# Flask Starter Lab™ - v2.1 (Suits Level 3 NCEA)

*2019-2021 - Steve Dunford*
*dfs@burnside.school.nz*

## Requirements:

- Python 3.something
- Your IDE of choice (anything from Notepad++ to Eclipse is fine)
- SQLiteStudio (or your preferred SQLite editor)
- The following packages (PIP install these)
    - **flask** (the main Flask library)
    - **flask-sqlalchemy** (the database mapper - makes SQL easy)
    - **flask-wtf** (the Python HTML forms library for Flask)

## What is Flask?

Flask is a micro web framework for Python. In essence, a way to make dynamic websites and web apps. Django is probably the best-known Python-based web framework, Flask is probably second**. While they are quite similar, they are fundamentally different in the way they approach the problem.

Django is the proverbial sledgehammer to crack a walnut… while Flask is a small pocket-knife that you can easily extend with a nutcracker attachment if you need to.

## Can you build large sites with it?

https://www.vintageradio.co.nz is the largest one I've built personally - thousands of lines of code, over 1100 radios, around 500 chassis', growing daily. Possibly slightly larger than this is Pintrest which allegedly moved from Django to Flask. LinkedIn also reportedly uses Flask.

## How do you host a Flask app?

I recommend PythonAnywhere - They provide free basic hosting for Flask, and they're scalable and affordable - and they give great personal service when you need it. Best of all, they didn't pay me a cent to say that - I recommend them simply because they've been awesome for me.
*#not-sponsored-at-all*

Flask will also run in www.repl.it but it's not always stable, or responsive, and often sqlite3 databases will self-destruct for no obvious reason at all and need to be replaced with a good copy. It's a handy place to start building Flask though, since its all online and has a fairly good basic IDE.

*** probably*

# Part 1 - Flask Hello World

To create a simple Flask app, we only need one file - let's call it **routes.py** because that's predominantly what it will contain.

```
from flask import Flask

app = Flask(__name__)


# basic route
@app.route('/')
def root():
    return ('<h1 style="color:red";>Hello World</h1>')


if __name__ == "__main__":
    app.run(debug=True, host="127.0.0.1", port=5000)
```

In repl.it the filename will be **main.py** due to their system setup - and the app.run line needs to be: `app.run(debug=True, host="0.0.0.0", port=8080)`

And so now we have a flask app.  Running that file (`python routes.py` from the command line in the folder you saved the file in will do this) will start Flask's internal web server and give you the address to browse to, which should be http://127.0.0.1:5000 (alternatively just click the run button if you're using repl.it).

## Flask Hello World - Better?

To extend this simple web app to use templates, which will (eventually) allow us to unleash the power of Jinja2 (a very awesome templating language that allows us to build better webpages), we use `render_template()` - and so our modified basic route will now look like this:

```
# better basic route
@app.route('/')
def root():
    return render_template('home.html')
```

We also need to import render_template - its part of the Flask package, so just add it on to the first line:

```
from flask import Flask, render_template
```

**render_template** takes HTML, or snippets of HTML, and creates a finished HTML document which is presented to the browser. In this case, our HTML is going to be in home.html. By default this needs to be in a folder called templates, so create that:

**home.html**

```
<h1 style="color:green";>Hello World</h1>
```

Our file system should now look like this:

```
routes.py
templates
 ├ home.html
```

You should see that aside from the 'Hello World' changing from red to green, nothing else has changed - so why bother? Well, we're just dealing with simple text here but think about a full-blown website - even a simple static site with, for example, just three pages: you have to duplicate the header, footer and nav bar three times in three different files and the only content to change is, well, the content. Scale this up to a larger site and you have a maintenance nightmare… imagine changing a site-wide link in the navigation if you have 200 pages or so! But templates solve this nightmare for us. Let's create a few more files and implement our site in small slices:

```
routes.py
templates
 ├ home.html
 ├ layout.html
 ├ header.html
 ├ nav.html
 ├ footer.html
```

And each file can be pretty simple for now:

