main.py

CODE WITH REPL.IT

FROM BEGINNER TO EXPERT
THROUGH GUIDED TUTORIALS

repl.it

CONTENT BY RITZA

Programming walkthroughs: Coding with Python and Repl.it

Gareth Dwyer

Contents

Introduction

This book is built from a collection of tutorials written over several months. Each chapter is mainly independent so there is no need to follow the chapters in order: pick a project that you are excited about and get coding!

While independent, some chapters share topics. A summary of each chapter is given below.

Beginner web scraping with Python and Repl.it

This is a beginner's introduction to web scraping where the reader will be shown how the web works and how HTML is structured and will build a Python scraper to download specific content from a specific site.

Building news word clouds using Python and Repl.it

This can be followed independently of the previous tutorial but assumes some of the knowledge shared there. The reader will build a web scraper using the Python Newspaper library to remove extra "boilerplate" content.

Building a Discord Bot with Python and Repl.it

Here the reader learns how to build a basic echo bot using the Discord API. It leaves the reader with a basic bot set up that can be customized further.

Building a Discord bot with Node.js and Repl.it

This is simply a translation of the Python version, but it's still recommended for developers to go through both so that they can see the differences (and similarities) between the languages in a real project.

Creating and hosting a basic web application with Django and Repl.it

Here the reader learns how to set up a Django project from scratch and host it with Repl.it.

Building a CRM app with NodeJS, Repl.it, and MongoDB

In this tutorial the reader builds a CRM (Customer Relationship Manager) with NodeJS, showing basic crud applications. This is not a translation of the Django tutorial but many of the same concepts are covered.

Introduction to Machine Learning with Python and repl.it

Here the reader learns about some basic Machine Learning (text classification) using Repl.it. This is a high level tutorial showing how to train a classifier and generate predictions and does not get into the maths involved.

Quicksort tutorial: Python implementation with line by line explanation

This is the most abstract and theoretical of the collection and shows the reader how to implement quicksort from scratch. This is more useful for interview practice or understanding university homework.

Beginner web scraping with Python and Repl.it

In this guide, we'll walk through how to automatically grab data from web sites. Most websites are created with a *human* audience in mind - you use a search engine or type a URL into your web browser, and see information displayed on the page. Sometimes, we want to *automatically* extract and process this data, and this is where web scraping can save us from boring repetitive labour. We can create a custom computer program to visit web sites, extract specific data and process this data in a specific way.

We'll be extracting news data from the bbc.com[1] news website, but you should be able adapt it to extract information from any website that you want with a bit of trial and error.

For example, you might need to:

- extract numbers from a report that is released weekly and published online
- grab the schedule for your favourite sports team as it's released
- find the release dates for upcoming movies in your favourite genre
- be notified automatically when a website changes

There are many other use cases for web scraping. However, you should also note that copyright law and web scraping laws are complex and differ by country. While people generally don't mind if you aren't blatantly copying their content or doing web scraping for commercial gain, there have been some legal cases involving scraping data from LinkedIn[2], media attention from scraping data from OKCupid[3], and in general web scraping can violate law, go against a particular website's terms of service, or breach ethical guidelines.

With the disclaimer out of the way, let's learn how to scrape!

Overview and requirements

Specifically, in this tutorial we'll cover

- What a website really is and how HTML works
- Viewing HTML in your web browser

[1] https://bbc.com/news
[2] https://techcrunch.com/2016/08/15/linkedin-sues-scrapers/
[3] https://www.engadget.com/2016/05/13/scientists-release-personal-data-for-70-000-okcupid-profiles/

- Using Python to download web pages
- Using BeautifulSoup[4] to extract parts of scraped data

We'll be using the online programming environment repl.it[5] so you won't need to install any software locally to follow along step by step. If you want to adapt this guide to your own needs, you should create a free account by going to repl.it[6] and following their sign up process.

It would help if you have basic familiarity with Python or another high level programming language, but we'll be explaining each line of code we write in detail so you should be able to keep up, or at least replicate the result, even if you don't.

