Chapter 6
Introduction to Flask

Content

Installation

- Flask is a small framework by most standards—small enough to be called a "micro- framework," and small enough that once you become familiar with it, you will likely be able to read and understand all of its source code.

- Flask has three main dependencies. The routing, debugging, and Web Server Gateway Interface (WSGI) subsystems come from Werkzeug; the template support is provided by Jinja2; and the command-line integration comes from Click.

- In this chapter, you will learn how to install Flask. The only requirement is a computer with Python installed.

Installing Python Packages with pip

To install Flask into the virtual environment, make sure the *venv* virtual environment is activated, and then run the following command:

```
(venv) $ pip install flask
```

When you execute this command, *pip* will not only install Flask, but also all of its dependencies. You can check what packages are installed in the virtual environment at any time using the `pip freeze` command:

```
(venv) $ pip freeze  click==6.7
Flask==0.12.2  itsdangerous==0.24
Jinja2==2.9.6  MarkupSafe==1.0
Werkzeug==0.12.2
```

You can also verify that Flask was correctly installed by starting the Python inter- preter and trying to import it:

```
(venv) $ python
>>> import flask
```

Initialization

All Flask applications must create an *application instance*. The web server passes all requests it receives from clients to this object for handling, using a protocol called Web Server Gateway Interface (WSGI, pronounced “wiz-ghee”). The application instance is an object of class `Flask`, usually created as follows:

```
from flask import Flask
app = Flask(__name__)
```

The only required argument to the `Flask` class constructor is the name of the main module or package of the application. For most applications, Python’s_`name`_variable is the correct value for this argument.

Routes and View Functions

The most convenient way to define a route in a Flask application is through the app.route decorator exposed by the application instance. The following example shows how a route is declared using this decorator:

```
@app.route('/')
def index():
  return '<h1>Hello World!</h1>'.
```

The following example uses app.add_url_rule() to register an index() function that is equivalent to the one shown previously:

```
def index():
        return '<h1>Hello World!</h1>'

      app.add_url_rule('/', 'index', index)
```

The following exam- ple defines a route that has a dynamic component:

```
@app.route('/user/<name>')
def user(name):
  return '<h1>Hello, {}!</h1>'.format(name)
```

The dynamic components in routes are strings by default but can also be of different types. For example, the route `/user/<int:id>` would match only URLs that have an integer in the `id` dynamic segment, such as */user/123*. Flask supports the types `string`, `int`, `float`, and `path` for routes. The `path` type is a special string type that can include forward slashes, unlike the `string` type.

```
return '<h1>Hello World!</h1>'
```

Development Web Server

To start the *hello.py* application from the previous section, first make sure the virtual environment you created earlier is activated and has Flask installed in it. For Linux and macOS users, start the web server as follows:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
*Serving Flask app "hello"
*Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

For Microsoft Windows users, the only difference is in how the FLASK_APP environment variable is set:

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
*Serving Flask app "hello"
*Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Once the server starts up, it goes into a loop that accepts requests and services them. This loop continues until you stop the application by pressing Ctrl+C.

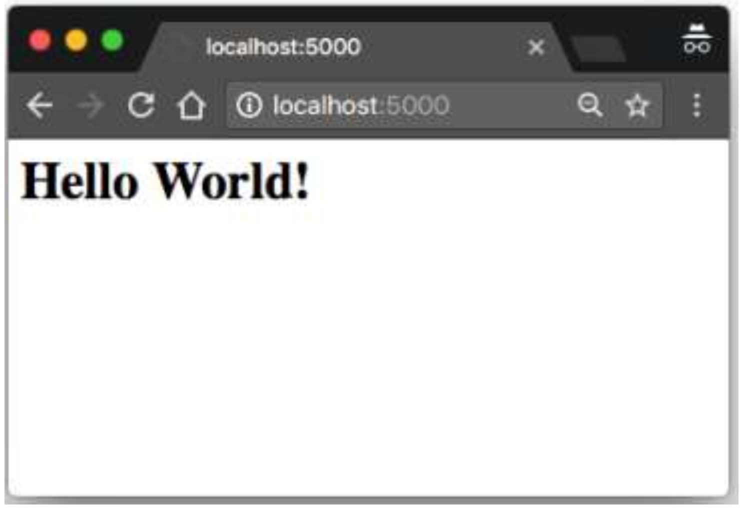
With the server running, open your web browser and type **`http://localhost:5000/`** in the address bar.