**header.html**:

```
<header>
 <h1>MINIimdb</h1>
</header>
```

**footer.html**:

```
<footer>
 <hr>
 <h2>&copy; 2019 SuperMegaPyConCorp</h2>
</footer>
```

**nav.html**:
```
<nav>
 <hr>
 <a href="/">Home</a>
 <a href="/all_movies">List All Movies</a>
 <hr>
</nav>
```

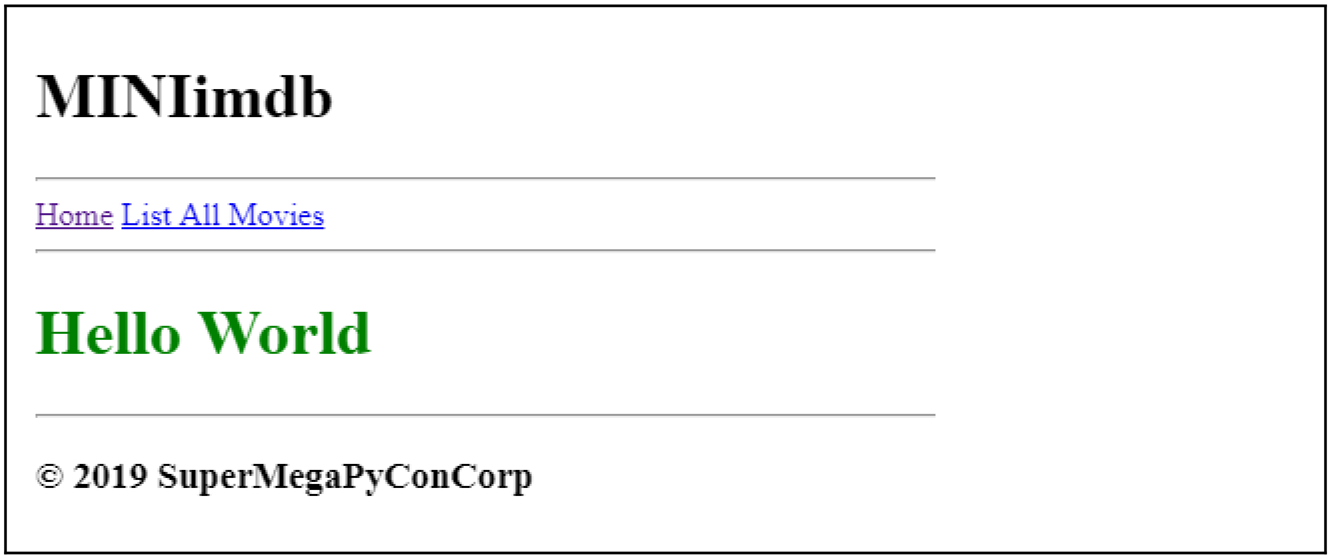
And the important one that pulls them all together - **layout.html** which is where most of the traditional HTML goes.  This file uses the **include** command to pull in all the other sections we just made (and a space for some content):

**layout.html**
```
<!doctype html>
<html>
 <head>
   <title>Example Site - {{page_title}}</title>
 </head>
 <body>
   {% include 'header.html' %}
   {% include 'nav.html' %}
   {% block content %}{% endblock %}
   {% include 'footer.html' %}
 </body>
</html>
```

There are two things we've done here that won't do anything yet - the link to **'/all_movies'** in nav.html, and the **{{ page title }}** in the <title> tag… we'll get to those shortly…

Finally, our home.html file will need a couple of extra lines to take advantage of all this:
```
{% extends 'layout.html' %}
{% block content %}

<h1 style="color:green";>Hello World</h1>

{% endblock %}
```

The site should now work, and look amazing!


MINIimdb

Home List All Movies

Hello World

© 2019 SuperMegaPyConCorp

Ok, so maybe not amazing - but we've now set the stage for a site that is easy to maintain in terms of layout - since the header, nav and footer sections are only created once.

To create any new page in the site now you simply need a file with `{% extends 'layout.html' %}` at the top to have it call layout.html, and `{% block content %}` next to insert itself into the correct part of the layout, then the last line is `{% endblock %}` (required in the same way you open and close HTML tags). Then **render_template('your_new_file.html')** will do the magic.

You've also seen a couple of Jinja2 features used here - *{% %}* and *{{ }}*.

**{% %}** is the command wrapper in Jinja 2 - all 'code' commands get wrapped with this.

**{{ }}** is the variable wrapper - any variable inside this will have its contents displayed.

**{# #}** this one isn't being used here, but it's for comments, and unlike `<!-- HTML comments -->` they don't show up in your rendered web page, which can be very handy.

