships. It takes the on brunt of the work in *associating* one-of-the-many on one side (a student) to one-of-the-many on the other (a course). Call this table `registrations`, and it represents individual registrations of a student to a class. It has two foreign keys, for students and for courses.

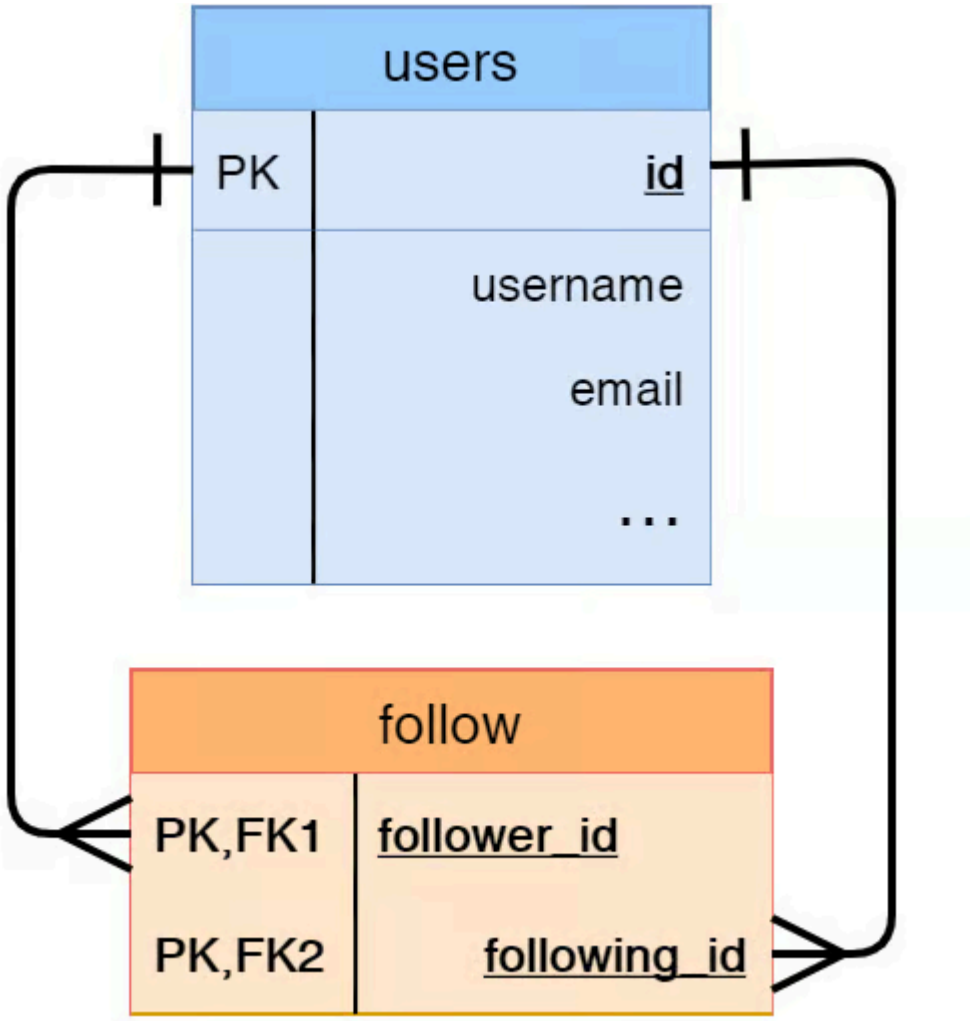


To query this many-to-many relationship requires two steps. First, in order to get the list of courses taken by a student, you query the `registrations` table for rows that match the student's ID. That's the first one-to-many relationship. Then, the many-to-one relationship between `registrations` and `courses` is traversed to obtain all the lists of courses. This works the other way, too, in that you can get the list of students who take a particular course.

Implementing Followers In The Database

While all that's fine and dandy, the students and courses example relates two different kinds of entities. You can find the courses a student takes and the students in a course, and there's an association table between the two. For your app, you're dealing entirely with users following other users. They are both `User` s! What is a student like you to do?

Fear not, for an association table can still be established between users and other users. It just means that both "one" sides of the two one-to-many relationships will point to the `users` table. This many-to-many relationship, decomposed into two one-to-many relationships, is *self-referential*.