Webpages: beauty and the beast

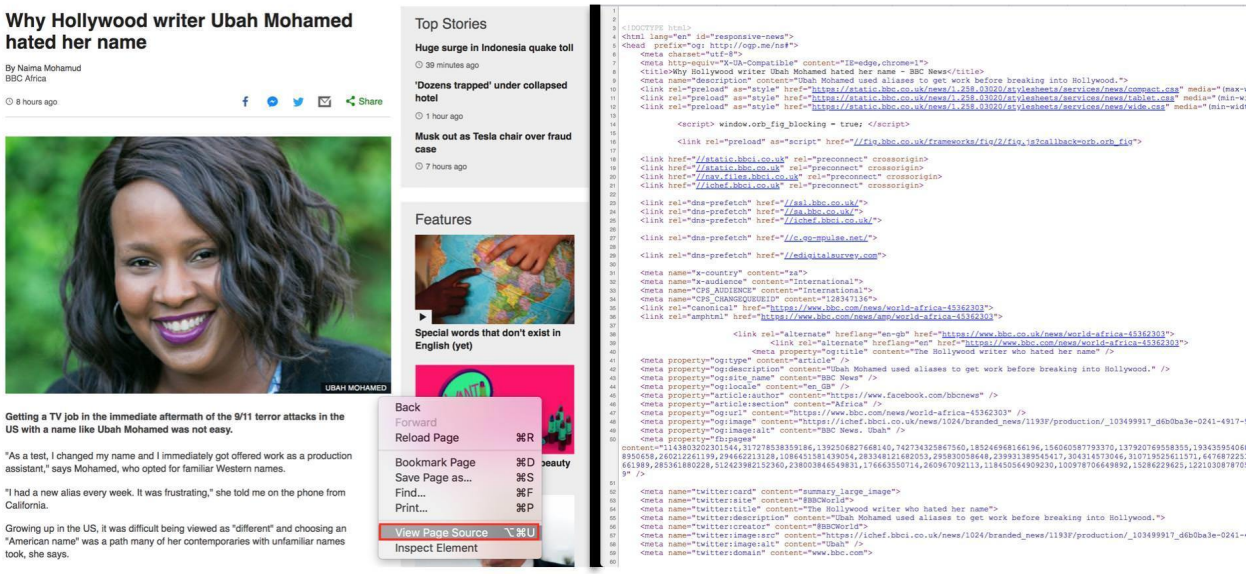
You have no doubt visited web pages using a web browser before. Websites exist in two forms:

1. The one you are used to where you can see text, images, and other media. Different fonts, sizes, and colours are used to display information in a useful and (normally) aesthetic way.
2. The "source" of the webpage. This is the computer code that tells your web browser (e.g. Mozilla Firefox or Google Chrome) what to display and how to display it.

Websites are coded in a combination of three computer languages: HTML, CSS and JavaScript. This in itself is a huge and complicated field with a messy history, but having a basic understanding of how some of it works is necessary to effectively automate web scraping. If you open any website in your browser and right click somehwere on the page, you'll see a menu which should include an option to "view source" – to inspect the true form of a website before it is interpreted by your web browser.

This is shown in the image below: a normal web page on the left an open context menu (displayed by right clicking on the page). Clicking this produces the result on the right – we can see the code that contains all the data and supporting information that the web browser needs to display the complete page. While the page on the left is easy to read and use, and looks good, the one on the right is a monstrosity. It takes some effort and experience to make any sense of it, but it's possible to do so and it's necessary if we want to write custom web scrapers.

[4] https://www.crummy.com/software/BeautifulSoup/
[5] https://repl.it
[6] https://repl.it



Normal and source view of the same BBC news article.

Navigating the source code using Find

The first thing to do is to work out how the two pages correspond: which parts of the normally displayed website match up to which parts of the code. You can use "find" Ctrl + F) in the source code view to find specific pieces of text that are visible in the normal view to help with this. In the story on the left we can see that the story starts with the phrase "Getting a TV job". If we search for this phrase in the code view, we can find the corresponding text within the code, on line 805.

Finding text in the source code of a web page.

