LECTURE 14

Web Frameworks

WEB DEVELOPMENT CONTINUED

Web frameworks are collections of packages or modules which allow developers to write web applications with minimal attention paid to low-level details like protocols, sockets and process management.

Common operations implemented by web frameworks:

- URL routing
- Output format templating.
- Database manipulation
- Basic security

WEB FRAMEWORKS

Python has a number of web framework options, but the two most popular are:

- Django
  - Follows MVC pattern.
  - Most popular.
  - Steeper learning curve.
  - More features built-in.
- Flask
  - “Micro”-framework: minimal approach.
  - You can get things up and going much faster.
  - Less built-in functionality.
  - Also a popular option.

INTRO TO FLASK

We'll be using Flask due to its ease-of-use. Django requires a bit more setup and familiarity with the MVC pattern, but once you're familiar with Flask, you'll find that learning Django is easier.

So, we have some of the backend functionality of our website completed (namely, factbook_scraper.py). Let's put this aside for a moment and create our website.

INTRO TO FLASK

```
travel_app/
  app/
    static/
    templates/
```

The very first thing is to create a directory structure for all of the files and components of our website and set up our virtual env. The root is travel_app.

- `travel_app/` will hold *all* of the files needed for our project.
- The `travel_app/app/` folder holds the application itself.
- `travel_app/app/static` will hold static files (images, css files, etc).
- `travel_app/app/templates` will hold our Jinja2 templates -- we'll come back to this in a bit!

INTRO TO FLASK

```
travel_app/
  app/
    static/
    templates/
  travel_env/
```

The very first thing is to create a directory structure for all of the files and components of our website and set up our virtual env. The root is travel_app.

```
travel_app$ virtualenv travel_env
travel_app$ source travel_env/bin/activate
(travel_env)travel_app$ pip install flask
```

For more info on the reasons for this structure, check out this link from the Flask docs.

INTRO TO FLASK

```
travel_app/
  app/
    __init__.py
    static/
    templates/
  travel_env/
```

We won't just be simply placing all of our code in a single module.

A typical flask application is rather large, and we want to make it as modular as possible.

So, we will be creating a *package*. Essentially, this means we're creating a directory of modules to house all of our code, but there's a bit more to it.

A package is a directory that contains a special file called __init__.py. This file can be empty, and it simply indicates that the directory is a package, and it can be imported the same way a module can be imported.

```
import app
```

INTRO TO FLASK

```
travel_app/
  app/
    __init__.py
    static/
    templates/
  travel_env/
```

The very first thing we'll do is create a Flask application object inside of app/__init__.py

```
from flask import Flask

t_app = Flask(__name__)  # Create a Flask application object called t_app
```

The argument to the Flask class is simply the name of the application package. It is used for resource lookups and can help with debugging. So why are we using __init__.py's __name__ attribute?

The __name__ attribute of a package's __init__ module is defined to be the package name! So, here it is "app".

INTRO TO FLASK

```
travel_app/
  app/
    __init__.py
    static/
    templates/
  travel_env/
```

The very first thing we'll do is create a Flask application object inside of app/__init__.py

```
from flask import Flask

t_app = Flask(__name__)   # Create a Flask application object called t_app
from app import views     # Import the views module from the app package
                          # note - we haven't made this module just yet!
```

Import appears at end to avoid circular dependencies: views will depend on t_app!

INTRO TO FLASK

travel_app/
app/
__init__.py
views.py
static/
templates/
travel_env/

Now, let's create our first view. Views are handlers for requests and are mapped to one or more URLs.

```
from app import t_app
@t_app.route('/')
@t_app.route('/index')
def index():
    return "Welcome to the Traveler's Companion!"
```

Notice the use of decorators!

We've created a view which displays a simple string to the user when they request URLs '/' and '/index'

INTRO TO FLASK

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
  travel_env/
```

One last thing we need to do to actually see some stuff. Create the script run.py to start up the development web server with our little application.