A Complete Application

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
        return '<h1>Hello World!</h1>'
if __name__ == '__main__':
        app.run()
```

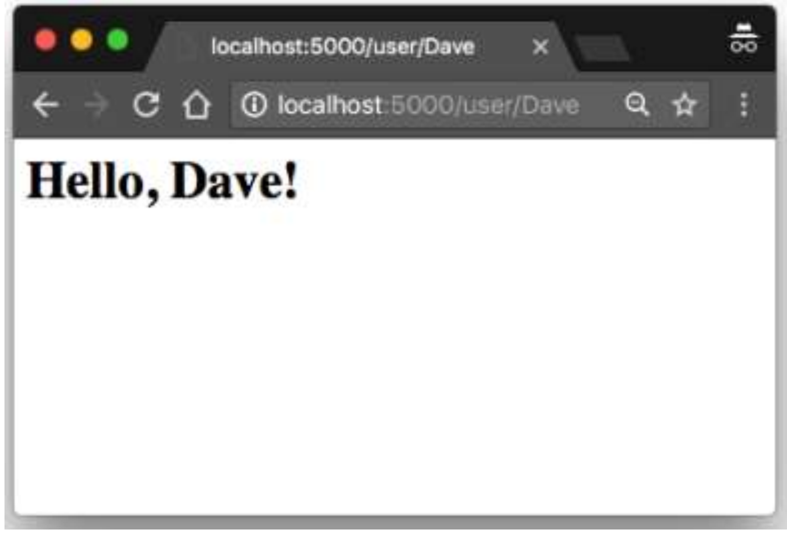
or

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



Hello World!

Dynamic Routes


localhost:5000/user/Dave

Hello, Dave!

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
        return '<h1>Hello World!</h1>'
@app.route('/user/<name>')
def user(name):
        return '<h1>Hello, {}!</h1>'.format(name)
```

Application and Request Contexts

To avoid cluttering view functions with lots of arguments that may not always be needed, Flask uses *contexts* to temporarily make certain objects globally accessible. Thanks to contexts, view functions like the following one can be written:

```
from flask import request

@app.route('/')
def index():
  user_agent = request.headers.get('User-Agent')
  return '<p>Your browser is {}</p>'.format(user_agent)
```

Responses

When a view function needs to respond with a different status code, it can add the numeric code as a second return value after the response text. For example, the following view function returns a 400 status code, the code for a bad request error:

```
@app.route('/')
def index():
  return '<h1>Bad Request</h1>', 400
```

The following example creates a response object and then sets a cookie in it:

```
from flask import make_response

@app.route('/')
def index():
  response = make_response('<h1>This document carries a
  cookie!</h1>')  response.set_cookie('answer', '42')
  return response
```

A redirect response can be generated manually with a three-value return or with a response object, but given its frequent use, Flask provides a `redirect()` helper function that creates this type of response:

```
from flask import redirect

@app.route('/')
def index():
  return redirect('http://www.example.com')
```

The following example returns status code 404 if the `id` dynamic argument given in the URL does not represent a valid user:

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
  user = load_user(id)
  if not user:
    abort(404)
  return '<h1>Hello, {}</h1>'.format(user.name)
```

Rendering Templates

By default Flask looks for templates in a *templates* subdirectory located inside the main application directory. For the next version of *hello.py*, you need to create the *templates* subdirectory and store the templates defined in the previous examples in it as *index.html* and *user.html*, respectively.

Example 3-1. templates/index.html: Jinja2 template

```
<h1>Hello World!</h1>
```

The response returned by the `user()` view function of Example 2-2 has a dynamic component, which is represented by a *variable*. Example 3-2 shows the template that implements this response.

Example 3-2. templates/user.html: Jinja2 template

```
<h1>Hello, {{ name }}!</h1>
```

Rendering Templates

By default Flask looks for templates in a *templates* subdirectory located inside the main application directory. For the next version of *hello.py*, you need to create the *templates* subdirectory and store the templates defined in the previous examples in it as *index.html* and *user.html*, respectively.

Example 3-3. hello.py: rendering a template
```
from flask import

Flask, render_template  # ...