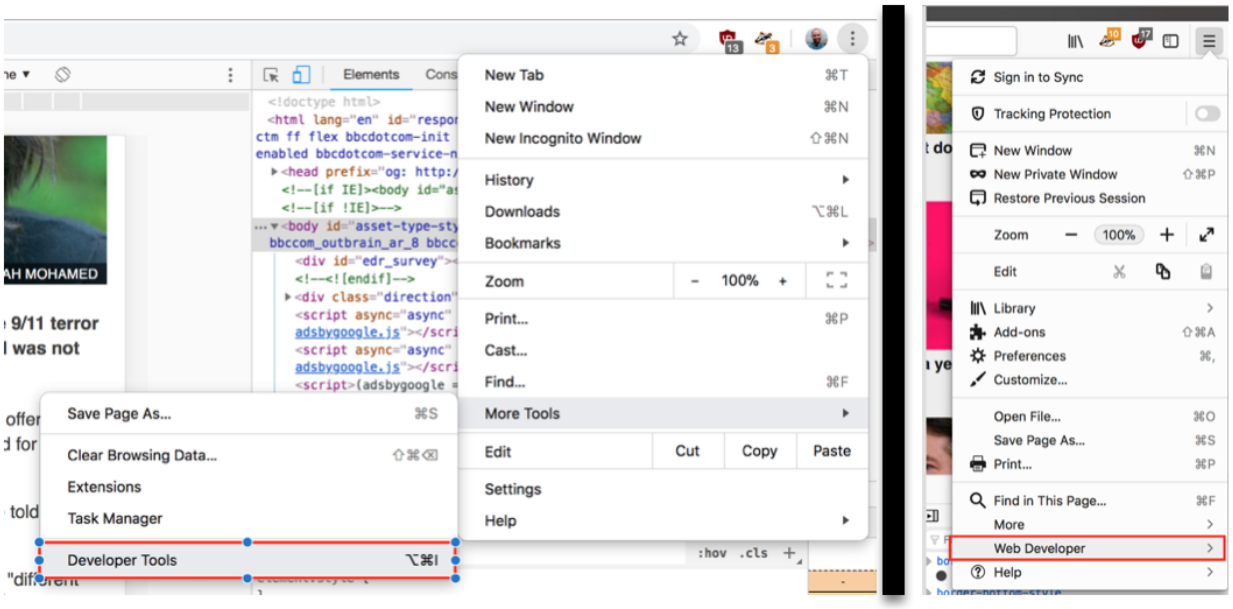
The `<p class="story-body__introduction">` just before the highlighted section is HTML code to specify that a paragraph (`<p>` in HTML) starts here and that this is a special kind of paragraph (an introduction to the story). The paragraph continues until the `</p>` symbol. You don't need to worry about understanding HTML completely, but you should be aware that it contains **both** the text data that makes up the news article and additional data about how to display the article.

A large part of web scraping is viewing pages like this to a) identify the data that we are interested in and b) to separate this from the markup and other code that it is mixed with. Even before we start writing our own code, it can be tricky to first understand other people's.

In most pages, there is a lot of code to define the structure, layout, interactivity, and other functionality of a web page, and relatively little that contains the actual text and images that we usually view. For especially complex pages, it can be quite difficult, even with the help of the find function, to locate the code that is responsible for a particular part of the page. To help with this, most web browsers come with so-called "developer tools", which are aimed primarily at programmers to assist in the creation and maintenance of web sites, but these tools are also very useful for doing web scraping.

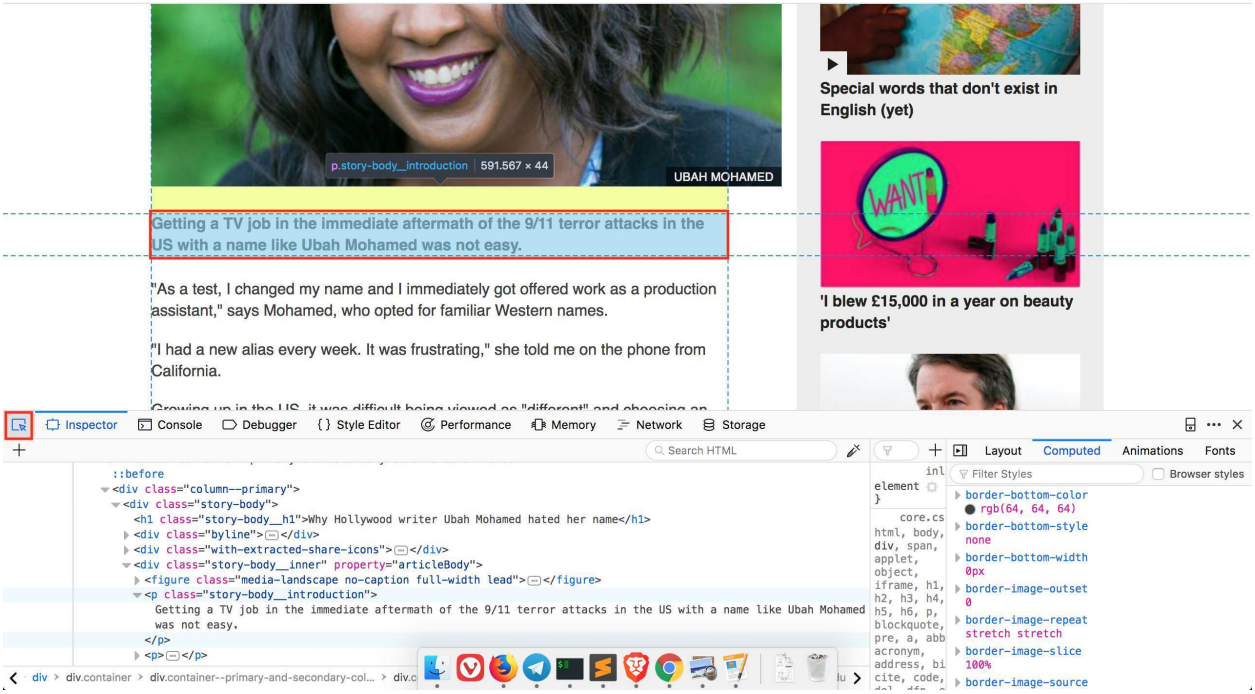
Navigating the source code using developer tools