```
#!travel_env/bin/python
from app import t_app
t_app.run(debug=True)
```

Note the line up top – we're telling the shell to use our isolated Python interpreter when we execute ./run.py.

INTRO TO FLASK

travel_app/
- run.py
- app/
  - __init__.py
  - views.py
  - static/
  - templates/
- travel_env/

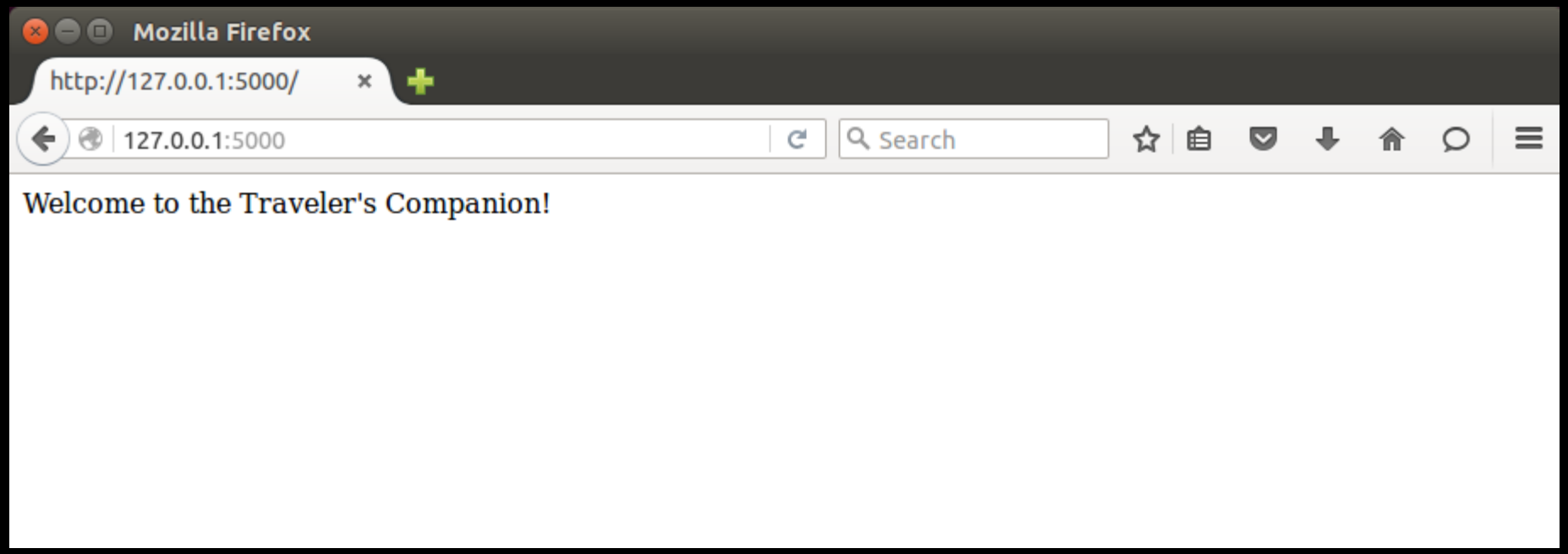
Now let's see what we have. Note that we're still using the virtual environment, even though we're not showing it in the command prompt.

```
ticket_app$ chmod a+x run.py
ticket_app$ ./run.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 206-691-942
```

INTRO TO FLASK

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
  travel_env/
```

