CODING NOMADS

14) User Roles Lesson

# Assign and Verify User Roles in Flask

12 min to complete · By Brandon Gigous

## Contents



Your changes in the last lesson will ultimately help you bring roles to life. In this lesson, you *shall* bring them to life! You'll be assigning and verifying roles like no tomorrow, making sure your users are doing what they are allowed and no more.

## Role Assignment

Whenever a `User` pops into existence is the perfect time to give them their starting role. Assigning the role any later might result in some hard-to-catch bugs, and there's no reason not to do it that soon anyway. That's because you have everything you need to know at the point of `User` creation to be able to give that user a role. Nearly all users who register are to be given the `"User"` role, but if a user's email matches the administrator's email, defined previously in the `RAGTIME_ADMIN` configuration key, then that user obviously gets the `"Administrator"` role.

When does a `User` get created? When the constructor is called, of course! So get to upgrading the `User` model by pulling up `models.py` again:

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        if self.role is None:
            if self.email == current_app.config['RAGTIME_ADMIN']:
                self.role = Role.query.filter_by(name='Administrator
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first
```

Notice the clever use of the database query for the `default` role. That is, if the administrator email didn't match what the user put in.

# Role Verification

To make a little easier on yourself, and your codebase, you can add some helper methods to make sure your users *can* do something. Meaning, do they have permission to moderate another comment, or to publish a composition, or to eat green eggs and ham? You can also define another function that's just for the `ADMIN`, as this permission will be checked often later:

```
from flask_login import AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...
    def can(self, perm):
        return self.role is not None and self.role.has_permission(pe

    def is_administrator(self):
        return self.can(Permission.ADMIN)


class AnonymousUser(AnonymousUserMixin):
```

```
        def can(self, perm):
            return False

        def is_administrator(self):
            return False

    login_manager.anonymous_user = AnonymousUser
```

These methods simply use the `Role`'s new `has_permission()` method you made earlier. The `can()` method is convenient for checking that a user has a given permission so that you can quickly check whether or not they can access a particular resource or perform a task.

You can also define a custom `AnonymousUser` class that defines the same methods to prevent any Python `NameErrors` from popping up. This class inherits from Flask-Login's `AnonymousUserMixin` so that these methods can be found whenever a user is anonymous. You must set the `login_manager.anonymous_user` attribute to have it use your custom class. No need for the user to log in first! Permissions can still be checked, but they always return `False`.


## Mentorship Makes the Difference!

# Custom Decorators For View Functions

You learned in the user authentication section that the `login_required` decorator can be used to prevent users from accessing view functions if they haven't logged in yet. Well, you can also do a similar thing with permissions. And what's more fun than making custom decorators? (Yes, tire swings win again...)

If you need to brush up on making decorators, now would be a good time. Make a new file, `app/decorators.py` , and let's have some fun:

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import Permission

def permission_required(permission):
    def decorator(f):
```

```
            @wraps(f)
            def decorated_function(*args, **kwargs):
                if not current_user.can(permission):
                    abort(403)
                return f(*args, **kwargs)
            return decorated_function
        return decorator

    def admin_required(f):
        return permission_required(Permission.ADMIN)(f)
```

