Flask Application Development Documentation

Release 1.1

H. Turgut Uyar

Oct 14, 2016

CONTENTS

1 Basics **3**

1.1 Applications . . . 3

1.2 Dynamic Content . . . 5

1.3 Adding Links . . . 6

2 Application Structure **9**

2.1 Base Templates . . . 9

2.2 URL Generation . . . 10

2.3 Application Organization . . . 12

3 Data Model **15**

3.1 Model Classes . . . 15

3.2 Control Flow in Templates . . . 17

3.3 Parametric Routes . . . 18

4 Forms **21**

4.1 Posting Data . . . 21

4.2 Posting Lists . . . 24

4.3 Input Validation . . . 25

5 Logins **29**

5.1 Storing Passwords . . . 29

5.2 Login Management . . . 30

6 Data Persistence **37**

7 Solutions to Exercises **39**

Author H. Turgut Uyar

Formats You can read this tutorial online at the following address:

http://web.itu.edu.tr/uyar/fad/

The document is also available in the following formats:

- PDF
- Compressed HTML

You can download the code examples using the link below:

- Compressed Code

Releases v1.1 - October 2016

- Added more narrative.
- Updated examples.
- Used better application structuring practices.
- Added section on input validation.
- Added chapter on logins.

v1.0 - October 2015

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:

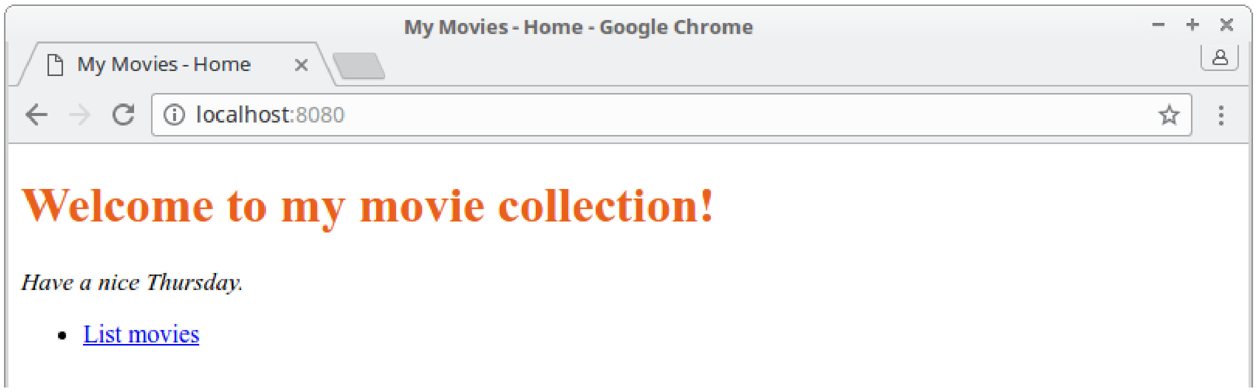
http://flask.pocoo.org/

CHAPTER

ONE

BASICS

In this tutorial, we're going to develop an application for managing a movie collection. We'll start by creating a simple home page containing only a title. Then, we'll add a greeting message based on the current date to show how dynamic content is generated. Next, we'll create another page, again containing only a title. This second page will be populated with the list of movies in later chapters. Finally, we'll provide a link from the home page to the movie list page. In the end, the home page will look as shown in Fig. 1.1.



Welcome to my movie collection!

Have a nice Thursday.

- List movies

Fig. 1.1: Home page containing a greeting message and a link to the movie list page.

Applications

Using Flask, we can create a web application in just one source file. In Listing 1.1, you can see the code for a very simple application that serves only one page with only a welcome message in it. Save this code in a file named `server.py` and start it using the command `python server.py`. Now, when you visit the address `http://localhost:8080/`, you should see the welcome message.

Listing 1.1: Application with only a welcome message. (file: `server.py`, version: `v0101`)

```python
from flask import Flask


app = Flask(__name__)


@app.route('/')
def home_page():
    return '<h1>Welcome to my movie collection!</h1>'

```

```python
11
12 if __name__ == '__main__':
13     app.run(host='0.0.0.0', port=8080, debug=True)
```

Let's go over this code line by line:

- Line 1: The `Flask` application class gets imported. Applications are instances of this class.
- Line 4: An application of this class gets instantiated.
- Line 13: The application gets started. The `host` parameter specifies that the server should be publicly available and the `port` parameter specifies that it should run on port 8080. We also set the application to run in debug mode to get better feedback about errors.

  Warning: Debug mode causes security problems. Therefore this mode should be enabled only during development.

- Lines 7-9: A function is defined to handle requests to the home page. It simply returns a string containing the welcome message. The `route` decorator registers the / URL with this function. So, whenever there is a request to the / URL, this function will be invoked and its result will be sent back to the client as the response.

Tip: To learn the conventions about indentation, spacing and other style issues, read the Python Style Guide:

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

Note that the request handler function returns a string that contains only an HTML element but not a full HTML page. We could write a full HTML string as the return value of the function as in:

```python
return '<!DOCTYPE html><html><head>...<h1>Welcome...</body></html>'
```

But this would not only make the code less readable -since it would cause a very long line or span over multiple lines-, it would also be very difficult to maintain. The better way is to use a template which will contain the HTML code. Flask looks for template files in the `templates` folder, so we can create a file named `home.html` in that folder to contain our home page template given in Listing 1.2.

Listing 1.2: Initial home page template. (file: `templates/home.html`, version: `v0102`)

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My Movies - Home</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>Welcome to my movie collection!</h1>
  </body>
</html>
```

Now, as shown in Listing 1.3, we can import a function for rendering templates (line 1) and the handler can call it to render the home page (line 9).

Listing 1.3: Application using a template. (file: `server.py`, version: `v0102`)

```python
from flask import Flask, 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=8080, debug=True)
```

Static files like images and style sheets are placed into the `static` folder. Our template (Listing 1.2) contains a style sheet reference on line 6. To make this reference work, we can create a file named `style.css` with the contents given in Listing 1.4.

Listing 1.4: Custom style sheet for application. (file: `static/style.css`, version: `v0102`)

```css
h1 {
    color: #E9601A;
}
```

Dynamic Content

Let's add a greeting message to our home page based on the current date. We first change our template to add the necessary markup (Listing 1.5, line 10).

Listing 1.5: Home page markup with dynamic greeting message. (file: `templates/home.html`, version: `v0103`)

```html
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>My Movies - Home</title>
6      <link rel="stylesheet" href="/static/style.css" />
7    </head>
8    <body>
9      <h1>Welcome to my movie collection!</h1>
10     <div class="greeting">Have a nice {{ day_name }}.</div>
11   </body>
12 </html>
```

In templates, parts between `{{` and `}}` are handled dynamically. That means, they will be replaced by the value of the expression between them. The parameters in the expression have to be supplied by the function that renders this template. In our example, the `home_page` function has to send a value for `day_name` as a parameter. Listing 1.6 shows how the handler function decides the name of the current day and sends it to the template (lines 11-13).

Listing 1.6: Handler function sending parameter. (file: `server.py`, version: `v0103`)

```python
from flask import Flask, render_template

from datetime import datetime


app = Flask(__name__)


@app.route('/')
def home_page():
    now = datetime.now()
    day = now.strftime('%A')
    return render_template('home.html', day_name=day)


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, debug=True)
```

Let's also change the look of the greeting message in the style sheet (Listing 1.7, lines 5-7).

Listing 1.7: Style sheet with greeting message style. (file: `static/style.css`, version: `v0103`)

```css
1  h1 {
2      color: #E9601A;
3  }
4
5  div.greeting {
6      font-style: italic;
7  }
```

The application, as implemented so far, runs as follows:

- An application object gets instantiated.
- This object registers the `home_page` function to respond to requests for the `/` route.
- The application starts running on the localhost on port 8080 and waits for requests.
- When a request comes for `http://localhost:8080/`, the `home_page` function gets invoked.
- This function gets the current time and renders the home page template, passing the name of the current day as a parameter to it. The server sends the rendered document back to the client.
- The client parses the response and encounters the style sheet link. It makes a second request to `http://localhost:8080/static/style.css`.
- The server sends back the `style.css` file.

Adding Links

Now we want to create a second page which will be used for listing the movies in our collection. At first, the page will only contain some static text; it will be populated in later chapters. The template for the page is given in Listing 1.8.

Listing 1.8: Initial template for movie list page. (file: `templates/movies.html`, version: `v0104`)

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My Movies - Movie List</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>Movies</h1>
  </body>
</html>
```

Our application needs a handler function to respond to requests for this page. We register the `movies_page` function to handle the `/movies` route (Listing 1.9, lines 16-18).