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

@app.route('/user/<name>')
def user(name):
  return render_template('user.html', name=name)
```

Variables

The following are some more examples of variables used in templates:

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar]
}}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

Variables can be modified with *filters*, which are added after the variable name with a pipe character as separator. For example, the following template shows the `name` vari- able capitalized:

```
Hello, {{ name|capitalize }}
```

Bootstrap Integration with Flask-Bootstrap

Bootstrap is an open-source web browser framework from Twitter that provides user interface components that help create clean and attractive web pages that are compatible with all modern web browsers used on desktop and mobile platforms.

The extension is called Flask-Bootstrap, and it can be installed with *pip*:

```
(venv) $ pip install flask-bootstrap
```

Example 3-4. hello.py: Flask-Bootstrap initialization

```
from flask_bootstrap import Bootstrap
# ...
bootstrap = Bootstrap(app)
```

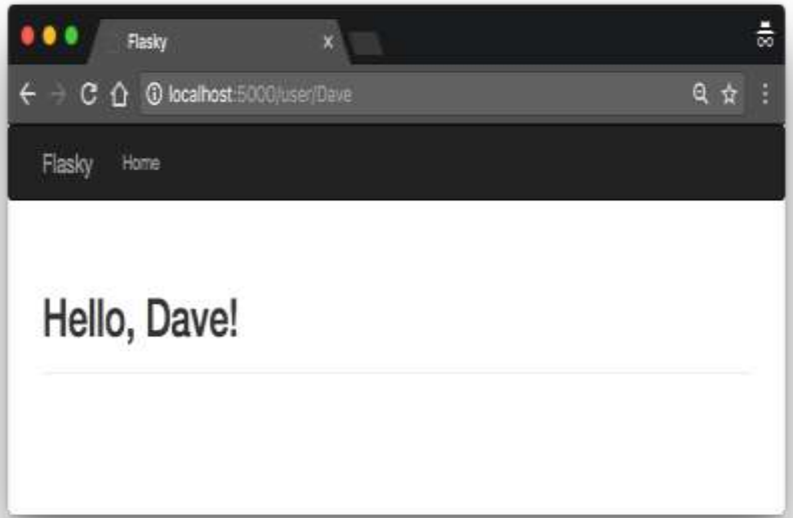
Example 3-5. templates/user.html: template that uses Flask-Bootstrap

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
       data-toggle="collapse" data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Flasky</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
      </ul>
    </div>
  </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
  <div class="page-header">
    <h1>Hello, {{ name }}!</h1>
  </div>
</div>
{% endblock %}
```



Custom Error Pages

When you enter an invalid route in your browser's address bar, you get a code 404 error page. Compared to the Bootstrap-powered pages, the default error page is now too plain and unattractive, and it has no consistency with the actual pages generated by the application.

Example 3-6. hello.py: custom error pages

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

@app.errorhandler(500)
def internal_server_error(e):
  return render_template('500.html'), 500
```

Error handlers return a response, like view functions, but they also need to return the numeric status code that corresponds to the error, which Flask conveniently accepts as a second return value.

Example 3-7. templates/base.html: base application template with navigation bar

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
       data-toggle="collapse" data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Flasky</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
      </ul>
    </div>
  </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
  {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

Example 3-8. templates/404.html: custom code 404 error page using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Not Found</h1>
</div>
{% endblock %}
```

Example 3-9. templates/user.html: simplified page template using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Hello, {{ name }}!</h1>
</div>
{% endblock %}
```

Example 3-10 shows how the application can include a *favicon.ico* icon in the base template for browsers to show in the address bar.

Example 3-10. templates/base.html: favicon definition

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static',
 filename='favicon.ico') }}"  type="image/x-icon">
<link rel="icon" href="{{ url_for('static',
 filename='favicon.ico') }}"  type="image/x-icon">
{% endblock %}
```

Localization of Dates and Times with Flask-Moment

Handling of dates and times in a web application is not a trivial problem when users work in different parts of the world.

There is an excellent open source library written in JavaScript that renders dates and times in the browser called Moment.js. Flask-Moment is an extension for Flask appli- cations that makes the integration of Moment.js into Jinja2 templates very easy. Flask- Moment is installed with *pip*:

```
(venv) $ pip install flask-moment
```

The extension is initialized in a similar way to Flask-Bootstrap. The required code is shown in Example 3-11.

Example 3-11. hello.py: initializing Flask-Moment

```
from flask_moment import Moment  moment = Moment(app)
```

Example 3-12. templates/base.html: importing the Moment.js library

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

To work with timestamps, Flask-Moment makes a `moment` object available to templates. Example 3-13 demonstrates passing a variable called `current_time` to the tem- plate for rendering.