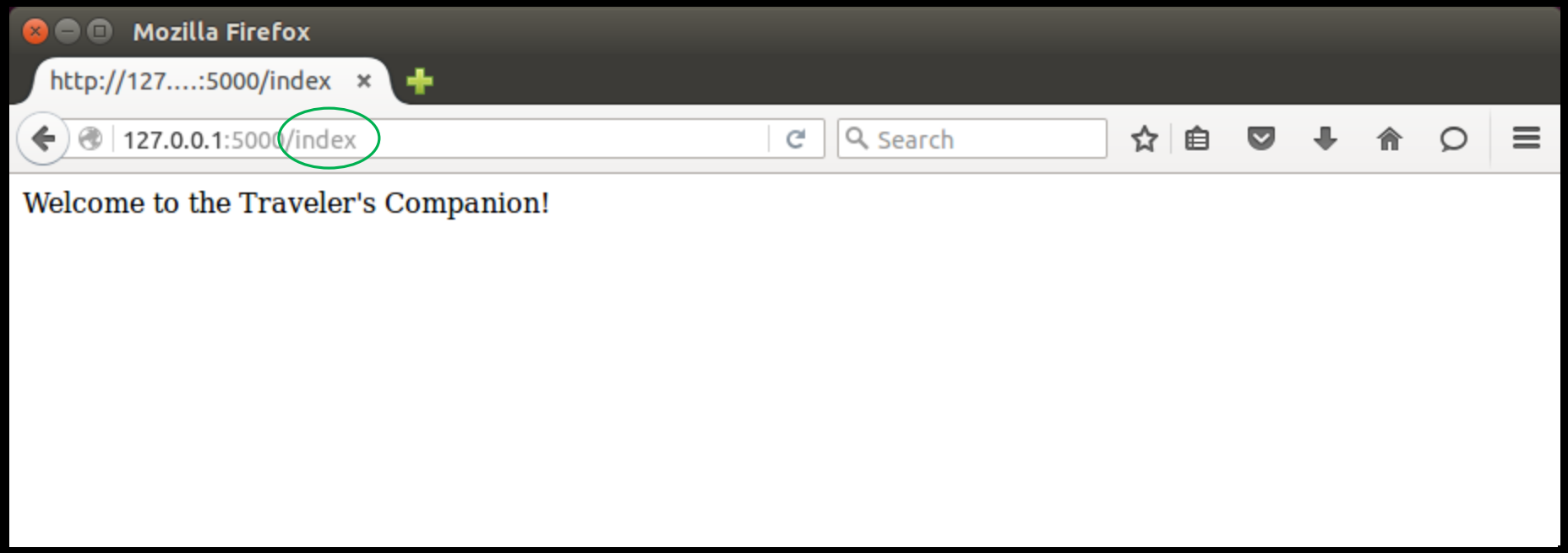
Check out our website by opening a browser and navigating to http://127.0.0.1:5000/



INTRO TO FLASK

We associated both '/' and '/index' with our view function index().

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
  travel_env/
```



TEMPLATING

So far, our only view function merely outputs a single string.

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
  travel_env/
```

```
from app import t_app
@t_app.route('/')
@t_app.route('/index')
def index():
    return "Welcome to the Traveler's Companion!"
```

TEMPLATING

travel_app/
- run.py
- app/
  - __init__.py
  - views.py
  - static/
  - templates/
- travel_env/

We could just as easily return html to be rendered in the browser.