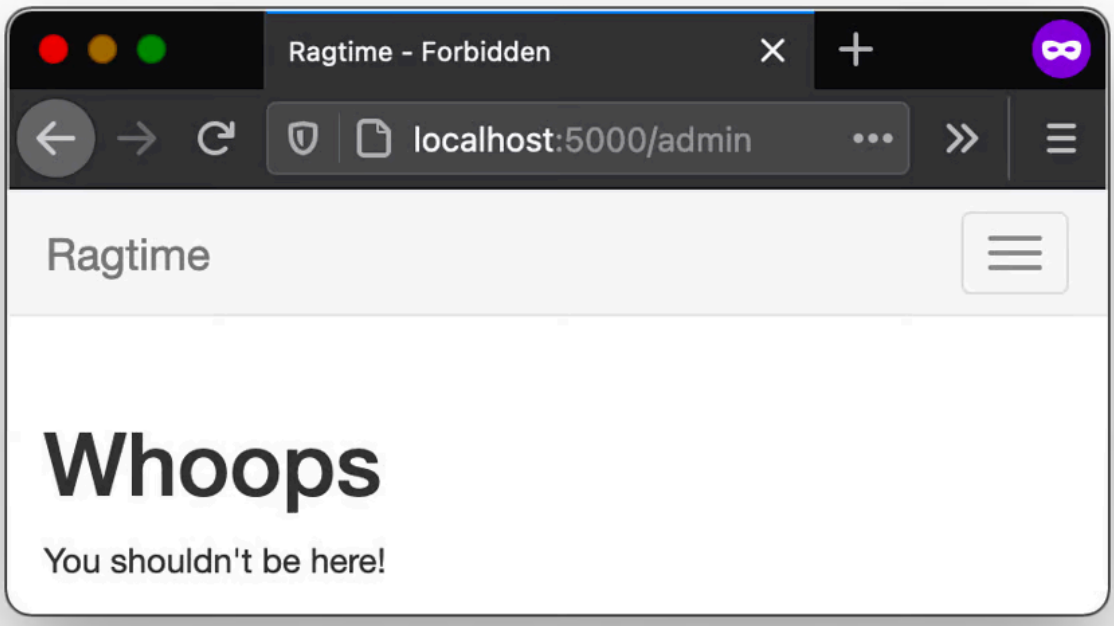
The `permission_required` decorator is a generic permissions check: the view function can do what it does *if the user has the specificed permission*. Otherwise, an abort is returned as the response, specifically with a 403 Forbidden status code. The `admin_required` decorator piggy-backs off the other decorator, and only checks if the user has the `ADMIN` permission.

"That's great," you're probably thinking, "but how exactly do I use them?" Ah yes, of course. To apply decorators for view functions in the correct order, refer to the following examples:

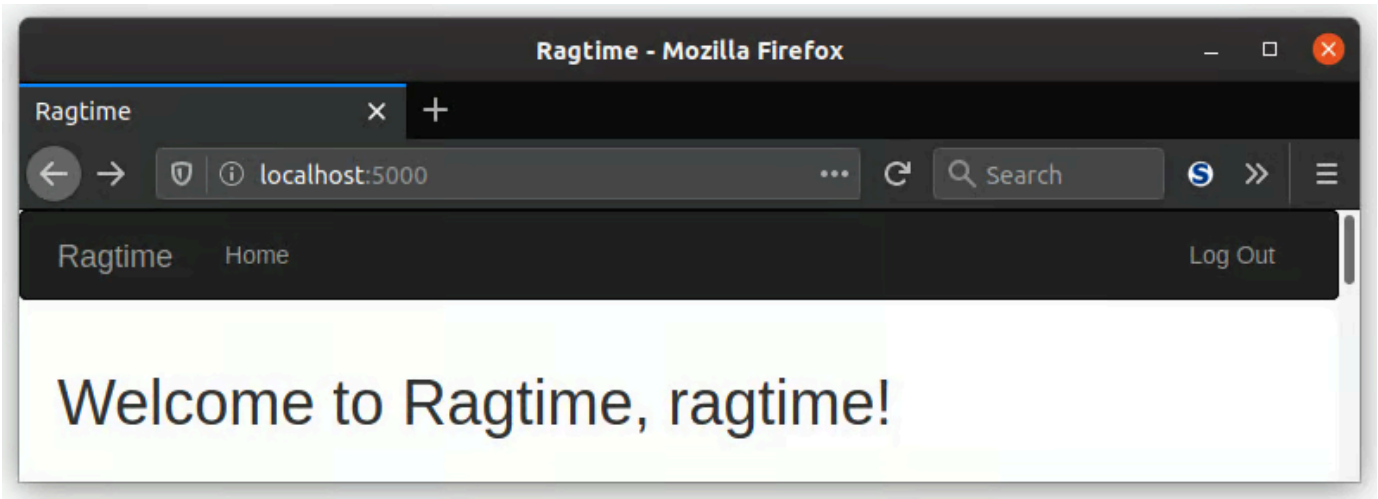
```
from .decorators import admin_required, permission_required

# route() comes first
# then check if user authenticated
# then check their permission
# The topmost decorators are "evaluated" before the others

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return "Welcome, administrator!"


@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
```

```
    def for_moderators_only():
        return "Greetings, moderator!"
```