Listing 1.9: Application with a movie list page request handler. (file: `server.py`, version: `v0104`)

```python
1  from flask import Flask, render_template
2  
3  from datetime import datetime
4  
5  
6  app = Flask(__name__)
7  
8  
9  @app.route('/')
10 def home_page():
11     now = datetime.now()
12     day = now.strftime('%A')
13     return render_template('home.html', day_name=day)
14 
15 
16 @app.route('/movies')
17 def movies_page():
18     return render_template('movies.html')
19 
20 
21 if __name__ == '__main__':
22     app.run(host='0.0.0.0', port=8080, debug=True)
```

Our last step will be to provide a link from the home page to the movie list page. To achieve this, we just have to modify the home page markup (Listing 1.10, lines 12-14)

Listing 1.10: Home page markup with link to movie list page. (file: `templates/home.html`, version: `v0104`)

```html
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>My Movies - Home</title>
6      <link rel="stylesheet" href="/static/style.css" />
7    </head>
8    <body>
9      <h1>Welcome to my movie collection!</h1>
10     <div class="greeting">Have a nice {{ day_name }}.</div>
```

```

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

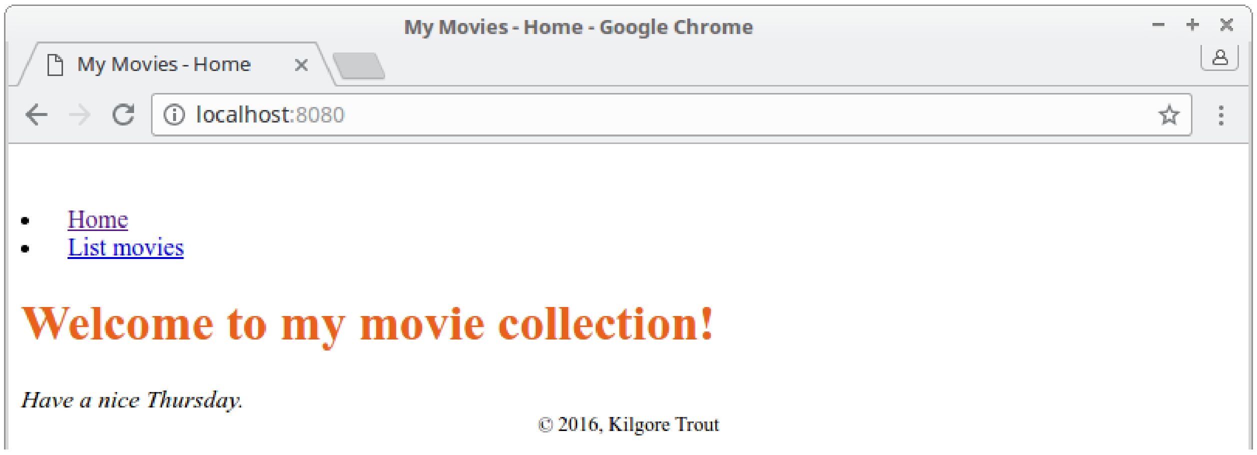
Exercise Add a link from the movie list page to the home page. (*Solution*)

CHAPTER

TWO

APPLICATION STRUCTURE

The pages in a web application share some components such as headers, footers, and navigation panels. In this chapter, we'll see how to implement such components without repeating code. We'll create a base template that will contain the components that all pages in the application will include. See Fig. 2.1 for the screenshot of the resulting home page.



- Home
- List movies

Welcome to my movie collection!

Have a nice Thursday.

© 2016, Kilgore Trout

Fig. 2.1: Home page containing navigation panel and footer.

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 a copyright notice but we don't 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.

We'll put our base template into the `layout.html` file with the contents given in Listing 2.1.

Listing 2.1: Initial base template. (file: `templates/layout.html`, version: `v0201`)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My Movies - {% block title %}{% endblock %}</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
```

```
8    <body>
9      {% block content %}{% endblock %}
10
11     <footer>
12       &copy; 2016, Kilgore Trout
13     </footer>
14   </body>
15 </html>
```

The `{% block ... %}` and `{% endblock %}` directives mark the parts that will be filled in by the pages that use this base template. Since there can be more than one such block in a template, blocks have to be named so that they can be distinguished from each other. The name is the identifier that follows the `block` keyword in the opening directive.

All remaining parts will be directly included. This means that there are two parts in this template that will be filled in by pages:

- The block named `title` marks the page title after the common prefix (line 5).
- The block named `content` marks the page content (line 9). But note that the footer is not part of the replaced block, so it will be included in all pages (lines 11-13).

Now the home page template will extend this base template (Listing 2.2, line 1). The `title` block (line 2) will replace the `title` slot in the layout, and the `content` block (lines between 3 and 10) will replace the `content` slot. When processed, this template will give the same markup as the previous one without the base template (Listing 1.10).

Listing 2.2: Home page template extending the base template. (file: `templates/home.html`, version: `v0201`)

```
1  {% extends "layout.html" %}
2  {% block title %}Home{% endblock %}
3  {% block content %}
4      <h1>Welcome to my movie collection!</h1>
5      <div class="greeting">Have a nice {{ day_name }}.</div>
6
7      <ul>
8        <li><a href="/movies">List movies</a></li>
9      </ul>
10 {% endblock %}
```

Let's also add the following rule to the style sheet to format the footer:

```
footer {
    text-align: center;
    font-size: 80%;
}
```

URL Generation

In our HTML markup we've used fixed URLs to link to pages: `/` for the home page, `/movies` for the movies page, and `/static/style.css` for the style sheet. These URLs assume that all our URLs start from the base of the server URL (`http://localhost:8080`). But in many cases -for example, if we want to run multiple applications on the same port–, we'll want our application URLs to start from a subpath on the server, such as `http://localhost:8080/mymovies`. So we would want our URLs to be `http://localhost:8080/mymovies/`, `http://localhost:8080/mymovies/movies`, and `http://localhost:8080/mymovies/static/style.css`. But using fixed URLs wouldn't work in this case because for example `/movies` would always link to `http://localhost:8080/movies`.

We can solve this issue using a Flask function called `url_for` which, given the handler function, returns the URL for its page on that application. So, for example, instead of writing the URL for the movies page as `/movies`, we call the `url_for` function and give it the handler (the `movies_page` function) to get its URL (Listing 2.3, line 8).

Listing 2.3: Home page template with URL generation. (file: `templates/home.html`, version: `v0202`)

```
1  {% extends "layout.html" %}
2  {% block title %}Home{% endblock %}
3  {% block content %}
4      <h1>Welcome to my movie collection!</h1>
5      <div class="greeting">Have a nice {{ day_name }}.</div>
6
7      <ul>
8        <li><a href="{{ url_for('movies_page') }}">List movies</a></li>
9      </ul>
10 {% endblock %}
```

For static files, we have to use the handler name `static` and give the relative path of the requested file (Listing 2.4, lines 6-7).

Listing 2.4: Base template with URL generation for static files. (file: `templates/layout.html`, version: `v0202`)

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>My Movies - {% block title %}{% endblock %}</title>
6      <link rel="stylesheet"
7            href="{{ url_for('static', filename='style.css') }}" />
8    </head>
9    <body>
10     {% block content %}{% endblock %}
11
12     <footer>
13       &copy; 2016, Kilgore Trout
14     </footer>
15   </body>
16 </html>
```

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 would be a good idea.

We'll add the navigation links as part of a header section in our base template (Listing 2.5, lines 10-15).

Listing 2.5: Base template with navigation links. (file: `templates/layout.html`, version: `v0203`)

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>My Movies - {% block title %}{% endblock %}</title>
6      <link rel="stylesheet"
7            href="{{ url_for('static', filename='style.css') }}" />
8    </head>
9    <body>
10     <header>
11       <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>
      &copy; 2016, Kilgore Trout
    </footer>
  </body>
</html>
```

Now the home page doesn't need to add the navigation links anymore (Listing 2.6).

Listing 2.6: Home page template that gets navigation links from base template. (file: `templates/home.html`, version: `v0203`)

```
--- ./code/v0202/templates/home.html
+++ ./code/v0203/templates/home.html
@@ -3,8 +3,4 @@
 {% block content %}
     <h1>Welcome to my movie collection!</h1>
     <div class="greeting">Have a nice {{ day_name }}.</div>
-
-    <ul>
-      <li><a href="{{ url_for('movies_page') }}">List movies</a></li>
-    </ul>
 {% endblock %}
```

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

Exercise Add Bootstrap to your base template and arrange your pages to use it. (*Solution*)

Application Organization

When our project gets larger, we will want to separate the source code into multiple files. In our example, it would be a good idea to move the functions that handle page requests into another file. But note that, if we take the `home_page` and `movies_page` functions into a different file and leave the global `app` variable in the original file, we will not be able to set up the routes anymore.

To solve this, we are going to use a Flask feature called "blueprints". Blueprints are similar to applications in that they can handle routes, but they need to be registered with an application before they can be used.

Let's create a `handlers.py` file to contain our handler functions (Listing 2.7). Again, note that we now route for a blueprint instead of the `app` application object (lines 9 and 16).

Listing 2.7: Module for page request handler functions. (file: `handlers.py`, version: `v0204`)

```
1  from flask import Blueprint, render_template
2
3  from datetime import datetime
4
5
6  site = Blueprint('site', __name__)
7
```

```python

@site.route('/')
def home_page():
    now = datetime.now()
    day = now.strftime('%A')
    return render_template('home.html', day_name=day)


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

Next, we have to register this blueprint with our application object (Listing 2.8, line 8). We also create the application object through a function so that the application setup will be easier to manage (lines 6-9):

Listing 2.8: Application created by a function. (file: `server.py`, version: `v0204`)

```python
1  from flask import Flask
2
3  from handlers import site
4
5
6  def create_app():
7      app = Flask(__name__)
8      app.register_blueprint(site)
9      return app
10
11
12 def main():
13     app = create_app()
14     app.run(host='0.0.0.0', port=8080, debug=True)
15
16
17 if __name__ == '__main__':
18     main()
```

Using a blueprint changes the way we generate URLs for page handler functions. Now function names have to be prefixed with the blueprint name, as in `site.home_page` instead of just `home_page`. So, the navigation links in the base layout change:

```diff
--- ./code/v0203c/templates/layout.html
+++ ./code/v0204/templates/layout.html
@@ -28,10 +28,10 @@
         <div id="navbar" class="collapse navbar-collapse">
           <ul class="nav navbar-nav">
             <li class="active">
-              <a href="{{ url_for('home_page') }}">Home</a>
+              <a href="{{ url_for('site.home_page') }}">Home</a>
             </li>
             <li>
-              <a href="{{ url_for('movies_page') }}">List movies</a>
+              <a href="{{ url_for('site.movies_page') }}">List movies</a>
             </li>
           </ul>
         </div>
```

The `url_for` function call for the style sheet doesn't change.

The Flask framework also gives us some utilities to make application configuration easier. The required settings can be placed into a Python source file and then the application can be configured from that file. For example, let us create a `settings.py` file (Listing 2.9) and put two settings about the application into it: one for controlling whether the application runs in debug mode or not, and the other for the port number the application will run on:

Listing 2.9: Settings file with debug mode and port. (file: `settings.py`, version: `v0205`)

```
DEBUG = True
PORT = 8080
```

Now let's use this file to configure the application (Listing 2.10, line 8). Any setting made in the file will be accessible through a mapping object named `app.config`. That means, you can get their values using indexing or the `get` method (lines 15-16).

Listing 2.10: Application configured through settings file. (file: `server.py`, version: `v0205`)

```
1  from flask import Flask
2  
3  from handlers import site
4  
5  
6  def create_app():
7      app = Flask(__name__)
8      app.config.from_object('settings')
9      app.register_blueprint(site)
10     return app
11 
12 
13 def main():
14     app = create_app()
15     debug = app.config['DEBUG']
16     port = app.config.get('PORT', 5000)
17     app.run(host='0.0.0.0', port=port, debug=debug)
18 
19 
20 if __name__ == '__main__':
21     main()
```

Note: If the `DEBUG` parameter is set in the settings, Flask will automatically use it in the configuration. So the `main` function can be written as:

```
def main():
    app = create_app()
    port = app.config.get('PORT', 5000)
    app.run(host='0.0.0.0', port=port)
```

CHAPTER

THREE

DATA MODEL

In this chapter, we'll create the Python classes in our data model, i.e. the classes for movies and movie collections. And then we'll populate the movie list page with some test data. The resulting page will look as in Fig. 3.1. [1]

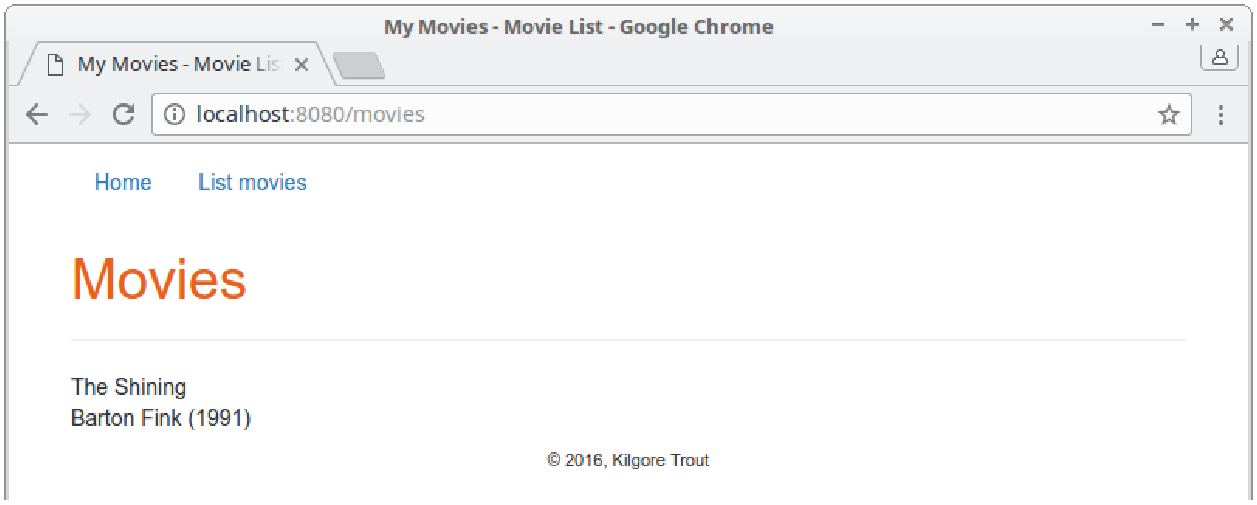


Fig. 3.1: Movie list page populated with test data.

Model Classes

To represent movies in the application, we implement a simple `Movie` class (Listing 3.1). Initially, a movie has only a title and a year.

Listing 3.1: Initial implementation of the movie model class. (file: `movie.py`, version: `v0301`)

```python
class Movie:
    def __init__(self, title, year=None):
        self.title = title
        self.year = year
```

Next, we implement a class to represent a movie collection. It contains a dictionary of movie objects and some methods for interacting with it such as for adding or deleting movies (Listing 3.2). For every movie, it generates an id value which it uses as the key into the dictionary for retrieval operations. The last used id value is stored and incremented

[1] The screenshots after this point assume that the Bootstrap exercise in the previous chapter has been completed.

whenever a new movie gets added (lines 4 and 7). The value is also set as an attribute in the movie object itself (line 9). This will help us get the id value that is assigned to a movie when it is added to the collection.

Listing 3.2: Initial implementation of the Store class. (file: `store.py`, version: `v0301`)

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

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

We want the movie collection to be an attribute of our application object to make it accessible to all components in the application (Listing 3.3). The application creation function will establish that connection through the `store` attribute (line 13). Since we don't have a data source to supply us with movie data at the moment, it also adds some sample data so that we can test the application (lines 14-15).

Listing 3.3: Application creation with data store. (file: `server.py`, version: `v0301`).

```
1  from flask import Flask
2
3  from handlers import site
4  from movie import Movie
5  from store import Store
6
7
8  def create_app():
9      app = Flask(__name__)
10     app.config.from_object('settings')
11     app.register_blueprint(site)
12
13     app.store = Store()
14     app.store.add_movie(Movie('The Shining'))
15     app.store.add_movie(Movie('Barton Fink', year=1991))
16
17     return app
18
19
20 def main():
21     app = create_app()
22     port = app.config.get('PORT', 5000)
```

```
23      app.run(host='0.0.0.0', port=port)
24
25
26  if __name__ == '__main__':
27      main()
```

Control Flow in Templates

Now that we have a movie collection in the application object, the `movies_page` handler can send it to the template to be displayed (Listing 3.4). But first we need to get the application this blueprint is registered with (line 2), and then access its collection (line 19).

Listing 3.4: Movies page handler getting data from application. (file: `handlers.py`, version: `v0301`)

```
1   from flask import Blueprint, render_template
2   from flask import current_app
3
4   from datetime import datetime
5
6
7   site = Blueprint('site', __name__)
8
9
10  @site.route('/')
11  def home_page():
12      now = datetime.now()
13      day = now.strftime('%A')
14      return render_template('home.html', day_name=day)
15
16
17  @site.route('/movies')
18  def movies_page():
19      movies = current_app.store.get_movies()
20      return render_template('movies.html', movies=sorted(movies.items()))
```

The movies are sent to the template as a sequence of pairs (line 22), where each pair consists of the movie id and the movie object itself. Since the default sorting mechanism on tuples sorts on the first element, the sequence will be sorted according to the id values.

To display the movies, we need an iteration mechanism in the movie list template (Listing 3.5). The `{% for ... %}` and `{% endfor %}` directives let us iterate over the elements of a collection and generate that markup for each element. In our example (lines 10-15), this will generate a `li` element for each movie in the `movies` list.

For every movie, we want to show the year in parentheses if the year is set, but if not, we don't want to display parentheses without content. So we use an `{% if ... %}` directive to include that markup only if the `year` attribute of the movie is not `None` (line 13). Also, we don't want to generate an empty `ul` element if there are no movies in the collection (lines 8 and 17).

Listing 3.5: Movie list template using iteration. (file: `templates/movies.html`, version: `v0301`)

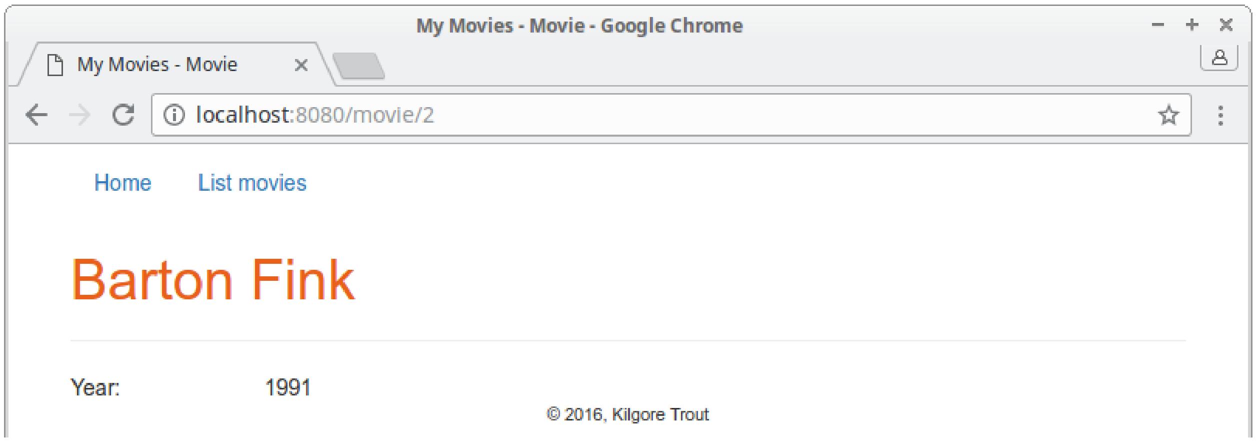
```
1   {% extends "layout.html" %}
2   {% block title %}Movie List{% endblock %}
3   {% block content %}
4         <div class="page-header">
5           <h1>Movies</h1>
```