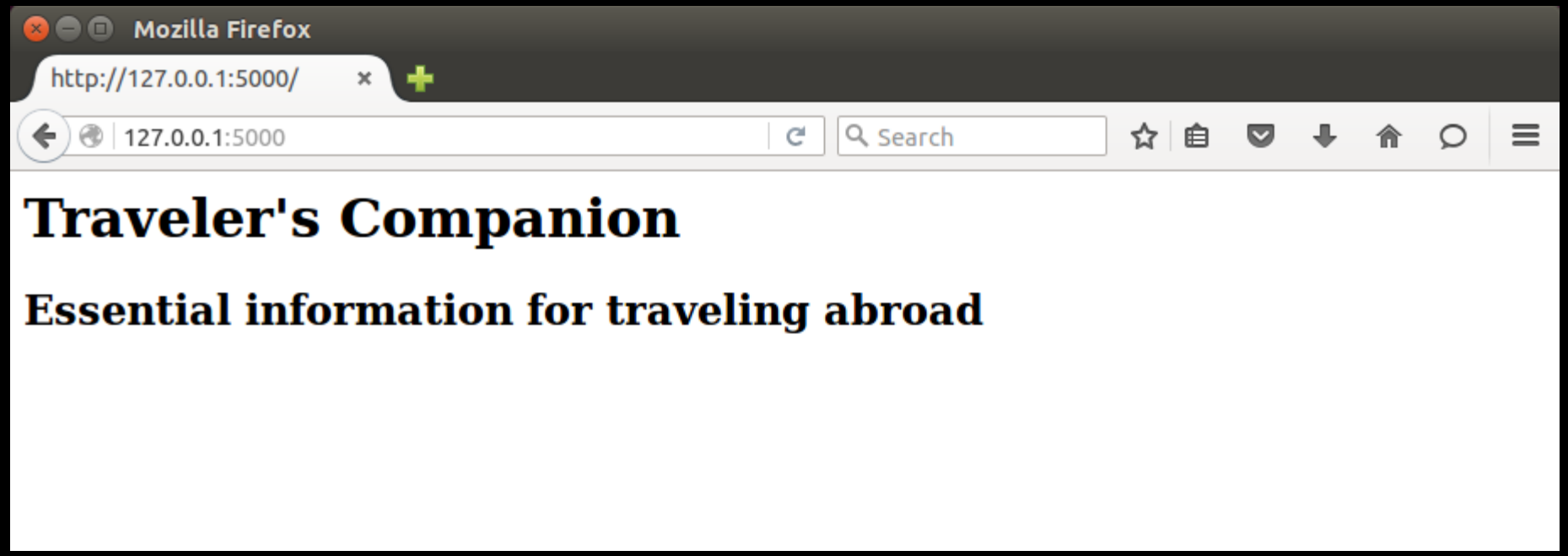
```
from app import t_app

@t_app.route('/')
@t_app.route('/index')
def index():
    return '''<html>
              <h1>Traveler's Companion</h1>
              <h2>Essential information for traveling abroad</h2>
              </html>
            '''
```

TEMPLATING

```
travel_app/
    run.py
    app/
        __init__.py
        views.py
        static/
        templates/
    travel_env/
```

We could just as easily return html to be rendered in the browser.



TEMPLATING

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
      base.html
  travel_env/
```

That method is so ugly though. So let's separate the presentation from the logic by creating a template base.html inside of templates/.

```
<html>
  <head>
    <title>Traveler's Companion</title>
  </head>
  <body>
    <h1>Traveler's Companion</h1>
    <h2>Essential information for traveling abroad</h2>
    {% block content %} {% endblock %}
  </body>
</html>
```

We're using the Jinja2 templating engine, which is included with Flask.

TEMPLATING

Jinja2 templates allow us to do a lot more than just write HTML – we can write *extensible* HTML.

- `{{ arg }}` corresponds to template arguments. We can pass `arg=val` to our template.
- `{%  %}` encompasses control statements (`if`, `for`, `block`, etc).
  - e.g. `{% if arg %} <h1>arg</h1> {% endif %}`
- `{% block content %}{% endblock %}` identifies a portion of the html (called "content") in which more content could be inserted.

TEMPLATING

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
      base.html
  travel_env/
```

That method is so ugly though. So let's separate the presentation from the logic by creating a template base.html inside of templates/.

```
<html>
  <head>
    <title>Traveler's Companion</title>
  </head>
  <body>
    <h1>Traveler's Companion</h1>
    <h2>Essential information for traveling abroad</h2>
    {% block content %} {% endblock %}
  </body>
</html>
```

Now we have a template called base.html which has some basic elements but also includes a control statement which allows us to derive and extend the template.

TEMPLATING

travel_app/
- run.py
- app/
  - __init__.py
  - views.py
  - static/
  - templates/
    - base.html
    - index.html
- travel_env/

Now we'll create templates/index.html which derives from base.html and adds some more content.

```
{% extends "base.html" %}
{% block content %}
Perform a search to find information about your destination!
<ul>
  <li> Capital </li>
  <li> Currency and Exchange Rates </li>
  <li> Diplomatic Representation Contact Information </li>
  <li> etc. </li>
</ul>
{% endblock %}
```

Now we have a template called index.html which inherits from base.html and specifies the matching block statement "content" .