With this new `follows` association table, you can can represent users following other users. You'll learn about that in the next lesson, if you keep *following* the trail...

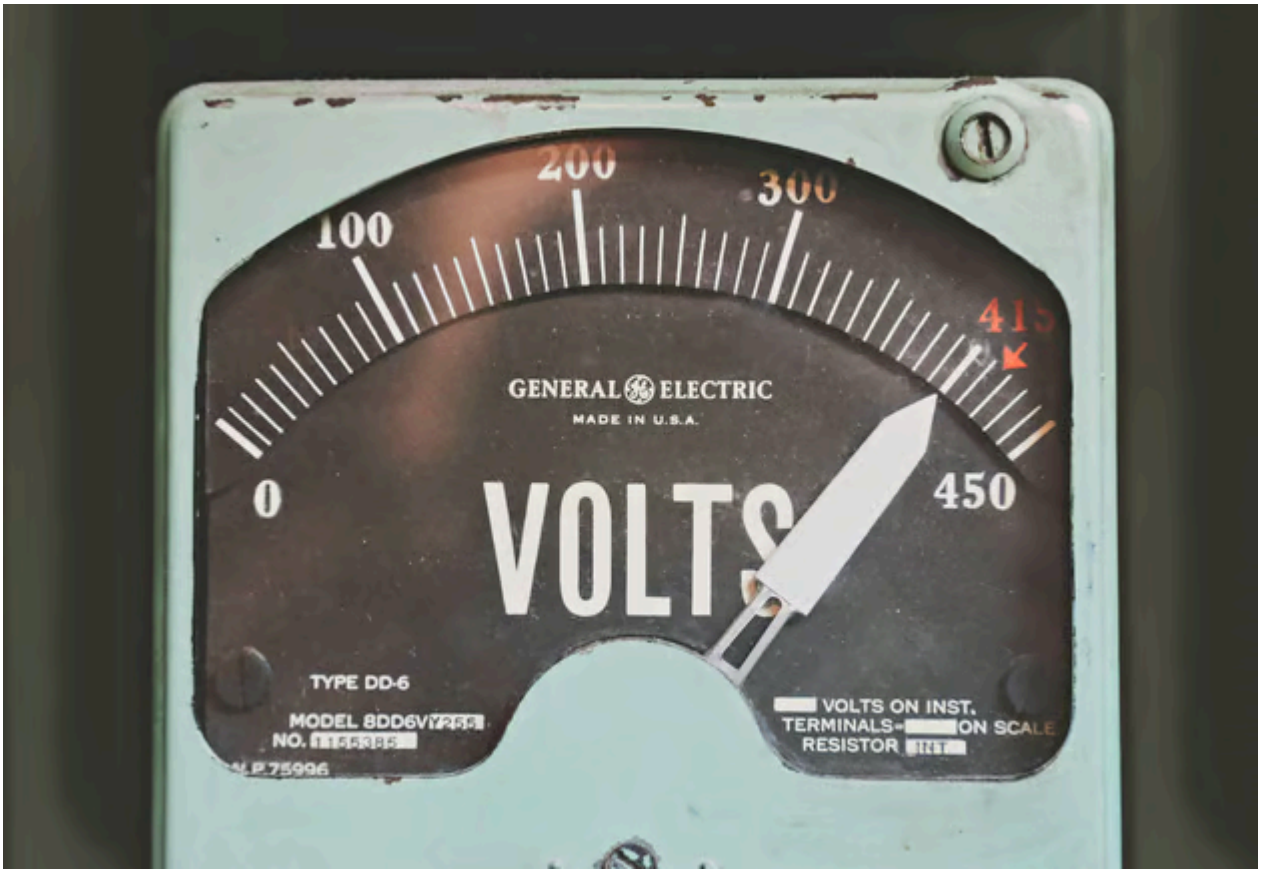
You learned about association tables earlier in the lesson and how they can reconcile the more complex details of many-to-many relationships. You also learned about self-referential many-to-many relationships, which just means instead of linking two related but *different* entities, they link records in the *same* table to each other. In this lesson,

you'll build one of these association tables called `follows` to help you create a self-referential relationship between users.

