Setting up our project and a Git repository

We could simply edit our Hello World application to add the desired functionality, but it's cleaner to start a new project. We'll create a new Git repository, a new Python file, a new `.wsgi` file, and a new Apache configuration file. We'll do this for each of the projects in the book, which means that all three of the projects as well as the original Hello World application will be accessible from our web server.

Setting up is very similar to what we did for our Hello World application in *Chapter 1, Hello, World!* but we'll briefly go through the steps again as we don't have to repeat a lot of the configuration and installation, as follows:

1. Log in to your GitHub or BitBucket account and create a new repository called `headlines`. Take note of the URL you're given for this blank repository.
2. On your local machine, create a new directory called `headlines` in your home directory or wherever you put the `firstapp` directory.
3. Create a new file in this directory called `headlines.py`.
4. In your terminal, change the directory to the `headlines` directory and initialize the Git repository by executing the following commands:

```
cd headlines
git init
git remote add origin <your headlines git URL>
git add headlines.py
git commit -m "initial commit"
git push -u origin master
```

Now, we're almost ready to push code to our new repository; we just need to write it first.

Creating a new Flask application

To begin with, we'll create the skeleton of our new Flask application, which is pretty much the same as our Hello World application. Open `headlines.py` in your editor and write the following code:

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def get_news():
  return "no news is good news"

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

This works exactly as before. You can run it in your terminal with `python headlines.py`. Open a browser and navigate to `localhost:5000` to see the **no news is good news** string displayed. However, although the old adage may be true, it's bad news that our app does not do anything more useful than this. Let's make it display actual news to our users.

Introduction to RSS and RSS feeds

RSS is an old but still widely used technology to manage content feeds. It's been around for such a long time that there's some debate as to what the letters RSS actually stand for, with some saying Really Simple Syndication and others Rich Site Summary. It's a bit of a moot point as everyone just calls it RSS.

RSS presents content in an ordered and structured format using XML. It has several uses, with one of the more common uses being for people to consume news articles. On news websites, news is usually laid out similarly to a print newspaper with more important articles being given more space and also staying on the page for longer. This means that frequent visitors to the page will see some content repeatedly and have to look out for new content. On the other hand, some web pages are updated only very infrequently, such as some authors' blogs. Users have to keep on checking these pages to see whether they are updated, even when they haven't changed most of the time. RSS feeds solve both of these problems. If a website is configured to use RSS feeds, all new content is published to a feed. A user can subscribe to the feeds of his or her choice and consume these using an RSS reader. New stories from all feeds he or she has subscribed to will appear in the reader and disappear once they are marked as read.

As RSS feeds have a formal structure, they allow us to easily parse the headline, article text, and date programmatically in Python. We'll use some RSS feeds from major news publications to display news to our application's users.

Although RSS follows a strict format and we could, with not too much trouble, write the logic to parse the feeds ourselves, we'll use a Python library to do this. The library abstracts away things such as different versions of RSS and allows us to access the data we need in a completely consistent fashion.

There are several Python libraries that we could use to achieve this. We'll select `feedparser`. To install it, open your terminal and type the following:

```
pip install --user feedparser
```

Now, let's go find an RSS feed to parse! Most major publications offer RSS feeds, and smaller sites built on popular platforms, such as WordPress and Blogger, will often have RSS included by default as well. Sometimes, a bit of effort is required to find the RSS feed; however, as there is no standard as to where it should be located, you'll often see the RSS icon somewhere on the homepage (look at the headers and footers), which looks similar to this:

Also, look for links saying **RSS** or **Feed**. If this fails, try going to `site.com/rss` or `site.com/feed`, where `site.com` is the root URL of the site for which you're looking for RSS feeds.

We'll use the RSS feed for the main BBC news page. At the time of writing, it is located at `http://feeds.bbci.co.uk/news/rss.xml`. If you're curious, you can open the URL in your browser, right-click somewhere on the page, and click on **View Source** or an equivalent. You should see some structured XML with a format similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
  <channel>
    <title>FooBar publishing</title>
    <link>http://dwyer.co.za</link>
    <description>A mock RSS feed</description>
    <language>en-gb</language>
    <item>
      <title>Flask by Example sells out</title>
      <description>Gareth Dwyer's new book,
      Flask by Example sells out in minutes</description>
      <link>http://dwyer.co.za/book/news/flask-by-example</link>
      <guid isPermalink="false">http://dwyer.co.za/book/news/
      flask-by-example</guid>
      <pubDate>Sat, 07 Mar 2015 09:09:19 GMT</pubDate>
    </item>
  </channel>
</rss>
```

At the very top of the feed, you'll see a line or two that describes the feed itself, such as which version of RSS it uses and possibly some information about the styles. After this, you'll see information relating to the publisher of the feed followed by a list of `<item>` tags. Each of these represents a *story*—in our case, a news article. These items contain information such as the headline, a summary, the date of publication, and a link to the full story. Let's get parsing!

Using RSS from Python

In our `headlines.py` file, we'll make modifications to import the `feedparser` library we installed, parse the feed, and grab the first article. We'll build up HTML formatting around the first article and show this in our application. If you're not familiar with HTML, it stands for **Hyper Text Markup Language** and is used to define the look and layout of text in web pages. It's pretty straightforward, but if it's completely new to you, you should take a moment now to go through a beginner tutorial to get familiar with its most basic usage. There are many free tutorials online, and a quick search should bring up dozens. A popular and very beginner-friendly one can be found at `http://www.w3schools.com/html/`.

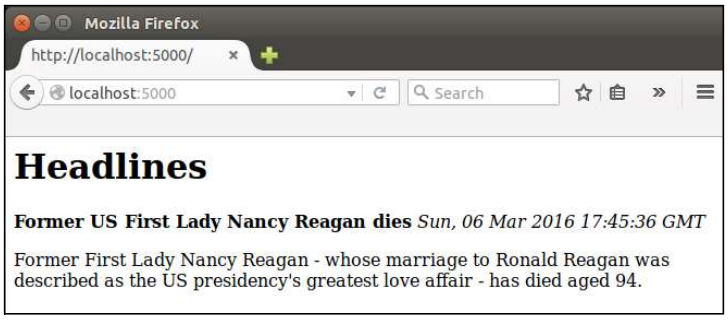
Our new code adds the import for the new library, defines a new global variable for the RSS feed URL, and further adds a few lines of logic to parse the feed, grab the data we're interested in, and insert this into some very basic HTML. It looks similar to this:

```
import feedparser

from flask import Flask

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/rss.xml"


@app.route("/")
def get_news():
  feed = feedparser.parse(BBC_FEED)
  first_article = feed['entries'][0]
  return """<html>
    <body>
        <h1> BBC Headlines </h1>
        <b>{0}</b> <br/>
        <i>{1}</i> <br/>
```

```
            <p>{2}</p> <br/>
        </body>
    </html>""".format(first_article.get("title"), first_article.
    get("published"), first_article.get("summary"))

    if __name__ == "__main__":
      app.run(port=5000, debug=True)
```

The first line of this function passes the BBC feed URL to our `feedparser` library, which downloads the feed, parses it, and returns a Python dictionary. In the second line, we grabbed just the first article from the feed and assigned it to a variable. The `entries` entry in the dictionary returned by `feedparser` contains a list of all the items that include the news stories we spoke about earlier, so we took the first one of these and got the headline or `title`, the date or the `published` field, and the summary of the article (that is, `summary`) from this. In the `return` statement, we built a basic HTML page all within a single triple-quoted Python string, which includes the `<html>` and `<body>` tags that all HTML pages have as well as an `<h1>` heading that describes what our page is; `<b>`, which is a *bold* tag that shows the news headline; `<i>`, which stands for the *italics* tag that shows the date of the article; and `<p>`, which is a paragraph tag to show the summary of the article. As nearly all items in an RSS feed are optional, we used the `python.get()` operator instead of using index notation (square brackets), meaning that if any information is missing, it'll simply be omitted from our final HTML rather than causing a runtime error.

For the sake of clarity, we didn't do any exception handling in this example; however, note that `feedparser` may well throw an exception on attempting to parse the BBC URL. If your local Internet connection is unavailable, the BBC server is down, or the provided feed is malformed, then `feedparser` will not be able to turn the feed into a Python dictionary. In a real application, we would add some exception handling and retry the logic here. In a real application, we'd also never build HTML within a Python string. We'll look at how to handle HTML properly in the next chapter. Fire up your web browser and take a look at the result. You should see a very basic page that looks similar to the following (although your news story will be different):



This is a great start, and we're now serving dynamic content (that is, content that changes automatically in response to user or external events) to our application's hypothetical users. However, ultimately, it's not much more useful than the static string. Who wants to see a single news story from a single publication that they have no control over?

To finish off this chapter, we'll look at how to show an article from different publications based on URL routing. That is, our user will be able to navigate to different URLs on our site and view an article from any of several publications. Before we do this, let's take a slightly more detailed look at how Flask handles URL routing.

URL routing in Flask

Do you remember that we briefly mentioned Python decorators in the previous chapter? They're represented by the funny `@app.route("/")` line we had above our main function, and they indicate to Flask which parts of our application should be triggered by which URLs. Our base URL, which is usually something similar to `site.com` but in our case is the IP address of our VPS, is omitted, and we will specify the rest of the URL (that is, the path) in the decorator. Earlier, we used a single slash, indicating that the function should be triggered whenever our base URL was visited with no path specified. Now, we will set up our application so that users can visit URLs such as `site.com/bbc` or `site.com/cnn` to choose which publication they want to see an article from.

The first thing we need to do is collect a few RSS URLs. At the time of writing, all of the following are valid:

- **CNN**: `http://rss.cnn.com/rss/edition.rss`
- **Fox News**: `http://feeds.foxnews.com/foxnews/latest`
- **IOL**: `http://www.iol.co.za/cmlink/1.640`

First, we will consider how we might achieve our goals using static routing. It's by no means the best solution, so we'll implement static routing for only two of our publications. Once we get this working, we'll consider how to use dynamic routing instead, which is a simpler and more generic solution to many problems.

Instead of declaring a global variable for each of our RSS feeds, we'll build a Python dictionary that encapsulates them all. We'll make our `get_news()` method generic and have our decorated methods call this with the relevant publication. Our modified code looks as follows:

