Flask Application Development Documentation

Release 1.0

H. Turgut Uyar

August 13, 2015

CONTENTS

Author H. Turgut Uyar

This tutorial aims to demonstrate how to build an application using the Flask web framework. You can find more information about the Flask project on its web site:

http://flask.pocoo.org/

Warning: Please keep in mind that this is not a "best practices" tutorial; it just tries to make it easier for a beginner to understand the Flask framework. Although I have prepared this tutorial to the best of my understanding on the subject, it may still contain errors, omissions, and improper uses of Python or Flask features.

CHAPTER

ONE

A PYTHON PRIMER

1.1 Data Types

tuple: different types, fixed number of elements, immutable

    singleton tuples: (x,)

list: preferably same type, mutable

    slicing, dicing

set: unordered, no duplicates, mutable

dictionary: key-value, table, mutable, unordered, .items()

everything is an object

assignments over references: mutable - immutable

None

1.2 Expressions

is

enumerate

sorted

1.3 Functions

def

1.4 Classes

class

self

no getters, setters

adding attributes after creating object

1.5 Strings

representation: '...', "...", """..."""

unicode, bytes, encode/decode

string formatting: %, .format

1.6 Packages

import ...

import ... as ...

from .. import ...

if __name__ == '__main__':

package: directory, module: file

1.7 Advanced

tuple unpacking

*args, **kwargs

CHAPTER

TWO

BASICS

We will first start by creating a simple home page containing only a page title. Then we will add the current date and time to this page to show how dynamic elements are handled in Flask. Next, we will create a second page also containing only a page title. This second page will be populated with the list of movies in later chapters. Finally, we will provide a link from the home page to the movie list page. In the end, the home page will look as shown in *Screenshot: Home page containing a title and a link to the movie list page.*.

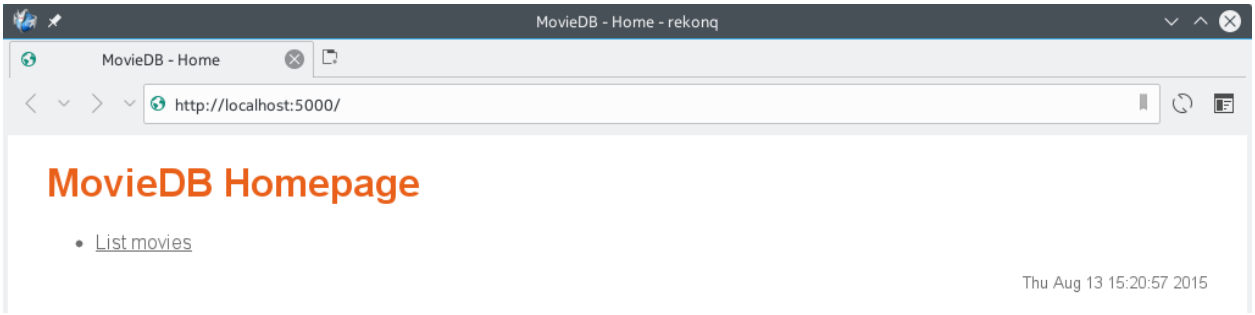

Figure 2.1: Screenshot: Home page containing a title and a link to the movie list page.

`server.py`: *Listing: Initial application*.

Listing Initial application.

```
from flask import Flask


app = Flask(__name__)


@app.route('/')
def home_page():
    return "<h1>MovieDB Homepage</h1>"


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Tip: read the Python Style Guide:

https://www.python.org/dev/peps/pep-0008/

---

routes and functions:

request to "/" -> response returned by "home_page" function

- line 1: import application class `Flask`
- line 4: instantiate application (global variable)
- line 13: start the application

Tip: start the application in debug mode for easier debugging:

```
app.run(host='0.0.0.0', port=5000, debug=True)
```

---

note:

- line 9: returns partial HTML, not a full page -> use a template

project structure:

templates in `templates` folder

static files (style sheets, images) in `static` folder

hierarchy:

```
flask-moviedb
|- server.py
|- templates
|  |- home.html
|- static
|  |- style.css
```

Add the file `template/home.html` as given in *Listing: Initial home page template* and change the file `server.py` as given in *Listing: Application with template*.

Listing Initial home page template.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>MovieDB - Home</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>MovieDB Homepage</h1>
  </body>
</html>
```

Listing Application with template.

```
from flask import Flask
from flask import render_template


app = Flask(__name__)


@app.route('/')
def home_page():
    return render_template('home.html')


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

- line 2: new import for rendering template
- line 10: return rendered template

Let us also apply a style sheet to our page. Add the file `static/style.css` as given in *Listing: Initial style sheet*.

note: line 6 in *Listing: Initial home page template*

Listing Initial style sheet.