Optimizing the Association Table

When dealing with many-to-many relationships between entities, association tables are a must. But technically, an association table consists only of the information needed to link the two together. That means just two foreign keys in the table, one for the first side of the many-to-many relationship, and another for the second side.

Sometimes, though, it's useful to keep track of information *about* that relationship. In the case of a user following some other user, it could be useful to know *when* that follow occurred. That way, you could show followers based on chronological order. The good news is that SQLAlchemy has the power to let you add this information! A super-association table of sorts.



Follow Model

Now that you're familiar with all the details of association tables, go ahead and start getting your hands dirty with some code. Type this out in `models.py`:

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer,
                            db.ForeignKey('users.id'),
                            primary_key=True)
    following_id = db.Column(db.Integer,
                             db.ForeignKey('users.id'),
                             primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

Different versions of the word "follow" can get confusing, so to clarify: `follower_id` is the ID of the user who follows another, and `following_id` is the user who is followed. The *follower* user is *following* the second user.

Now that that's out of the way, the code shouldn't be too surprising, as you have already made models with certain columns being primary keys and others having foreign keys. What you probably *haven't* seen is both of them together. What gives here? Of course, both foreign keys must be users because, thus far, users can't follow anyone else in your web app. (And if you did have something like that, you'd need another association table.) By setting `primary_key` to `True` for both columns, both foreign keys *together* form the primary key. That means the *pair* of IDs is the key.

Finally, the last column, `timestamp`, is simply the time the *follower* started *following* the other. Each row that is created in this table automatically sets `timestamp` to the time the row is inserted.