```
import feedparser
from flask import Flask
```

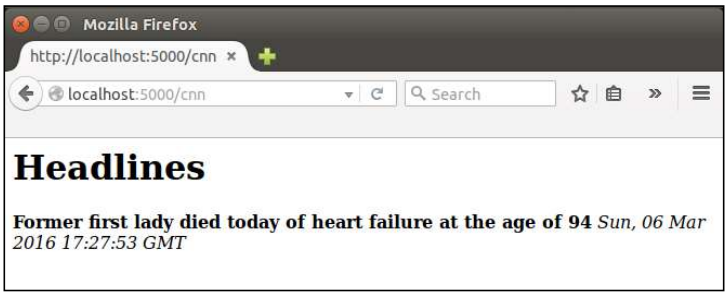
```
app = Flask(__name__)

RSS_FEEDS = {'bbc': 'http://feeds.bbci.co.uk/news/rss.xml',
             'cnn': 'http://rss.cnn.com/rss/edition.rss',
             'fox': 'http://feeds.foxnews.com/foxnews/latest',
             'iol': 'http://www.iol.co.za/cmlink/1.640'}

@app.route("/")
@app.route("/bbc")
def bbc():
    return get_news('bbc')

@app.route("/cnn")
def cnn():
    return get_news('cnn')

def get_news(publication):
  feed = feedparser.parse(RSS_FEEDS[publication])
  first_article = feed['entries'][0]
  return """<html>
    <body>
        <h1>Headlines </h1>
        <b>{0}</b> </ br>
        <i>{1}</i> </ br>
        <p>{2}</p> </ br>
    </body>
</html>""".format(first_article.get("title"), first_article.
get("published"), first_article.get("summary"))


if __name__ == "__main__":
  app.run(port=5000, debug=True)
```

Common mistakes:

If you're copying or pasting functions and editing the `@app.route` decorator, it's easy to forget to edit the function name. Although the name of our functions is largely irrelevant as we don't call them directly, we can't have different functions share the same name as the latest definition will always override any previous ones.

We still return the BBC news feed by default, but if our user visits the CNN or BBC routes, we will explicitly take the top article from respective publication. Note that we can have more than one decorator per function so that our `bbc()` function gets triggered by a visit to our base URL or to the `/bbc` path. Also, note that the function name does not need to be the same as the path, but it is a common convention that we followed in the preceding example.

Following this, we can see the output for our application when the user visits the `/cnn` page. The headline displayed is now from the CNN feed.



Now that we know how routing works in Flask, wouldn't it be nice if it could be even simpler? We don't want to define a new function for each of our feeds. What we need is for the function to dynamically grab the right URL based on the path. This is exactly what dynamic routing does.

In Flask, if we specify a part of our URL path in angle brackets `< >`, then it is taken as a variable and is passed to our application code. Therefore, we can go back to having a single `get_news()` function and pass in a `<publication>` variable, which can be used to make the selection from our dictionary. Any variables specified by the decorator must be accounted for in our function's definition. The first few lines of the updated `get_news()` function are shown as follows:

```
@app.route("/")
@app.route("/<publication>")
def get_news(publication="bbc"):
    # rest of code unchanged
```

In the code shown earlier, we added `<publication>` to the route definition. This creates an argument called `publication`, which we need to add as a parameter of the function directly below the route. Thus, we can keep our default value for the publication parameter as `bbc`, but if the user visits CNN, Flask will pass the `cnn` value as the publication argument instead.

The rest of the code remains unchanged, but it's important to delete the now unused `bbc()` and `cnn()` function definitions as we need the default route to activate our `get_news()` function instead.

It's easy to forget to *catch* the URL variables in the function definition. Any dynamic part of the route must contain a parameter of the same name in the function in order to use the value, so look out for this. Note that we gave our publication variable a default value of `bbc` so that we don't need to worry about it being undefined when the user visits our base URL. However, again, our code will throw an exception if the user visits any URL that we don't have as a key in our dictionary of feeds. In a real web application, we'd catch cases such as this and show an error to the user, but we'll leave error handling for later chapters.

Publishing our Headlines application

This is as far as we'll take our application in this chapter. Let's push the results to our server and configure Apache to display our headlines application instead of our Hello World application by default.

First, add your changes to the Git repository, commit them, and push them to the remote. You can do this by running the following commands (after opening a terminal and changing directory to the headlines directory):

```
git add headlines.py
git commit -m "dynamic routing"
git push origin master
```

Then, connect to the VPS with SSH and clone the new project there using the following commands:

```
ssh -i yourkey.pem root@123.456.789.123
cd /var/www
git clone https://<yourgitrepo>
```

Don't forget to install the new library that we now depend on. Forgetting to install dependencies on your server is a common error that can lead to a frustrating debugging. Keep this in mind. The following is the command for this:

```
pip install --user feedparser
```

Now, create the `.wsgi` file. I assume that you named your Git project `headlines` when creating the remote repository and that a directory named `headlines` was created in your `/var/www` directory when you did the preceding Git clone command. If you called your project something else and now have a directory with a different name, rename it to headlines (otherwise, you'll have to adapt a lot of the configuration we're about to do accordingly). To rename a directory in Linux, use the following command:

```
mv myflaskproject headlines
```

The command used earlier will rename the directory called `myflaskproject` to `headlines`, which will ensure that all the configuration to follow will work. Now, run the following:

```
cd headlines
nano headlines.wsgi
```

Then, insert the following:

```
import sys
sys.path.insert(0, "/var/www/headlines")
from headlines import app as application
```

Exit Nano by hitting the *Ctrl* + *X* key combo and enter *Y* when prompted to save changes.

Now, navigate to the `sites-available` directory in Apache and create the new `.conf` file using the following commands:

```
cd /etc/apache2/sites-available
nano headlines.conf
```

Next, enter the following:

```
<VirtualHost *>
    ServerName example.com

    WSGIScriptAlias / /var/www/headlines/headlines.wsgi
    WSGIDaemonProcess headlines
    <Directory /var/www/headlines>
       WSGIProcessGroup headlines
       WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Save the file and quit nano. Now, disable our old site, enable the new one, and restart Apache by running the following commands:

```
sudo a2dissite hello.conf
sudo a2enssite headlines.conf
sudo service apache2 reload
```

Try and visit the IP address of your VPS from your local machine, and if all went as expected, you should see the news headline as before! If not, don't worry. It's easy to make a mistake in some piece of configuration. It's most likely that your `headlines.wsgi` or `headlines.conf` file has a small error. The easiest way to find this is by looking at the most recent errors in your Apache error log, which would have been triggered when you attempted to visit the site. View this again with the following command:

```
sudo tail -fn 20 /var/log/apache2/error.log
```

Summary

That's it for this chapter. The major takeaways of this chapter were taking a look at how routing, both static and dynamic, are handled in Flask. You also learned a fairly messy way of formatting data using HTML and returning this to the user.

In the next chapter, we'll take a look at cleaner ways to separate our HTML code from our Python code using Jinja templates. We'll also have our app display more than a single news story.

3
Using Templates in Our Headlines Project

In the last chapter, we saw one way of combining static HTML with dynamic content for creating a web page. But it's messy, and we don't want to hack away at Python strings for building our web pages. Mixing HTML and Python is not ideal for a few reasons: for one, it means if we ever want to change static text, such as that which appears in our headings, we have to edit our Python files, which also involves reloading these files into Apache. If we hire frontend developers to work on HTML, we run the risk of them breaking the unfamiliar Python code by mistake, and it's far more difficult to structure any other frontend code such as JavaScript and CSS correctly. Ideally, we should aim for complete segregation between the frontend and backend components. We can achieve this to a large extent using Jinja, but as with most aspects of life, some compromise will be necessary.

By the end of this chapter, we'll have extended our application to display more than a single headline for the chosen publication. We'll display several articles for each publication, each one having a link to the original article, and our logic and view components will largely be separated. In this chapter, we'll cover the following topics:

- Introducing Jinja
- Basic use of Jinja templates
- Advanced use of Jinja templates

Introducing Jinja

Jinja is a Python template engine. It allows us to easily define dynamic blocks of HTML which are populated by Python. HTML templates are useful even for static websites which have multiple pages. Usually, there are some common elements, such as headers and footers, on every page. Although it is possible to maintain each page individually for static websites, this requires that a single change be made in multiple places if the change is made to a shared section. Flask was built on top of Jinja, so although it is possible to use Jinja without Flask, Jinja is still an inherent part of Flask, and Flask provides several methods to work directly with Jinja. Generally, Flask assumes nothing about the structure of your application except what you tell it, and prefers providing functionality through optional plugins. Jinja is somewhat of an exception to this. Flask gives you Jinja by default, and assumes that you store all your Jinja templates in a subdirectory of your application named `templates`.

Once we've created templates, we'll make calls from our Flask app to render these templates. Rendering involves parsing the Jinja code, inserting any dynamic data, and creating pure HTML to be returned to a user's browser. All of this is done behind the scenes though, so it can get a bit confusing as to what is being done where. We'll take things one step at a time.

Basic use of Jinja templates

The first step to using Jinja templates is creating a directory in our application to contain our template files, so navigate to your `headlines` directory, and create a directory called `templates`. Unlike the previous steps, this name is expected by other parts of the application and is case sensitive, so take care while creating it. At the most basic level, a Jinja template can just be an HTML file, and we'll use the `.html` extension for all our Jinja templates. Create a new file in the `templates` directory called `home.html`. This will be the page that our users see when visiting our application, and will contain all the HTML that we previously had in a Python string.

We'll only be using Jinja to build HTML files in this book, but Jinja is flexible enough for use in generating any text-based format. Although we use the `.html` extension for our Jinja templates, the files themselves will not always be pure HTML.

For now, put the following static HTML code into this file. We'll look at how to pass dynamic data between Python and our templates in the next step.

```
<html>
    <head>
        <title>Headlines</title>
```

```
    </head>
    <body>
        <h1>Headlines</h1>
        <b>title</b><br />
        <i>published</i><br />
        <p>summary</p>
    </body>
</html>
```

Now in our Python code, instead of building up the string and returning that in our routing function, we'll render this template and return it. In `headlines.py`, add an import at the top:

```
from flask import render_template
```

The `render_template` function is the magic which takes a Jinja template as input and produces pure HTML, capable of being read by any browser, as the output. For now, some of the magic is lost, as we'll give it pure HTML as input and view the same as output in our browser.