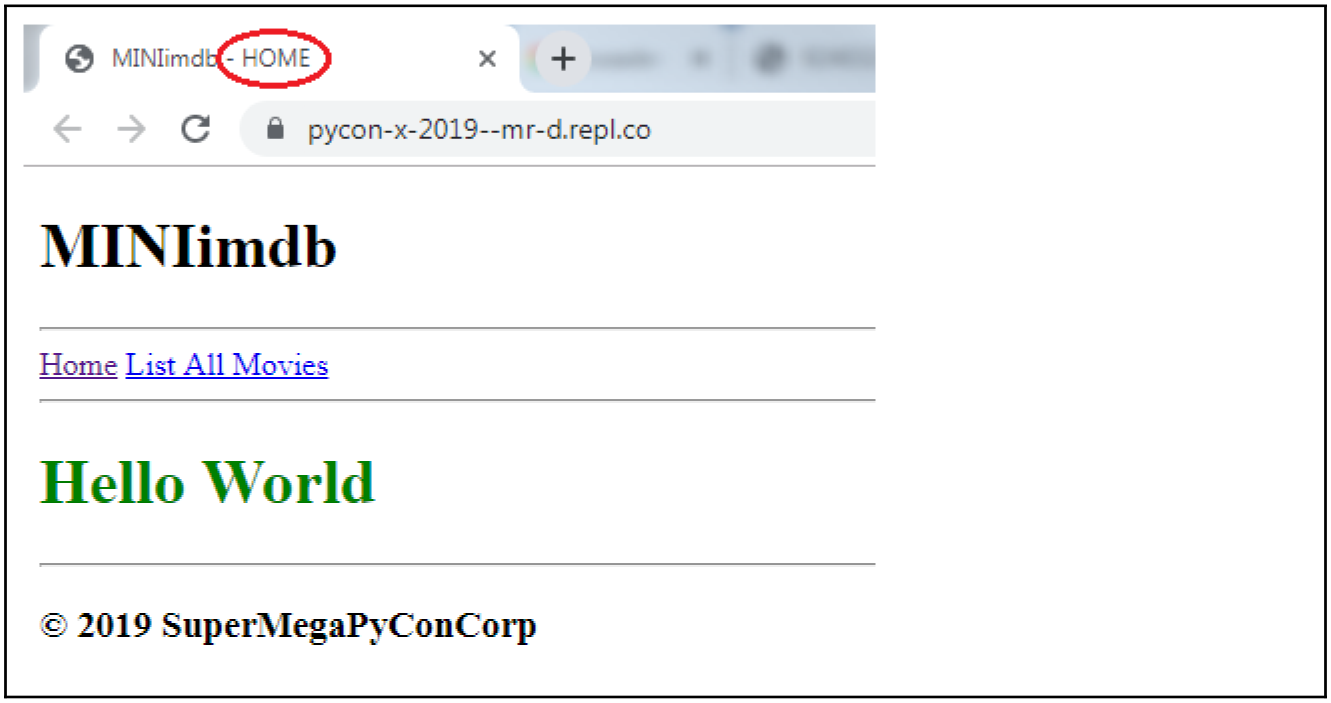
Let's experiment with a variable. We're set up with one in the title tag in layout.html, so how do we give it a value? It just needs to go in with the template being rendered, so change the line:

```
    return render_template('home.html')
```

To this:

```
    return render_template('home.html', page_title='HOME')
```

Now the tab should show the title we've set:



MINIimdb

Home List All Movies

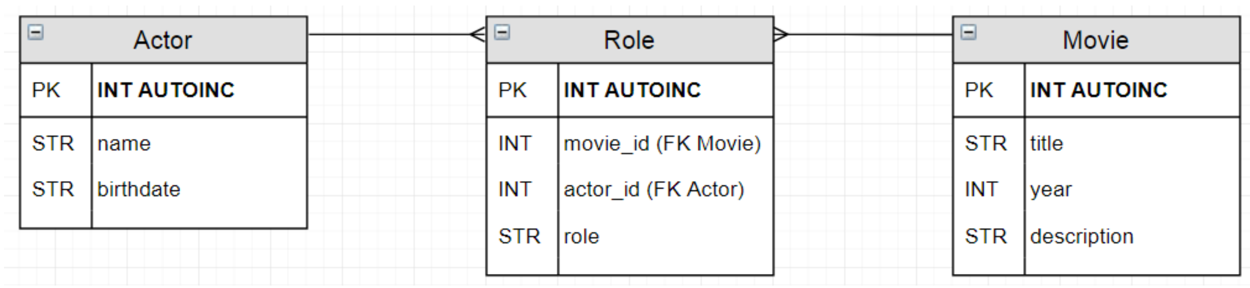
Hello World

© 2019 SuperMegaPyConCorp

Now we have a basic site which works.  It doesn’t do anything, but it works.  Welcome to Flask!

## Part 2 - The Database

Flask imposes no rules about what database you have to use - in this instance we're going with SQLite3, because its a nice easy single-file solution.  The example database has three tables:



The role table is an association table to get around the many-to-many relationship between Actors and Movies (one actor can be in many movies, and one movie can contain many actors).