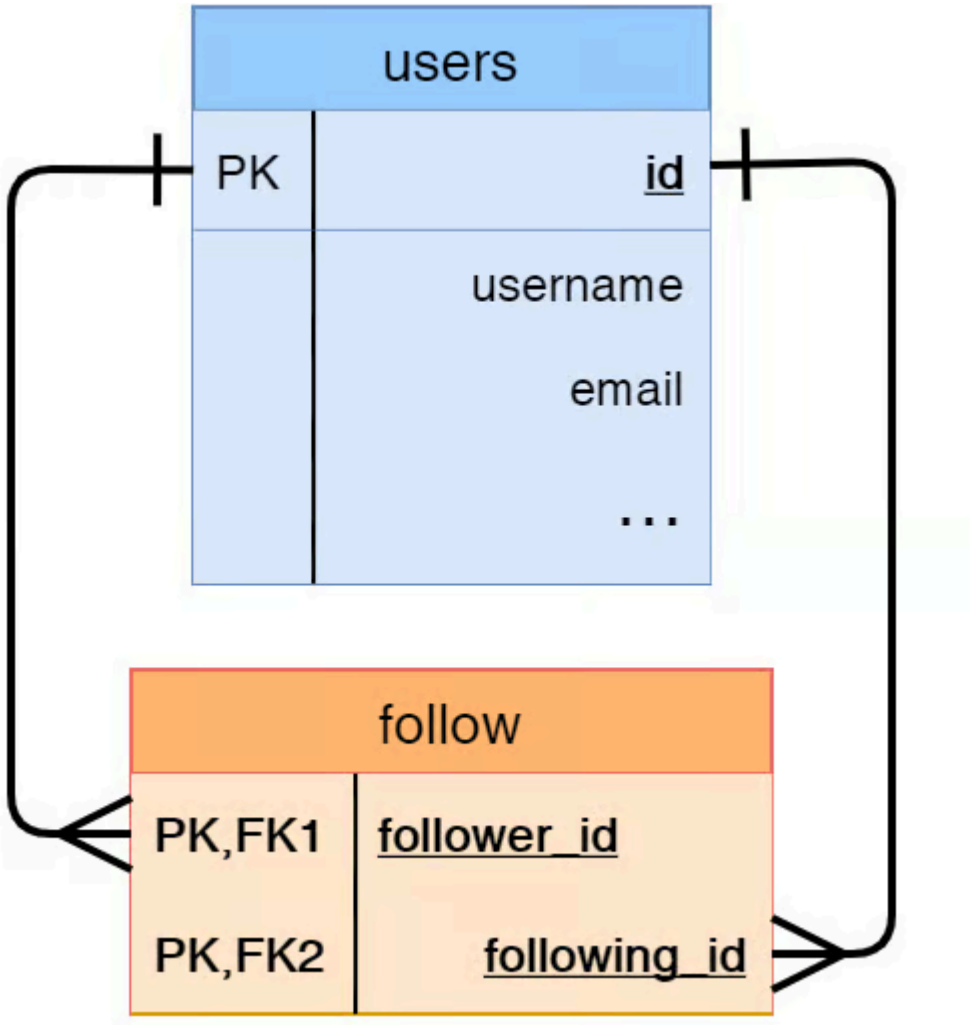
Associating The Association

Great, now you have your souped up association table. It's time to give it something to associate *with*, and that something is the `users` table.

Your `User` model will need to connect to the `Follow` model:

```
class User(UserMixin, db.Model):
    # ...
    following = db.relationship('Follow',
                                foreign_keys=[Follow.follower_id],
                                backref=db.backref('follower', lazy='
                                lazy='dynamic',
                                cascade='all, delete-orphan')
    followers = db.relationship('Follow',
                                 foreign_keys=[Follow.following_id],
                                 backref=db.backref('following', lazy
                                 lazy='dynamic',
                                 cascade='all, delete-orphan')
```

With this, you create the two one-to-many relationships.



`foreign_keys` : `foreign_keys` keyword argument must be specified since there are two foreign keys in the association table. Without it, SQLAlchemy wouldn't know which foreign key to use as it would be too ambiguous.

`backref` : Next is the `backref` argument which is a reference *back* to the `User` *from* the `Follow` model. If you remember the discussion about relationships back in the Database Management section, `backref` adds an attribute to the model given in the first argument of `db.relationship()` . So, from an instance of `Follow` , you can refer to `follower` as the `User` who follows the other. You can refer to `following` as the `User` being followed. So you can get the "one" sides, all because of the work done by `backref` . With `User` , you just need to use `following` and `followers` to get you the list of other users that the user follows or who follows them.

`joined` **lazy mode:** But what's with the `db.backref()` or the extra `lazy` argument? Well, the `db.backref()` is just a way to define extra arguments for the `backref` argument in `db.relationship()` . The `joined` lazy mode causes any related `Follow` objects to be loaded immediately from the *join* query. Say a user, George, has 10 followers. If you have George's `User` object as `george` , calling

`george.followers.all()` gives a list of 10 `Follow` instances. Each of those instances has the `follower` and `following` back reference attributes set to the corresponding users. With `joined`, all this happens in a single database query. You can get all of the information you need loaded and ready to go just from the `george.followers.all()`. The default `select` lazy mode would instead require 11 total database queries! Ten of them would be needed to query the individual users in each `Follow` instance.

`dynamic` **lazy mode:** As for the second `lazy` argument on the `User` side, it is set to the `dynamic` lazy mode similar to the relationship between `Role` and `Users`. The relationship attributes, in this case, return a query object so that additional filters can be added, instead of just returning the items directly.

`cascade`**:** The `cascade` argument changes how actions are propagated from a parent to related objects. Usually, the default cascade options are appropriate for most cases, but in this case, you'll want to change the default for the case when any `Users` are deleted from the database. By default, when a `User` gets deleted, any values in the `follows` table that match that User are set to a null value. Instead, the `delete-orphan` is added so that all entries that match the deleted user in the `follows` table are *also* deleted, destroying the link.

Info: The `cascade` argument takes a comma-separated list of options. The `all` option represents all cascade options except `delete-orphan`.

Python Helper Methods

Phew, now that that's all done, you'll want to define helper methods. These will be defined in the `User` model so that you can more easily handle database operations with followers and those they follow:

```
class User(UserMixin, db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, following=user)
            db.session.add(f)

    def unfollow(self, user):
```

```
        f = self.following.filter_by(following_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
        if user.id is None:
            return False
        return self.following.filter_by(
            following_id=user.id).first() is not None

    def is_a_follower(self, user):
        if user.id is None:
            return False
        return self.followers.filter_by(
            follower_id=user.id).first() is not None
```

The first method, `follow()`, performs the "follow" action accordingly in the database, in which a new row is inserted in the `follows` table linking a user to the passed-in `user`. The `unfollow()` method does the reverse: it will delete the row pertaining to the user that is to be unfollowed.