```
1  body {
2    background-color: white;
3    color: #6F6F6F;
4    font-family: 'Droid Sans', 'Helvetica', 'sans';
5    font-size: 12pt;
6    margin: 2ex 2em;
7  }
8
9  h1, h2, h3, h4 {
10   color: #E9601A;
11 }
12
13 a {
14   color: #6F6F6F;
15   text-decoration: underline;
16 }
```

2.1 Dynamic Content

Let us add the current date and time to the home page. We first change the `template/home.html` file as in *Listing: Home page template showing the current date and time*. Note that, the only difference is the added footer section (lines 11-13).

Listing Home page template showing the current date and time.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8" />
5      <title>MovieDB - Home</title>
6      <link rel="stylesheet" href="/static/style.css" />
7    </head>
8    <body>
9      <h1>MovieDB Homepage</h1>
10
11     <footer>
12       <div id="datetime">{{ current_time }}</div>
13     </footer>
14   </body>
15 </html>
```

`{{ expression }}` -> put value of expression

- line 12: `{{ current_time }}`: `current_time` is a parameter to this template; it should be supplied by the calling function (`home_page`)

Listing: Application providing the current date and time

Listing Application providing the current date and time.

```
1  import datetime
2
3  from flask import Flask
4  from flask import render_template
5
6
7  app = Flask(__name__)
8
9
10 @app.route('/')
11 def home_page():
12     now = datetime.datetime.now()
13     return render_template('home.html', current_time=now.ctime())
14
15
16 if __name__ == '__main__':
17     app.run(host='0.0.0.0', port=5000)
```

- line 1: import for date and time operations
- line 12: get current time
- line 13: send to template

Finally, we would like control how the date and time will be displayed, so we add the lines given in *Listing: Stylesheet for date and time* to the `style.css` file.

Listing Style rules for date and time.

```
div#datetime {
  font-size: 80%;
  text-align: right;
}
```

The application, as implemented so far, will run as follows:

- An application object will be instantiated.
- This object will register the `home_page` function to respond to requests for the "/" route.
- The application starts running on the localhost on port 5000 and waits for requests.
- When a request comes for "http://localhost:5050/" the `home_page` function gets invoked.
- This function gets the current time and renders the template `home.html`, passing the current time as a parameter to it.

2.2 Adding Links

Now we want to create a second page which will be responsible for listing the movies in the collection. At first, the page will only contain some static text, it will be populated in later chapters. Create the HTML file `templates/movies.html` which will serve as a template for the `/movies` route. Arrange the HTML file as given in *Listing: Initial movie list page template*. Again, since there is no dynamic element in this template, the registered `movies_page` function only renders a static template (see *Listing: Application with movies page*, lines 16-18).

Listing Initial movie list page template.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>MovieDB - Movie List</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h2>Movie List</h2>
  </body>
</html>
```

Listing Application with movies page.

```
import datetime

from flask import Flask
from flask import render_template


app = Flask(__name__)


@app.route('/')
def home_page():
    now = datetime.datetime.now()
    return render_template('home.html', current_time=now.ctime())


@app.route('/movies')
def movies_page():
    return render_template('movies.html')


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Our next step will be to provide a link from the home page to the movie list page. In order to achieve this, we modify the `HomePage.html` as in *Listing: Home page template with link to movie list page*. The code for adding the link is on line 12.

Listing Home page template with link to movie list page.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>MovieDB - Home</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>MovieDB Homepage</h1>

    <ul>
      <li><a href="/movies">List movies</a></li>
    </ul>

    <footer>
      <div id="datetime">{{ current_time }}</div>
```

```
17      </footer>
18    </body>
19  </html>
```

2.3 Exercise

- Add a link from the movie list page to the home page. (*Solution*)

CHAPTER

THREE

APPLICATION STRUCTURE

The pages in a web application share some components such as navigation panels. In this chapter, we will see how to implement such components without repeating code. We will create a base page that will contain the components that all pages in the application will include. See *Screenshot: Home page containing navigation panel.* for the screenshot of the resulting home page.

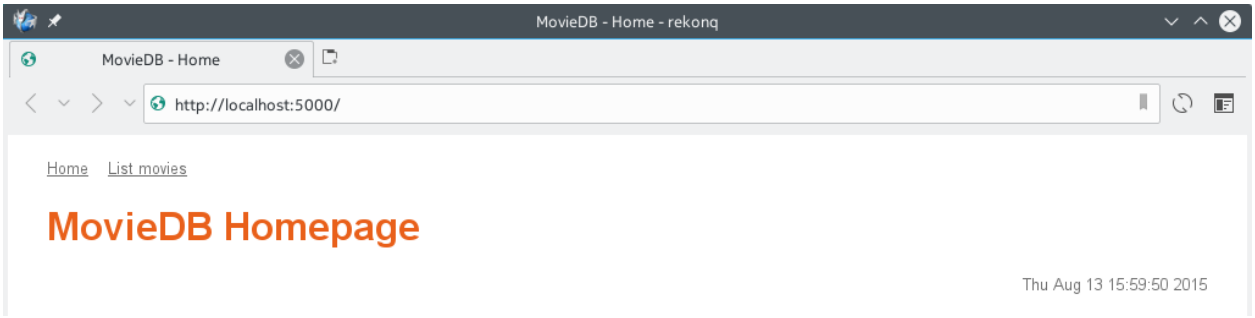

Figure 3.1: Screenshot: Home page containing navigation panel.

3.1 Base Templates

Adding shared components to multiple pages in an application is a tedious and error-prone approach. We would like to be able to specify these at one point and let pages get them from that single source. For instance, we want all our pages to use the same footer that contains the date and time but we do not want to add a footer component to every page separately. Instead, we would like to add it to a base template and extend all pages from this base template.

Add a new file named `templates/layout.html` as given in *Listing: Base template with standard footer*. Note that, the date and time label is already contained in this page (lines 11-13).

Listing Base template with standard footer.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8" />
5      <title>MovieDB - {% block title %}{% endblock %}</title>
6      <link rel="stylesheet" href="/static/style.css" />
7    </head>
8    <body>
9      {% block content %}{% endblock %}
10
11     <footer>
```

```
12        <div id="datetime">{{ current_time }}</div>
13      </footer>
14    </body>
15  </html>
```