```
      </div>

      {% if movies %}
      <ul class="list-unstyled">
        {% for movie_id, movie in movies %}
        <li>
          {{ movie.title }}
          {% if movie.year %} ({{ movie.year }}) {% endif %}
        </li>
        {% endfor %}
      </ul>
      {% endif %}
{% endblock %}
```

Parametric Routes

Next, we want to add a page that will display a movie. To identify a movie in the collection, we will use its id value as part of its URL. For example, the URL `/movie/1` will refer to the movie with id 1 in the collection. The movie page will look like in Fig. 3.2.



Barton Fink

Year: 1991

© 2016, Kilgore Trout

Fig. 3.2: The movie display page.

Let's start with the movie page template. Assuming it gets the movie object as a parameter named `movie`, the template code will be as in Listing 3.6.

Listing 3.6: Movie page template. (file: `templates/movie.html`, version: `v0302`)

```
{% extends "layout.html" %}
{% block title %}Movie{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>{{ movie.title }}</h1>
      </div>

      {% if movie.year %}
      <div class="row">
        <div class="col-sm-2">Year:</div>
        <div class="col-sm-10">{{ movie.year }}</div>
      </div>
```

```
    {% endif %}
{% endblock %}
```

The handler function (Listing 3.7, lines 23-26) has to get the movie object from the collection. To do this, it needs the id value of the movie. Since we specified the movie URL to contain the id, we can retrieve it from the route and use it to query the collection object. In Flask, routes can contain sections that will be mapped to parameters. This example states that the second part of the URL will be mapped to the `movie_id` parameter in the function after getting converted to an integer (line 23).

Listing 3.7: Movie page handler with parametric route. (file: `handlers.py`, version: `v0302`)

```
1  from flask import Blueprint, render_template
2  from flask import current_app
3  
4  from datetime import datetime
5  
6  
7  site = Blueprint('site', __name__)
8  
9  
10 @site.route('/')
11 def home_page():
12     now = datetime.now()
13     day = now.strftime('%A')
14     return render_template('home.html', day_name=day)
15 
16 
17 @site.route('/movies')
18 def movies_page():
19     movies = current_app.store.get_movies()
20     return render_template('movies.html', movies=sorted(movies.items()))
21 
22 
23 @site.route('/movie/<int:movie_id>')
24 def movie_page(movie_id):
25     movie = current_app.store.get_movie(movie_id)
26     return render_template('movie.html', movie=movie)
```

Exercise Organize the movie list page so that the entries are links to pages that will display the selected movie. *Hint*: You will need to check the documentation for the `url_for` function about handling parametric routes. (*Solution*)

CHAPTER

FOUR

FORMS

In this chapter, we'll implement the operations that will let us modify the collection, like adding a new movie and deleting an existing movie. These operations require the use of forms containing components like text boxes and check boxes. The resulting pages are given in Fig. 4.1 and Fig. 4.2.

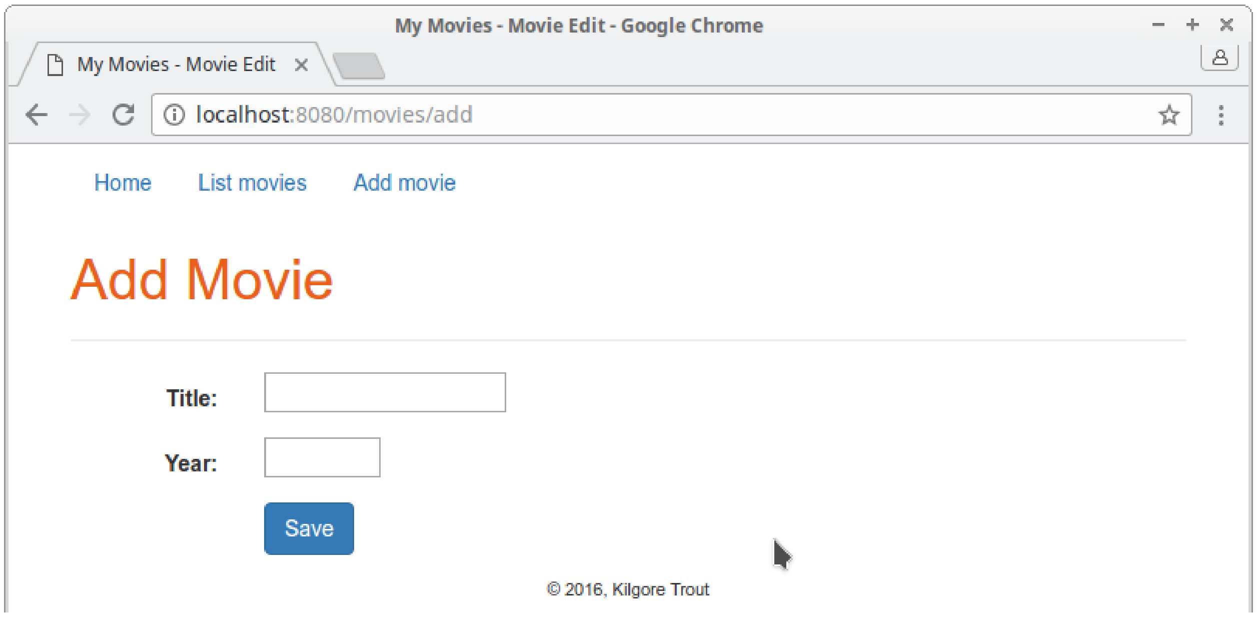


Fig. 4.1: Form for adding a movie.

Posting Data

First, we'll add a form to edit the data of a movie. Add an HTML template file named `templates/movie_edit.html` and modify its contents as in Listing 4.1. Note that this is mostly a static template, with only the minimum and maximum allowed year values taken as parameters for the year input (lines 20-21).

Listing 4.1: Form for adding a movie. (file: `templates/movie_edit.html`, version: `v0401`)

```
1 {% extends "layout.html" %}
2 {% block title %}Movie Edit{% endblock %}
3 {% block content %}
```

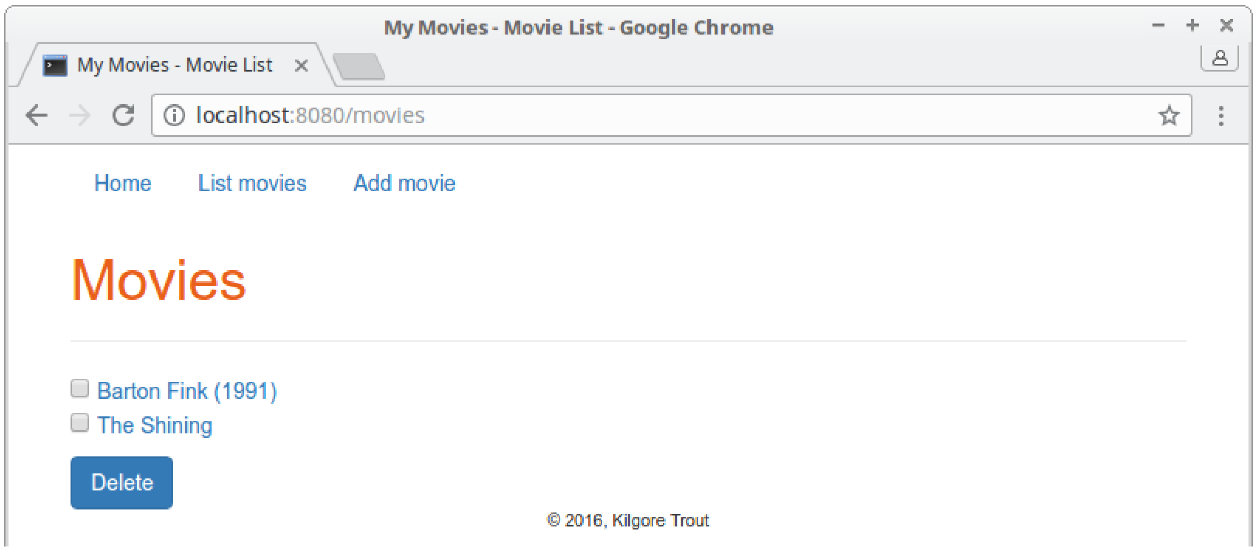


Fig. 4.2: Form for deleting movies.

```
4        <div class="page-header">
5          <h1>Edit Movie</h1>
6        </div>
7
8        <form role="form" action="" method="post" name="movie_edit"
9              class="form-horizontal">
10         <div class="form-group">
11           <label for="title" class="control-label col-sm-2">Title:</label>
12           <div class="col-sm-10">
13             <input type="text" name="title" required autofocus />
14           </div>
15         </div>
16
17         <div class="form-group">
18           <label for="year" class="control-label col-sm-2">Year:</label>
19           <div class="col-sm-10">
20             <input type="number" name="year" size="4"
21                    min="{{ min_year }}" max="{{ max_year }}" />
22           </div>
23         </div>
24
25         <div class="form-group">
26           <div class="col-sm-offset-2 col-sm-10">
27             <button type="submit" name="save"
28                     class="btn btn-primary">Save</button>
29           </div>
30         </div>
31       </form>
32 {% endblock %}
```

Since the `action` attribute of the `form` element is empty (line 8), when the submit button is clicked, the form data will be posted to the same address as the current one. Therefore, the handler for this page has to be able to handle two types of requests:

- Requests with no data. These use the HTTP GET method. The response will be an empty form for a new movie.

- Request with the data of the movie to add. These use the HTTP POST method. In this case, after the movie is added, we want to redirect the client to the display page for the added movie instead of showing a new type of page.

In order for a handler function to be able to respond to both GET and POST methods, these need to be allowed when registering the route (Listing 4.2, line 1).

Flask provides a `request` object that contains the data of the current request. We can use the `method` attribute of this object to determine whether the current request uses the GET or POST method (lines 3 and 6). If this is a GET request, the empty edit form template will be rendered (lines 4-5). If it's a POST request, a new movie will be created using the posted data (lines 7-9), and added to the collection (line 10). Afterwards, the client will be redirected to the movie display page for the newly added movie (line 11). Remember that when a movie is added to the collection (line 10), the id value the store assigns to this movie is set as its `_id` attribute (Listing 3.2, line 9) and that value is needed when rendering the movie page template (line 11).

Listing 4.2: Handler for the movie addition page. (file: `handlers.py`, version: `v0401`)