Two things to keep in mind here. First, the `follow()` method does not need to add `timestamp` when creating the new `Follow` instance. That's because it's already set to the current date and time by default. Second, neither the `follow()` and `unfollow()` methods have a `db.session.commit()` because there might be other operations that need to be done alongside a user following or unfollowing another.

Next is `is_following()`, and as you probably guessed, it determines if the user is following the specified user. And `is_a_follower` determines if another user is a follower. Both make sure the `user` has been assigned an `id` before querying for it in the database, as that would cause an exception.

You are well on your way to getting followers following all sorts of followed users.

While you might have the models, relationships, and helper methods in place to allow users to follow others, you still need to build the view functions and templates to give your users the means to do it. In this lesson, you'll define the various view functions needed for following, unfollowing, and everything in between.

(Un)Following Other Users

Your view functions are what actually allow your users to do things on your webapp, and that includes following other users. The `follow()` view function is defined below:

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash("That is not a valid user.")
        return redirect(url_for('.home'))
    if current_user.is_following(user):
        flash("Looks like you are already following that user.")
        return redirect(url_for('.user', username=username))
```

```
        current_user.follow(user)
        db.session.commit()
        flash(f"You are now following {username}")
        return redirect(url_for('.user', username=username))
```

The requested user to follow, specified in the URL, is loaded and verified 1) to exist and 2) they are already being followed. Given those checks pass, the user is followed and the session is finally committed to the database.

Task: Implement the `unfollow()` view function. This is very similar to the `follow()` view function.

View Functions To Display Users

It's also useful to display the list of users that follow a particular user. The view function to get and paginate that list of users is shown below:

```
# Who my followers are
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash("That is not a valid user.")
        return redirect(url_for('.home'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        page,
        per_page=current_app.config['RAGTIME_FOLLOWERS_PER_PAGE'],
        error_out=False)
    # convert to only follower and timestamp
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
    return render_template('followers.html',
                           user=user,
                           title="Followers of",
                           endpoint='.followers',
```