As you can see, it's not really a mini-IMDb… but it wants to be when it grows up.

You can use raw SQL queries in Flask quite happily, but we're going to go one better and use SQLAlchemy, an Object Relational Mapper that sits between our code and the database.  To do this we'll need another file, models.py, that describes the mapping:

**models.py**

```
from main import db

class Movie(db.Model):
    __tablename__ = 'Movie'

    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    year = db.Column(db.Integer)
    description = db.Column(db.String())


class Actor(db.Model):
    __tablename__ = 'Actor'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable = False)
    birthdate = db.Column(db.String(30))
```

```
class Role(db.Model):
    __tablename__ = 'Role'

    id = db.Column(db.Integer, primary_key=True)
    movie_id = db.Column(db.Integer, db.ForeignKey('movie.id'),
nullable = False)
    actor_id = db.Column(db.Integer, db.ForeignKey('actor.id'),
nullable = False)
    role = db.Column(db.String(80), nullable = False)
```

This sets up the mapping, now we just need to make use of it, so routes.py (or main.py on repl.it) is going to need some changes to make use of it:

```
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///movies.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# need db to import models
import models


# home route
@app.route('/')
def home():
    return render_template('home.html', page_title='IT WORKS!')


if __name__ == "__main__":
    app.run(debug=True, host="127.0.0.1", port=5000)
```

It's worth noting at this point that cramming everything into routes.py is a very simplistic approach, and there are far better ways to lay out a Flask app - it's certainly not scalable to do it this way.  But, baby steps.

Now we can add a new route with a simple query in it to test this configuration
```
# list all the movies
@app.route('/all_movies')
def all_movies():
    movies = models.Movie.query.all()
    return render_template('all_movies.html', page_title="ALL
MOVIES", movies=movies)
```

As a comparison, If we were doing straight SQL queries without SQLAlchemy then it would be like this:
```
# list all the movies
@app.route('/all_movies')
def all_movies():
    conn = sqlite3.connect('movies.db')
    cur = conn.cursor()
    cur.execute("SELECT * FROM Movie;")
    results = cur.fetchall()
    return render_template("all_movies.html", page_title="ALL
MOVIES", results=results)
```

Straight SQL is more complex to query, but virtually no configuration is required - you just need `import sqlite3` - however, the results come back as a list of tuples, not objects - making them more awkward to access (eg: **actor[1]** rather than **actor.name**).

Anyway, back to the **/all_movies** route - you'll see that it's trying to render a template based on all_movies.html, so we'd better create that:

**all_movies.html**
```
{% extends 'layout.html' %}
{% block content %}

{% for movie in movies %}
    <p>{{ movie.title }}</p>
{% endfor %}


{% endblock %}
```
That's a really basic loop to show the names of each movie in the database - note that Jinja2 loops (and if statements for that matter) have to be closed with an **endfor** (or an **endif**).

Sometimes it can be useful to have a loop counter, similar to using enumerate in Python - try changing the movie.title line as follows:
```
<p>{{ loop.index }} - {{ movie.title }}</p>
```

Note: If you need a zero-indexed counter, use `{{ loop.index0 }}`

# Part 3 - A Dynamic Route

The obvious next step is a details page for individual movies, so let's look at how we do that.  First, let’s modify the **all_movies.html** file to link each movie:

**All_movies.html**

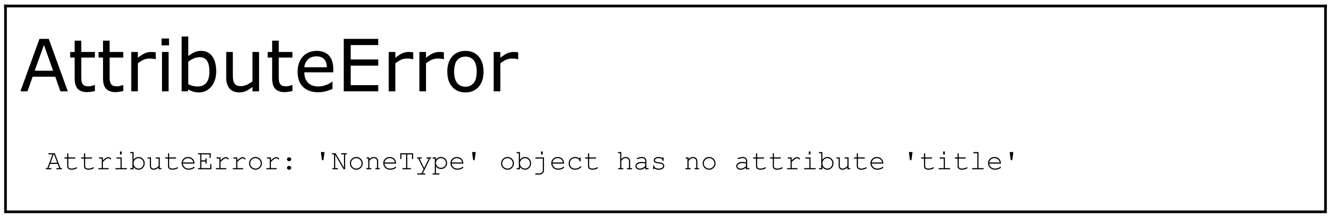
```
{% extends 'layout.html' %}
{% block content %}

{% for movie in movies %}
    <p><a href="/movie/{{ movie.id }}">{{ movie.title }}</a></p>
{% endfor %}

{% endblock %}
```