You can open the developer tools for your browser from the main menu, with Google Chrome shown on the left and Mozilla Firefox on the right below. If you're using a different web browser, you should be able to find a similar setting.



Opening Developer Tools in Chrome (left) and Firefox (right)

Activating the tool brings up a new panel in your web browser, normally at the bottom or on the right-hand side. The tool contains an "Inspector" panel and a selector tool, which can be chosen by pressing the icon highlighed in red below. Once the selector tool is active, you can click on parts of the web page to view the corresponding source code. In the image below, we selected the same first paragraph in the normal view and we can see the `<p class=story-body__introduction">` code again in the panel below.



Viewing the code for a specific element using developer tools

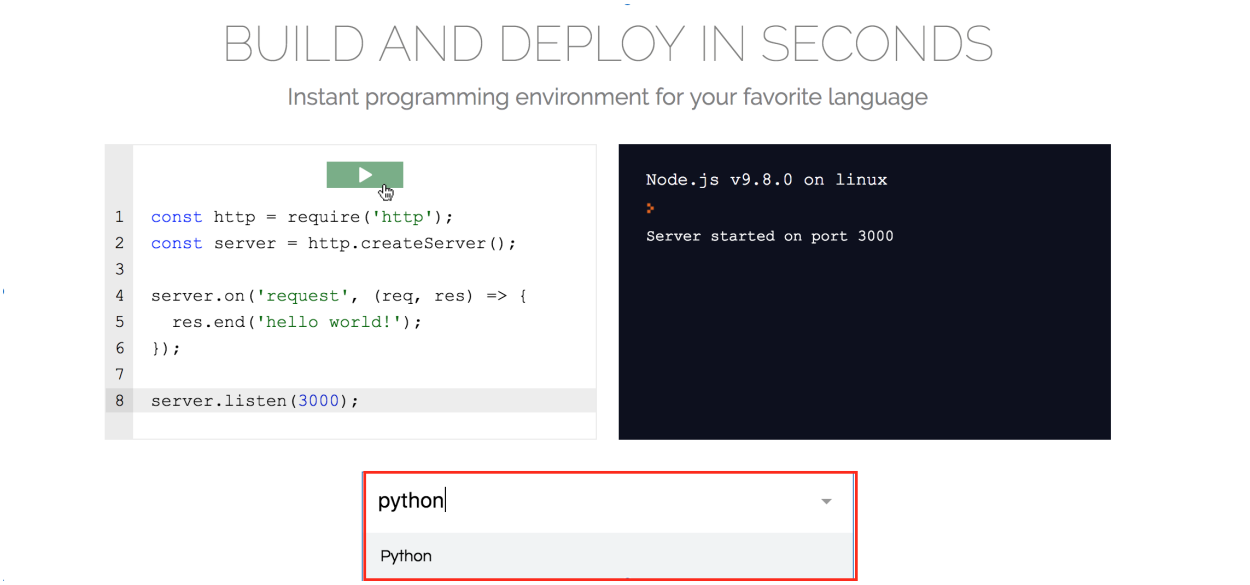
The Developer Tools are significantly more powerful than using the simple find tool, but they are also more complicated. You should choose a method based on your experience and the complexity of the page that you are trying to analyze.