- `{% block ... %}`
- different in every page: page title (line 5), page content (line 9)

Now the `home.html` template will extend this `layout.html` template (see *Listing: Home template extending the base template*) and it will not contain the label component for the footer anymore.

- just fill in the title (line 2) and content (lines 3-9) blocks

Listing Home template extending the base template.

```
1  {% extends "layout.html" %}
2  {% block title %}Home{% endblock %}
3  {% block content %}
4      <h1>MovieDB Homepage</h1>
5
6      <ul>
7        <li><a href="/movies">List movies</a></li>
8      </ul>
9  {% endblock %}
```

3.2 URL Generation

- writing absolute URL paths is not safe
- what if we want to change the base URL to `http://localhost:5000/moviedb/`?
- movie list page now on `http://localhost:5000/moviedb/movies`, so the link `/movies` would point to `http://localhost:5000/movies`
- generate URL for a function: `url_for`

Listing Home page template with URL generation.

```
1  {% extends "layout.html" %}
2  {% block title %}Home{% endblock %}
3  {% block content %}
4      <h1>MovieDB Homepage</h1>
5
6      <ul>
7        <li><a href="{{ url_for('movies_page') }}">List movies</a></li>
8      </ul>
9  {% endblock %}
```

- similarly, generate URL for style sheet

Listing Base template with URL generation.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8" />
5      <title>MovieDB - {% block title %}{% endblock %}</title>
6      <link rel="stylesheet"
7            href="{{ url_for('static', filename='style.css') }}" />
```

```
  </head>
  <body>
    {% block content %}{% endblock %}

    <footer>
      <div id="datetime">{{ current_time }}</div>
    </footer>
  </body>
</html>
```

3.3 Navigation

Another improvement concerns the navigation. We might need links to the home page or the movie list page from many pages in the application. So, having a global navigation mechanism where all such links will be available in all pages might be a good idea.

Add a header containing navigation links to the base template as in *Listing: Base template with navigation*.

Listing Base template with navigation.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>MovieDB - {% block title %}{% endblock %}</title>
    <link rel="stylesheet"
          href="{{ url_for('static', filename='style.css') }}" />
  </head>
  <body>
    <header>
      <nav>
        <li><a href="{{ url_for('home_page') }}">Home</a></li>
        <li><a href="{{ url_for('movies_page') }}">List movies</a></li>
      </nav>
    </header>

    {% block content %}{% endblock %}

    <footer>
      <div id="datetime">{{ current_time }}</div>
    </footer>
  </body>
</html>
```

Now the templates of our actual pages (the home page and the movie list page) don't need to add the navigation links (see *Listing: Home page template with navigation links*, lines 3-5).

Listing Home page template with navigation links.

```
{% extends "layout.html" %}
{% block title %}Home{% endblock %}
{% block content %}
    <h1>MovieDB Homepage</h1>
{% endblock %}
```

To display the navigation links side by side instead of as a list, let us add the following lines to the style sheet:

```
nav li {
  display: inline;
  margin-right: 1em;
  font-size: 80%;
}
```

3.4 Exercise

- Arrange the movie list page so that it will use the same navigation panel and footer. (*Solution*)

CHAPTER

FOUR

DATA MODEL

In this chapter, we will create the Python classes in our data model, that is, the classes for movies and movie collections. First we will implement the movie class and fill the movie list page with some in-place generated test data. Then we will implement the movie collection class which will hold some application-wide test data. The resulting movie list page will look like in *Screenshot: Movie list page populated with test data.*.

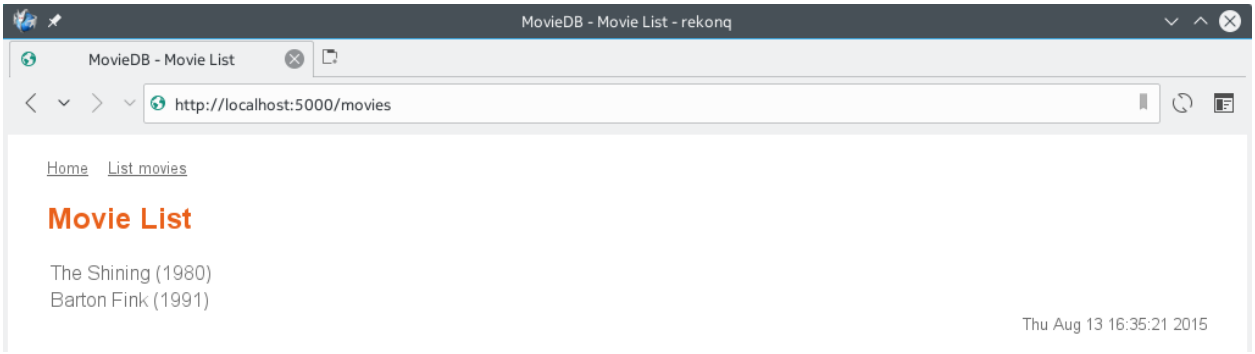

Figure 4.1: Screenshot: Movie list page populated with test data.