This gives a series of links which match the pattern of **/movie/<int>** which Flask can capture.  Here's how that looks:

```
# details of one movie
@app.route('/movie/<int:id>')
def movie(id):
    movie = models.Movie.query.filter_by(id=id).first()
    title = movie.title
    return render_template('movie.html', page_title=title,
movie=movie)
```

The route **/movie/<int:id>** means it will only match if an int is used… trying to go to /movie/bob for example, will be ignored by this route.  The int then needs to be passed into the function as an argument (**def movie(id):**)

Then we do a query on the movie table, but filter it by the id value, and request only one result by calling **.first()** - although there is an issue here that we’re not dealing with - what if there is no movie with the id value given?  The movie variable will be None, and the view will crash.



AttributeError

```
AttributeError: 'NoneType' object has no attribute 'title'
```

A test could be set up to catch that and do something about it, or you could just call it a 404 - fortunately SQLAlchemy has a tidy way to handle that - **.first_or_404()**

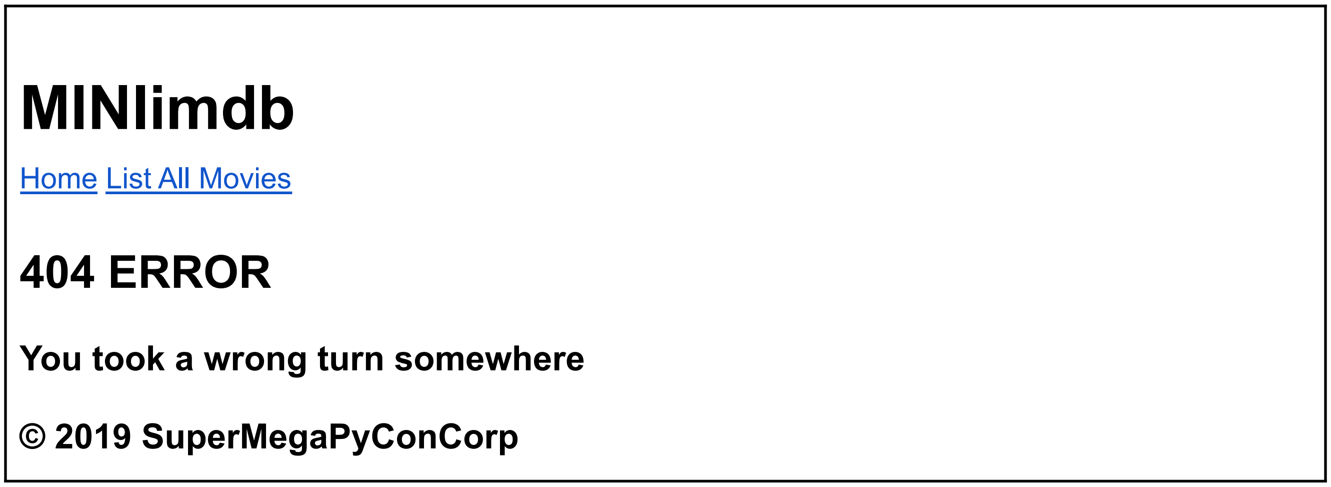
To handle 404's, Flask has a built-in route we can use:

```
@app.errorhandler(404)
def page_not_found(e):
   return render_template("404.html")
```

Now we just need a simple page to display when this happens:
**404.html**

```
{% extends 'layout.html' %}
{% block content %}

<h2>404 ERROR</h2>
<h3>You took a wrong turn somewhere</h3>

{% endblock %}
```