Example 3-13. hello.py: adding a datetime variable

```
from datetime import datetime  @app.route('/')
def index():
  return render_template('index.html',
            current_time=datetime.utcnow())
```

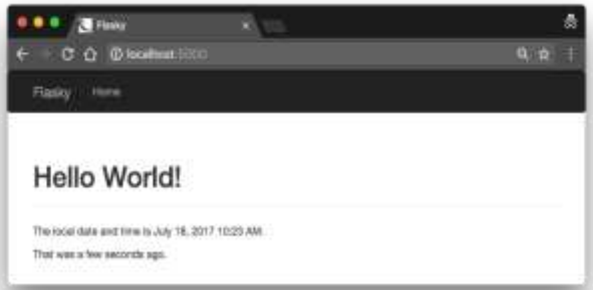
Example 3-14 shows how this `current_time` template variable is rendered.

Example 3-14. templates/index.html: timestamp rendering with Flask-Moment

```
<p>The local date and time is {{ moment(current_time).format('LLL')
}}.</p>
<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

The `format('LLL')` function renders the date and time according to the time zone and locale settings in the client computer. The argument determines the rendering style, from `'L'` to `'LLLL'` for four different levels of verbosity. The `format()` function can also accept a long list of custom format specifiers.

Figure 3-3 shows how the *http://localhost:5000/* route looks after the two timestamps are added to the *index.html* template.



Hello World!

The local date and time is July 18, 2017 10:23 AM.

That was a few seconds ago.

Web Forms

Flask-WTF and its dependencies can be installed with *pip*:

```
(venv) $ pip install flask-wtf
```

Configuration

Example 4-1. hello.py: Flask-WTF configuration

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

Form Classes

Example 4-2 shows a simple web form that has a text field and a submit button.

Example 4-2. hello.py: form class definition

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class NameForm(FlaskForm):
  name = StringField('What is your name?',
  validators=[DataRequired()])  submit = SubmitField('Submit')
```

HTML Rendering of Forms

Form fields are callables that, when invoked from a template, render themselves to HTML. Assuming that the view function passes a `NameForm` instance to the template as an argument named `form`, the template can generate a simple HTML form as follows:

```
<form method="POST">
  {{ form.hidden_tag() }}
  {{ form.name.label }} {{ form.name() }}
  {{ form.submit() }}
</form>
```

Any key- word arguments added to the calls that render the fields are converted into HTML attributes for the field—so, for example, you can give the field `id` or `class` attributes and then define CSS styles for them:

```
<form method="POST">
  {{ form.hidden_tag() }}
  {{ form.name.label }} {{ form.name(id='my-text-field') }}
  {{ form.submit() }}
</form>
```

Using Flask-Bootstrap, the previous form can be rendered as follows:

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

Example 4-3. templates/index.html: using Flask-WTF and Flask-Bootstrap to render a form

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif
  %}!</h1>
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

Form Handling in View Functions

In the new version of *hello.py*, the `index()` view function will have two tasks. First it will render the form, and then it will receive the form data entered by the user. Example 4-4 shows the updated `index()` view function.

Example 4-4. hello.py: handle a web form with GET and POST request methods

```
@app.route(’/’, methods=[’GET’, ’POST’])
def index():
 name = None
 form = NameForm()
 if form.validate_on_submit():   name =
  form.name.data   form.name.data = ’’
 return render_template(’index.html’, form=form, name=name)
```