TEMPLATING

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
      base.html
      index.html
  travel_env/
```

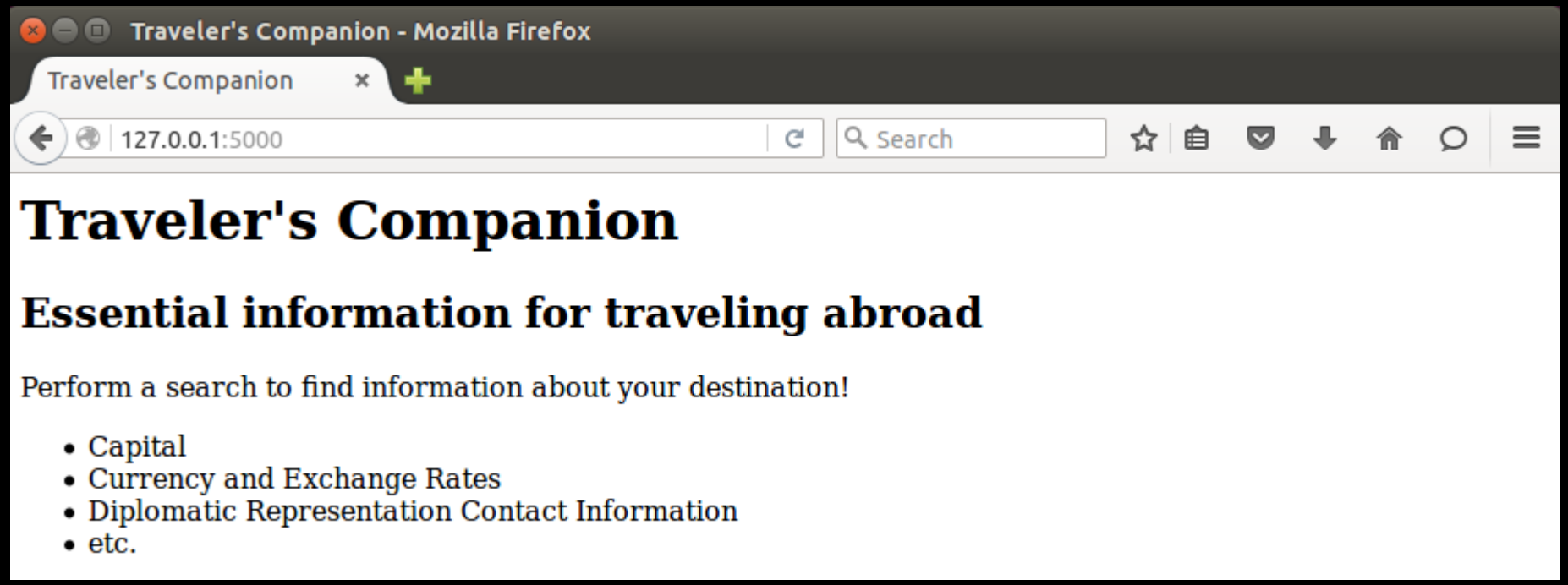
One last thing we need to do. Let's update our views to access our template.

```
from flask import render_template
from app import t_app

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

Invokes Jinja2 templating engine

TEMPLATING


Traveler's Companion - Mozilla Firefox
Traveler's Companion
127.0.0.1:5000
Search
Traveler's Companion
Essential information for traveling abroad
Perform a search to find information about your destination!
Capital
Currency and Exchange Rates
Diplomatic Representation Contact Information
etc.

FORMS

```
travel_app/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
      base.html
      index.html
  travel_env/
```

Obviously, we don't just want to display text to the user, we want to allow them to perform searches. Naturally, we turn our attention to forms.

```
(travel_env)travel_app$ pip install flask-wtf
```

Let's install the WTForms extension which integrates WTForms, a form validation and rendering library, with Flask.

FORMS

When using extensions with Flask, typically you will have to perform some kind of configuration. This can be done in the "config.py" file at the root of your project.

config.py

```
WTF_CSRF_ENABLED = True
SECRET_KEY = 'somesuperdupersecretkey'
```

For the WTForms extension, we may want to specify whether cross-site request forgery protection is enabled and the key to use to validate the form.