Rendering a basic template

In your `get_news()` function, remove the `return` statement, which contains our triple-quoted HTML string as well. Leave the previous lines which grab the data from `feedparser`, as we'll be using that again soon.

Update the `return` statement, so that the `get_news()` function now looks as follows:

```
@app.route("/")
@app.route("/<publication>"
def get_news(publication="bbc"):
  feed = feedparser.parse(RSS_FEEDS[publication])
  first_article = feed['entries'][0]
  return render_template("home.html")
```

Although our current HTML file is pure HTML and not yet using any of the Jinja syntax that we'll see later, we're actually already doing quite a bit of magic. This call looks in our `templates` directory for a file named `home.html`, reads this, parses any Jinja logic, and creates an HTML string to return to the user. Once you've made both the preceding changes, run your application again with `python headlines.py`, and navigate to `localhost:5000` in your browser.

Again, we've gone a step backwards in order to advance. If you run the app and view the result in your browser now, you should see something similar to our original page, except that instead of the real news data, you'll just see the strings **title**, **published**, and **summary** as seen in the following image:

Let's take a look at how to populate these fields inside our `render_template` call so that we can see real news content again.

Passing dynamic data to our template

First, in our Python file, we'll pass each of these as named variables. Update the `get_news()` function again, and pass all the data that you need to display to the user as arguments to `render_template()`, as follows:

```
@app.route("/")
@app.route("/<publication>"
def get_news(publication="bbc"):
  feed = feedparser.parse(RSS_FEEDS[publication])
  first_article = feed['entries'][0]
  render_template("home.html",
                  title=first_article.get("title"),
                  published=first_article.get("published"),
                  summary=first_article.get("summary"))
```

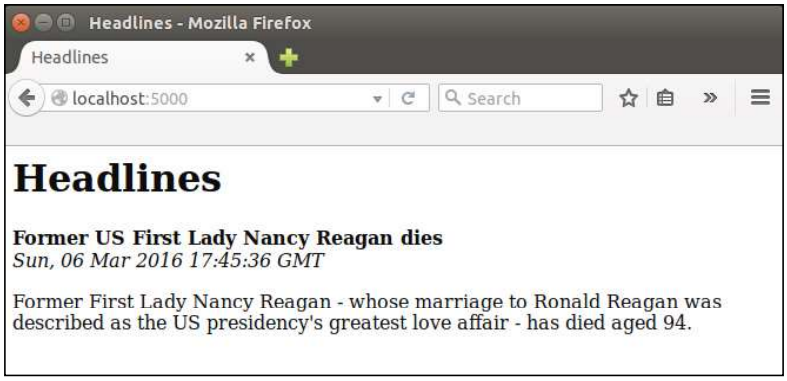
The `render_template` function takes the filename of the template as its first argument, and can then take an arbitrary number of named variables as subsequent arguments. The data in each of these variables will be available to the template, using the variable name.

Displaying dynamic data in our template

In our `home.html` file, we simply need to put two braces on either side of our placeholders. Change it to look like the following:

```
<html>
    <head>
        <title>Headlines</title>
    </head>
    <body>
        <h1>Headlines</h1>
        <b>{{title}}</b><br />
        <i>{{published}}</i><br />
        <p>{{summary}}</p>
    </body>
</html>
```

Double braces, {{ }}, indicate to Jinja that anything inside them should not be taken as literal HTML code. Because our *placeholders*, *title*, *published*, and *summary*, are the same as our Python variable names passed into the `render_template` call, just adding the surrounding braces means that the `render_template` call will substitute these for the real data, returning a pure HTML page. Try it out to make sure that we can see real news data again, as seen in the following image:



Advanced use of Jinja templates

Now we have perfect separation of our backend and frontend components, but our application doesn't do anything more than it did before. Let's take a look at how to display multiple news articles from a selected publication. We don't want to add three new arguments to our `render_template` call for each article (or dozens of additional arguments if we ever decide that we want to display more than just the title, date, and summary of an article).

Fortunately, Jinja can take over some of the logic from Python. This is where we have to be careful: we spent all that effort to separate our logic and view components, and when we discover how powerful the Jinja language actually is, it's tempting to move a lot of the logic into our template files. This would leave us back where we started with code that is difficult to maintain. However, in some cases it's necessary for our frontend code to handle some logic, such as now where we don't want to pollute our backend code with too many repeated arguments.

Using Jinja objects

The first thing to learn is how Jinja handles objects. All of the basic Python data structures, such as variables, objects, lists, and dictionaries, can be understood by Jinja and can be processed in a very similar way to what we are used to in Python. For example, instead of passing each of the three components of our article separately to our template, we could have passed in the `first_article` object and dealt with the separation in Jinja. Let's see how to do that. Change the Python code to pass in a single-named argument to `render_template`, that is `first_article`, and the frontend code to grab the bits we need from this.

The `render_template` call should now look like this:

```
render_template("home.html", article=first_article)
```

The template now has a reference called `article`, which we can use to get the same result as before. Change the relevant part of the home.html to read as follows:

```
<b>{{article.title}}</b><br />
<i>{{article.published</i><br />
<p>{{article.summary}}</p>
```

Note that accessing items from a dictionary is slightly different in Jinja as compared to Python. We use a full stop to access properties, so to access the title of the article, we use `{{article.title}}` as in the preceding example, instead of the Python equivalent `article["title"]` or `article.get("title")`. Our code is again neater, but yet again has no additional functionality.

Adding looping logic to our template

Without much extra effort, we can make the whole list of articles available to Jinja. In the Python code, change the `render_template` call to read as follows:

```
render_template("home.html", articles=feed['entries'])
```

You can remove the line directly above the preceding one in the code which defines the `first_article` variable, as we won't need it any more. Our template now has access to the full list of articles that we fetch through `feedparser`.

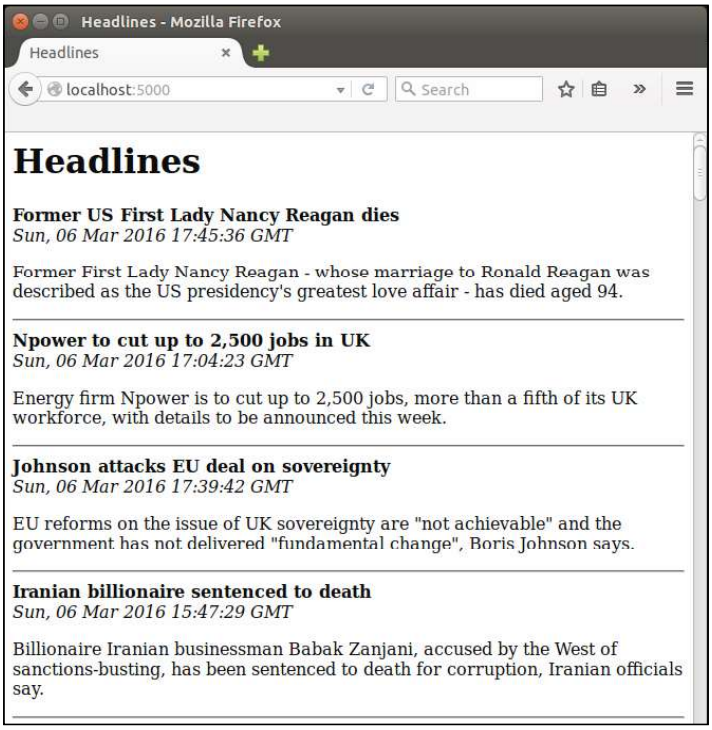
In our Jinja template, we could now `add {{articles}}` or `{{articles[0]}}` to see a full dump of all the information we're now passing, or just a dump of the first article respectively. You can try this as an intermediate step if you're curious, but in our next step we'll be looping through all the articles and displaying the information we want.

By giving our template more data to work with, we're passing along some of the logic responsibility that should ideally be handled by our Python code, but we can also deal with this very cleanly in Jinja. Similar to the way we use double braces, `{{ }}`, to indicate variables, we use the brace and percentage combination, `{% %}`, to indicate control logic. This will be clearer by looking at an example. Change the `<body>` part of the template code to read as follows:

```
<body>
    <h1>Headlines</h1>
    {% for article in articles %}
        <b>{{article.title}}</b><br />
        <i>{{article.published}}</i><br />
        <p>{{article.summary}}</p>
        <hr />
    {% endfor %}
</body>
```

We can see that the Jinja for loop is similar to Python. It loops through the *articles* list that we've passed in from the Python code, and creates a new variable, `article`, for each iteration of the loop, each time referring to the next item in the list. The `article` variable can then be used like any other Jinja variable (using the double braces). Because whitespace in Jinja is irrelevant, unlike Python, we must define where our loop ends with the `{% endfor %}` line. Finally, the `<hr />` in HTML creates a horizontal line which acts as a separator between each article.

Run the application locally with the new template file, and view the results in your browser. You should see something similar to the following image:



Adding hyperlinks to our template

Now we want to link each headline to the original article. Our user will probably find this useful—if a headline seems interesting, he or she can easily get to the full text of the article to read it. The owner of the RSS feed will also often require or request that anyone who uses the feed links back to the original articles. (Again, check for terms and conditions as published along with most big feeds.) Because we're passing the whole `article` object to our template already, we won't need to make any further changes to our Python code to achieve this; we simply need to make use of the extra data already available to us.

