What is Werkzeug?

Werkzeug is a collection of libraries that can be used to create a WSGI (Web Server Gateway Interface) compatible web application in Python.

A WSGI (Web Server Gateway Interface) server is necessary for Python web applications since a web server cannot communicate directly with Python. WSGI is an interface between a web server and a Python-based web application.

Put another way, Werkzeug provides a set of utilities for creating a Python application that can talk to a WSGI server, like [Gunicorn](https://gunicorn.org/).

Want to learn more about WSGI?

Check out ['What is Gunicorn in Python?'](https://www.quora.com/What-is-Gunicorn-in-Python/answer/Michael-Herman-3) and take a look at the [Building a Python Web Framework](https://testdriven.io/courses/python-web-framework/wsgi/) course.

Werkzeug provides the following functionality (*which Flask uses*):

1. Request processing
2. Response handling
3. URL routing
4. Middleware
5. HTTP utilities
6. Exception handling

It also provides a basic development server with hot reloading.

Let's dive into an example of building a web application using Werkzeug. We'll also look at how Flask implements similar functionality.

Hello World App

As an introduction to Werkzeug, let's start by creating a "Hello World" app using some of the key functionality provided by Werkzeug.

You can find the source code for the project discussed in this article on GitLab: <https://gitlab.com/patkennedy79/werkzeug_movie_app>.

Installation

Start by creating a new project:

$ mkdir werkzeug\_movie\_app

$ cd werkzeug\_movie\_app

$ python3 -m venv venv

$ source venv/bin/activate

(venv)$

Install Werkzeug, Jinja, and [redis-py](https://redis-py.readthedocs.io/):

(venv)$ pip install Werkzeug Jinja2 redis

(venv)$ pip freeze > requirements.txt

[Redis](https://redis.io/) will be used as the data storage solution for storing movie data.

Application

Werkzeug is a collection of libraries used to build a WSGI-compatible web application. It doesn't provide a high-level class, like [Flask](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L103), for scaffolding out a full web application. Instead, you need to create the application yourself from Werkzeug's libraries.

Create a new *app.py* file in the top-level folder of your project:

**from** **werkzeug.wrappers** **import** Request, Response

**class** **MovieApp**(object):

*"""Implements a WSGI application for managing your favorite movies."""*

**def** \_\_init\_\_(self):

**pass**

**def** dispatch\_request(self, request):

*"""Dispatches the request."""*

**return** Response('Hello World!')

**def** wsgi\_app(self, environ, start\_response):

*"""WSGI application that processes requests and returns responses."""*

request = Request(environ)

response = self.dispatch\_request(request)

**return** response(environ, start\_response)

**def** \_\_call\_\_(self, environ, start\_response):

*"""The WSGI server calls this method as the WSGI application."""*

**return** self.wsgi\_app(environ, start\_response)

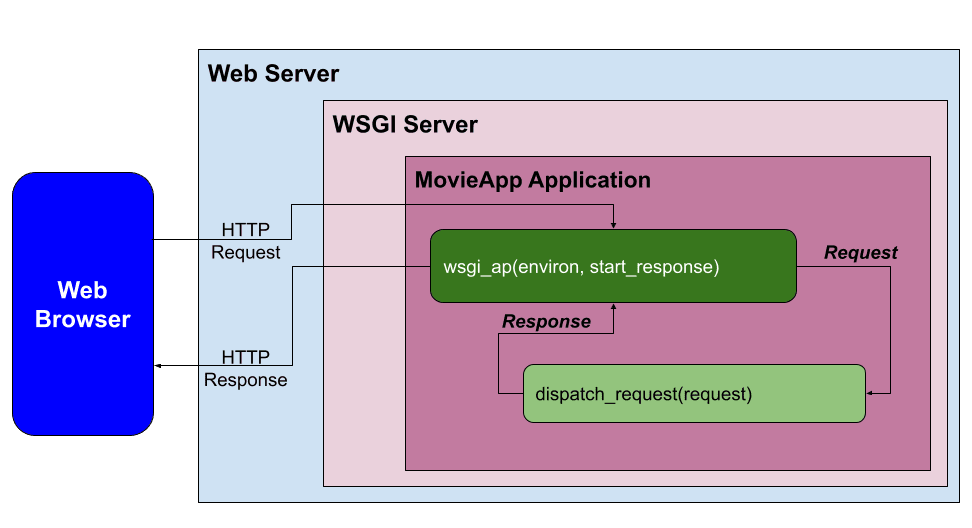
**def** create\_app():

*"""Application factory function that returns an instance of MovieApp."""*

app = MovieApp()

**return** app

The MovieApp class implements a WSGI-compatible web application, which processes requests from different users and generates responses back to the users. Here's the flow of how this class interfaces with a WSGI server:



When a request comes in, it's processed in wsgi\_app():

**def** wsgi\_app(self, environ, start\_response):

*"""WSGI application that processes requests and returns responses."""*

request = Request(environ)

response = self.dispatch\_request(request)

**return** response(environ, start\_response)

The environment (environ) is automatically processed in the Request class to create a request object. The request is then processed in dispatch\_request(). For this initial example, dispatch\_request() returns a response of 'Hello World!'. The response is then returned from wsgi\_app().

Flask Comparison:

MovieApp is an simplified version of the [Flask](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L103) class.

Within the Flask class, the [wsgi\_app()](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L2417) is the actual WSGI application that interfaces with the WSGI server. Also, [dispatch\_request()](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L1914) and [full\_dispatch\_request()](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L1938) are used to do the request dispatching, which matches the URL to the applicable view function and handles exceptions.

Development Server

Add the following code to the bottom of *app.py* to run the Werkzeug development server:

**if** \_\_name\_\_ == '\_\_main\_\_':

*# Run the Werkzeug development server to serve the WSGI application (MovieApp)*

**from** **werkzeug.serving** **import** run\_simple

app = create\_app()

run\_simple('127.0.0.1', 5000, app, use\_debugger=**True**, use\_reloader=**True**)

Run the application:

(venv)$ python app.py

Navigate to [http://localhost:5000](http://localhost:5000/) to see the 'Hello World!' Message.

Flask Comparison:

Within the Flask class, there's an equivalent [run()](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L889) method that utilizes the Werkzeug development server.

Middleware for Serving Static Files

In web applications, middleware is a software component that can be added to the request/response processing pipeline to perform a specific function.