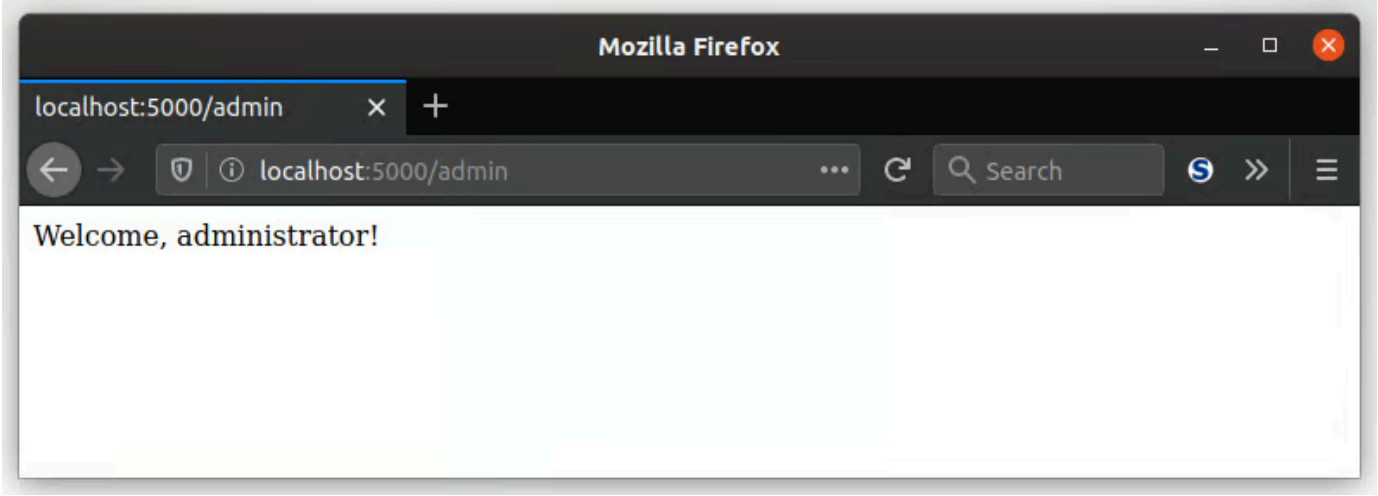
To demonstrate, try logging out or logging in as a regular ol' user. Then try to access the admin page at `localhost:5000/admin`. You'll see something like this:




Now log in with your admin account, the one that has the administrator's email. Register it first if you need to.




Then, try accessing the admin page, and you will have access!




# Injecting Permissions

While that heading might sound like someone can acquire certain permissions through a medical procedure, that's not what's meant here. Not people or users, but templates. The reason? First, a bit of background. Although your users can be prevented from seeing certain pages entirely via your new decorators, sometimes you want them to see the page but not everything in it. Not certain buttons or perhaps your users see different text depending on what permissions they have. You won't want to put in the literal permission values just to check permissions. What if they change?!

Rather, you can **inject** your `Permissions` class into your templates globally. Wait, what? You already know that within your templates, you already have access to things like `current_app` and `current_user`. How did they get there? They were *injected* automatically by Flask and Flask-Login, respectively. You can do the same for your own custom classes and variables by using the Flask `app_context_processor` decorator:

```
# in app/main/__init__.py

# ...
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
# ...
```

You'll notice it's also pretty similar to your `shell_context_processor` decorated function! Now, all templates globally have access to the `Permission` class, so you check user permissions much more easily.

# Unit Tests

**Optional Task:** Write unit tests that test the following:

- That your permission helper functions work
- That your roles' names are successfully assigned after `Role.insert_roles()` is called
- That the User, Mod, and Admin roles each have the correct permissions
- That a new user has the User role automatically assigned by default



# ALMOST DONE. Before you go...

Don't forget to insert your new roles and give your existing users their roles! This is an exercise for the reader. ;)

**Note:** And *definitely* don't forget to perform another database migration!

Welp, that's it. You've just gotten through user roles! You are given permission to proceed to the next section, which is about fleshing out your users' profile pages.

# Summary: How to Assign a User Role

- The best time to assign a User a role is when the user is created

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        if self.role is None:
            if self.email == current_app.config['RAGTIME_ADMIN']:
                self.role = Role.query.filter_by(name='Administra
            if self.role is None:
                self.role = Role.query.filter_by(default=True).fi
```

- Add some helper methods, `can()` and `is_administrator()` so your users can do something
- The `permission_required` decorator is a generic permissions check.The `admin_required` decorator piggy-backs off the other decorator, and only checks if the user has the `ADMIN` permission.
- You can **inject** your `Permissions` class into your templates globally in order for a user to see parts of a page instead of the whole thing.