In the template file, search for the following:

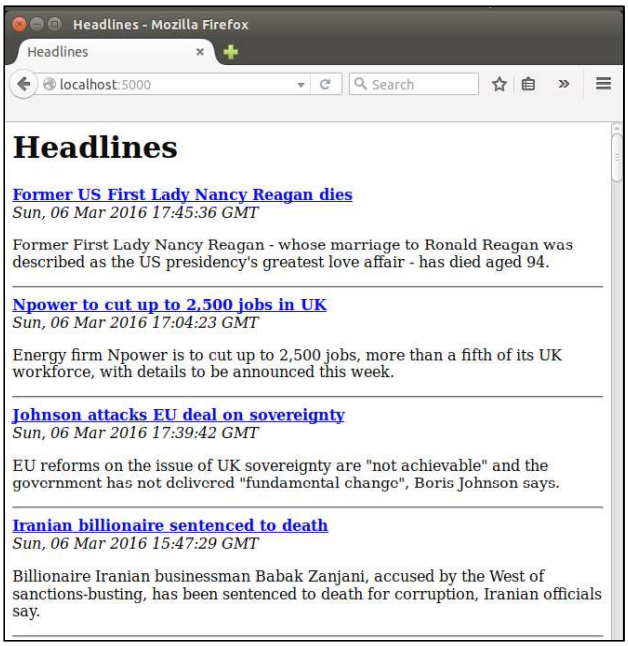
```
<b>{{article.title}}</b><br />
```

Change this line to the following:

```
<b><a href="{{article.link}}">{{article.title}}</a></b><br />
```

If you're new to HTML, then there's quite a bit going on here. Let's pull it apart: the `<a>` tag in HTML indicates a hyperlink (usually displayed by default as blue and underlined in most browsers), the `href` attribute specifies the destination or URL of the link, and the link ends with the `</a>` tag. That is, any text between `<a>` and `</a>` will be clickable, and will be displayed differently by our user's browser. Note that we can use the double braces to indicate a variable even within the double quotation marks used to define the destination attribute.

If you refresh the page in your browser, you should now see the headlines as bold links, as in the following image, and clicking on one of the links should take you to the original article.



Pushing our code to the server

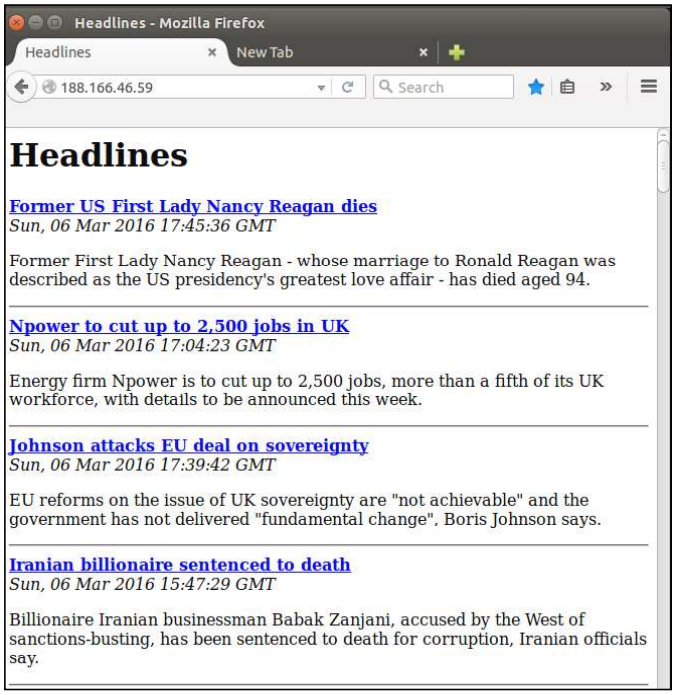
Now is a good time to push the code to our VPS. This is the last time we'll break down the steps of how to do this, but hopefully, you'd be familiar enough with Git and Apache by now that there won't be anything unexpected. On your local machine, from the `headlines` directory, run:

```
git add headlines.py
git add templates
git commit -m "with Jinja templates"
git push origin master
```

And on your VPS (SSH into it as usual), change to the appropriate directory, pull the updates from the Git repository, and restart Apache to reload the code:

```
cd /var/www/headlines
git pull
sudo service apache2 reload
```

Make sure everything has worked by visiting the IP address of your VPS from the web browser on your local machine and checking that you see the same output that we saw locally, as seen in the following image:



Summary

We now have a basic news summary site! You can display recent news from a number of different websites, see the headline, date, and summary for each recent article, and can click on any headline to visit the original article. You've only seen a tiny sample of the power of the Jinja language though—as we expand this project and other projects in future chapters, you'll see how it can be used for inheritance, conditional statements, and more.

In the next chapter, we'll add weather and currency information to our application, and look at ways to interact with our users.

4
User Input for Our Headlines Project

Remember how we allowed the user to specify the publication to be viewed by using `<variable>` parts in our URL? Although we were effectively getting input from our user, it's a way of retrieving input that has some pretty heavy limitations. Let's look at some more powerful ways to interact with our users, and add some more useful information to our application. We'll be making quite a few incremental changes to our code files from here on, so remember that you can always refer to the accompanying code bundle if you need an overview at any point.

In this chapter, we'll look at some more flexible and powerful ways to get input. We'll also bump into some more advanced Git features along the way, and take a moment to explain how to use them.

We'll cover the following topics in this chapter:

- Getting user input using HTTP GET
- Getting user input using HTTP POST
- Adding weather and currency data

Getting user input using HTTP GET

HTTP GET requests are the simplest way of retrieving input from the user. You might have noticed question marks in URLs while browsing the Web. When submitting a term in the search box on the website, your search term will usually appear in the URL, and look something like this:

```
example.com/search?query=weather
```

The bit after the question mark represents a named GET argument. The name is `query` and the value, `weather`. Although arguments like these are usually automatically created through HTML input boxes, the user can also manually insert them into the URL, or they can be part of a clickable link that is sent to the user. HTTP GET is designed to get limited, non-sensitive information from the user in order for the server to return a page as requested by the GET arguments. By convention, GET requests should never modify the server state in a way that produces side effects, that is, the user should be able to make exactly the same request multiple times and always be given exactly the same results.

GET requests are, therefore, ideal for allowing our user to specify which publication to view. Let's extend our Headlines project to incorporate selecting a headline based on a GET request. First, let's modify the Python code to do the following:

- Import the request context from Flask
- Remove the dynamic URL variable
- Check to see if the user has entered a valid publication as a GET argument
- Pass the user query and the publication to the template

Update the `headlines.py` file as follows:

```
import feedparser
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

RSS_FEEDS = {'bbc': 'http://feeds.bbci.co.uk/news/rss.xml',
             'cnn': 'http://rss.cnn.com/rss/edition.rss',
             'fox': 'http://feeds.foxnews.com/foxnews/latest',
             'iol': 'http://www.iol.co.za/cmlink/1.640'}

@app.route("/")
def get_news():
        query = request.args.get("publication")
        if not query or query.lower() not in RSS_FEEDS:
                publication = "bbc"
        else:
                publication = query.lower()
        feed = feedparser.parse(RSS_FEEDS[publication])
        return render_template("home.html",
        articles=feed['entries']
```

```
if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

The first new change is a new import for Flask's request context. This is another piece of Flask magic that makes our life easier. It provides a global context which our code can use to access information about the latest request made to our application. This is useful for us, because the GET arguments that our user passes along as part of a request are automatically available in `request.args`, from which we can access key-value pairs as we would with a Python dictionary (although it is immutable). The request context simplifies some other parts of request handling as well, which means that we don't have to worry about threads or the ordering of requests. You can read more about how the request context works, and what it does, at the following website:

`http://flask-cn.readthedocs.org/en/latest/reqcontext/`

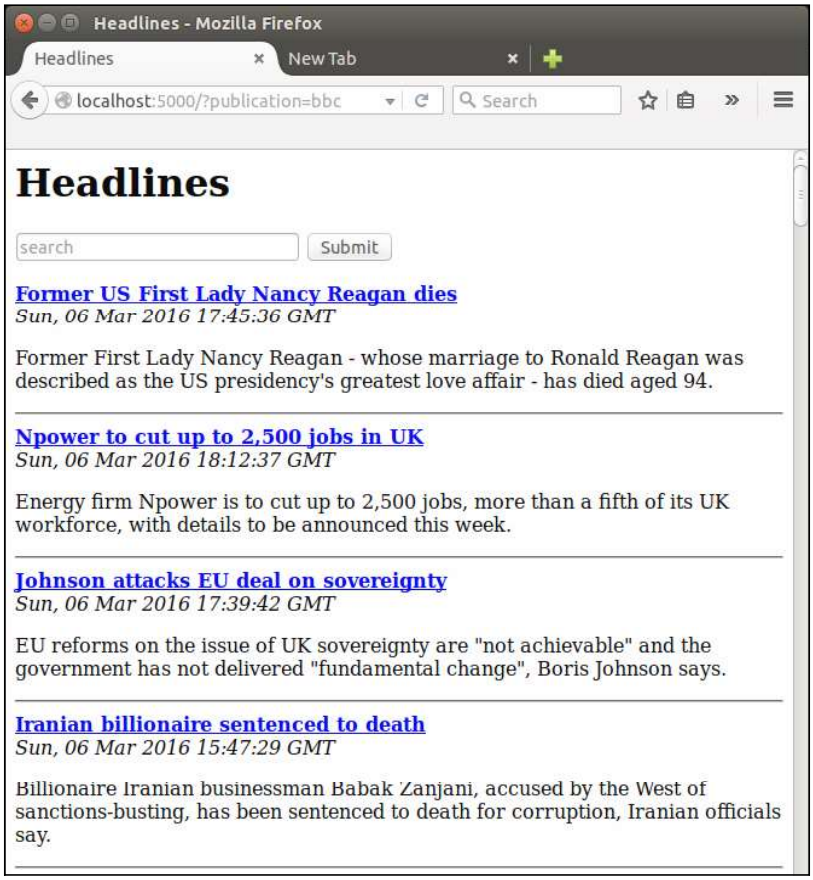
We check to see if this has the publication key set by using the `get()` method, which returns `None`. if the key doesn't exist. If the argument is there, we make sure that the value is valid (that is, it is accounted for by our `RSS_FEEDS` mapping), and if it is, we return the matching publication.

We can test out the code by visiting our URL followed by the `get` argument, for example: `localhost:5000/?publication=bbc`. Unfortunately, from our user's experience, we've made the application less user-friendly, instead of more. Why did we do this? It turns out that our user doesn't have to modify the URL by hand—with a very small change, we can have the URL arguments populated automatically so that the user doesn't have to touch the URL at all. Modify the `home.html` template, and add the following HTML below the heading:

```
<form>
  <input type="text" name="publication" placeholder="search" />
  <input type="submit" value="Submit" />
</form>
```

This is quite straightforward, but let's pick it apart to see how it all works. First we create an HTML form element. By default, this will create an HTTP GET request when submitted, by passing any inputs as GET arguments into the URL. We have a single text input which has the name `publication`. This name is important as the GET argument will use this. The `placeholder` is optional, but it will give our user a better experience as the browser will use it to indicate what the text field is intended for. Finally, we have another input of type `submit`. This automatically creates a nice **Submit** button for our form which, when pressed, will grab any text in the input and submit it to our Python backend.

Save the template, and reload the page to see how it works now. You should see the input form at the top of the page, as seen in the following screenshot. We've gained a lot of functionality for four lines of HTML, and now we can see that, although GET arguments initially looked like they were creating more mission and admin, they actually make our web application much simpler and more user-friendly.



Getting user input using HTTP POST

The alternative to HTTP GET is HTTP POST, and it may not always be immediately obvious which one to use. HTTP POST is used to post larger chunks of data or more sensitive data to the server. Data sent through POST requests is not visible in the URL, and although this does not make it inherently more secure (it does not by default provide encryption or validation), it does offer some security advantages. URLs are often cached by the browser and suggested through autocomplete features next time the user types in a similar URL.

Data sent through GET requests may, therefore, be retained. Using POST also prevents someone from seeing the data by looking over the user's shoulder (shoulder surfing). Passwords especially are often obscured on input by using HTML password fields, making them appear as asterisks (********) or dots (••••••••) in the browser. The data would still be clearly visible in the URL if sent using GET however, and so POST should always be used instead.

Although our search query is hardly confidential or excessively long, we're going to take a moment now to see how we'd implement the same functionality using POST instead of GET. If you just want to get ahead with finishing off our Headlines application, feel free to skip this section, but keep in mind that we'll be using POST requests in later projects without extended explanation. Once we're done with the POST example, we'll revert our application to the state it is currently in (using the GET request), as this is much more suitable for our use case.

Creating a branch in Git

To make a change to our code base that we're not sure if we want, we'll use Git's branch functionality. Think of a branch as being like a fork in a road, except we can at any time change our mind and go back to the decision point. First, we need to make sure our current branch (master) is up to date—that all our local changes are committed. Open a terminal, and run the following commands from the headlines directory:

```
git add headlines.py
git add templates/home.html
git commit -m "Using GET"
git push origin master
```

We don't strictly need to push it to the server—Git keeps a full revision history locally, and our changes would still be theoretically safe without the push. However, our code is in a working state, so there's no harm making the backup to remote. Now we're going to create the new branch and switch to using it to make our next set of changes:

```
git branch post-requests
git checkout post-requests
```

We're now working in a new branch of our codebase. Usually, we'd eventually merge this branch back into our master branch, but in our case, we'll just abandon it once we're done with what we need. It's quite hard to visualize what's happening as Git does most things behind the scenes, so it's worth reading up about Git if you're interested, and are likely to use it for future projects. Otherwise, just think of this as a checkpoint so that we can freely experiment without the worry of messing up our code.

Adding POST routes in Flask

To use a POST request, we need to make some small changes to our Python and HTML code. In the `headlines.py` file, make the following changes:

- Change `request.args.get` to `request.form.get`
- Change `@app.route("/")` to `@app.route("/", methods=['GET', 'POST'])`

The reason for the first change is that we are now grabbing the user data from a form, so Flask automatically makes this available to us in `request.form`. This works the same way as `request.get` except that it gathers data from POST requests instead of from GETs. The second change is not quite as obvious. What we haven't mentioned before is that all route decorators can specify how the function can be accessed: either through GET requests, POST requests, or both. By default, only GET is permitted, but we now want our default page to be accessible by either GET (when we just visit the home main page and are given BBC as a default), or POST (for when we've requested the page through our form with the additional query data). The `methods` parameter accepts a list of HTTP methods which should be permitted to access that particular route of our application.

Making our HTML form use POST

Our template needs similar changes. Change the opening `<form>` tag in the `home.html` file to read:

```
<form action="/" method="POST">
```

Just as with Flask, HTML forms use GET by default, so we have to explicitly define that we want to use POST instead. The `action` attribute isn't strictly necessary, but usually, when we use POST, we redirect users to a confirmation page or similar, and the URL for the following page would appear here. In this case, we're explicitly saying that we want to be redirected to the same page after our form has been submitted.

Save the changes to the Python and HTML files, and refresh the page in your browser to see the changes take effect. The functionality should be exactly the same except that we don't see any data in the URL. This can be cleaner for many applications, but in our case, it is not what we want. For one, we'd like the search term to be cached by our users' browsers. If a user habitually makes a query for FOX, we want the browser to be able to autocomplete this after he begins typing in the URL for our application. Furthermore, we'd like our users to be able to easily share links that include the query.

If a user (let's call him Bob) sees a bunch of interesting headlines after typing **cnn** into our application, and wants to share all of these headlines with another user (Jane), we don't want Bob to have to message Jane, telling her to visit our site, and type a specific query into the search form. Instead, Bob should be able to share a URL that allows Jane to directly visit the page exactly as he saw it (for example, `example.com/?publication=cnn`). Jane can simply click on the link sent by Bob and view the same headlines (assuming she visits our page before the RSS feed is updated).

Reverting our Git repository

We need to revert the code to how we had it before. Because all the changes in the previous section were made in our experimental post-request branch, we don't need to manually re-edit the lines we changed. Instead, we'll commit our changes to this branch, and then switch back to our master branch, where we'll find everything as we left it. In your terminal, run the following:

```
git add headlines.py
git add templates/home.html
git commit -m "POST requests"
git checkout master
```

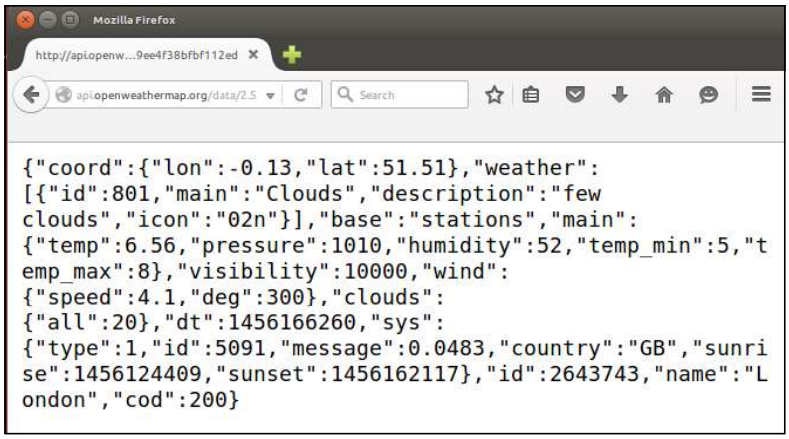
Open the `headlines.py` and `templates/home.html` files to be sure, but they should be exactly as we left them before experimenting with POST!

Adding weather and currency data

Now let's add some more functionality. We're showing media headlines from three different sources, but our user is probably interested in more than current affairs. We're going to see how easy it is to display the current weather and some exchange rates at the top of the page. For the weather data, we'll be using the OpenWeatherMap API, and for currency data, we'll be using Open Exchange Rates. At the time of writing, these APIs are freely available, although they both require registration.

Introducing the OpenWeatherMap API

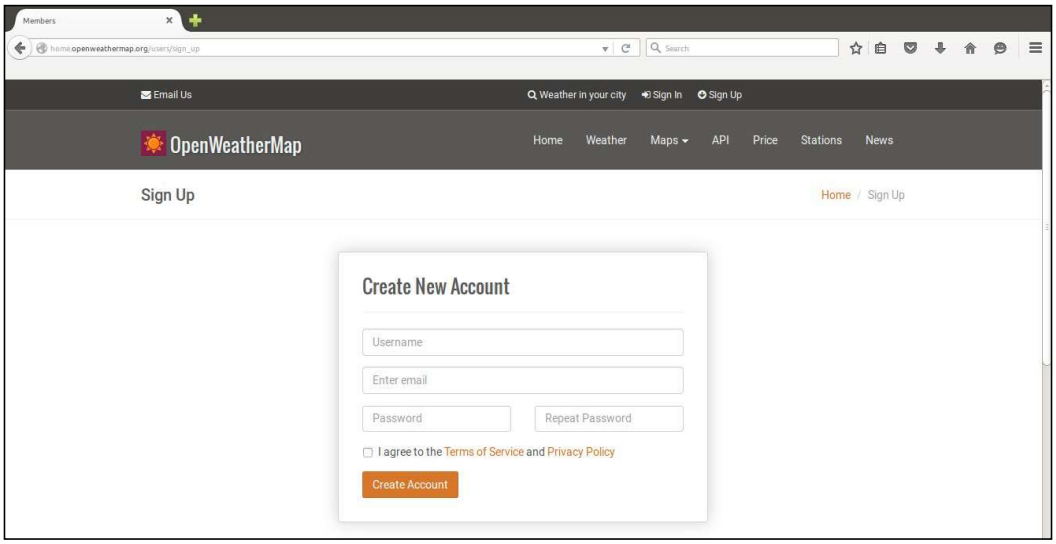
In your web browser, visit the URL `http://api.openweathermap.org/data/2.5/weather?q=London,uk&units=metric&appid=cb932829eacb6a0e9ee4f38bfbf112ed`. You should see something that looks similar to the following screenshot:



This is the JSON weather data for London which is designed to be read automatically instead of by humans. Before looking at how to go about reading this data into our Headlines application, note that the URL we visited has an `appid` parameter. Even though the weather data is provided for free, every developer who accesses the data needs to sign up for a free account with OpenWeatherMap, and get a unique API key to pass as the value for the `appid` parameter. This is to prevent people from abusing the API by making too many requests, and hogging the available bandwidth. At the time of writing, OpenWeatherMap allows 60 calls to the API per minute and 50,000 per day as part of their free access plan, so it's unlikely that we'll be hitting these limits for our project.

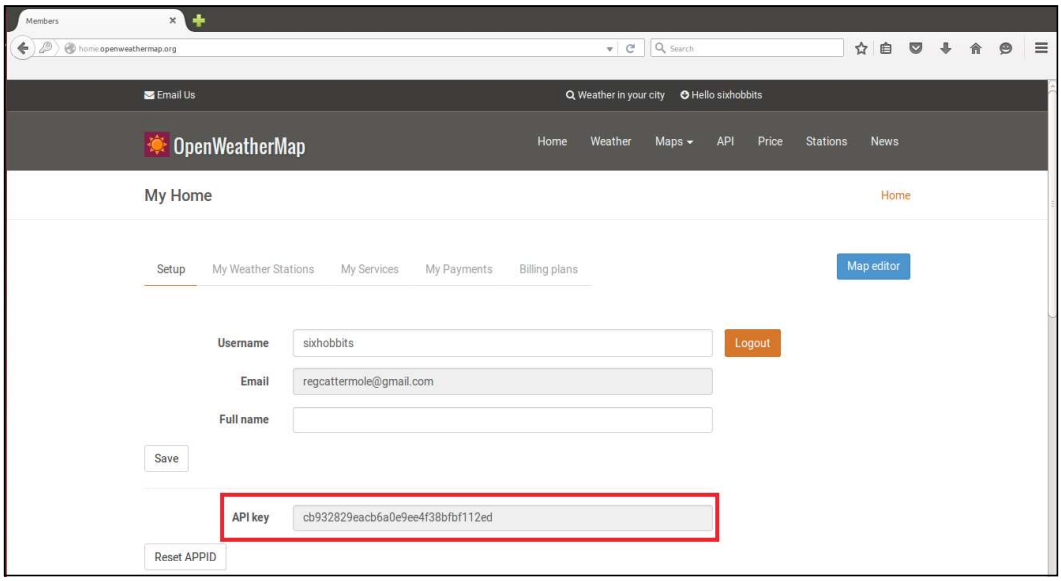
Signing up with OpenWeatherMap

You should sign up for your own API key instead of using the one published in this book. Generally, your API key should remain a secret, and you should avoid sharing it (especially avoid publishing it in a book). To get your own API key, head over to `www.openweathermap.org`, and complete their sign-up progress by clicking the sign-up link at the top of the page. Fill out an e-mail address, username, and password. The registration page should look similar to the following screenshot:



Retrieving your OpenWeatherMap API key

Once you've signed up, you'll be able to log into OpenWeatherMap. You can the find your personal API key by navigating to `home.openweathermap.org` and scrolling down to the **API key** text box. You should see your API key as indicated by the red rectangle in the following image:



Copy the key to your clipboard, as we'll be using it in our Python code soon.

Parsing JSON with Python

Now we can access structured weather data over HTTP by using a URL. But doing so in our browser isn't much good, as we want to read this data automatically from our Python code. Luckily, Python has a bunch of useful standard libraries for exactly this use case!

Introducing JSON

JSON is a structured data format very similar to a Python dictionary, as should be apparent from the preceding sample. In fact, in this case it's identical, and we could very simply convert it to a Python dictionary to use in our Flask application by loading it as a string and running the built-in Python `eval` function on it. However, JSON is not always identical to a Python dictionary. For example, it uses `true` and `false` instead of `True` and `False` (note the case difference)—and passing anything that we don't have full control over to `eval()` is generally a bad idea. Therefore, we'll use the `Python json` library to safely parse it. We'll also use the Python `urllib2` library to download the data from the web, and the Python `urllib` library to correctly encode URL parameters.

Retrieving and parsing JSON in Python

For retrieving and parsing JSON in Python, the first step is to add the three new imports that we need to our `headlines.py` file:

```
import json
import urllib2
import urllib
```

Style tip:

For good Python style, keep the imports ordered alphabetically. You can read more about the conventions for ordering imports at the following site: `https://www.python.org/dev/peps/pep-0008/#imports`

Now add a new function, `get_weather()`, which will make a call to the weather API with a specific query. It's pretty straightforward, and looks like the following code. Replace the `<your-api-key-here>` placeholder with the API key that you copied from the OpenWeatherMap page.

```
def get_weather(query):
    api_url = http://api.openweathermap.org/data/2.5/
    weather?q={}&units=metric&appid=<your-api-key-here>
```

```
    query = urllib.quote(query)
    url = api_url.format(query)
    data = urllib2.urlopen(url).read()
    parsed = json.loads(data)
    weather = None
    if parsed.get("weather"):
        weather = {"description":
                   parsed["weather"][0]["description"],
                  "temperature":parsed["main"]["temp"],
                  "city":parsed["name"]
                   }
    return weather
```

We use the same URL we looked at earlier in our browser, but we make the query part-configurable so that the city for which we retrieve the weather data is dynamic. We use `urllib.quote()` on the query variable, as URLs cannot have spaces in them, but the names of the cities that we want to retrieve weather for may contain spaces. The `quote()` function handles this for us by, for example, translating a space to "`%20`", which is how spaces are represented in URLs. Then we load the data over HTTP into a Python string by using the `urllib2` library. As in our feedparsing example, downloading data over the Internet is always potentially unstable, and for a real-world application, we would need to add some exception handling, and retry logic here.

We then use the json library's `loads()` function (load string) to convert the JSON string that we downloaded into a Python dictionary. Finally, we manually build up a simpler Python dictionary based on the JSON one returned by the API, as OpenWeatherMap supplies a whole bunch of attributes that we don't need for our application.

Using our weather code

Now make two small changes to the `get_news()` function in order to use our `get_weather()` function. We need to call the `get_weather()` function (for now we'll just pass in London as a constant), and then pass the weather data to our template. The `get_news()` function should now look as follows:

```
@app.route("/")
def get_news():
        query = request.args.get("publication")
        if not query or query.lower() not in RSS_FEEDS:
                publication = "bbc"
        else:
                publication = query.lower()
        feed = feedparser.parse(RSS_FEEDS[publication])
        weather = get_weather("London,UK")
```

```
            return render_template("home.html",
            articles=feed["entries"],
            weather=weather)
```

This now loads the simplified data for London into the `weather` variable, and passes it along to our template file so that we can display the data to our users.

Displaying the weather data

Now we just need to adapt our template to account for the extra data. We'll display the weather data just above the news headlines, and add some level 2 headings to keep the different sections of our application organized.

Add the following three lines to the `home.html` template, right after the opening `<h1>` tag:

```
<body>
  <h1>Headlines</h1>
  <h2>Current weather</h2>
  <p>City: <b>{{weather.city}}</b></p>
  <p>{{weather.description}} |{{weather.temperature}}&#8451;</p>
  <h2>Headlines</h2>
```

There's nothing here that we haven't seen before. We simply grab the sections we want out of our weather variable using braces. The funny `&#8451;` part is to display the symbol for degrees Celsius. If you're one of those people who is able to make sense of the notion of Fahrenheit, then remove the `&units=metric` from the API URL (which will tell OpenWeatherData to give us the temperatures in Fahrenheit), and display the *F* symbol for our users by using `&#8457;` in your template instead.

Allowing the user to customize the city

As mentioned earlier, we would not always want to display the weather for London. Let's add a second search box for city! Searching is usually hard, because data input by users is never consistent, and computers love consistency. Luckily, the API that we're using does a really good job of being flexible, so we'll just pass on the user's input directly, and leave the difficult bit for others to deal with.

Adding another search box to our template

We'll add the search box to our template exactly as before. This form goes directly under the *Current weather* heading in the `home.html` file.

```
<form>
  <input type="text" name="city" placeholder="weather search">
```

```
  <input type="submit" value="Submit">
</form>
```

The form defined in the preceding code snippet simply uses a named text input and a submit button, just like the one we added for the publication input.

Using the user's city search in our Python code

In our Python code, we need to look for the `city` argument in the GET request. Our `get_news()` function is no longer well-named, as it does more than simply getting the news. Let's do a bit of refactoring. Afterwards, we'll have a `home()` function that makes calls to get the news and the weather data (and later on the currency data), and our `get_news()` function will again only be responsible for getting news. We're also going to have quite a few defaults for different things, so instead of hard-coding them all, we'll add a `DEFAULTS` dictionary as a global, and whenever our code can't find information in the GET arguments, it'll fall back to getting what it needs from there. The changed parts of our code (excluding the imports, global URLs, and the main section at the end) now look like this:

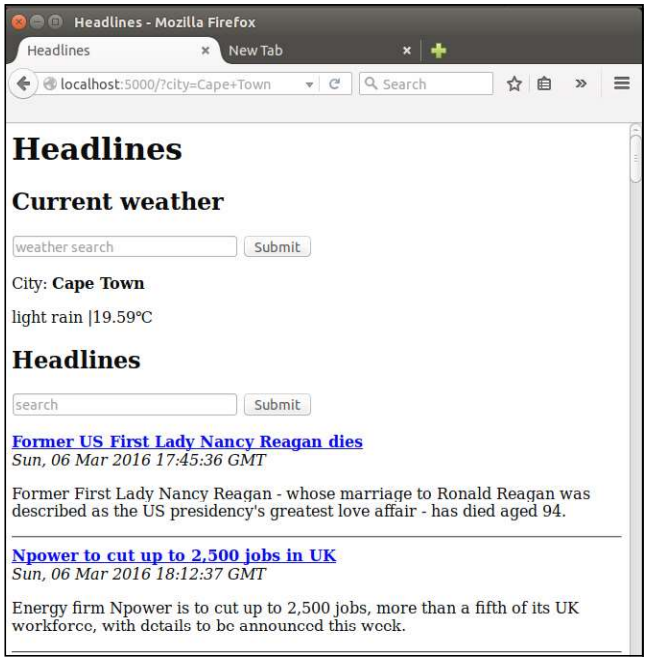
```
# ...