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/
      base.html
      index.html
  travel_env/
```

FORMS

Now, we just modify __init__.py to set up our Flask object to use the settings in our config file.

```
from flask import Flask

t_app = Flask(__name__)
t_app.config.from_object('config')

from app import views
```

t_app.config is the fancy dictionary known as a configuration object. It holds all the config options for the Flask object.

travel_app/
- config.py
- run.py
- app/
  - __init__.py
  - views.py
  - static/
  - templates/
    - base.html
    - index.html
- travel_env/

FORMS

Let's create our first form. All we want right now is to be able to accept some string which represents the user's search term for our get_info function.

We'll create a forms.py inside of app/.

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    static/
    templates/
      base.html
      index.html
  travel_env/
```

FORMS

Let's create our first form. All we want right now is to be able to accept some string which represents the user's search term for our get_info function.

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    static/
    templates/
      base.html
      index.html
  travel_env/
```

```python
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import DataRequired

class CountrySearch(FlaskForm):
    country_name = StringField('country_name', validators=[DataRequired()])
```

FORMS

Next, we create a new template for the page on which we will display the form.

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    static/
    templates/
      base.html
      index.html
      search.html
  travel_env/
```

```
{% extends "base.html" %}
{% block content %}
<h3>{{title}}</h3>
<form action="" method="post" name="country_search">
{{form.csrf_token}}
<p> Please enter the country you'd like to find information for:
  {{form.country_name(size=30)}}
</p>
<input type="submit" value="Search!">
</form>
{% endblock %}
```

For CSRF protection

FORMS

Lastly, to see our form in action, we need to create a route for it.

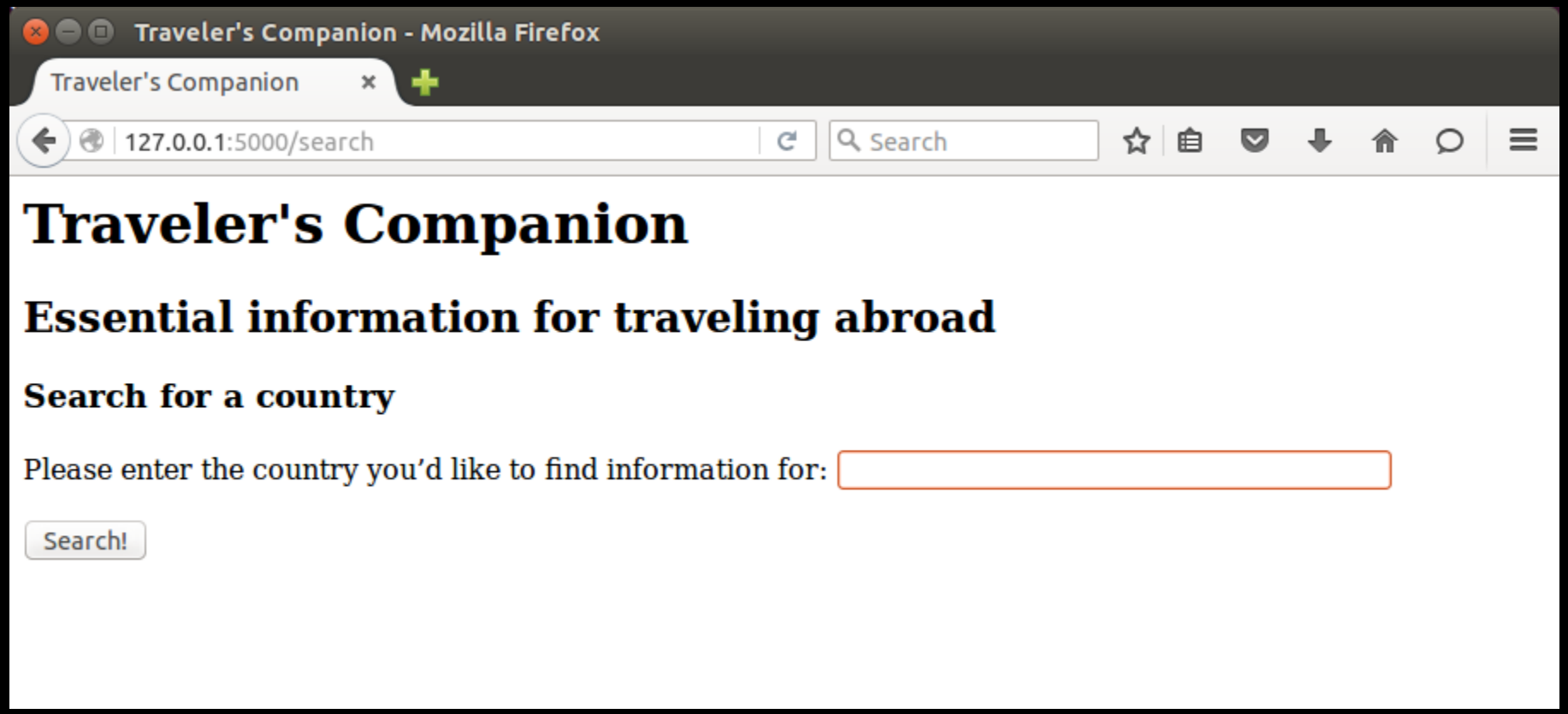
```
from flask import render_template
from app import t_app
from app.forms import CountrySearch

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

@t_app.route('/search', methods=['GET', 'POST'])
def search():
    form = CountrySearch()
    return render_template('search.html', title='Search for a country', form=form)
```