```
1  @site.route('/movies/add', methods=['GET', 'POST'])
2  def movie_add_page():
3      if request.method == 'GET':
4          return render_template('movie_edit.html',
5                                 min_year=1887, max_year=datetime.now().year)
6      else:
7          title = request.form['title']
8          year = int(request.form['year']) if request.form['year'] else None
9          movie = Movie(title, year=year)
10         current_app.store.add_movie(movie)
11         return redirect(url_for('site.movie_page', movie_id=movie._id))
```

The `request` and `redirect` names need to be imported from Flask:

```
from flask import Blueprint, redirect, render_template, url_for
from flask import current_app, request
```

Now that we can add movies to the collection, we don't need the sample movie data in the `server.py` file anymore (Listing 4.3).

Listing 4.3: Handler for the movie addition page. (file: `server.py`, version: `v0401`)

```
--- ./code/v0302b/server.py
+++ ./code/v0401/server.py
@@ -1,7 +1,6 @@
 from flask import Flask

 from handlers import site
-from movie import Movie
 from store import Store


@@ -11,8 +10,6 @@
     app.register_blueprint(site)

     app.store = Store()
-    app.store.add_movie(Movie('The Shining'))
-    app.store.add_movie(Movie('Barton Fink', year=1991))

     return app

```

To make this page accessible from other pages, we add a link to the global navigation panel (Listing 4.4, lines 9-11).

Listing 4.4: Navigation with link to movie addition page. (file: `templates/layout.html`, version: `v0401`)

```
 1 <div id="navbar" class="collapse navbar-collapse">
 2   <ul class="nav navbar-nav">
 3     <li class="active">
 4       <a href="{{ url_for('site.home_page') }}">Home</a>
 5     </li>
 6     <li>
 7       <a href="{{ url_for('site.movies_page') }}">List movies</a>
 8     </li>
 9     <li>
10       <a href="{{ url_for('site.movie_add_page') }}">Add movie</a>
11     </li>
12   </ul>
13 </div>
```

Posting Lists

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. There will also be a delete button which, when clicked, will cause all checked movies to be deleted. First, we change the template for the movie list page as in Listing 4.5.

Listing 4.5: Movie list template with check boxes for entries. (file: `templates/movies.html`, version: `v0402`)

```
 1 {% extends "layout.html" %}
 2 {% block title %}Movie List{% endblock %}
 3 {% block content %}
 4       <div class="page-header">
 5         <h1>Movies</h1>
 6       </div>
 7 
 8       {% if movies %}
 9       <form role="form" action="" method="post" name="movies"
10             class="form-horizontal">
11         <ul class="list-unstyled">
12           {% for movie_id, movie in movies %}
13           <li>
14             <input type="checkbox" name="movie_ids" value="{{ movie_id }}" />
15             <a href="{{ url_for('site.movie_page', movie_id=movie_id) }}">
16               {{ movie.title }}
17               {% if movie.year %} ({{ movie.year }}) {% endif %}
18             </a>
19           </li>
20           {% endfor %}
21         </ul>
22 
23         <button type="submit" name="delete"
24                 class="btn btn-primary">Delete</button>
25       </form>
26       {% endif %}
27 {% endblock %}
```

Since this page is now going to be used to post data to the server, the elements are placed into a form (between lines 9-10 and 25). The data will be posted to the same address, and the keys of the selected movies will be accumulated in the `movie_ids` variable (line 14). And finally, there is the `button` element (lines 23-24).

Like the `movie_add_page` handler, now the `movies_page` handler also needs to handle two types of requests. One for displaying the movies and letting the user select the movies to delete (`GET` method) and one for deleting the movies selected by the user (`POST` method).

Listing 4.6: Handler for the movie removal page. (file: `handlers.py`, version: `v0402`)

```
1  @site.route('/movies', methods=['GET', 'POST'])
2  def movies_page():
3      if request.method == 'GET':
4          movies = current_app.store.get_movies()
5          return render_template('movies.html', movies=sorted(movies.items()))
6      else:
7          movie_ids = request.form.getlist('movie_ids')
8          for movie_id in movie_ids:
9              current_app.store.delete_movie(int(movie_id))
10         return redirect(url_for('site.movies_page'))
```

When posting a collection of data, we have to use the `getlist` method (Listing 4.6, line 7). In this example, the ids of the selected movies selected will be packed into a list and assigned to the `movie_ids` variable.

Exercise Add a link to the movie display page which, when clicked, will take the user to the movie edit page. The URL of the edit page has to have the extra path `/edit` after the movie display page URL. That means, for example, the edit page of movie with key 1 has to be `/movie/1/edit`. After saving, the movie has to be updated in the collection (not added a second time). (*Solution*)

Input Validation

A very important aspect of web application development is making sure that the data received from the outside world contains sensible values. This not only makes the application more robust with respect to user errors, it's also crucial for preventing security issues. Any received data, such as values supplied by users, or values read from databases must be validated.

At the moment, when editing movies, our handler functions assume that the received movie data is valid. The HTML markup defines constraints for the inputs to enforce validity:

1. The `title` input is marked as required (Listing 4.1, line 13). So if the user leaves this field blank, the browser will issue an error message.
2. The `year` input has the type `number` with minimum and maximum allowed values (lines 20-21). So the browser won't allow the user to send non-numeric symbols, or any value that's not within this range.

But these mechanisms are not sufficient:

1. Since the `title` input is required it can't be left blank, but a title consisting of only whitespace is accepted.
2. Even worse, an attacker can bypass this form and post any arbitrary data to the corresponding URL directly; therefore circumventing any rule defined in the HTML markup. A non-numeric value for year can be sent by issuing the following command from the command line:

```
$ curl --data "title=Grease&year=abc" http://localhost:8080/movies/add
...
ValueError: invalid literal for int() with base 10: 'abc'
```

To solve these problems, we'll start by adding a function to handle validation (Listing 4.7). This function will return whether there is any errors in the inputs. It takes the `form` data mapping as input and attaches two attributes to it: one for data and one for the error messages (lines 2-3). For every input, it either adds an entry into the error mapping if the value is not validated (as in line 6 for title), or it adds an entry into the data mapping (as in lines 8 for title). When adding the data entry, it also does type conversion for the year input (lines 15 and 19). If all data is valid, errors will be empty.

Listing 4.7: Movie data validator function. (file: `handlers.py`, version: `v0403`)

```
 1  def validate_movie_data(form):
 2      form.data = {}
 3      form.errors = {}
 4
 5      if len(form['title'].strip()) == 0:
 6          form.errors['title'] = 'Title can not be blank.'
 7      else:
 8          form.data['title'] = form['title']
 9
10      if not form['year']:
11          form.data['year'] = None
12      elif not form['year'].isdigit():
13          form.errors['year'] = 'Year must consist of digits only.'
14      else:
15          year = int(form['year'])
16          if (year < 1887) or (year > datetime.now().year):
17              form.errors['year'] = 'Year not in valid range.'
18          else:
19              form.data['year'] = year
20
21      return len(form.errors) == 0
```

Next, our movie addition page handler will call this function to validate the form it received (Listing 4.8, line 6). If the inputs are valid, the data for the new movie can now be taken from the `data` mapping (lines 8-9) that was attached to the form by the validator. If the validator returns errors, the handler renders the movie edit form again (lines 14-15). By passing `request.form` to the template (through line 13), it sets the default values in the form so that the user won't have to enter all fields again. Note that, in the `GET` method case, the code follows the lines 4 and 14, doing the same operations as before.

Listing 4.8: Movie addition with validation. (file: `handlers.py`, version: `v0403`)

```
 1  @site.route('/movies/add', methods=['GET', 'POST'])
 2  def movie_add_page():
 3      if request.method == 'GET':
 4          form = {'title': '', 'year': ''}
 5      else:
 6          valid = validate_movie_data(request.form)
 7          if valid:
 8              title = request.form.data['title']
 9              year = request.form.data['year']
10              movie = Movie(title, year=year)
11              current_app.store.add_movie(movie)
12              return redirect(url_for('site.movie_page', movie_id=movie._id))
13          form = request.form
14      return render_template('movie_edit.html', form=form,
15                                 min_year=1887, max_year=datetime.now().year)
```

Also remember that the validator attaches error messages to its form parameter and these messages can now be displayed when the form is re-rendered after an invalid submission (Listing 4.9, lines 15-19 and 29-33).

Listing 4.9: Movie edit form with error messages. (file: `templates/movie_edit.html`, version: `v0403`)

```
{% extends "layout.html" %}
{% block title %}Movie Edit{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>Edit Movie</h1>
      </div>

      <form role="form" action="" method="post" name="movie_edit"
            class="form-horizontal">
        <div class="form-group">
          <label for="title" class="control-label col-sm-2">Title:</label>
          <div class="col-sm-10">
            <input type="text" name="title" required autofocus
                   value="{{ form['title'] }}" />
            {% if 'title' in form.errors %}
            <div class="alert alert-danger">
                {{ form.errors['title'] }}
            </div>
            {% endif %}
          </div>
        </div>