4.1 Movies

To represent movies in the application, we implement a `Movie` class (see *Listing: Movie class*. At first, it only has a title and a year.

Listing Movie class.

```
1 class Movie:
2     def __init__(self, title, year=None):
3         self.title = title
4         self.year = year
```

- every movie has a key (id) in the collection

4.2 Displaying Lists

To display a list of movies, we change the code for the `templates/movies.html` template as given in *Listing: Movie list template*. This code will generate a `tr` element for each movie in the `movies` list (lines 7-11). Note that, `movies` is a parameter that has to be passed by the calling function and it is assumed to be a list of `Movie` objects.

Listing Movie list template.

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
    <h2>Movie List</h2>

    <table>
      {% for key, movie in movies %}
      <tr>
        <td>{{ movie.title }} ({{ movie.year }})</td>
      </tr>
      {% endfor %}
    </table>
{% endblock %}
```

- {% for .. %}
- {% if .. %}

Listing Movie list page handler with sample movie data.

```
import datetime

from flask import Flask
from flask import render_template

from movie import Movie


app = Flask(__name__)


@app.route('/')
def home_page():
    now = datetime.datetime.now()
    return render_template('home.html', current_time=now.ctime())


@app.route('/movies')
def movies_page():
    movie1 = Movie('The Shining', year=1980)
    movie2 = Movie('Barton Fink', year=1991)
    movies = [(1, movie1), (2, movie2)]
    now = datetime.datetime.now()
    return render_template('movies.html', movies=movies,
                           current_time=now.ctime())


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Since we don't have a data source to supply us with the movie information at the moment, we generate a sample list (lines 20-22).

- line 6: import model class
- line 24: send movies as parameter to template

4.3 Movie Collections

Next, we will implement a class that will represent a movie collection. This class will contain a dictionary of movie objects and some methods to interact with that list such as adding or deleting movies (see *Listing: Movie collection class*).

- save last used key

Listing Movie collection class.

```
1  class Store:
2      def __init__(self):
3          self.movies = {}
4          self.last_key = 0
5
6      def add_movie(self, movie):
7          self.last_key += 1
8          self.movies[self.last_key] = movie
9
10     def delete_movie(self, key):
11         del self.movies[key]
12
13     def get_movie(self, key):
14         return self.movies[key]
15
16     def get_movies(self):
17         return sorted(self.movies.items())
```

- `sorted` will sort on first element of tuple (key)

Warning: Note that there is no error checking whatsoever in this code. A real example should check for existence of keys and other possible issues.

In our earlier example, the list of movies was generated by the page that lists the movies. This is obviously not the proper place to generate the collection because there should be one movie collection object which is accessible to all components in the application. Therefore, the better place to keep that list would be in the application object (see *Listing: Application containing the collection object*). So, when the application starts running, we add a collection object member to it (line 27), and provide the sample data (lines 28-29) in order to test our application.

Listing Application containing the collection object.

```
1  import datetime
2
3  from flask import Flask
4  from flask import render_template
5
6  from movie import Movie
7  from store import Store
8
9
10 app = Flask(__name__)
11
12
13 @app.route('/')
14 def home_page():
15     now = datetime.datetime.now()
16     return render_template('home.html', current_time=now.ctime())
17
```

```
18
19 @app.route('/movies')
20 def movies_page():
21     now = datetime.datetime.now()
22     movies = app.store.get_movies()
23     return render_template('movies.html', movies=movies,
24                            current_time=now.ctime())
25
26
27 if __name__ == '__main__':
28     app.store = Store()
29     app.store.add_movie(Movie('The Shining', year=1980))
30     app.store.add_movie(Movie('Barton Fink', year=1991))
31     app.run(host='0.0.0.0', port=5000)
```

Now that the `movies_page` function does not generate the list of movies, it has to retrieve them from the application (line 22).