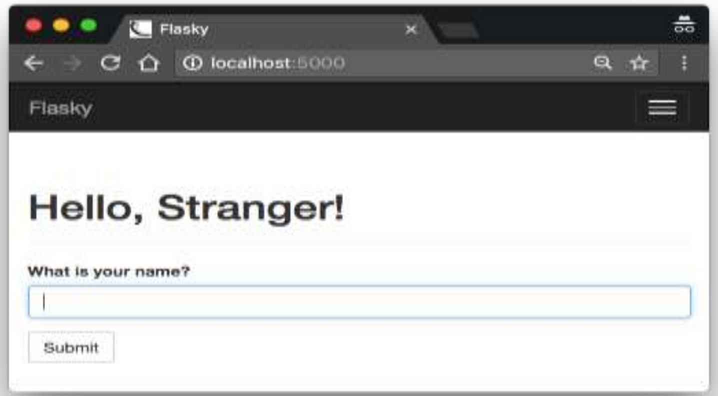


Figure 4-1. Flask-WTF web form

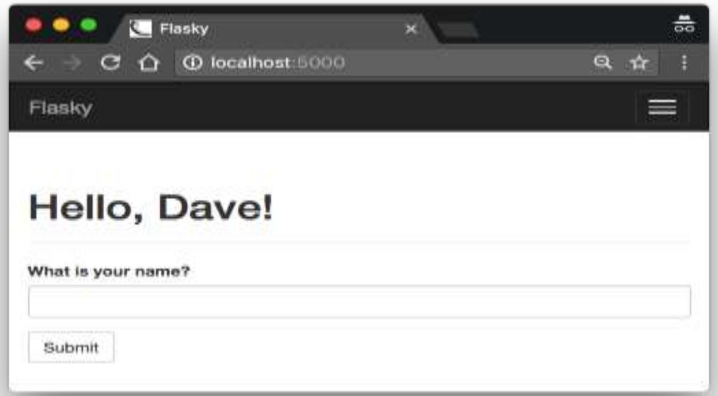


Figure 4-2. Web form after submission

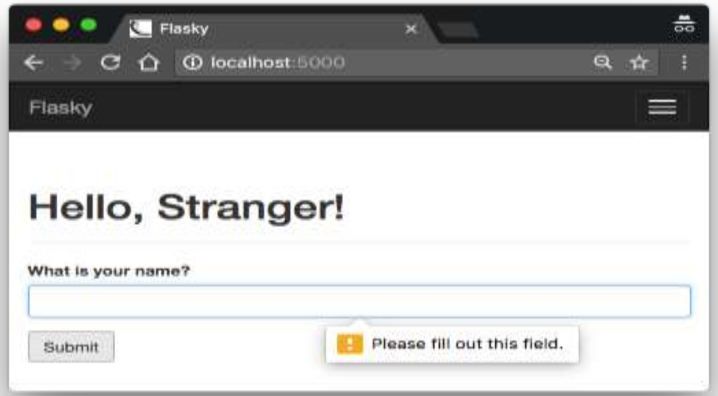


Figure 4-3. Web form after failed validator

Redirects and User Sessions

Example 4-5 shows a new version of the `index()` view function that implements redi- rects and user sessions.

Example 4-5. hello.py: redirects and user sessions

```
from flask import Flask, render_template, session, redirect,

url_for  @app.route('/', methods=['GET', 'POST'])
def index():
  form = NameForm()
  if form.validate_on_submit():  session['name'] =
    form.name.data  return redirect(url_for('index'))
  return render_template('index.html', form=form, name=session.get('name'))
```

In the previous version of the application, a local `name` variable was used to store the name entered by the user in the form. That variable is now placed in the user session as `session['name']` so that it is remembered beyond the request.

Message Flashing

Flask includes this functionality as a core feature. Example 4-6 shows how the `flash()` function can be used for this purpose.

Example 4-6. hello.py: flashed messages

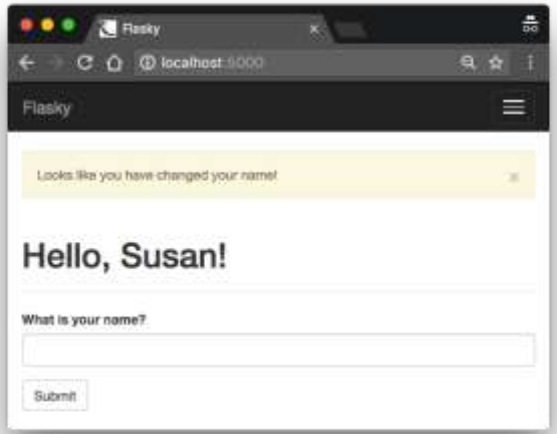
```
from flask import Flask, render_template, session, redirect,

url_for, flash  @app.route('/', methods=['GET', 'POST'])
def index():
 form = NameForm()
 if form.validate_on_submit():  old_name =
  session.get('name')
  if old_name is not None and old_name != form.name.data:
   flash('Looks like you have changed your name!')
  session['name'] = form.name.data
  return redirect(url_for('index'))
 return render_template('index.html',
  form = form, name = session.get('name'))
```

Example 4-7. templates/base.html: rendering of flashed messages

```
{% block content %}
<div class="container">
  {% for message in get_flashed_messages() %}
  <div class="alert alert-warning">
    <button type="button" class="close" data-
    dismiss="alert">&times;</button>
    {{ message }}
  </div>
  {% endfor %}

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



Looks like you have changed your name!

Hello, Susan!

What is your name?

In this example, messages are rendered using Bootstrap's alert CSS styles for warning messages (one is shown in Figure 4-4).