```
                pagination=pagination,
                follows=follows)
```

First thing to do is to get the user in question. If the user doesn't exist, the user is told so through a notification. A pagination object is then created from the user's followers. Since the query for followers returns a list of `Follow` instances, only the follower users are needed as you already know they follow the user in question. Another list is created instead that gives only the follower users and the timestamp to keep rendering simple.

Task: Create a `following()` view function. This is similar to the `followers()` view function, except you will show the users a particular user follows using the `user.following` relationship. It will also pass variables to the `followers.html` file.

You are so close to having a real social media app! Especially one where users can, y'know, be social and keep up with those they follow.

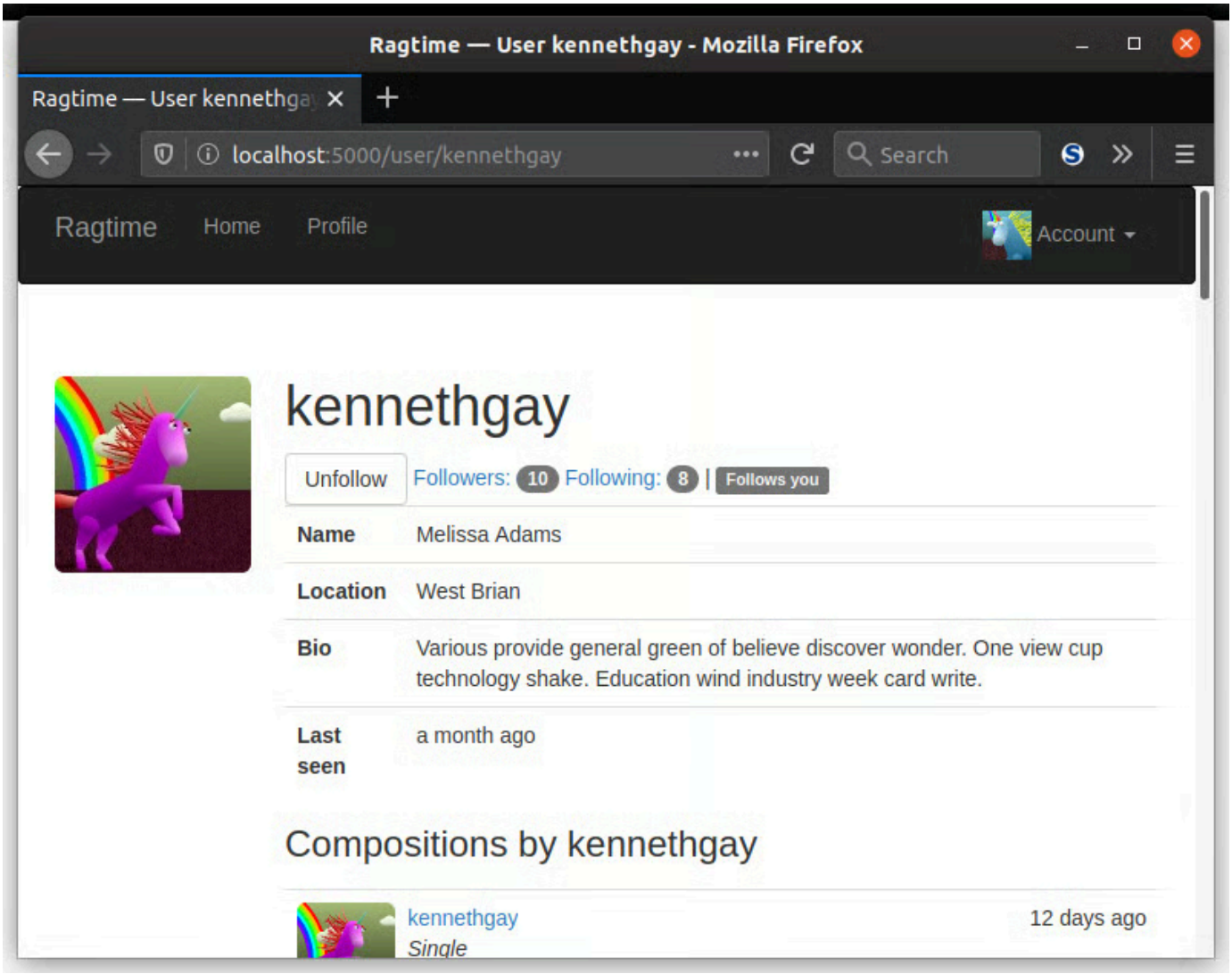
Of course, what comes with view functions are always templates, whether they are rendering the contents of a view function or providing links for another. This lesson tasks you with defining buttons for following and unfollowing, as well as links to see followers and those they are following.

Add To User Template

You are tasked with adding to the user template the following:

- Follow button: This button will only show if the user has permission to follow others and if the user is not the current user. Have this link to your `follow()` view function. It should display "unfollow" if the current user already follows the user.
- Show how many followers the displayed user has. You can use the Bootstrap `badge` class on a `<span>` tag to show it prettily. This should also link to the `followers()` view function.
- Show how many users the displayed user is following. This should link to the `following()` view function.
- If the displayed user is not the current user, show "Follows you". You can use the Bootstrap `label` class on a `<span>` tag to show it prettily.