4.4 Exercise

- (*Warning*: needs explaining parametric URLs like `/movie/1`)

  Add a page that will display a movie (see *Screenshot: Movie display page.*). Then, organize the movie list page so that the entries are links to pages that will display the selected movie (see *Screenshot: Movie list page with links to movie display pages.*). (*Solution*)

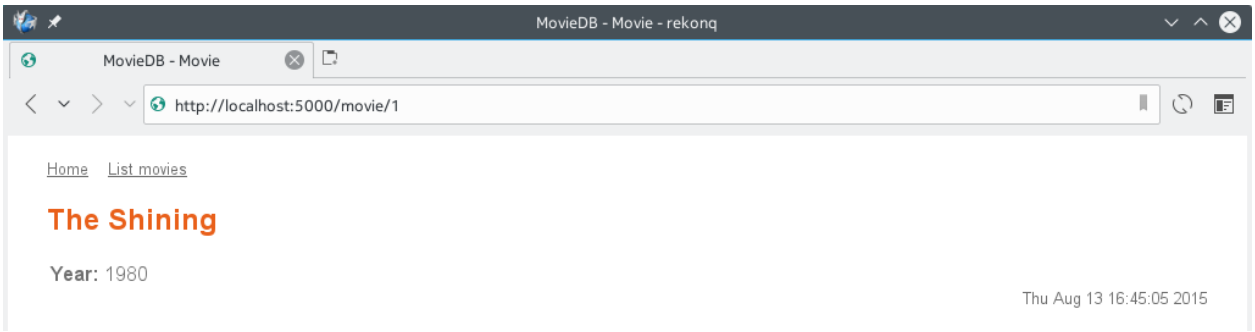

Figure 4.2: Screenshot: Movie display page.

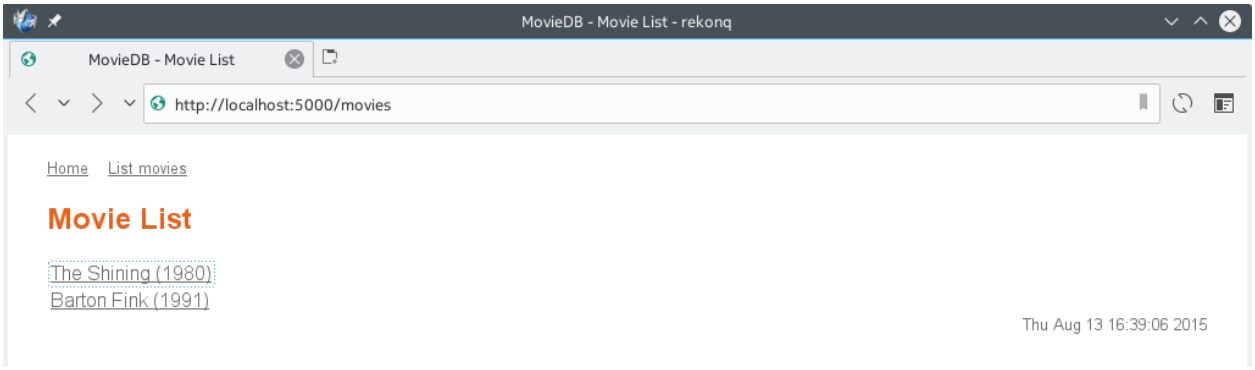

Figure 4.3: Screenshot: Movie list page with links to movie display pages.

CHAPTER

FIVE

FORMS

In this chapter, we will implement the parts that will enable us to modify the collection. This includes the operations to add a new movie and to delete an existing movie. All these operations require the use of forms containing components such as text boxes and check boxes. The resulting pages are given in *Screenshot: Movie edit page.* and *Screenshot: Movie list page with delete option.*.

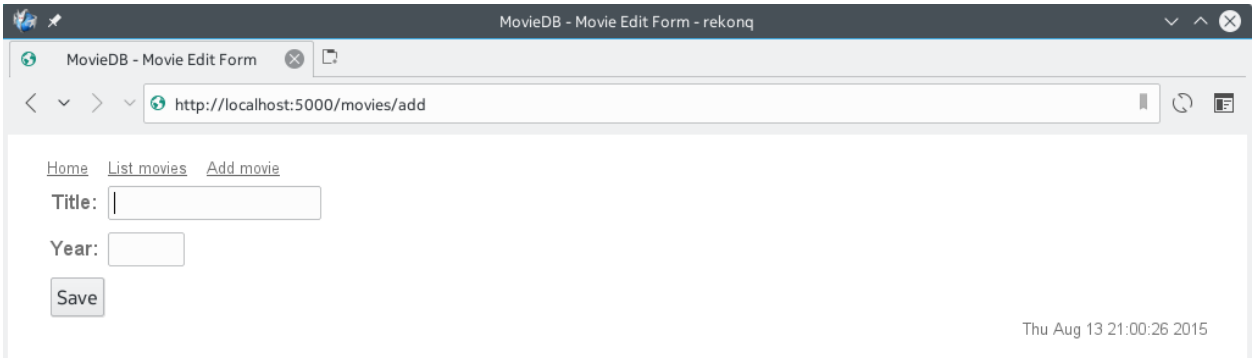

Figure 5.1: Screenshot: Movie edit page.