One important function for a web server/application to perform is serving static files (CSS, JavaScript, and image files). Werkzeug provides a middleware for this functionality called [SharedDataMiddleware](https://github.com/pallets/werkzeug/blob/1.0.1/src/werkzeug/middleware/shared_data.py#L32).

SharedDataMiddleware is ideally suited for working with the Werkzeug development server to serve static files.

For a production environment, you'll want to switch out the Werkzeug development server and SharedDataMiddleware for a web server such as [Nginx](https://www.nginx.com/) and a WSGI server such as Gunicorn.

To utilize SharedDataMiddleware, start by adding a new folder called "static" to the project with "css" and "img" folders:

├── app.py

├── requirements.txt

└── static

├── css

└── img

Within the "static/img" folder, add the Flask logo from <https://gitlab.com/patkennedy79/werkzeug_movie_app/-/blob/main/static/img/flask.png>. Save it as *flask.png*.

Next, expand the application factory function:

**def** create\_app():

*"""Application factory function that returns an instance of MovieApp."""*

app = MovieApp()

app.wsgi\_app = SharedDataMiddleware(app.wsgi\_app, {

'/static': os.path.join(os.path.dirname(\_\_file\_\_), 'static')

})

**return** app

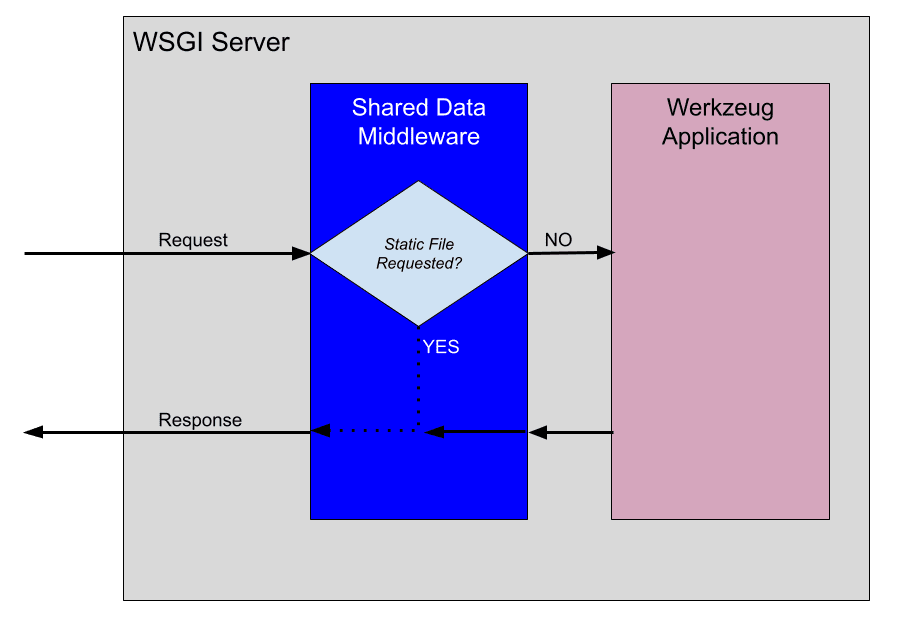
Update the imports at the top:

**import** **os**

**from** **werkzeug.middleware.shared\_data** **import** SharedDataMiddleware

**from** **werkzeug.wrappers** **import** Request, Response

Now when a request is processed by the Werkzeug applicaiton (app), it will first be routed to SharedDataMiddleware to determine if a static file has been requested:



If a static file is requested, SharedDataMiddleware will generate the response with the static file. Otherwise, the request is passed down the chain to the Werkzeug app for processing in wsgi\_app().

To see SharedDataMiddleware in action, run the server and navigate to <http://localhost:5000/static/img/flask.png> to view the Flask logo.

For a full list of middleware solutions provided by Werkzeug, check out the [Middleware](https://werkzeug.palletsprojects.com/en/1.0.x/middleware/) docs.

Flask Comparison:

Flask does not utilize the SharedDataMiddleware. It takes a different approach for serving static files. By default, if a static folder [exists](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L597), Flask automatically [adds a new URL rule](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L601) to serve the static files up.

To illustrate this concept, run flask routes in the top-level project of a Flask application and you will see:

(venv)$ flask routes

Endpoint Methods Rule

----------- ------- -----------------------

index GET /

static GET /static/<path:filename>

Templates

As is typically done in a Flask project, we'll use Jinja for the templating engine for our app.

Start by adding a new folder called "templates" to the project:

├── app.py

├── requirements.txt

├── static

│   ├── css

│   └── img

│   └── flask.png

└── templates

In order to utilize Jinja, expand the constructor of the MovieApp class:

**def** \_\_init\_\_(self):

*"""Initializes the Jinja templating engine to render from the 'templates' folder."""*

template\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'templates')

self.jinja\_env = Environment(loader=FileSystemLoader(template\_path),

autoescape=**True**)

Add the import:

**from** **jinja2** **import** Environment, FileSystemLoader

Flask Comparison:

Flask utilizes Jinja [Environment](https://github.com/pallets/flask/blob/1.1.2/src/flask/templating.py#L36) as well to create the templating engine.

Within the MovieApp class, add a new render\_template() method:

**def** render\_template(self, template\_name, \*\*context):

*"""Renders the specified template file using the Jinja templating engine."""*

template = self.jinja\_env.get\_template(template\_name)

**return** Response(template.render(context), mimetype='text/html')

This method takes the template\_name and any variables to pass to the templating engine (\*\*context). It then generates a Response using the render() method from Jinja.

Flask Comparison:

Doesn't the render\_template() function look familiar? The Flask [flavor](https://github.com/pallets/flask/blob/1.1.2/src/flask/templating.py#L125) is one of the most used functions in Flask.

To see render\_template() in action, update dispatch\_request() to render a template:

**def** dispatch\_request(self, request):

*"""Dispatches the request."""*

**return** self.render\_template('base.html')

All requests to the app will now render the *templates/base.html* template.