And test it with a non-existent movie:
**/movie/100101001**


MINIimdb

Home List All Movies

404 ERROR

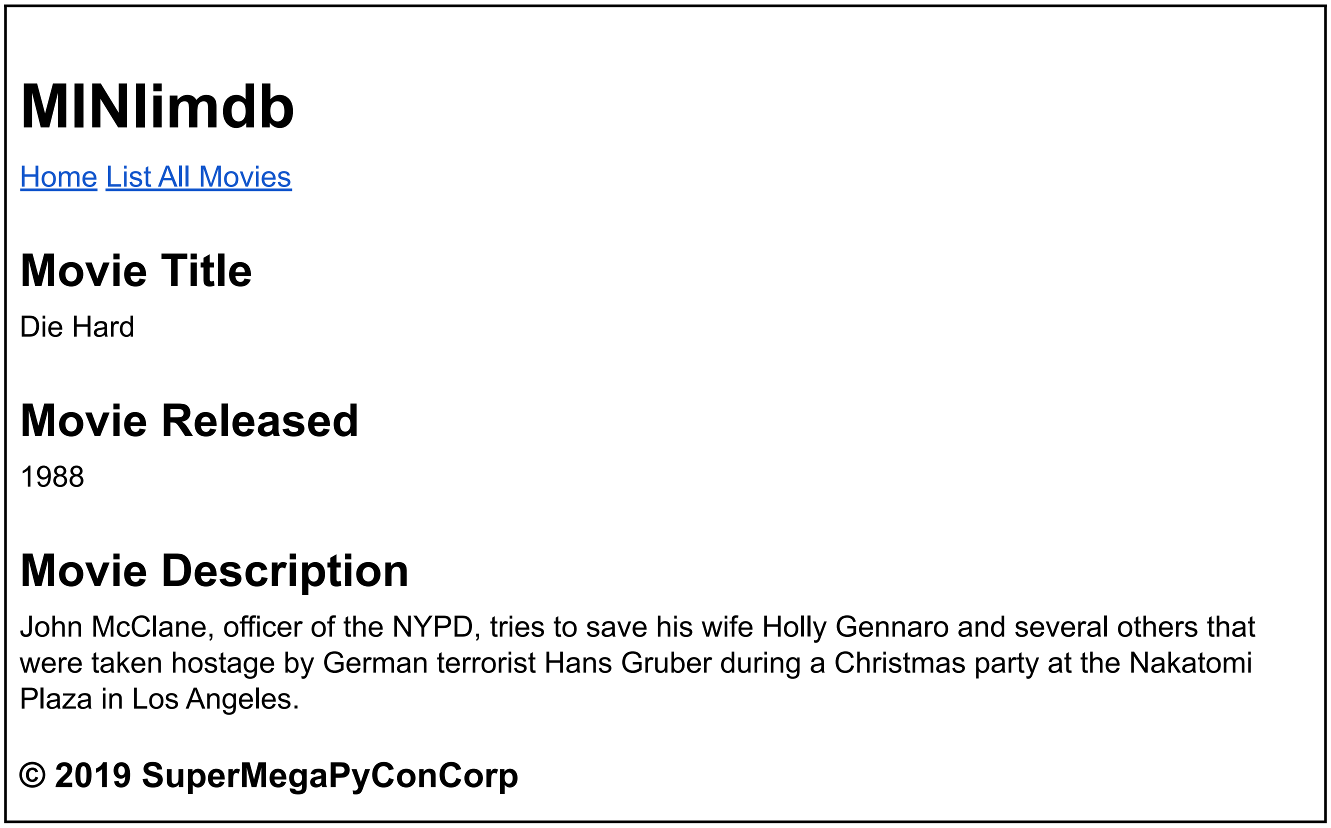
You took a wrong turn somewhere

© 2019 SuperMegaPyConCorp

So it's still pretty ugly, but this isn't about CSS - it's about Flask… The CSS is your job…

Of course, we got caught up in the what-ifs and never checked what happens when we look at a legitimate movie:

**/movie/1**



# MINIimdb

Home List All Movies

## Movie Title

Die Hard

## Movie Released

1988

## Movie Description

John McClane, officer of the NYPD, tries to save his wife Holly Gennaro and several others that were taken hostage by German terrorist Hans Gruber during a Christmas party at the Nakatomi Plaza in Los Angeles.

**© 2019 SuperMegaPyConCorp**

You now have the know-how to do the same thing for Actors.

But what about getting a list of actors for a given movie? In traditional SQL this would need a join - in SQLAlchemy its called a relationship - something we set up once and can use whenever we need it. Head over to models.py and make the following changes:

**Movie class**:
Tack the following line on the end, under description

```
    actors = db.relationship('Role', back_populates='movie')
```

**Actor class**:
Tack the following line on the end, under birthdate:

```
    movies = db.relationship('Role', back_populates='actor')
```

Then add a __str__ function under that so we can get the Actors name easily by simply viewing the actor object.

```
def __str__(self):
    return self.name
```

**Role class**:
Add this on the bottom of the role class under role

```
    actor = db.relationship('Actor', back_populates='movies')
    movie = db.relationship('Movie', back_populates='actors')
```

And all together:

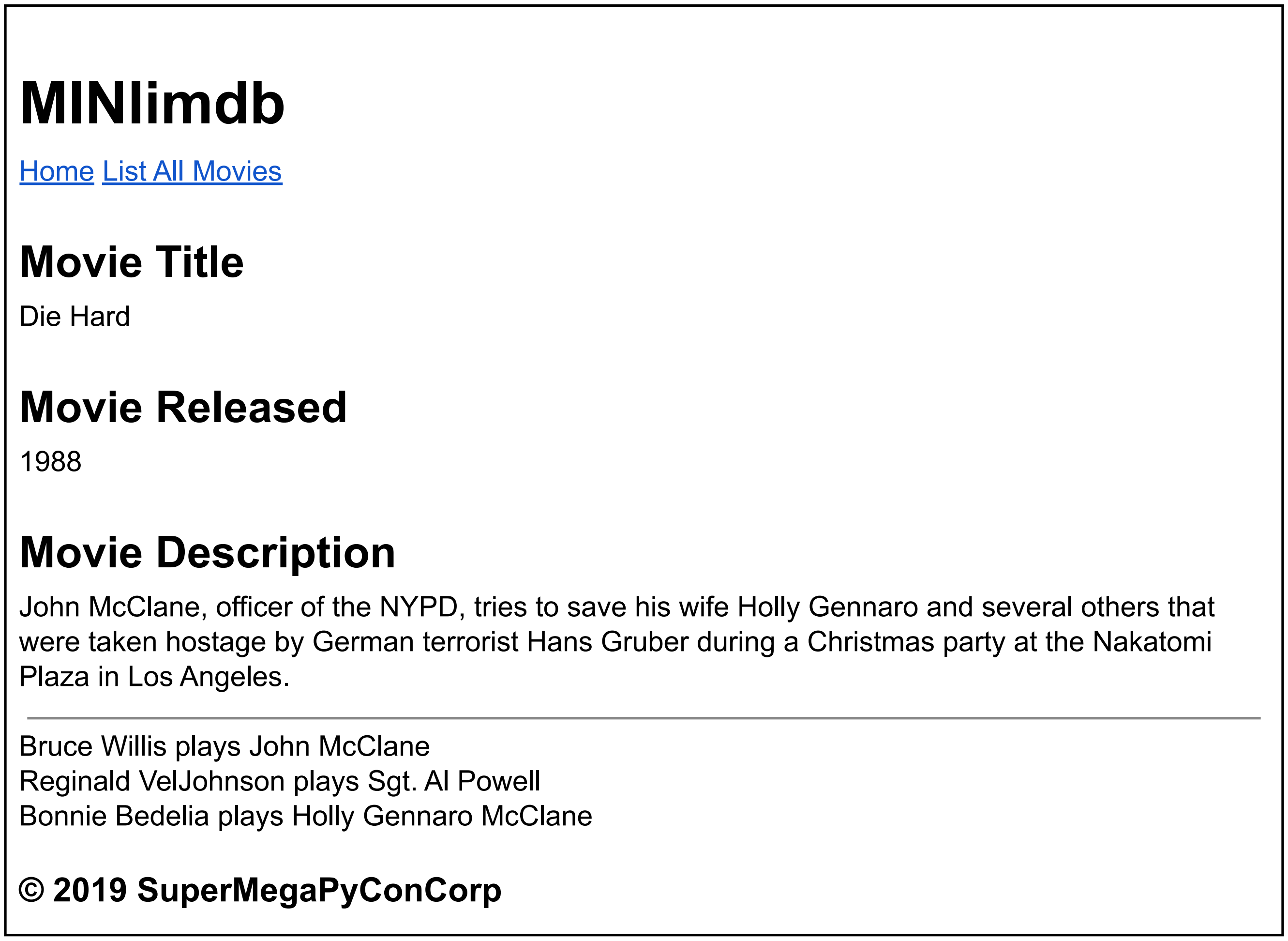
```
from main import db

class Movie(db.Model):
    __tablename__ = 'Movie'

    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    year = db.Column(db.Integer)
    description = db.Column(db.String())
    actors = db.relationship('Role', back_populates='movie')

class Actor(db.Model):
    __tablename__ = 'Actor'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable = False)
    birthdate = db.Column(db.String(30))
    movies = db.relationship('Role', back_populates='actor')

    def __str__(self):
        return self.name

class Role(db.Model):
    __tablename__ = 'Role'

    id = db.Column(db.Integer, primary_key=True)
    movie_id = db.Column(db.Integer, db.ForeignKey('Movie.id'),
nullable = False)
    actor_id = db.Column(db.Integer, db.ForeignKey('Actor.id'),
nullable = False)
    role = db.Column(db.String(80), nullable = False)

    actor = db.relationship('Actor', back_populates='movies')
    movie = db.relationship('Movie', back_populates='actors')
```