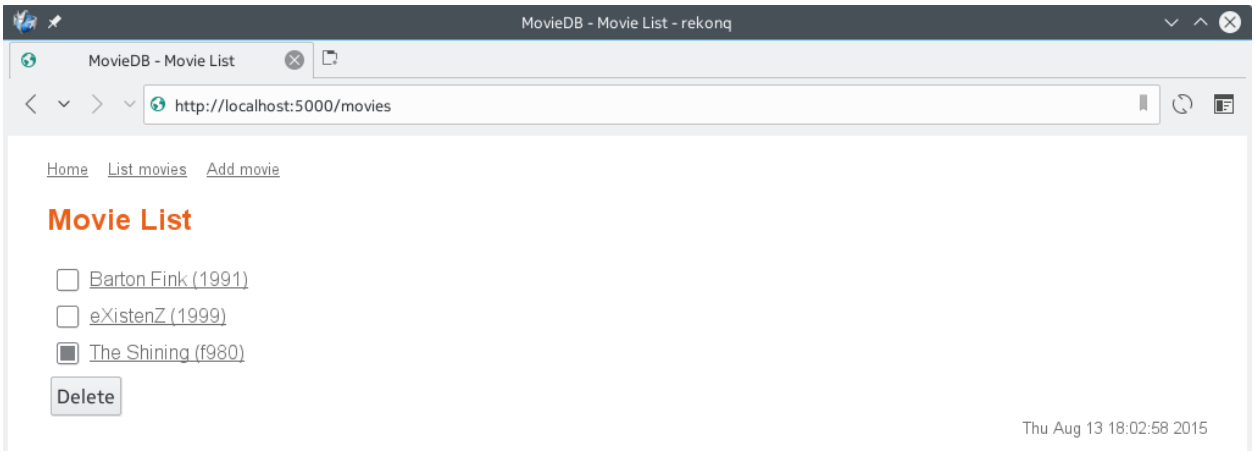

Figure 5.2: Screenshot: Movie list page with delete option.

- adding a new movie and updating an existing movie will use the same edit form

5.1 Text Boxes

First, we will add a new page to edit the data of a movie. Add an HTML template named `templates/movie_edit.html` and modify its contents as in *Listing: Page template for editing a movie*.

Listing Page template for editing a movie.

```
{% extends "layout.html" %}
{% block title %}Movie Edit Form{% endblock %}
{% block content %}
    <form action="{{ url_for('movies_page') }}" method="post">
      <table>
        <tr>
          <th>Title:</th>
          <td>
            <input type="text" name="title" required autofocus />
          </td>
        </tr>
        <tr>
          <th>Year:</th>
          <td>
            <input type="text" name="year" size="4" />
          </td>
        </tr>
      </table>
      <input value="Save" name="save" type="submit" />
    </form>
{% endblock %}
```

- action for adding a movie: post to `/movies` page

Listing Application with movie adding support.

```
import datetime

from flask import Flask
from flask import redirect
from flask import render_template
from flask import request
from flask import url_for

from movie import Movie
from store import Store


app = Flask(__name__)


@app.route('/')
def home_page():
    now = datetime.datetime.now()
    return render_template('home.html', current_time=now.ctime())


@app.route('/movies', methods=['GET', 'POST'])
def movies_page():
    if request.method == 'GET':
        now = datetime.datetime.now()
        movies = app.store.get_movies()
```

```
27          return render_template('movies.html', movies=movies,
28                                 current_time=now.ctime())
29      else:
30          title = request.form['title']
31          year = request.form['year']
32          movie = Movie(title, year)
33          app.store.add_movie(movie)
34          return redirect(url_for('movie_page', key=app.store.last_key))
35
36
37  @app.route('/movies/add')
38  def movie_edit_page():
39      now = datetime.datetime.now()
40      return render_template('movie_edit.html', current_time=now.ctime())
41
42
43  @app.route('/movie/<int:key>')
44  def movie_page(key):
45      now = datetime.datetime.now()
46      movie = app.store.get_movie(key)
47      return render_template('movie.html', movie=movie,
48                             current_time=now.ctime())
49
50
51  if __name__ == '__main__':
52      app.store = Store()
53      app.run(host='0.0.0.0', port=5000)
```

- line 22: multiple HTTP methods for same URL
- lines 24-28: get movies
- lines 29-34: add new movie
- lines 37-40: route to movie edit form

Note that, now that we can add movies to the collection, we don't need the sample movie data anymore and we can delete the relevant lines (lines 52-53).

> **Warning:** `key=app.store.last_key` on line 34 probably not a very good idea.

To make this page accessible from other pages, we add a link to the global navigation panel (*Listing: Base template with navigation to add a movie.*)

Listing Base template with navigation to add a movie.

```
1   <!DOCTYPE html>
2   <html>
3     <head>
4       <meta charset="utf-8" />
5       <title>MovieDB - {% block title %}{% endblock %}</title>
6       <link rel="stylesheet"
7             href="{{ url_for('static', filename='style.css') }}" />
8     </head>
9     <body>
10      <header>
11        <nav>
12          <li><a href="{{ url_for('home_page') }}">Home</a></li>
13          <li><a href="{{ url_for('movies_page') }}">List movies</a></li>
```

```
        <li><a href="{{ url_for('movie_edit_page') }}">Add movie</a></li>
      </nav>
    </header>

    {% block content %}{% endblock %}

    <footer>
      <div id="datetime">{{ current_time }}</div>
    </footer>
  </body>
</html>
```

5.2 Check Boxes

Our next step is to delete movies from the collection. We will change the movie list page so that there will be a check box next to every movie and a delete button at the bottom of the page. When the delete button is pressed all the checked movies will be deleted. First, we change the template for the movie list page as in *Listing: Movie list template with check boxes for entries*.