<!DOCTYPE html>

<**html** lang="en">

<**head**>

<**meta** charset="UTF-8">

<**title**>Werkzeug Movie App</**title**>

*<!-- CSS file for styling the application -->*

<**link** rel="stylesheet" href="/static/css/style.css" type="text/css">

</**head**>

<**body**>

<**h1**>Werkzeug Movie App</**h1**>

{% block body %}

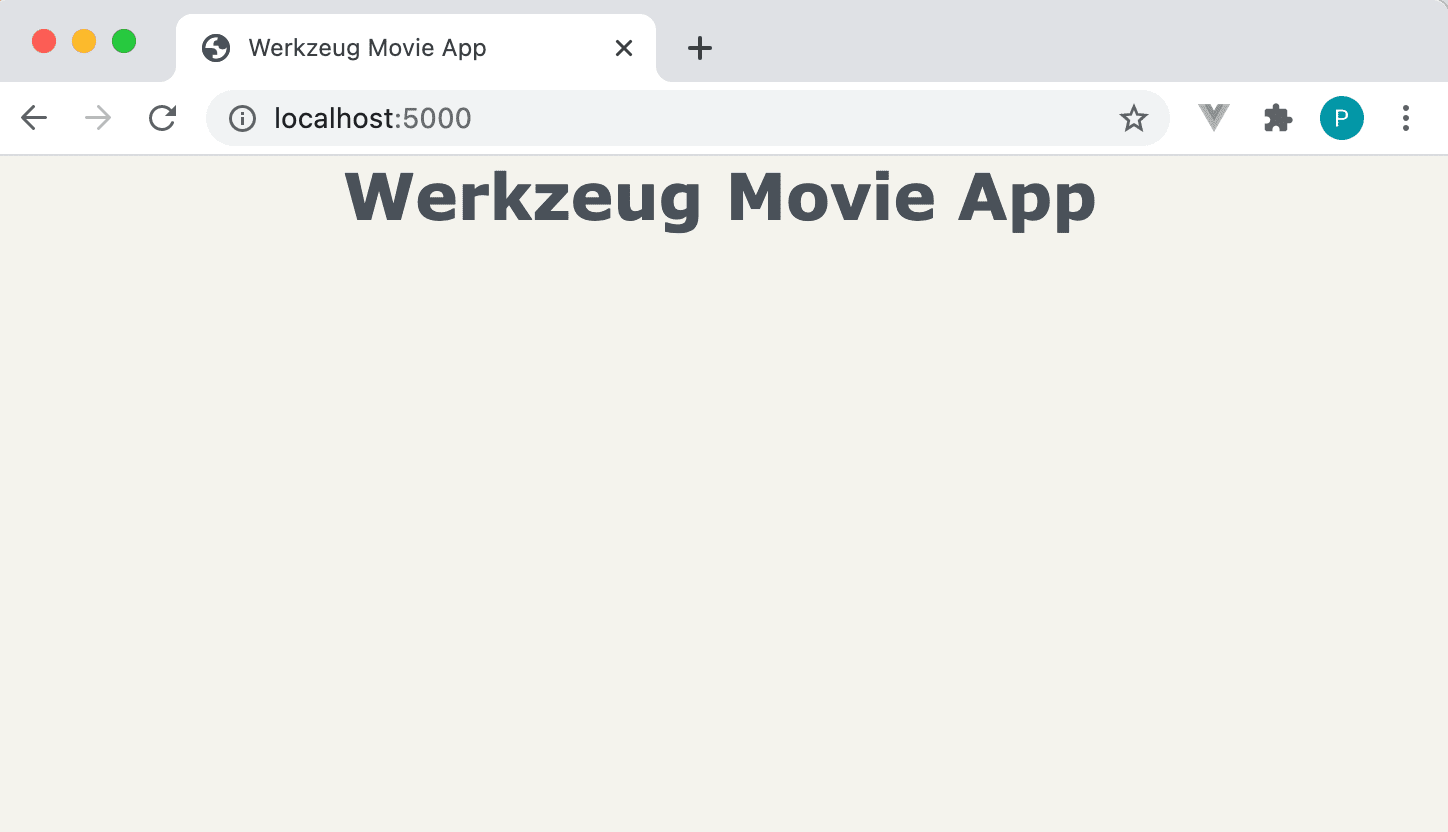
{% endblock %}

</**body**>

</**html**>

Make sure to add this template to the "templates" folder and save a copy of <https://gitlab.com/patkennedy79/werkzeug_movie_app/-/blob/main/static/css/style.css> to *static/css/style.css*.

Run the server. Navigate to [http://localhost:5000](http://localhost:5000/). You should now see:



Routing

Routing means to match the URL to the appropriate view function. Werkzeug provides a [Map](https://werkzeug.palletsprojects.com/en/1.0.x/routing/#werkzeug.routing.Map) class that allows you to match URLs to view functions using [Rule](https://werkzeug.palletsprojects.com/en/1.0.x/routing/#werkzeug.routing.Rule) objects.

Let's create the Map object in the MovieApp constructor to illustrate how this works:

**def** \_\_init\_\_(self):

*"""Initializes the Jinja templating engine to render from the 'templates' folder."""*

template\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'templates')

self.jinja\_env = Environment(loader=FileSystemLoader(template\_path),

autoescape=**True**)

self.url\_map = Map([

Rule('/', endpoint='index'),

Rule('/movies', endpoint='movies'),

])

Don't forget the import:

**from** **werkzeug.routing** **import** Map, Rule

Each Rule object defines a URL and the view function (endpoint) to call if the URL is matched:

self.url\_map = Map([

Rule('/', endpoint='index'),

Rule('/movies', endpoint='movies'),

])

For example, when the homepage ('/') is requested, the index view function should be called.

Flask Comparison:

One of the amazing features of Flask is the [@route decorator](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L1288), which is used to assign a URL to a view function. This decorator [updates the url\_map](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L1278) for the Flask app, similar to the hand-coded url\_map that we defined above.

In order to utilize the URL mapping, dispatch\_request() needs to be updated:

**def** dispatch\_request(self, request):

*"""Dispatches the request."""*

adapter = self.url\_map.bind\_to\_environ(request.environ)

**try**:

endpoint, values = adapter.match()

**return** getattr(self, endpoint)(request, \*\*values)

**except** HTTPException **as** e:

**return** e

Now when a request comes in to dispatch\_request(), the url\_map will be utilized to attempt to match() the URL to an entry. If the URL requested is included in the url\_map, then the applicable view function (endpoint) will be called. If the URL is not found in the url\_map, then an exception is raised.

Exception handling will be covered shortly!