All of this is to allow us to access one table from another. We can do this now by adding an extra loop onto the bottom of the movie.html page:

```
{% for person in movie.actors %}
 {{person.actor}} plays {{person.role}}<br />
{% endfor %}
```

It's worth noting that SQLAlchemy has some great documentation. The setup for this project comes from https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html#association-object. It's more complex than a normal many-to-many relationship due to the role column the association table has. If we were doing a Pizza site, for example, and needed toppings on pizzas but no further description then the setup is easier - https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html#many-to-many

The movie details page now looks like this:



MINIimdb

Home List All Movies

Movie Title

Die Hard

Movie Released

1988

Movie Description

John McClane, officer of the NYPD, tries to save his wife Holly Gennaro and several others that were taken hostage by German terrorist Hans Gruber during a Christmas party at the Nakatomi Plaza in Los Angeles.

Bruce Willis plays John McClane
Reginald VelJohnson plays Sgt. Al Powell
Bonnie Bedelia plays Holly Gennaro McClane

© 2019 SuperMegaPyConCorp

# Part 4 - A Simple Form

Forms can be built using straight HTML, but the Flask-WTF package does a great job of making them simple and safe.  Let's look at a simple drop-down (select) list to choose a movie - mainly because there's a trick to setting up a drop-down that is worth knowing.

First, create a file called **forms.py** and put the following in it

```
from flask_wtf import FlaskForm
from wtforms import SelectField
from wtforms.validators import DataRequired

class Select_Movie(FlaskForm):
    movies = SelectField('movies', validators=[DataRequired()],
coerce=int)
```

This creates a drop-down list form that will only validate if something was selected, and which will return an int.

The main thing to know is that you don't set up the drop-down choices here - that is done when the form is created and just before the template is rendered - let's take a look at the route:

```
@app.route('/choose_movie', methods=['GET', 'POST'])
def choose_movie():
 form = Select_Movie()
 movies = models.Movie.query.all()
 form.movies.choices = [(movie.id, movie.title) for movie in
movies]
 if request.method=='POST':
   if form.validate_on_submit():
     return redirect(url_for('movie', id=form.moviename.data))
   else:
     abort(404)
 return render_template('movies.html', title='Select A Movie',
form=form)
```

There's a lot going on here - let's look at it line by line.  First up, this route accepts GET and POST requests (by default they only accept GET), which means you can submit (POST) your choice on the form and the route can deal with it.

On line 3 the form is instantiated as 'form'.

Then a query is made to get all the movies

Then the form is populated with tuples (id, movie title). The ID is what will be submitted, and the title is what will be seen in the drop-down list. It uses a list comprehension to populate it.

Then we check to see if this route was accessed via POST (ie: submit was pressed), if so then check if the form validated (this form just needs to have a value selected to validate).

If the form validates then redirect using url_for (you can redirect directly by entering the route, but this method handles changes to route names elegantly).

If the form doesn't validate, then throw a 404.

For all of this, we need to import *abort*, *request*, *redirect* and *url_for* from Flask, as well as our new form - and then add a few more things to the top of routes.py - this is how it needs to look:

```
from flask import Flask, render_template, abort, request, redirect,
url_for
from flask_sqlalchemy import SQLAlchemy
from forms import Select_Movie

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///movies.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.secret_key = 'correcthorsebatterystaple'

db = SQLAlchemy(app)

WTF_CSRF_ENABLED = True
WTF_CSRF_SECRET_KEY = 'sup3r_secr3t_passw3rd'

# need db to import models
import models
```

We've added a secret_key to the config, which is required for forms to work, and some CSRF prevention configuration, and we've imported the Select_Movie form so it can be used.

## PART 5 - Movie Tagging

The Role table is a complex many-to-many relationship, but what about simple many-to-many relationships, where we don't care about the content in a table, and just want to 'see through' it to the other table? For example, content tagging.

```python
MovieTag = db.Table('MovieTag', db.Model.metadata,
        db.Column('Movie_id', db.Integer, db.ForeignKey('Movie.id')),
        db.Column('Tag_id', db.Integer, db.ForeignKey('Tag.id'))
)


class Tag(db.Model):
    __tablename__ = 'Tag'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    description = db.Column(db.Text(500))

    movies = db.relationship('Movie', secondary=MovieTag, back_populates='tags')


class Movie(db.Model):
    __tablename__ = 'Movie'

    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    year = db.Column(db.Integer)
    description = db.Column(db.String())

    actors = db.relationship('Role', back_populates='movie')
    tags = db.relationship('Tag', secondary=MovieTag, back_populates='movies')
```

*Last Update: 25 June 2020*