Listing Movie list template with check boxes for entries.

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
    <h2>Movie List</h2>

    <form action="{{ url_for('movies_page') }}" method="post">
      <table>
        {% for key, movie in movies %}
        <tr>
          <td><input type="checkbox" name="movies_to_delete"
                     value="{{ key }}" /></td>
          <td>
            <a href="{{ url_for('movie_page', key=key) }}">
              {{ movie.title }} ({{ movie.year }})
            </a>
          </td>
        </tr>
        {% endfor %}
      </table>
      <input type="submit" value="Delete" name="delete" />
    </form>
{% endblock %}
```

- action for deleting movie: post to `/movies` page
- if `movies_to_delete` in request, delete movies, otherwise add new movie

Listing Application code for deleting movies.

```
@app.route('/movies', methods=['GET', 'POST'])
def movies_page():
    if request.method == 'GET':
        now = datetime.datetime.now()
        movies = app.store.get_movies()
        return render_template('movies.html', movies=movies,
                               current_time=now.ctime())
```

```
    elif 'movies_to_delete' in request.form:
        keys = request.form.getlist('movies_to_delete')
        for key in keys:
            app.store.delete_movie(int(key))
        return redirect(url_for('movies_page'))
    else:
        title = request.form['title']
        year = request.form['year']
        movie = Movie(title, year)
        app.store.add_movie(movie)
        return redirect(url_for('movie_page', key=app.store.last_key))
```

5.3 Exercise

- Add a link to the movie display page which, when clicked, will take the user to the movie edit page. After saving, the movie should be updated in the collection (not added a second time). (*Solution*)

CHAPTER

SIX

DATA PERSISTENCE

A major problem with the application as implemented so far is that the data the user enters do not persist. Every time, the application starts with an empty collection and added movies are lost when the application is shut down. In this chapter we will see how to store the data in a database. This chapter has nothing to do with Wicket; it just stores the data in an SQL database and assumes that the reader knows about SQL.

To keep things simple and to avoid database installation or configuration issues, we will an SQLite database.

Create a database with the name `movies.sqlite` in your home directory and create a table in it using the following SQL statement:

```
CREATE TABLE MOVIE (
   ID INTEGER PRIMARY KEY AUTOINCREMENT,
   TITLE VARCHAR(80) NOT NULL,
   YR INTEGER
)
```

Tip: You can use the SQLite Manager add-on for Firefox to manage SQLite databases:

https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/

The methods for adding a movie, removing a movie, updating a movie, getting a movie by id, and getting all movies are given below. These are simple JDBC operations and are not within the scope of this tutorial.

```
1  import sqlite3 as dbapi2
2
3  from movie import Movie
4
5
6  class Store:
7      def __init__(self, dbfile):
8          self.dbfile = dbfile
9          self.last_key = None
10
11     def add_movie(self, movie):
12         with dbapi2.connect(self.dbfile) as connection:
13             cursor = connection.cursor()
14             query = "INSERT INTO MOVIE (TITLE, YR) VALUES (?, ?)"
15             cursor.execute(query, (movie.title, movie.year))
16             connection.commit()
17             self.last_key = cursor.lastrowid
18
19     def delete_movie(self, key):
20         with dbapi2.connect(self.dbfile) as connection:
21             cursor = connection.cursor()
22             query = "DELETE FROM MOVIE WHERE (ID = ?)"
```

```
            cursor.execute(query, (key,))
            connection.commit()

    def update_movie(self, key, title, year):
        with dbapi2.connect(self.dbfile) as connection:
            cursor = connection.cursor()
            query = "UPDATE MOVIE SET TITLE = ?, YR = ? WHERE (ID = ?)"
            cursor.execute(query, (title, year, key))
            connection.commit()

    def get_movie(self, key):
        with dbapi2.connect(self.dbfile) as connection:
            cursor = connection.cursor()
            query = "SELECT TITLE, YR FROM MOVIE WHERE (ID = ?)"
            cursor.execute(query, (key,))
            title, year = cursor.fetchone()
        return Movie(title, year)

    def get_movies(self):
        with dbapi2.connect(self.dbfile) as connection:
            cursor = connection.cursor()
            query = "SELECT ID, TITLE, YR FROM MOVIE ORDER BY ID"
            cursor.execute(query)
            movies = [(key, Movie(title, year))
                      for key, title, year in cursor]
        return movies
```

Finally, we have to change the collection object of the application. The database is assumed to be a file named `movies.sqlite` in the user's home directory.