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    static/
    templates/
      base.html
      index.html
      search.html
  travel_env/
```

FORMS


Traveler's Companion - Mozilla Firefox
Traveler's Companion
127.0.0.1:5000/search
Search
Traveler's Companion
Essential information for traveling abroad
Search for a country
Please enter the country you'd like to find information for:
Search!

FORMS

travel_app/
- config.py
- run.py
- app/
  - __init__.py
  - views.py
  - forms.py
  - static/
  - templates/
    - base.html
    - index.html
    - search.html
- travel_env/

Now, we're displaying a form so let's actually grab some input.

```
from flask import render_template
from app import t_app
from app.forms import CountrySearch
…
@t_app.route('/search', methods=['GET', 'POST'])
def search():
    form = CountrySearch()
    if form.validate_on_submit():
        country = form.country_name.data
    return render_template('search.html', title='Search for a country', form=form)
```

form.validate_on_submit() returns false if the user hasn't entered data, true if they have and validators are met.

FORMS

If we gather data, and all of the validators are met, we still render search, but with post info.

```
from flask import render_template
…
from app.factbook_scraper import get_info

@t_app.route('/search', methods=['GET', 'POST'])
def search():
    info = None
    form = CountrySearch()
    if form.validate_on_submit():
        country = form.country_name.data
        info = get_info(country)
    return render_template('search.html', title='Search for a country',
                           form=form, info=info)
```

travel_app/
- config.py
- run.py
- app/
  - __init__.py
  - views.py
  - forms.py
  - factbook_scraper.py
  - static/
  - templates/
    - base.html
    - index.html
    - search.html
- travel_env/

FORMS

So here's what we're doing:

- When a user navigates to /search, they can enter their search criteria.
- When a form is submitted, we use the search information to call our scraping module.
- Then, we render the search template again with the country information.

Now, we need to update the template for search.html to display the information.

Note: this is not an ideal setup – we're mixing backend logic with our application.
We'll call factbook_scraper directly to build our site for now.

DISPLAYING RESULTS

To remind ourselves, here's the contents of search.html.

```
{% extends "base.html" %}
{% block content %}
<h3>{{title}}</h3>
<form action="" method="post" name="country_search">
{{form.csrf_token}}
<p> Please enter the country you'd like to find information for:
  {{form.country_name(size=30)}}
</p>
<input type="submit" value="Search!">
</form>
{% endblock %}
```

DISPLAYING RESULTS

We enclose our additions to search.html using an if condition – if info is empty, we don't display the results. This happens when a user first navigates to the search page.