        <div class="form-group">
          <label for="year" class="control-label col-sm-2">Year:</label>
          <div class="col-sm-10">
            <input type="number" name="year" size="4"
                   min="{{ min_year }}" max="{{ max_year }}"
                   value="{{ form['year'] }}" />
            {% if 'year' in form.errors %}
            <div class="alert alert-danger">
                {{ form.errors['year'] }}
            </div>
            {% endif %}
          </div>
        </div>

        <div class="form-group">
          <div class="col-sm-offset-2 col-sm-10">
            <button type="submit" name="save"
                    class="btn btn-primary">Save</button>
          </div>
        </div>
      </form>
{% endblock %}
```

And similar arrangements have to be made in the movie editing handler (Listing 4.10).

Listing 4.10: Movie editing with validation. (file: `handlers.py`, version: `v0403`)

```
@site.route('/movie/<int:movie_id>/edit', methods=['GET', 'POST'])
def movie_edit_page(movie_id):
    movie = current_app.store.get_movie(movie_id)
    if request.method == 'GET':
        form = {'title': movie.title,
                'year': movie.year if movie.year else ''}
    else:
        valid = validate_movie_data(request.form)
        if valid:
            movie.title = request.form.data['title']
            movie.year = request.form.data['year']
            current_app.store.update_movie(movie)
            return redirect(url_for('site.movie_page', movie_id=movie._id))
        form = request.form
    return render_template('movie_edit.html', form=form,
                           min_year=1887, max_year=datetime.now().year)
```

Exercise Replace the input validation mechanism with an implementation that uses the Flask-WTF plugin. Also make use of its CSRF protection features. For handling `number` type inputs, use the wtforms_components package. (*Solution*)

CHAPTER

FIVE

LOGINS

As our next step, we want to make sure that only authenticated users will be able to modify the collection. So we'll need a way of logging users in to and out of our site. We'll create two users: the `normaluser` (password `normaluserpw`) will be able to edit existing movies, but only the `admin` (password `adminpw`) user will be able to add or delete movies.

Storing Passwords

Applications should never store passwords in plain text. The correct way to store a password is after hashing it. The passlib package provides the necessary tools for storing passwords securely. After installing it, you can hash a password as shown in an example Python session below:

```
>>> from passlib.apps import custom_app_context as pwd_context
>>> password = 'adminpw'
>>> hashed = pwd_context.encrypt(password)
>>> hashed
'$6$rounds=702382$2wkwLY3/D2qLgX7K$tAQC6L3G8D/GWHzx5FQIP/4.LtSNT/
↪q5KG0avCcf0MKsRucwUXCW.nhmjkpkVaSCbvb1QOQJz321fKZ0DxSJP/'
```

Later, when a user supplies a password, you can check whether it's the correct password or not by verifying it against the hashed value:

```
>>> pwd_context.verify('admin_pw', hashed)
False
>>> pwd_context.verify('adminpw', hashed)
True
```

Although the proper place to store hashed passwords would be a file or a database, for this example we're going to store them in the settings file as a map from usernames to passwords (Listing 5.1). The `ADMIN_USERS` setting lists the usernames who have administrative privileges.

Listing 5.1: Setting for passwords. (file: `settings.py`, version: `v0501`)

```
DEBUG = True
PORT = 8080
SECRET_KEY = 'secret'
WTF_CSRF_ENABLED = True

PASSWORDS = {
    'admin': '$6$rounds=702382$2wkwLY3/D2qLgX7K$tAQC6L3G8D/GWHzx5FQIP/4.LtSNT/
↪q5KG0avCcf0MKsRucwUXCW.nhmjkpkVaSCbvb1QOQJz321fKZ0DxSJP/',
    'normaluser': '$6$rounds=627096$AscnqlDtN2bWotwE
↪$2s38G2w3xupwFf7woYv8XTWvwYc9sBk7t0reSU0VLZhXQCd6FFjlkysWpy8eYL06SQGcMgFQfvuP2XRB/
↪BeTb.'
}

ADMIN_USERS = ['admin']
```

Login Management

For handling login management, we'll use the Flask-Login plugin. This plugin requires us to implement a class for representing users on our site (Listing 5.2). Our user data basically consists of a username and a password. Flask-Login assumes that there will be a unique string value for identifying each user. In most cases, this will be the string value of the database id number for a user. In our example, we will use the username for this purpose. For details about how Flask-Login represents users, check out its documentation.

Listing 5.2: Model class for users. (file: `user.py`, version: `v0501`)

```
1   from flask import current_app
2   from flask_login import UserMixin
3
4
5   class User(UserMixin):
6       def __init__(self, username, password):
7           self.username = username
8           self.password = password
9           self.active = True
10          self.is_admin = False
11
12      def get_id(self):
13          return self.username
14
15      @property
16      def is_active(self):
17          return self.active
18
19
20  def get_user(user_id):
21      password = current_app.config['PASSWORDS'].get(user_id)
22      user = User(user_id, password) if password else None
23      if user is not None:
24          user.is_admin = user.username in current_app.config['ADMIN_USERS']
25      return user
```

We also implement a function that given a user's id, returns the user object associated with that id (lines 20-25). It first checks whether there is a entry in the passwords map, and if so, creates the user object (lines 21-22). It also sets the `is_admin` property if the user is listed in the `ADMIN_USERS` settings (line 24).

To incorporate Flask-Login, we have to instantiate a `LoginManager` (lines 2 and 9) and add it to our application (line 22). If a visitor makes a request to a protected page without logging in, we can redirect the request to a login page by setting the `login_view` property of the login manager (line 23). And finally, a user loader function will be responsible for creating a user object from the user id in the session (lines 6 and 12-14).

Listing 5.3: Application with login manager. (file: `server.py`, version: `v0501`)

```
1   from flask import Flask
2   from flask_login import LoginManager
3
4   from handlers import site
```

```python
from store import Store
from user import get_user


lm = LoginManager()


@lm.user_loader
def load_user(user_id):
    return get_user(user_id)


def create_app():
    app = Flask(__name__)
    app.config.from_object('settings')
    app.register_blueprint(site)

    lm.init_app(app)
    lm.login_view = 'site.login_page'

    app.store = Store()

    return app


def main():
    app = create_app()
    port = app.config.get('PORT', 5000)
    app.run(host='0.0.0.0', port=port)


if __name__ == '__main__':
    main()
```

Now, we can simply protect a page by adding the `login_required` decorator to its handler. For example, to make sure that only authenticated users can edit movie data we can protect the `movie_edit_page` handler (Listing 5.4, line 2).

Listing 5.4: Protecting the movie edit page. (file: `handlers.py`, version: `v0501`)

```python
1  @site.route('/movie/<int:movie_id>/edit', methods=['GET', 'POST'])
2  @login_required
3  def movie_edit_page(movie_id):
4      movie = current_app.store.get_movie(movie_id)
5      form = MovieEditForm()
6      if form.validate_on_submit():
7          movie.title = form.data['title']
8          movie.year = form.data['year']
9          current_app.store.update_movie(movie)
10         flash('Movie data updated.')
11         return redirect(url_for('site.movie_page', movie_id=movie._id))
12     form.title.data = movie.title
13     form.year.data = movie.year if movie.year else ''
14     return render_template('movie_edit.html', form=form)
```

To reflect this protection in the template and not show the edit button to anonymous visitors, we can organize the movie display template as in Listing 5.5 (lines 15 and 22).

Listing 5.5: Showing the edit button only to authenticated users. (file: `templates/movie.html`, version: `v0501`)

```
{% extends "layout.html" %}
{% block title %}Movie{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>{{ movie.title }}</h1>
      </div>

      {% if movie.year %}
      <div class="row">
        <div class="col-sm-2">Year:</div>
        <div class="col-sm-10">{{ movie.year }}</div>
      </div>
      {% endif %}

      {% if current_user.is_authenticated %}
      <div class="row">
        <div class="col-sm-2"></div>
        <div class="col-sm-10">
          <a href="{{ request.path }}/edit" class="btn btn-primary">Edit</a>
        </div>
      </div>
      {% endif %}
{% endblock %}
```

Another feature provided by Flask is to easily display messages to logged-in users. The `flash` function registers a message that the user will see on the next page (Listing 5.4, line 10). To display these messages, we have to add the necessary markup to the base layout template (Listing 5.9, lines 64-66).

To prevent the normal user from adding or deleting movies, we can also add user role based protections to parts of the handlers (Listing 5.6, lines 7-8 and 19-20). In this example, the `abort` function will cause an "Unauthorized" error due to the 401 HTTP status code.

Listing 5.6: Protecting the movie add and delete operations. (file: `handlers.py`, version: `v0501`)

```
1  @site.route('/movies', methods=['GET', 'POST'])
2  def movies_page():
3      if request.method == 'GET':
4          movies = current_app.store.get_movies()
5          return render_template('movies.html', movies=sorted(movies.items()))
6      else:
7          if not current_user.is_admin:
8              abort(401)
9          movie_ids = request.form.getlist('movie_ids')
10         for movie_id in movie_ids:
11             current_app.store.delete_movie(int(movie_id))
12         flash('%d movies deleted.' % len(movie_ids))
13         return redirect(url_for('site.movies_page'))
14
15
16 @site.route('/movies/add', methods=['GET', 'POST'])
17 @login_required
18 def movie_add_page():
19     if not current_user.is_admin:
20         abort(401)
21     form = MovieEditForm()
```

```
    if form.validate_on_submit():
        title = form.data['title']
        year = form.data['year']
        movie = Movie(title, year=year)
        current_app.store.add_movie(movie)
        flash('Movie added.')
        return redirect(url_for('site.movie_page', movie_id=movie._id))
    return render_template('movie_edit.html', form=form)
```

We could also write the condition based on the username, as shown below. But role based protections are easier to maintain.