Add the import:

**from** **werkzeug.exceptions** **import** HTTPException

We've specified two view functions in the url\_map, so let's create them now within the MovieApp class:

**def** index(self, request):

**return** self.render\_template('base.html')

**def** movies(self, request):

**return** self.render\_template('movies.html')

While *templates/base.html* was created in the previous section, *templates/movies.html* needs to be created now:

{% extends "base.html" %}

{% block body %}

<**div** class="table-container">

<**table**>

*<!-- Table Header -->*

<**thead**>

<**tr**>

<**th**>Index</**th**>

<**th**>Movie Title</**th**>

</**tr**>

</**thead**>

*<!-- Table Elements (Rows) -->*

<**tbody**>

<**tr**>

<**td**>1</**td**>

<**td**>Knives Out</**td**>

</**tr**>

<**tr**>

<**td**>2</**td**>

<**td**>Pirates of the Caribbean</**td**>

</**tr**>

<**tr**>

<**td**>3</**td**>

<**td**>Inside Man</**td**>

</**tr**>

</**tbody**>

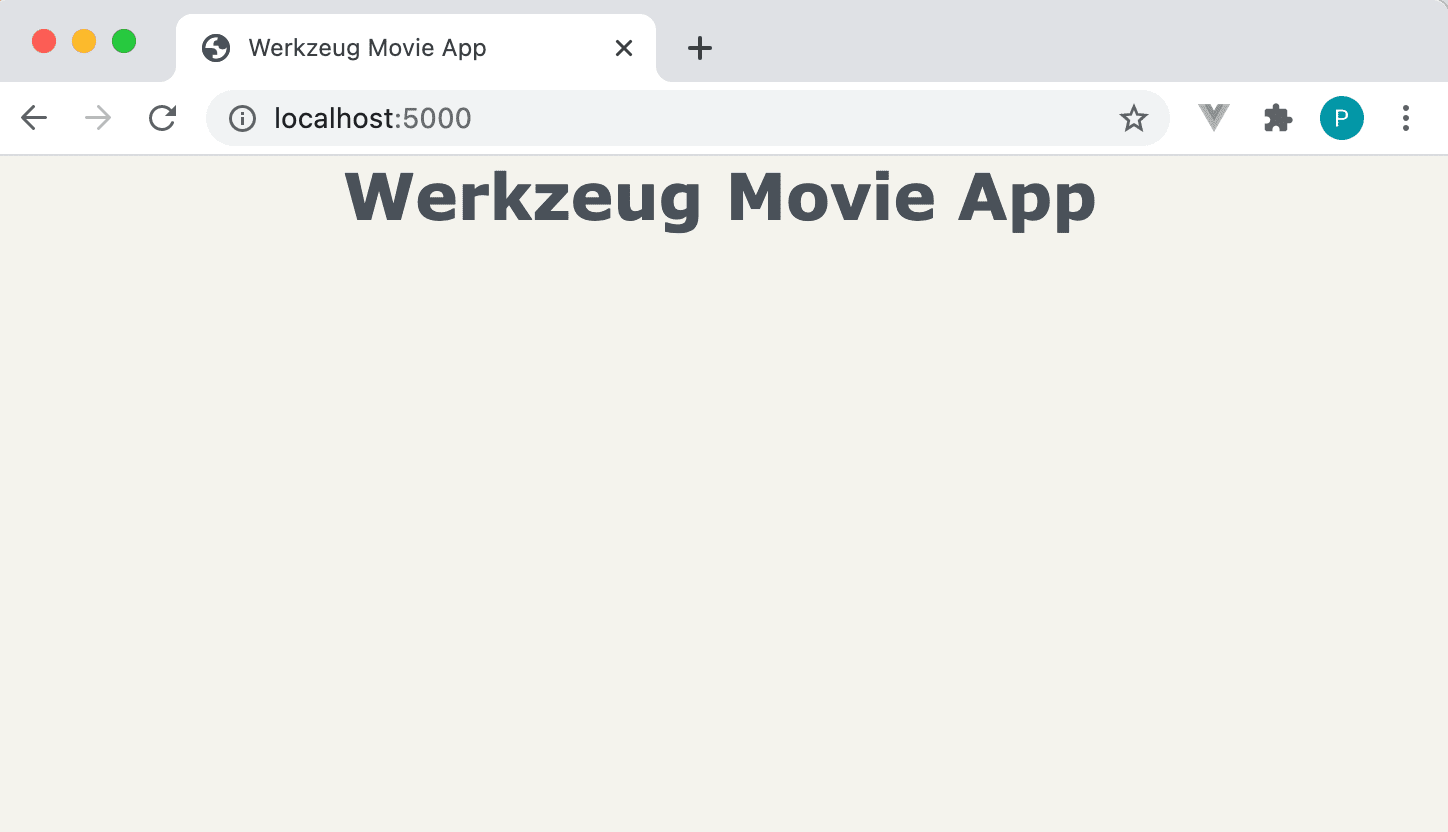
</**table**>

</**div**>

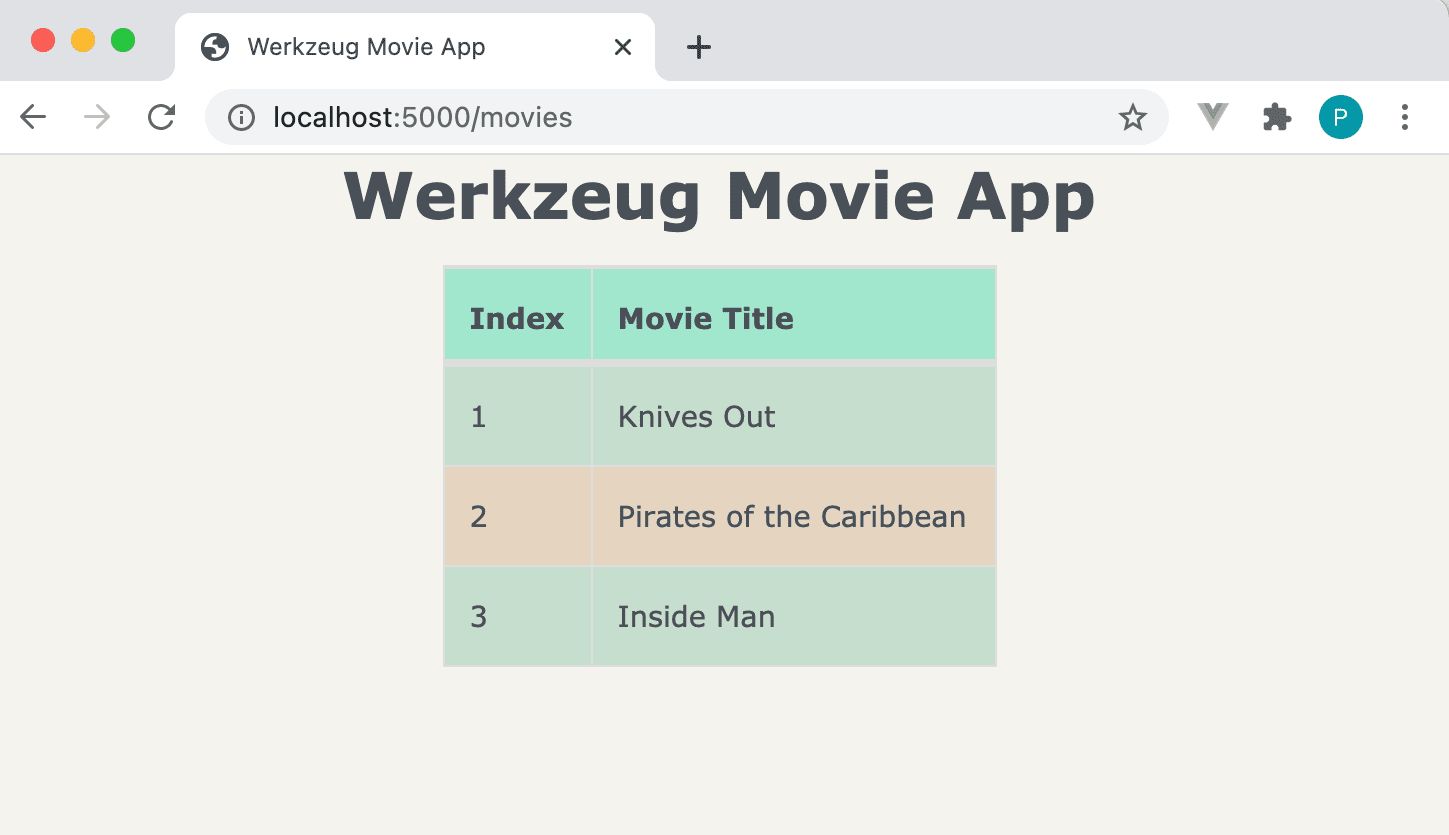
{% endblock %}

This template file utilizes template inheritance to use *base.html* as the parent template. It generates a table of three movies.

[http://localhost:5000](http://localhost:5000/) should look the same:

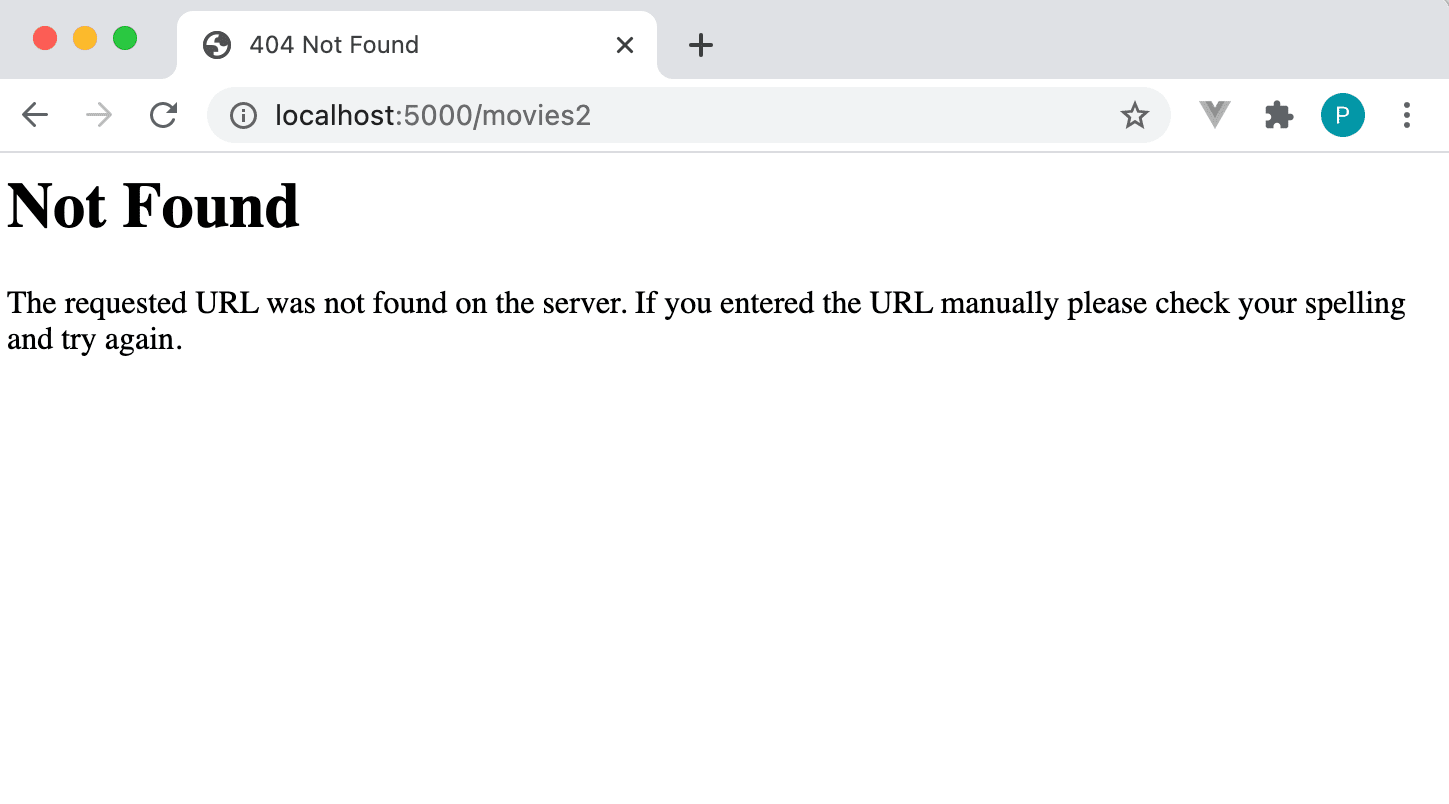


However, if you navigate to <http://localhost:5000/movies>, you'll now see the table of movies:



Exception Handling

Try navigating to <http://localhost:5000/movies2>:



The page returned is the default error page when a URL is not found in url\_map:

**def** dispatch\_request(self, request):

*"""Dispatches the request."""*

adapter = self.url\_map.bind\_to\_environ(request.environ)

**try**:

endpoint, values = adapter.match()

**return** getattr(self, endpoint)(request, \*\*values)

**except** HTTPException **as** e:

**return** e

Additionally, you should see the following in the console:

127.0.0.1 - - [07/Mar/2021 12:13:17] "GET /movies2 HTTP/1.1" 404 -

Let's create a custom error page by expanding dispatch\_request():

**def** dispatch\_request(self, request):

*"""Dispatches the request."""*

adapter = self.url\_map.bind\_to\_environ(request.environ)

**try**:

endpoint, values = adapter.match()

**return** getattr(self, endpoint)(request, \*\*values)

**except** NotFound:

**return** self.error\_404()

**except** HTTPException **as** e:

**return** e

Update the import:

**from** **werkzeug.exceptions** **import** HTTPException, NotFound

Now when a URL is not found in the url\_map, it will be handled by calling error\_404(). Create this new method within the MovieApp class:

**def** error\_404(self):

response = self.render\_template("404.html")

response.status\_code = 404

**return** response

Create *templates/404.html*:

{% extends "base.html" %}

{% block body %}

<**div** class="error-description">

<**h2**>Page Not Found (404)</**h2**>

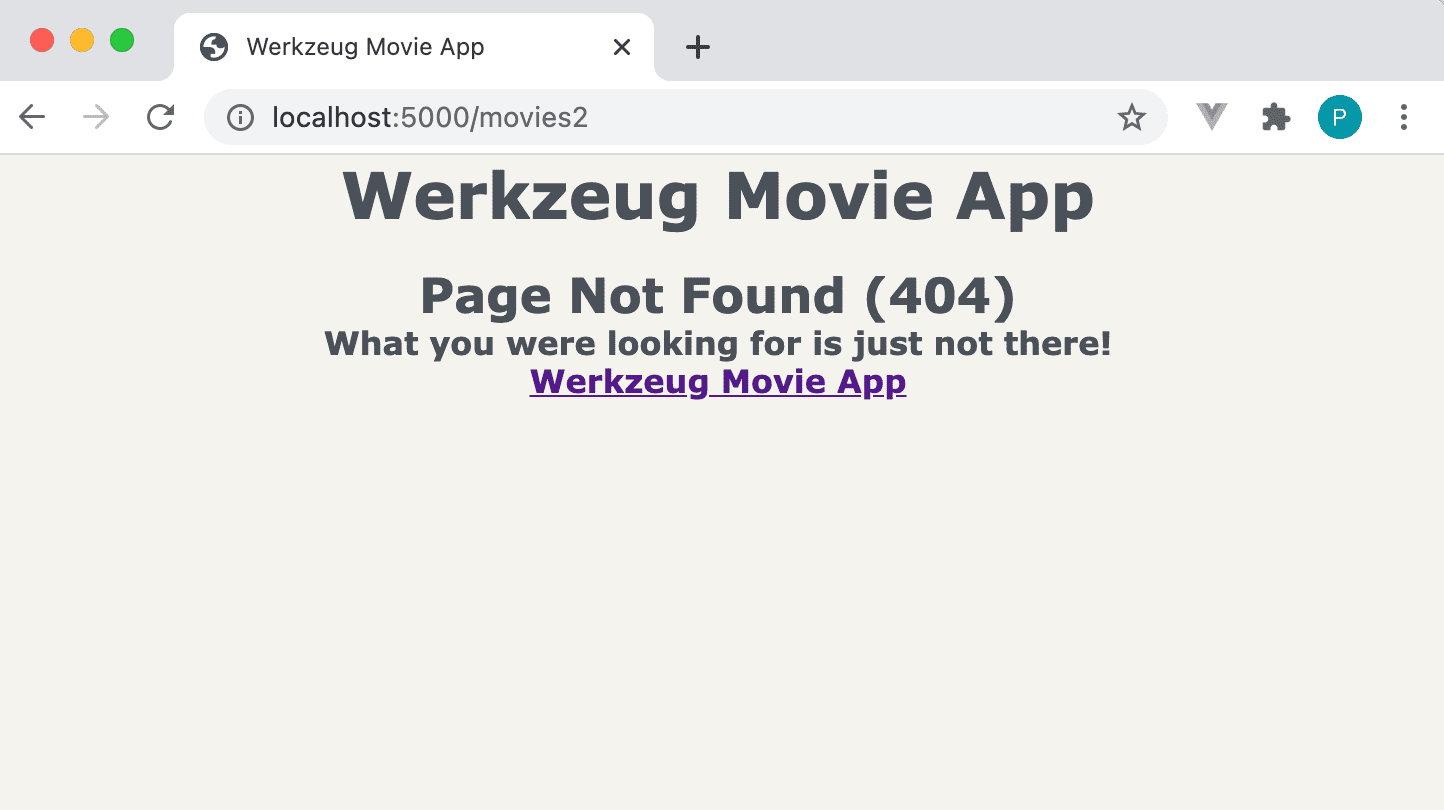
<**h4**>What you were looking for is just not there!</**h4**>

<**h4**><**a** href="/">Werkzeug Movie App</**a**></**h4**>

</**div**>

{% endblock %}

Now when you navigate to <http://localhost:5000/movies2>, you should see a friendly message:



Flask Comparison:

When full\_dispatch\_request() in the Flask class detects an exception, it will be handled gracefully in [handle\_user\_exceptions()](https://github.com/pallets/flask/blob/1.1.2/src/flask/app.py#L1781). Flask allows custom error pages for all HTTP error codes as well.

Request Processing

In this section, we'll add a form to the app to allow the user to input their favorite movies.

Redis

As mentioned, we'll be using Redis, an in-memory data structure store, to persist the movies due to its fast read/write speed and ease of setup.

[Install](https://redis.io/download) and run Redis.