```
% extends "base.html" %}
{% block content %}
…
{% if info %} <br/>
Capital: {{ info['Capital'] }} <br/>
Languages: {{ info['Languages'] }} <br/>
Currency: {{info['Exchange rates'] }} <br/>
Population: {{ info['Population'] }} <br/>
Area: {{ info['Area'] }}
{% endif %}
```

DISPLAYING RESULTS

DISPLAYING RESULTS

DISPLAYING RESULTS

Well, that looks…terrible. Let's make it look nicer. We'll add an external style sheet to our static folder and update our base.html.

```
<html>
  <head>
    <title>Traveler's Companion</title>
    <link rel="stylesheet" type="text/css"
        href="{{ url_for('static',filename='style.css') }}">
  </head>
  <body>
    <h1>Traveler's Companion</h1>
    <h2>Essential information for traveling abroad</h2>
    {% block content %} {% endblock %}
  </body>
</html>
```

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    factbook_scraper.py
    static/
      style.css
    templates/
      base.html
      index.html
      search.html
  travel_env/
```

DISPLAYING RESULTS

We can use a {{ … }} delimeter which will tell Jinja2 to evaluate the expression inside. Inside, we'll make a call to url_for which generates a URL to the target.

```
<html>
  <head>
    <title>Traveler's Companion</title>
    <link rel="stylesheet" type="text/css"
        href="{{ url_for('static',filename='style.css') }}">
  </head>
  <body>
    <h1>Traveler's Companion</h1>
    <h2>Essential information for traveling abroad</h2>
    {% block content %} {% endblock %}
  </body>
</html>
```

```
travel_app/
  config.py
  run.py
  app/
    __init__.py
    views.py
    forms.py
    factbook_scraper.py
    static/
      style.css
    templates/
      base.html
      index.html
      search.html
  travel_env/
```

DISPLAYING RESULTS

```
<html>
  <head>
    <title>Traveler's Companion</title>
    <link rel="stylesheet" type="text/css"
          href="{{ url_for('static',filename='style.css') }}">
  </head>
  <body>
    <h1>Traveler's Companion</h1>
    <h2>Essential information for traveling abroad</h2>
    <div id="hmenu">
      <ul>
        <li><a href="{{url_for('index')}}">Home</a></li>
        <li><a href="{{url_for('search')}}">Search</a></li>
      </ul>
    </div>
    {% block content %} {% endblock %}
  </body>
</html>
```

Let's add a navigation menu while we're at it.

I'm going to cheat a bit and use Bootstrap to get some quick styling in.

Some of the components in the html are a little different in the posted example, but the ideas are the same!

Traveler's Companion - Mozilla Firefox

Traveler's Companion

127.0.0.1:5000/search

Search

Traveler's Companion Essential information for traveling abroad

Home Search

Search for a country

Please enter the country you'd like to find information for: Norway

Search!

Capital	Oslo
Languages	Bokmal Norwegian (official), Nynorsk Norwegian (official), small Sami- and Finnish-speaking minorities
Currency	Norwegian kroner (NOK) per US dollar: 7.876 (2015 est.)
Population	5,207,689 (July 2015 est.)
Area	323,802 sq km

DISPLAYING RESULTS

We're probably not going to be featured on CSS Zen Garden or anything, but it looks better.

So that's zero to Flask in 40 slides. Some important things to note:

- Flask has a TON of extensions: check http://flask.pocoo.org/extensions/.
- As I said, we built the "back-end" functionality right into the app here. A smarter way would be to have a reactor loop that picks up scraping requests placed into a queue by the Flask app, and returns the results when it can. This way, we can perform non-blocking scraping in the case of a bunch of users accessing the site at once.
- We've been serving our site using Flask's development server (app.run()), which is great for development but you should use a full server for deployment. Let's explore self-hosting with Gunicorn and Nginx.