Once you're done, it will look something like this:



Showing Followers and Following

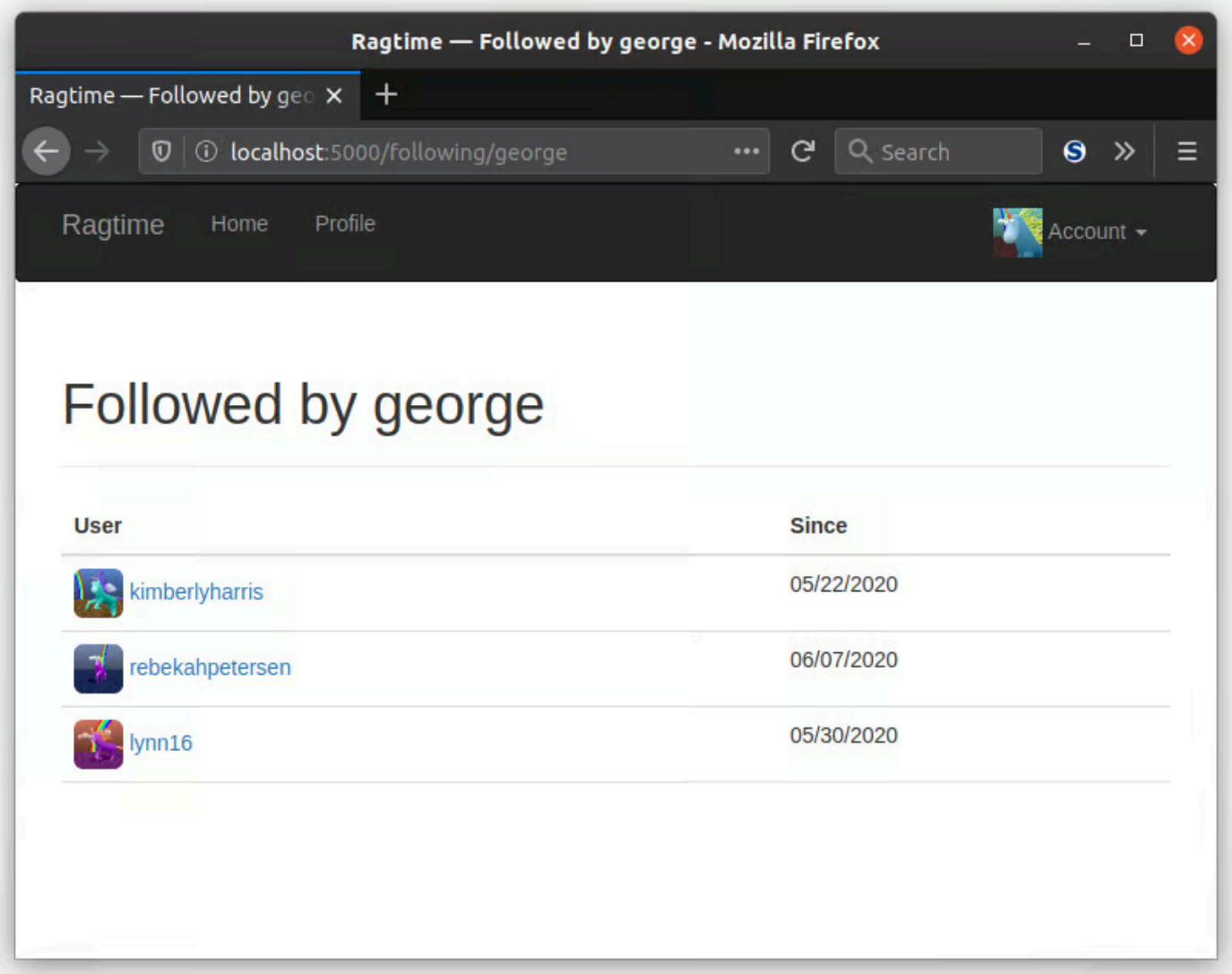
Once you get the follower information on the user page, the next thing to do is to show users who are followed by a user and who are following a user. This can all be done with the `followers.html` template. You referenced this template in your `followers()` and `following()` view functions.

- You can implement this as a two column table, with the first column showing the username of the user and their avatar. The second column will show the `timestamp`.
    - This table can use the Bootstrap `table`, `table-hover`, and custom `followers` class.

```
.table.followers tr {
    border-bottom: 1px solid #e0e0e0;
}
```

- You'll need to show the pagination widget at the bottom of the page.

The page will look something like this:



(note: there's no widget shown in the image because it's just one page)

Summary: Implementing User Following with Many-to-Many Relationships

In this lesson, you:

- Learned that a **many-to-many** relationship is when multiple records on one side are related to multiple records on the other.
- Grasped that to resolve many-to-many relationships, you need to create an **association table** with foreign keys from both related tables.
- Discovered how to represent many-to-many relationships with **self-referential** relationships when the same entity type is on both sides (such as users).
- Focused on how to define association tables and relationships in SQLAlchemy, including **primary key** and **foreign key** setup, and using `relationship()`, `backref`, and `lazy` loading strategies.

- Acquired the knowledge to set up helper methods like `follow()`, `unfollow()`, `is_following()`, and `is_follower()` for better abstraction and encapsulation of the follower functionality.
- Implemented the templates needed to render the new follow functionality.

Previous Next → Video: Association Table in SQLAlchemy