```
if __name__ == '__main__':
    app.store = Store(os.path.join(os.getenv('HOME'), 'movies.sqlite'))
    app.run(host='0.0.0.0', port=5000)
```

CHAPTER

SEVEN

SOLUTIONS TO EXERCISES

Chapter *Basics*

Exercise Add a link from the movie list page to the home page.

templates/movies.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>MovieDB - Movie List</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h2>Movie List</h2>

    <ul>
      <li><a href="/">Home</a></li>
    </ul>
  </body>
</html>
```

Chapter *Application Structure*

Exercise Arrange the movie list page so that it will use the same navigation panel and footer.

templates/movies.html

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
    <h2>Movie List</h2>
{% endblock %}
```

server.py

```
import datetime

from flask import Flask
from flask import render_template


app = Flask(__name__)


@app.route('/')
def home_page():
```

```
12      now = datetime.datetime.now()
13      return render_template('home.html', current_time=now.ctime())
14
15
16  @app.route('/movies')
17  def movies_page():
18      now = datetime.datetime.now()
19      return render_template('movies.html', current_time=now.ctime())
20
21
22  if __name__ == '__main__':
23      app.run(host='0.0.0.0', port=5000)
```

- current time has to be sent because footer expects it (lines 18-19)

Chapter *Data Model*

Exercise Add a page that will display a movie. Then, organize the movie list page so that the entries are links to pages that will display the selected movie.

templates/movie.html

```
1  {% extends "layout.html" %}
2  {% block title %}Movie{% endblock %}
3  {% block content %}
4      <h2>{{ movie.title }}</h2>
5      <table>
6          <tr>
7              <th>Year:</th>
8              <td>{{ movie.year }}</td>
9          </tr>
10     </table>
11 {% endblock %}
```

templates/movies.html

```
1  {% extends "layout.html" %}
2  {% block title %}Movie List{% endblock %}
3  {% block content %}
4      <h2>Movie List</h2>
5
6      <table>
7        {% for key, movie in movies %}
8        <tr>
9          <td>
10           <a href="{{ url_for('movie_page', key=key) }}">
11             {{ movie.title }} ({{ movie.year }})
12           </a>
13         </td>
14       </tr>
15       {% endfor %}
16     </table>
17 {% endblock %}
```

server.py

```
1  import datetime
2
3  from flask import Flask
4  from flask import render_template
```

```

from movie import Movie
from store import Store


app = Flask(__name__)


@app.route('/')
def home_page():
    now = datetime.datetime.now()
    return render_template('home.html', current_time=now.ctime())


@app.route('/movies')
def movies_page():
    movies = app.store.get_movies()
    now = datetime.datetime.now()
    return render_template('movies.html', movies=movies,
                           current_time=now.ctime())


@app.route('/movie/<int:key>')
def movie_page(key):
    now = datetime.datetime.now()
    movie = app.store.get_movie(key)
    return render_template('movie.html', movie=movie,
                           current_time=now.ctime())



if __name__ == '__main__':
    app.store = Store()
    app.store.add_movie(Movie('The Shining', year=1980))
    app.store.add_movie(Movie('Barton Fink', year=1991))
    app.run(host='0.0.0.0', port=5000)
```

Chapter *Forms*

Exercise Add a link to the movie display page which, when clicked, will take the user to the movie edit page. After saving, the movie should be updated in the collection (not added a second time).

templates/movie.html

```
{% extends "layout.html" %}
{% block title %}Movie{% endblock %}
{% block content %}
    <h2>{{ movie.title }}</h2>
    <table>
        <tr>
            <th>Year:</th>
            <td>{{ movie.year }}</td>
        </tr>
    </table>

    <p><a href="{{ request.path + '/edit' }}">Edit</a></p>
{% endblock %}
```

templates/movie_edit.html

```
{% extends "layout.html" %}
{% block title %}Movie Edit Form{% endblock %}
{% block content %}
    {% set key = request.path.split('/')[-2] %}
    {% set action = url_for('movie_page', key=key) if movie else url_for('movies_page') %}
    {% set title = movie.title if movie else '' %}
    {% set year = movie.year if movie else '' %}
    <form action="{{ action }}" method="post">
      <table>
        <tr>
          <th>Title:</th>
          <td>
            <input type="text" name="title" value="{{ title }}"
                   required autofocus />
          </td>
        </tr>
        <tr>
          <th>Year:</th>
          <td>
            <input type="text" name="year" size="4" value="{{ year }}" />
          </td>
        </tr>
      </table>
      <input value="Save" name="save" type="submit" />
    </form>
{% endblock %}
```

server.py

```
@app.route('/movies/add')
@app.route('/movie/<int:key>/edit')
def movie_edit_page(key=None):
    movie = app.store.get_movie(key) if key is not None else None
    now = datetime.datetime.now()
    return render_template('movie_edit.html', movie=movie,
                           current_time=now.ctime())


@app.route('/movie/<int:key>', methods=['GET', 'POST'])
def movie_page(key):
    if request.method == 'GET':
        now = datetime.datetime.now()
        movie = app.store.get_movie(key)
        return render_template('movie.html', movie=movie,
                               current_time=now.ctime())
    else:
        title = request.form['title']
        year = request.form['year']
        app.store.update_movie(key, title, year)
        return redirect(url_for('movie_page', key=app.store.last_key))
```

store.py

```
class Store:
    def __init__(self):
        self.movies = {}
        self.last_key = 0

    def add_movie(self, movie):
```

```
7           self.last_key += 1
8           self.movies[self.last_key] = movie
9
10      def delete_movie(self, key):
11          del self.movies[key]
12
13      def update_movie(self, key, title, year):
14          self.movies[key].title = title
15          self.movies[key].year = year
16
17      def get_movie(self, key):
18          return self.movies[key]
19
20      def get_movies(self):
21          return sorted(self.movies.items())
```