The quickest way to get Redis up and running is with Docker:

$ docker run --name some-redis -d -p 6379:6379 redis

To check that the Redis container is running:

$ docker ps

To stop the running Redis container:

$ docker stop some-redis # Use name of Docker container

If you're not a Docker user, check out these resources:

* [Install and Configure Redis on Mac OS X via Homebrew](https://medium.com/@petehouston/install-and-config-redis-on-mac-os-x-via-homebrew-eb8df9a4f298)
* [DigitalOcean - How to Install and Secure Redis on Ubuntu](https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-redis-on-ubuntu-18-04)

In order to utilize Redis, start by updating the MovieApp constructor to create an instance of StrictRedis:

**def** \_\_init\_\_(self, config): *# Updated!!*

*"""Initializes the Jinja templating engine to render from the 'templates' folder,*

*defines the mapping of URLs to view methods, and initializes the Redis interface."""*

template\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'templates')

self.jinja\_env = Environment(loader=FileSystemLoader(template\_path),

autoescape=**True**)

self.url\_map = Map([

Rule('/', endpoint='index'),

Rule('/movies', endpoint='movies'),

])

self.redis = StrictRedis(config['redis\_host'], config['redis\_port'],

decode\_responses=**True**) *# New!!*

Additionally, the constructor (\_\_init\_\_()) has an additional argument (config), which is used for creating the instance of StrictRedis.

Import:

**from** **redis** **import** StrictRedis

The configuration parameters that are passed in to the constructor need to be specified in the application factory function:

**def** create\_app():

*"""Application factory function that returns an instance of MovieApp."""*

app = MovieApp({'redis\_host': 'localhost', 'redis\_port': 6379})

app.wsgi\_app = SharedDataMiddleware(app.wsgi\_app, {

'/static': os.path.join(os.path.dirname(\_\_file\_\_), 'static')

})

**return** app

Form Processing

In order to allow the user to add a movie to the Redis storage, we need to add a new view function in url\_map:

**def** \_\_init\_\_(self, config):

*"""Initializes the Jinja templating engine to render from the 'templates' folder,*

*defines the mapping of URLs to view methods, and initializes the Redis interface."""*

...

self.url\_map = Map([

Rule('/', endpoint='index', methods=['GET']),

Rule('/movies', endpoint='movies', methods=['GET']),

Rule('/add', endpoint='add\_movie', methods=['GET', 'POST']), *# !!!*

])

...

The Rule entries in url\_map have been expanded to specify the HTTP methods that are allowed for each URL. Additionally, the '/add' URL has been added:

Rule('/add', endpoint='add\_movie', methods=['GET', 'POST']),

If the '/add' URL is requested with either the GET or POST methods, then the add\_movie() view function will be called.

Next, we need to create the add\_movie() view function in the MovieApp class:

**def** add\_movie(self, request):

*"""Adds a movie to the list of favorite movies."""*

**if** request.method == 'POST':

movie\_title = request.form['title']

self.redis.lpush('movies', movie\_title)

**return** redirect('/movies')

**return** self.render\_template('add\_movie.html')

Import:

**from** **werkzeug.utils** **import** redirect

If a GET request is made to '/add', then add\_movie() will render the *templates/add\_movie.html* file. If a POST request is made to '/add', then the form data is stored in the Redis storage in the movies list and the user is redirected to the list of movies.

Create the *templates/add\_movie.html* template file:

{% extends "base.html" %}

{% block body %}

<**div** class="form-container">

<**form** method="post">

<**div** class="field">

<**label** for="movieTitle">Movie Title:</**label**>

<**input** type="text" id="movieTitle" name="title"/>

</**div**>

<**div** class="field">

<**button** type="submit">Submit</**button**>

</**div**>

</**form**>

</**div**>

{% endblock %}

Display Movies

Since we're now storing the movies in Redis, the movie() view function needs to be updated to read from the movies list in Redis:

**def** movies(self, request):

*"""Displays the list of favorite movies."""*

movies = self.redis.lrange('movies', 0, -1)

**return** self.render\_template('movies.html', movies=movies)

The list of movies is being passed to the *templates/movies.html* template file, which needs to be updated to loop through this list to create the table of movies:

{% extends "base.html" %}

{% block body %}

<**div** class="table-container">

<**table**>

*<!-- Table Header -->*

<**thead**>

<**tr**>

<**th**>Index</**th**>

<**th**>Movie Title</**th**>

</**tr**>

</**thead**>

*<!-- Table Elements (Rows) -->*

<**tbody**>

{% for movie in movies %}

<**tr**>

<**td**>{{ loop.index }}</**td**>

<**td**>{{ movie }}</**td**>

</**tr**>

{% endfor %}

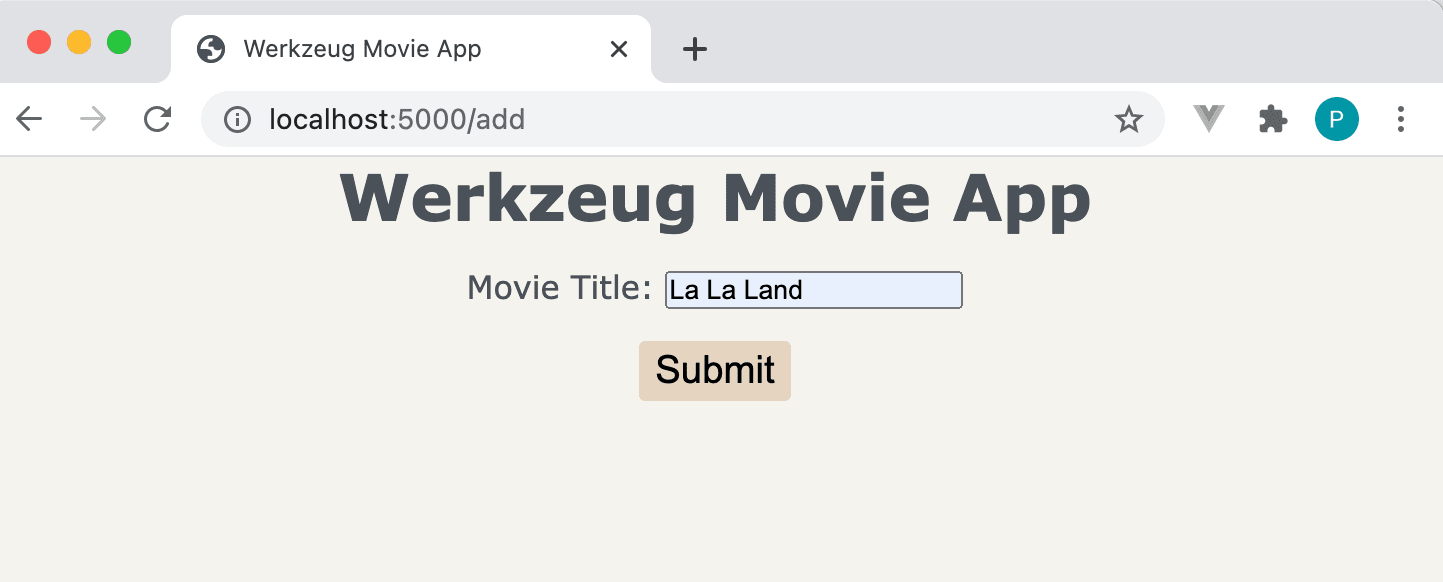
</**tbody**>

</**table**>

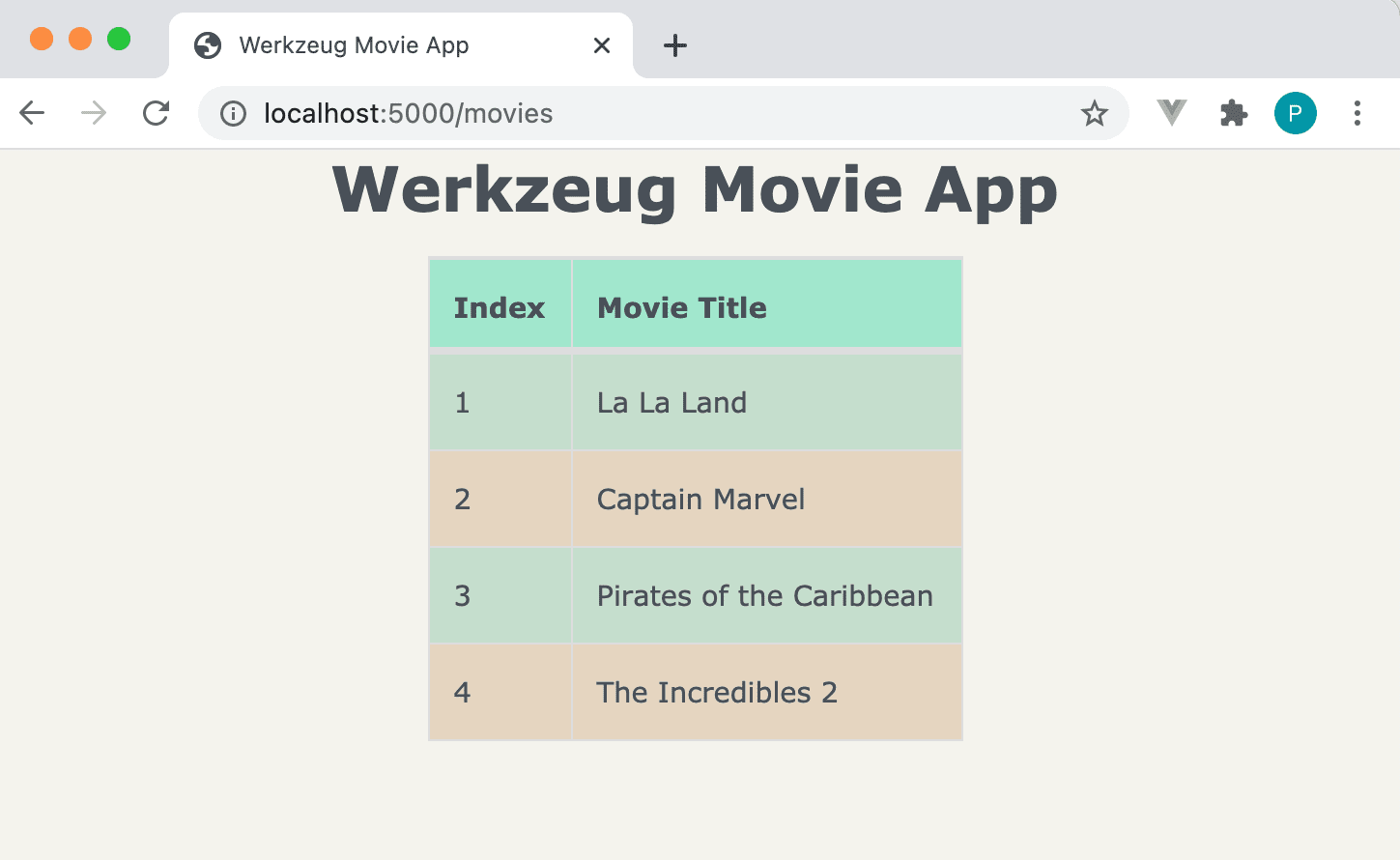
</**div**>

{% endblock %}

To see the form processing in action, start by navigating to <http://localhost:5000/add> and adding a new movie:



After submitting the form, you should be automatically redirected to the list of movies (which may include movies previously added):



That's it!

Why Not Use Werkzeug Instead of Flask?

Werkzeug provides much of the key functionality found in Flask, but Flask adds a number of powerful features, like:

1. Sessions
2. Application and Request contexts
3. Blueprints
4. Request callback functions
5. Utilities:
   1. @route decorator
   2. url\_for() function
6. CLI commands
7. Exception handling
8. Test client
9. Flask shell
10. Logging
11. Signals
12. Extensions

As with any web framework -- Don't re-invent the wheel! Flask is a much better option (when compared to Werkzeug) for web development based on its rich feature set and large collection of extensions.

Conclusion

This article provided an overview of Werkzeug, which is one of the key components of Flask, by showing how to build a simple web application using Werkzeug. While it's important to understand how the underlying libraries work in Flask, the complexity of creating a web application using Werkzeug should illustrate how easy it is to develop a web app using Flask!

Additionally, if you're interested in learning how to test a Werkzeug application, check out the tests for the Werkzeug Movie App: <https://gitlab.com/patkennedy79/werkzeug_movie_app/-/tree/main/tests>.

If you'd like to learn more about Flask, be sure to check out my course -- [Developing Web Applications with Python and Flask](https://testdriven.io/courses/learn-flask/).