DEFAULTS = {'publication':'bbc',
            'city': 'London,UK'}

@app.route("/")
def home():
    # get customized headlines, based on user input or default
    publication = request.args.get('publication')
    if not publication:
        publication = DEFAULTS['publication']
    articles = get_news(publication)
    # get customized weather based on user input or default
    city = request.args.get('city')
    if not city:
        city = DEFAULTS['city']
    weather = get_weather(city)
return render_template("home.html", articles=articles,
weather=weather)

def get_news(query):
    if not query or query.lower() not in RSS_FEEDS:
        publication = DEFAULTS["publication"]
    else:
        publication = query.lower()
    feed = feedparser.parse(RSS_FEEDS[publication])
```

```
        return feed['entries']

    def get_weather(query):
        query = urllib.quote(query)
        url = WEATHER_URL.format(query)
        data = urllib2.urlopen(url).read()
        parsed = json.loads(data)
        weather = None
        if parsed.get('weather'):
            weather =
            {'description':parsed['weather'][0]['description'],
             'temperature':parsed['main']['temp'],
             'city':parsed['name']
            }
        return weather
```

Now we have a good separation of concerns—our `get_weather()` function gets weather data, our `get_news()` function gets news, and our `home()` function combines the two and handles the user's input to display customized data to our visitors.

Checking our new functionality

If all went well, we should now have a site that displays customizable news and weather data. The weather search, as mentioned, is pretty flexible. Give it a go with some different inputs—you should see a page similar to the following image:



Handling duplicate city names

The OpenWeatherMap API handles duplicate city names well, although the defaults are sometimes a bit counter-intuitive. For example, if we search for Birmingham, we'll get the one in the USA. If we want to look for the Birmingham in the UK, we can search for Birmingham, UK. In order to not confuse our viewers, we'll make a small modification for displaying the country next to the city. Then they'll immediately be able to see if they get results for a city different from what they were expecting. If you examine the full API response from our weather call, you'll find the country code listed under `sys`—we'll grab that, add it to our custom dictionary, and then display it in our template.

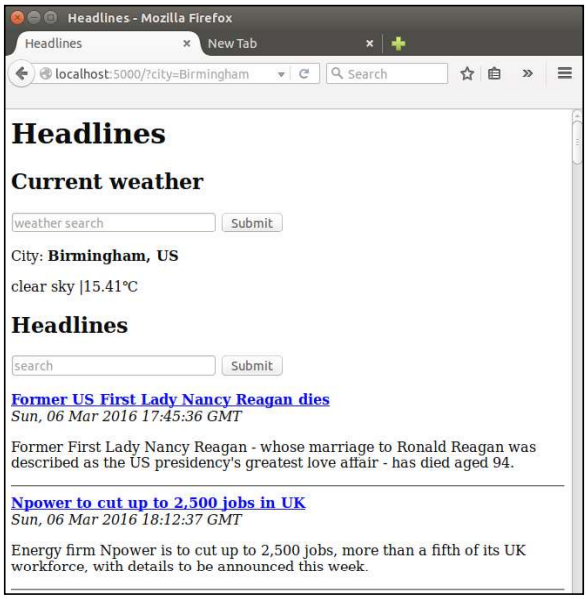
In the `get_weather` function, modify the line where we build the dictionary:

```
weather = {'description': parsed['weather'][0]['description'],
           'temperature': parsed['main']['temp'],
           'city': parsed['name'],
           'country': parsed['sys']['country']
          }
```

And in our template, modify the line where we display the city to read as follows:

```
<p>City: <b>{{weather.city}}, {{weather.country}}</b></p>
```

Check that its working – if you restart the application and reload the page, you should see that typing `Birmingham` into to the **Current weather** search box now displays the country code next to the city name.



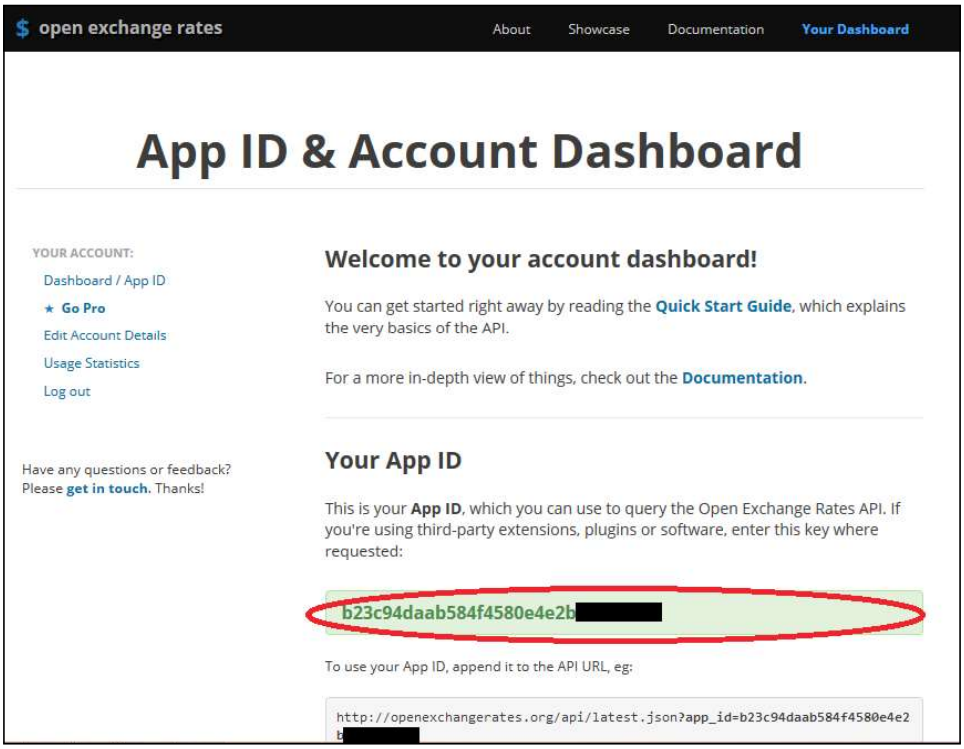
Currency

Currency data is considered more valuable than weather data. Many commercial services offer APIs that are frequently updated and very reliable. However, the free ones are a bit rare. One service that offers a limited API for free is Open Exchange Rates—and again, we need to register a free account to get an API key.

Getting an API key for the Open Exchange Rates API

Head over to `openexchangerates.com`, and complete their registration process. After clicking on the **Sign up** link, it may look like they only have paid plans, as these are more prominently displayed. However, underneath the large paid plan options, there is a single line of text describing their free offering with a link to select it. Click on this, and enter your details.

If you are not automatically redirected, head over to your dashboard on their site, and you'll see your **API key** (App ID) displayed. Copy this, as we'll need to add it to our Python code. You can see an example of where to find your API key in the following screenshot:



Using the Open Exchange Rates API

The `currency` API returns JSON just like the `weather` API, so we can integrate it into our Headlines application very easily. We need to add the URL as a global, and then add a new function to calculate rates. Unfortunately, the free version of the API is restricted to returning all the major currencies against the United States Dollar, so we will have to calculate our own approximate rates for conversions not involving the dollar, and rely on a perfect market to keep our information as accurate as possible (see `http://en.wikipedia.org/wiki/Triangular_arbitrage`).

Add the variable `CURRENCY_URL` to your globals below the existing `WEATHER_URL`, as seen in the following code snippet. You'll need to substitute your own App ID.

```
WEATHER_URL =
"http://api.openweathermap.org/data/2.5/weather?q={}
&units=metric&APPID=<your-api-key-here>"
CURRENCY_URL =
"https://openexchangerates.org//api/latest.json?
app_id=<your-api-key-here>"
```

Add the `get_rates()` function as follows:

```
def get_rate(frm, to):
        all_currency = urllib2.urlopen(CURRENCY_URL).read()

        parsed = json.loads(all_currency).get('rates')
        frm_rate = parsed.get(frm.upper())
        to_rate = parsed.get(to.upper())
        return to_rate/frm_rate
```

Note the calculation that we do at the end. If the request was from USD to any of the other currencies, we could simply grab the correct number from the returned JSON. But in this case, the calculation is simple enough, and it's therefore not worth adding the extra step of logic to work out if we need to do the calculation or not.

Using our currency function

Now we need to call the `get_rates()` function from our `home()` function, and pass the data through to our template. We also need to add default currencies to our `DEFAULTS` dictionary. Make the changes as indicated by the highlighted code that follows:

```
DEFAULTS = {'publication':'bbc',
            'city': 'London,UK',
            'currency_from':'GBP',
            'currency_to':'USD'
```

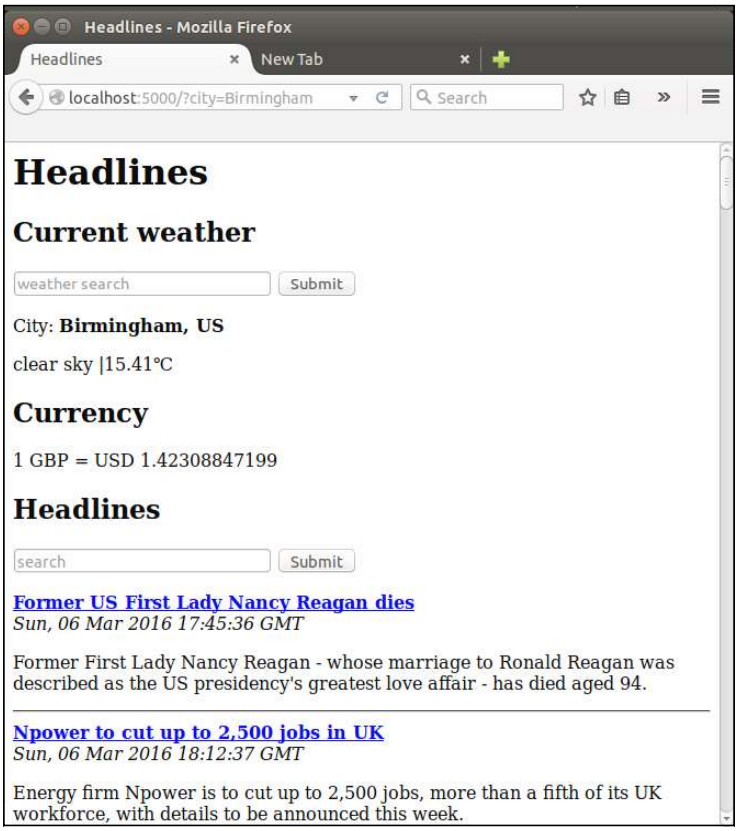
```
}