```
if not current_user.username == 'admin':
    abort(401)
```

Again, we can reflect the changes in the template (Listing 5.7). Here, we're not only hiding the delete button from non-administrative users (lines 25-28), we're also hiding the check boxes (lines 14-16) because they wouldn't make sense without a button.

Listing 5.7: Protecting the delete operation. (file: `templates/movies.html`, version: `v0501`)

```
1  {% extends "layout.html" %}
2  {% block title %}Movie List{% endblock %}
3  {% block content %}
4        <div class="page-header">
5          <h1>Movies</h1>
6        </div>
7
8        {% if movies %}
9        <form role="form" action="" method="post" name="movies"
10             class="form-horizontal">
11         <ul class="list-unstyled">
12           {% for movie_id, movie in movies %}
13           <li>
14             {% if current_user.is_admin %}
15             <input type="checkbox" name="movie_ids" value="{{ movie_id }}" />
16             {% endif %}
17             <a href="{{ url_for('site.movie_page', movie_id=movie_id) }}">
18               {{ movie.title }}
19               {% if movie.year %} ({{ movie.year }}) {% endif %}
20             </a>
21           </li>
22           {% endfor %}
23         </ul>
24
25         {% if current_user.is_admin %}
26         <button type="submit" name="delete"
27                 class="btn btn-primary">Delete</button>
28         {% endif %}
29       </form>
30       {% endif %}
31 {% endblock %}
```

Next, we'll add the login page template (Listing 5.8). Note that this template requires an extra style sheet (lines 3-4), so we'll put it into an extra header section (lines 2 and 5) that we define in the base layout (Listing 5.9, line 14).

Listing 5.8: Login page template. (file: `templates/login.html`, version: `v0501`)

```
{% extends "layout.html" %}
{% block extra_head %}
    <link rel="stylesheet"
          href="{{ url_for('static', filename='signin.css') }}" />
{% endblock %}
{% block content %}
      <form role="form" action="" method="post" name="login"
            class="form-signin">
        {{ form.csrf_token }}

        <h2 class="form-signin-heading">Sign in</h2>

        <div class="form-group">
          <label for="login-username" class="sr-only">Username:</label>
          {{ form.username(required=True, autofocus=True,
                           id='login-username', class='form-control',
                           placeholder='your username') }}
          {% for error in form.username.errors %}
            <span class="error">{{ error }}</span>
          {% endfor %}
        </div>

        <div class="form-group">
          <label for="login-password" class="sr-only">Password:</label>
          {{ form.password(id='login-password', class='form-control',
                           placeholder='your password') }}
          {% for error in form.password.errors %}
            <span class="error">{{ error }}</span>
          {% endfor %}
        </div>

        <button type="submit"
                class="btn btn-lg btn-primary btn-block">Log in</button>
      </form>
{% endblock %}
```

The full base layout template is given in Listing 5.9. Note the following changes:

- If the visitor is not logged in, a login link is displayed (lines 44-45).
- If the visitor is logged in, the username is displayed along with a logout link (lines 46-57).
- The menu item for adding movie is only shown to administrative users (lines 37-41).

Listing 5.9: Base layout with flashed messages. (file: `templates/layout.html`, version: `v0501`)

```
1   <!DOCTYPE html>
2   <html lang="en">
3     <head>
4       <meta charset="utf-8" />
5       <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6       <meta name="viewport" content="width=device-width, initial-scale=1" />
7
8       <title>My Movies - {% block title %}{% endblock %}</title>
9
10      <link rel="stylesheet"
```

```
          href="{{ url_for('static', filename='bootstrap.min.css') }}" />
    <link rel="stylesheet"
          href="{{ url_for('static', filename='style.css') }}" />
    {% block extra_head %}{% endblock %}
  </head>
  <body>
    <nav class="navbar navbar-fixed-top">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed"
                  data-toggle="collapse" data-target="#navbar"
                  aria-expanded="false" aria-controls="navbar">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
            <li class="active">
              <a href="{{ url_for('site.home_page') }}">Home</a>
            </li>
            <li>
              <a href="{{ url_for('site.movies_page') }}">List movies</a>
            </li>
            {% if current_user.is_admin %}
            <li>
              <a href="{{ url_for('site.movie_add_page') }}">Add movie</a>
            </li>
            {% endif %}
          </ul>
          <ul class="nav navbar-nav navbar-right">
            {% if not current_user.is_authenticated %}
            <li><a href="{{ url_for('site.login_page') }}">Log in</a></li>
            {% else %}
            <li class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown"
                 role="button" aria-expanded="false">
                 {{ current_user.username }} <span class="caret"></span>
              </a>
              <ul class="dropdown-menu" role="menu">
                <li>
                  <a href="{{ url_for('site.logout_page') }}">Log out</a>
                </li>
              </ul>
            </li>
            {% endif %}
          </ul>
        </div>
      </div>
    </nav>

    {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
    {% endfor %}

    <div class="container" role="main">
```

```
      {% block content %}{% endblock %}
    </div>

    <footer>
      &copy; 2016, Kilgore Trout
    </footer>

    <script src="{{ url_for('static', filename='jquery.min.js') }}"></script>
    <script src="{{ url_for('static', filename='bootstrap.min.js') }}"></script>
  </body>
</html>
```

Finally, the handlers for the login and logout pages are given in Listing 5.10. The supplied username will be used to get a user object from the registered users (lines 5-6). If such a user is found, it will have a `password` attribute which can then be checked against the password sent by the user (lines 8-9). The redirection to the next page (line 12) is needed to handle the cases where the user will be automatically redirected on login protected pages. For example, if an anonymous user visits the `/movies/add` page, he/she will be redirected to the login page (because of the `login_view` setting, Listing 5.3, line 23), and after successfully logging in, this part will redirect the user back to the movie addition page.

Listing 5.10: Handlers for login and logout pages. (file: `handlers.py`, version: `v0501`)

```
1  @site.route('/login', methods=['GET', 'POST'])
2  def login_page():
3      form = LoginForm()
4      if form.validate_on_submit():
5          username = form.data['username']
6          user = get_user(username)
7          if user is not None:
8              password = form.data['password']
9              if pwd_context.verify(password, user.password):
10                 login_user(user)
11                 flash('You have logged in.')
12                 next_page = request.args.get('next', url_for('site.home_page'))
13                 return redirect(next_page)
14         flash('Invalid credentials.')
15     return render_template('login.html', form=form)
16
17
18 @site.route('/logout')
19 def logout_page():
20     logout_user()
21     flash('You have logged out.')
22     return redirect(url_for('site.home_page'))
```

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.

```
import sqlite3 as dbapi2

from movie import Movie


class Store:
    def __init__(self, dbfile):
        self.dbfile = dbfile
        self.last_key = None

    def add_movie(self, movie):
        with dbapi2.connect(self.dbfile) as connection:
            cursor = connection.cursor()
            query = "INSERT INTO MOVIE (TITLE, YR) VALUES (?, ?)"
            cursor.execute(query, (movie.title, movie.year))
            connection.commit()
            self.last_key = cursor.lastrowid

    def delete_movie(self, key):
        with dbapi2.connect(self.dbfile) as connection:
```

```
21             cursor = connection.cursor()
22             query = "DELETE FROM MOVIE WHERE (ID = ?)"
23             cursor.execute(query, (key,))
24             connection.commit()
25 
26     def update_movie(self, key, title, year):
27         with dbapi2.connect(self.dbfile) as connection:
28             cursor = connection.cursor()
29             query = "UPDATE MOVIE SET TITLE = ?, YR = ? WHERE (ID = ?)"
30             cursor.execute(query, (title, year, key))
31             connection.commit()
32 
33     def get_movie(self, key):
34         with dbapi2.connect(self.dbfile) as connection:
35             cursor = connection.cursor()
36             query = "SELECT TITLE, YR FROM MOVIE WHERE (ID = ?)"
37             cursor.execute(query, (key,))
38             title, year = cursor.fetchone()
39         return Movie(title, year)
40 
41     def get_movies(self):
42         with dbapi2.connect(self.dbfile) as connection:
43             cursor = connection.cursor()
44             query = "SELECT ID, TITLE, YR FROM MOVIE ORDER BY ID"
45             cursor.execute(query)
46             movies = [(key, Movie(title, year))
47                           for key, title, year in cursor]
48         return movies
```

- note the list comprehension (lines 46-47)

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, debug=True)
```

CHAPTER

SEVEN

SOLUTIONS TO EXERCISES

Chapter *Basics*

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

Listing 7.1: Movie list template with link to the home page (file: `templates/movies.html`, version: `v0104b`).

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My Movies - Movie List</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>Movies</h1>

    <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.

Listing 7.2: Movie list template with navigation and footer. (file: `templates/movies.html`, version: `v0203b`)

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
    <h1>Movies</h1>
{% endblock %}
```

Exercise Add Bootstrap to your base template and arrange your pages to use it.

Download `bootstrap.min.css`, `bootstrap.min.js`, and `jquery.min.css`, and put them into the `static` folder.

Listing 7.3: Base template using Bootstrap. (file: `templates/layout.html`, version: `v0203c`)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <title>My Movies - {% block title %}{% endblock %}</title>

    <link rel="stylesheet"
          href="{{ url_for('static', filename='bootstrap.min.css') }}" />
    <link rel="stylesheet"
          href="{{ url_for('static', filename='style.css') }}" />
  </head>
  <body>
    <nav class="navbar navbar-fixed-top">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed"
                  data-toggle="collapse" data-target="#navbar"
                  aria-expanded="false" aria-controls="navbar">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
            <li class="active">
              <a href="{{ url_for('home_page') }}">Home</a>
            </li>
            <li>
              <a href="{{ url_for('movies_page') }}">List movies</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>

    <div class="container" role="main">
      {% block content %}{% endblock %}
    </div>

    <footer>
      &copy; 2016, Kilgore Trout
    </footer>

    <script src="{{ url_for('static', filename='jquery.min.js') }}"></script>
    <script src="{{ url_for('static', filename='bootstrap.min.js') }}"></script>
  </body>
</html>
```

Listing 7.4: Home page template using Bootstrap. (file: `templates/home.html`, version: `v0203c`)

```
{% extends "layout.html" %}
{% block title %}Home{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>Welcome to my movie collection!</h1>
      </div>
      <div class="greeting">Have a nice {{ day_name }}.</div>
{% endblock %}
```

Listing 7.5: Movie list page template using Bootstrap. (file: `templates/movies.html`, version: `v0203c`)

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>Movies</h1>
      </div>
{% endblock %}
```

Listing 7.6: Style sheet adjustment for Bootstrap. (file: `static/style.css`, version: `v0203c`)

```
body {
    padding-top: 2em;
}

h1 {
    color: #E9601A;
}

div.greeting {
    font-style: italic;
}

footer {
    text-align: center;
    font-size: 80%;
}
```

Chapter *Data Model*

Exercise Organize the movie list page so that the entries are links to pages that will display the selected movie.

Listing 7.7: Movie list page template with links to movie pages. (file: `templates/movies.html`, version: `v0302b`)

```
{% extends "layout.html" %}
{% block title %}Movie List{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>Movies</h1>
      </div>

      {% if movies %}
      <ul class="list-unstyled">
        {% for movie_id, movie in movies %}
        <li>
          <a href="{{ url_for('site.movie_page', movie_id=movie_id) }}">
            {{ movie.title }}
            {% if movie.year %} ({{ movie.year }}) {% endif %}
          </a>
        </li>
        {% endfor %}
      </ul>
```

```
    {% endif %}
{% endblock %}
```

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 has to be updated in the collection (not added a second time).

Listing 7.8: Movie display template with link to edit page. (file: `templates/movie.html`, version: `v0402b`)

```
{% extends "layout.html" %}
{% block title %}Movie{% endblock %}
{% block content %}
    <div class="page-header">
      <h1>{{ movie.title }}</h1>
    </div>

    {% if movie.year %}
    <div class="row">
      <div class="col-sm-2">Year:</div>
      <div class="col-sm-10">{{ movie.year }}</div>
    </div>
    {% endif %}

    <div class="row">
      <div class="col-sm-2"></div>
      <div class="col-sm-10">
        <a href="{{ request.path }}/edit" class="btn btn-primary">Edit</a>
      </div>
    </div>
{% endblock %}
```

Listing 7.9: Movie edit form with default values. (file: `templates/movie_edit.html`, version: `v0402b`)

```
{% extends "layout.html" %}
{% block title %}Movie Edit{% endblock %}
{% block content %}
    <div class="page-header">
      <h1>Edit Movie</h1>
    </div>

    <form role="form" action="" method="post" name="movie_edit"
          class="form-horizontal">
      <div class="form-group">
        <label for="title" class="control-label col-sm-2">Title:</label>
        <div class="col-sm-10">
          <input type="text" name="title" required autofocus
                 value="{{ form['title'] }}" />
        </div>
      </div>

      <div class="form-group">
        <label for="year" class="control-label col-sm-2">Year:</label>
        <div class="col-sm-10">
          <input type="number" name="year" size="4"
                 min="{{ min_year }}" max="{{ max_year }}"
                 value="{{ form['year'] }}" />
          </div>
        </div>

        <div class="form-group">
          <div class="col-sm-offset-2 col-sm-10">
            <button type="submit" name="save"
                    class="btn btn-primary">Save</button>
          </div>
        </div>
      </form>
{% endblock %}
```

Listing 7.10: Movie edit and add page handlers for updated template. (file: `handlers.py`, version: `v0402b`)

```
from flask import Blueprint, redirect, render_template, url_for
from flask import current_app, request

from datetime import datetime

from movie import Movie


site = Blueprint('site', __name__)


@site.route('/')
def home_page():
    now = datetime.now()
    day = now.strftime('%A')
    return render_template('home.html', day_name=day)


@site.route('/movies', methods=['GET', 'POST'])
def movies_page():
    if request.method == 'GET':
        movies = current_app.store.get_movies()
        return render_template('movies.html', movies=sorted(movies.items()))
    else:
        movie_ids = request.form.getlist('movie_ids')
        for movie_id in movie_ids:
            current_app.store.delete_movie(int(movie_id))
        return redirect(url_for('site.movies_page'))


@site.route('/movies/add', methods=['GET', 'POST'])
def movie_add_page():
    if request.method == 'GET':
        form = {'title': '', 'year': ''}
        return render_template('movie_edit.html', form=form,
                               min_year=1887, max_year=datetime.now().year)
    else:
        title = request.form['title']
        year = int(request.form['year']) if request.form['year'] else None
        movie = Movie(title, year=year)
        current_app.store.add_movie(movie)
        return redirect(url_for('site.movie_page', movie_id=movie._id))

```

```

@site.route('/movie/<int:movie_id>')
def movie_page(movie_id):
    movie = current_app.store.get_movie(movie_id)
    return render_template('movie.html', movie=movie)


@site.route('/movie/<int:movie_id>/edit', methods=['GET', 'POST'])
def movie_edit_page(movie_id):
    movie = current_app.store.get_movie(movie_id)
    if request.method == 'GET':
        form = {'title': movie.title,
                'year': movie.year if movie.year else ''}
        return render_template('movie_edit.html', form=form,
                               min_year=1887, max_year=datetime.now().year)
    else:
        movie.title = request.form['title']
        movie.year = int(request.form['year']) if request.form['year'] else None
        current_app.store.update_movie(movie)
        return redirect(url_for('site.movie_page', movie_id=movie._id))
```

Listing 7.11: Movie collection model class with update operation. (file: `store.py`, version: `v0402b`)

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

    def add_movie(self, movie):
        self.last_movie_id += 1
        self.movies[self.last_movie_id] = movie
        movie._id = self.last_movie_id

    def update_movie(self, movie):
        self.movies[movie._id] = movie

    def delete_movie(self, movie_id):
        del self.movies[movie_id]

    def get_movie(self, movie_id):
        return self.movies[movie_id]

    def get_movies(self):
        return self.movies
```

Exercise Replace the input validation mechanism with an implementation that uses the Flask-WTF plugin. Also make use of its CSRF protection features. For handling `number` type inputs, use the wtforms_components package.

Install `Flask-WTF` and `wtforms_components`. On your setup, you might need to put these into the `requirements.txt` file.

Listing 7.12: Movie edit form class with validation. (file: `forms.py`, version: `v0403b`)

```
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import DataRequired, NumberRange, Optional
from wtforms_components import IntegerField
```

```python

from datetime import datetime


class MovieEditForm(FlaskForm):
    title = StringField(
        'Title',
        validators=[
            DataRequired()
        ]
    )

    year = IntegerField(
        'Year',
        validators=[
            Optional(),
            NumberRange(min=1887, max=datetime.now().year)
        ]
    )
```

Listing 7.13: Handlers for WTF forms. (file: `handlers.py`, version: `v0403b`)

```python
@site.route('/movies/add', methods=['GET', 'POST'])
def movie_add_page():
    form = MovieEditForm()
    if form.validate_on_submit():
        title = form.data['title']
        year = form.data['year']
        movie = Movie(title, year=year)
        current_app.store.add_movie(movie)
        return redirect(url_for('site.movie_page', movie_id=movie._id))
    return render_template('movie_edit.html', form=form)

@site.route('/movie/<int:movie_id>/edit', methods=['GET', 'POST'])
def movie_edit_page(movie_id):
    movie = current_app.store.get_movie(movie_id)
    form = MovieEditForm()
    if form.validate_on_submit():
        movie.title = form.data['title']
        movie.year = form.data['year']
        current_app.store.update_movie(movie)
        return redirect(url_for('site.movie_page', movie_id=movie._id))
    form.title.data = movie.title
    form.year.data = movie.year if movie.year else ''
    return render_template('movie_edit.html', form=form)
```

Listing 7.14: Setting for CSRF and cookie protection. (file: `settings.py`, version: `v0403b`)

```python
DEBUG = True
PORT = 8080
SECRET_KEY = 'secret'
WTF_CSRF_ENABLED = True
```

Listing 7.15: Movie editing template for WTF forms. (file: `templates/movie_edit.html.py`, version: `v0403b`)

```
{% extends "layout.html" %}
{% block title %}Movie Edit{% endblock %}
{% block content %}
      <div class="page-header">
        <h1>Edit Movie</h1>
      </div>

      <form role="form" action="" method="post" name="movie_edit"
            class="form-horizontal">
        {{ form.csrf_token }}

        <div class="form-group">
          <label for="title" class="control-label col-sm-2">Title:</label>
          <div class="col-sm-10">
            {{ form.title(required=True, autofocus=True) }}
            {% for error in form.title.errors %}
              <div class="alert alert-danger">
                {{ error }}
              </div>
            {% endfor %}
          </div>
        </div>

        <div class="form-group">
          <label for="year" class="control-label col-sm-2">Year:</label>
          <div class="col-sm-10">
            {{ form.year(size='4') }}
            {% for error in form.year.errors %}
              <div class="alert alert-danger">
                {{ error }}
              </div>
            {% endfor %}
          </div>
        </div>

        <div class="form-group">
          <div class="col-sm-offset-2 col-sm-10">
            <button type="submit" name="save"
                    class="btn btn-primary">Save</button>
          </div>
        </div>
      </form>
{% endblock %}
```