@app.route("/")
def home():
    # get customized headlines, based on user input or default
    publication = request.args.get('publication')
    if not publication:
        publication = DEFAULTS['publication']
    articles = get_news(publication)
    # get customized weather based on user input or default
    city = request.args.get('city')
    if not city:
        city = DEFAULTS['city']
    weather = get_weather(city)
    # get customized currency based on user input or default
    currency_from = request.args.get("currency_from")
    if not currency_from:
        currency_from = DEFAULTS['currency_from']
    currency_to = request.args.get("currency_to")
    if not currency_to:
        currency_to = DEFAULTS['currency_to']
    rate = get_rate(currency_from, currency_to)
    return render_template("home.html", articles=articles,
    weather=weather,
                           currency_from=currency_from, currency_
to=currency_to, rate=rate)
```

Displaying the currency data in our template

Finally, we need to modify our template to display the new data. Underneath the weather section in `home.html`, add:

```
<h2>Currency</h2>
1 {{currency_from}} = {{currency_to}} {{rate}}
```

As always, check that everything is working in your browser. You should see the default currency data of the British Pound to US Dollar conversion as in the following image:



Adding inputs for the user to select currency

Now we need to add yet another user input to customize which currencies to display. We could easily add another text search like we did for the previous two, but this gets messy. We need two bits of input from the user: the *from* currency and the *to* currency. We could add two inputs, or we could ask the user to enter both into the same input, but the former makes our page pretty cluttered, and the latter means we need to worry about properly splitting the user input data (which is almost certainly not consistent). Instead, let's look at a different input element, the HTML `select`. You've almost certainly seen these on other web pages—they're drop-down menus with a list of values that the user can choose from. Let's see how to build them in HTML, and how to grab the data from them in Flask.

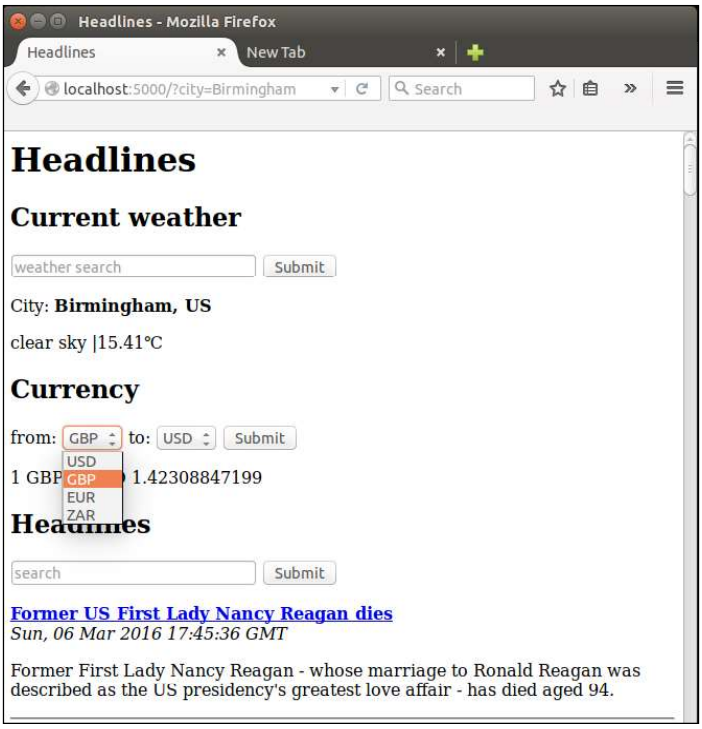
Creating an HTML select drop-down element

First, let's hard-code four currencies in each drop-down menu. The code should be inserted right below the **Currency** heading in the `home.html` template, and it looks like this:

```
<form>
    from: <select name="currency_from">
            <option value="USD">USD</option>
            <option value="GBP">GBP</option>
            <option value="EUR">EUR</option>
            <option value="ZAR">ZAR</option>
          </select>

     to: <select name="currency_to">
           <option value="USD">USD</option>
           <option value="GBP">GBP</option>
           <option value="EUR">EUR</option>
           <option value="ZAR">ZAR</option>
         </select>
         <input type="submit" value="Submit">
</form>
```

The name used for the GET request argument is an attribute of the select tag itself (similar to the name attribute used in our `<input type="text">` tags). In our case, these are `currency_from` and `currency_to`, which we specified in our Python code earlier. The value is slightly more tricky—we have the value that's passed in our GET request (for example, `currency_from=EUR`), and then the value that is displayed to the user. In this case, we'll use the same for both—the currency code—but this is not compulsory. For example, we could use the full name of the currency, such as United States Dollar, in the display value, and the code in the value that's passed in the request. The argument value is specified as an attribute of the option tags, each a child of `<select>`. The display value is inserted between the opening and closing `<option>` and `</option>` tags.

Test this out to make sure it's working, by saving the template and reloading the page. You should see drop-down inputs appear, as in the following image:



Adding all the currencies to the select input

Of course, we could do what we did in the preceding section for the full list. But we're programmers, not data capturers, so we'll make the list dynamic, insert the options using a `for` loop, and keep our template up-to-date and clean. To get the list of currencies, we can simply take the keys of our JSON `all_currency` object, in order to make our `get_rate()` function return a tuple—the calculated rate and the list of currencies. We can then pass the (sorted) list to our template, which can loop through them and use them to build the drop-down lists. The changes for this are shown as follows:

Make the following changes in the `home()` function:

```
    if not currency_to:
      currency_to=DEFAULTS['currency_to']
rate, currencies = get_rate(currency_from, currency_to)
return render_template("home.html", articles=articles,
weather=weather,
currency_from=currency_from, currency_to=currency_to,
rate=rate,
currencies=sorted(currencies))
```

In the `get_rate()` function:

```
frm_rate = parsed.get(frm.upper())
to_rate = parsed.get(to.upper())
return (to_rate / frm_rate, parsed.keys())
```

And in the `home.html` template:

```
<h2>Currency</h2>
<form>
        from: <select name="currency_from">
              {% for currency in currencies %}
                  <optionvalue="{{currency}}">
                  {{currency}}</option>
              {% endfor %}
              </select>

        to: <select name="currency_to">
             {% for currency in currencies %}
                 <option value="{{currency}}">
                 {{currency}}</option>
             {% endfor %}

            </select>
        <input type="submit" value="Submit">
</form>
1 {{currency_from}} = {{currency_to}} {{rate}}
```

Displaying the selected currency in the drop-down input

After this, we should easily be able to see the exchange rate for any currency we want. One minor irritation is that the dropdowns always display the top item by default. It would be more intuitive for our users if they displayed the currently selected value instead. We can do this by setting the `selected="selected"` attribute in our select tag and a simple, one-line Jinja `if` statement to work out which line to modify. Change the `for` loops for the currency inputs in our `home.html` template to read as follows:

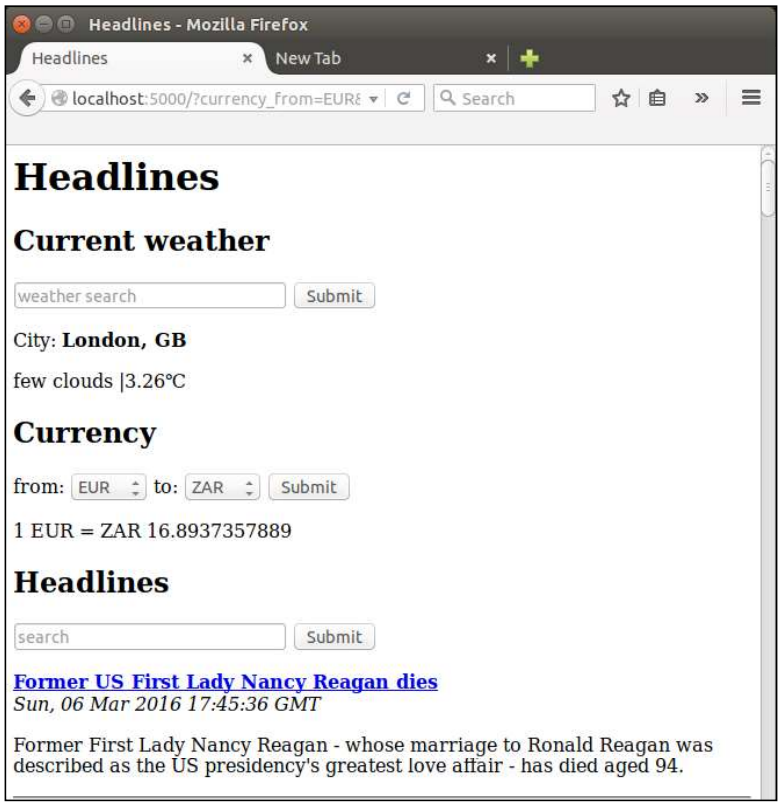
For the `currency_from` loop:

```
{% for currency in currencies %}
    <option value="{{currency}}"
    {{'selected="selected"' if currency_from==currency}}>
    {{currency}}</option>
{% endfor %}
```

For the `currency_to` loop:

```
{% for currency in currencies %}
    <option value="{{currency}}"
    {{'selected="selected"' if currency_to==currency}}>
    {{currency}}</option>
{% endfor %}
```

Reload the application and the page, and you should now be able to select any of the available currencies from both select inputs, and after the page has loaded with the desired currency data, the select inputs should automatically display the current currencies as well, as seen in the following image. After clicking on the select input, you should also be able to type on your keyboard and select the option based on the first letters of what you've typed.



We can now see news, weather, and currency data at the same time! You can refer to the complete code from the code bundle of the chapter.

Summary

In this chapter, we've looked at the difference between the HTTP GET and POST requests, and discussed where it's good to use which. Although we have no good use for HTTP POST at the moment, we will use it in future projects where we will be getting login data from our users. Luckily, the explanatory work we did with HTTP POST is not lost—we also took a look at some more advanced ways that Git can help us with version control, and our unused code is safely stored in a different branch of our code repository in case we need to refer back to it later. Last but not least, we added weather and currency data to our application, and looked at a few different options for allowing our user to input data into our application. We're nearly done with our first project!

In the next chapter, we'll do some cosmetic touch-ups, and look at remembering our users so that they don't have to carry out exactly the same actions every time they visit our site.