Downloading a web page with Python

Now that we've seen a bit more of how web pages are built in our browser, we can start retrieving and manipulating them using Python. Python is not a web browser, so we will not get a 'normal' representation of a web page through Python. Instead, we'll only be able to retrieve and manipulate the HTML source code.

We'll do this through a Python Repl using the `requests` library. Open repl.it[7] and choose to create a new Python repl.

[7] https://repl.it

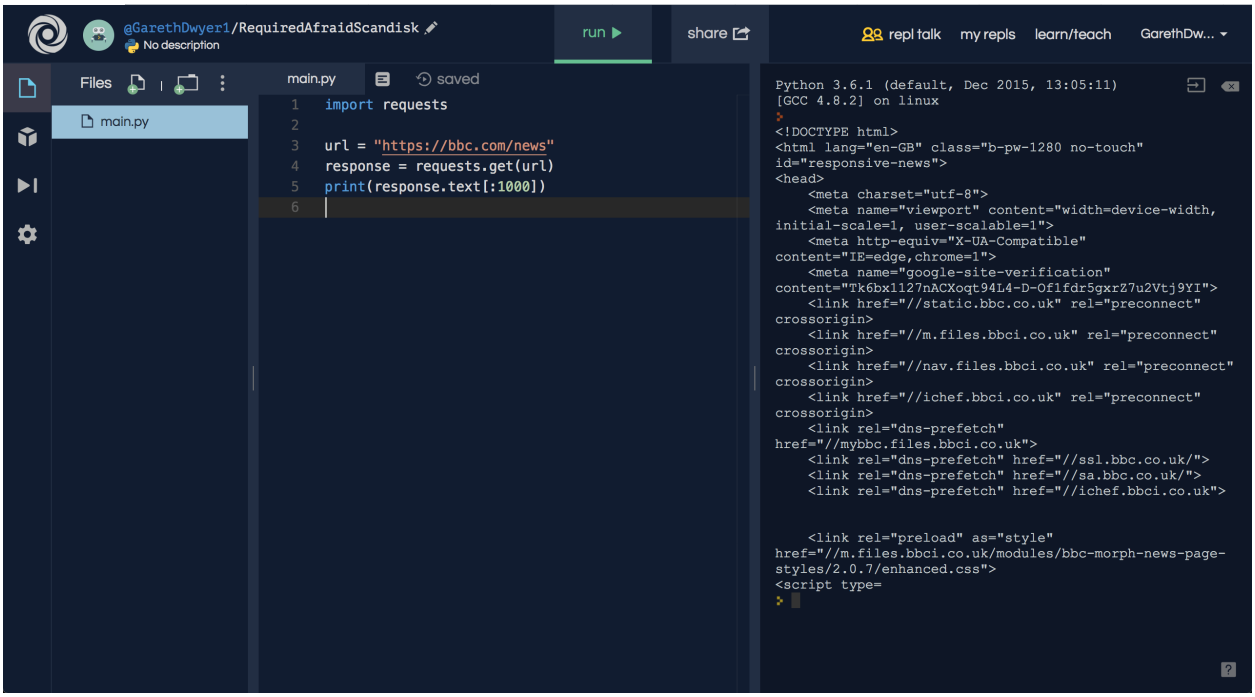


This will take you to a working Python coding environment where you can write and run Python code. To start with, we'll download the content from the BBC News homepage, and print out the first 1000 characters of HTML source code.

You can do this with the following four lines of Python:

```
import requests

url = "https://bbc.com/news"
response = requests.get(url)
print(response.text[:1000])
```

Put this code in the `main.py` file that Repl automatically creates for you and press the "Run" button. After a short delay, you should see the output in the output pane - the beginning of HTML source code, similar to what we viewed in our web browser above.



Downloading a single page using Python

Let's pull apart each of these lines.

- In line 1, we import the Python `requests` library, which is a library that allows to make web requests.
- In line 3, we define a variable containing the URL of the main BBC news site. You can visit this URL in your web browser to see the BBC News home page.
- In line 4, we pass the URL we defined to the `requests.get` function, which will visit the web page that the URL points to and fetch the HTML source code. We load this into a new variable called `response`.
- In line 5, we access the `text` attribute of our `response` object, which contains all of the HTML source code. We take only the first 1000 characters of this, and pass them to the `print` function, which simply dumps the resulting text to our output pane.

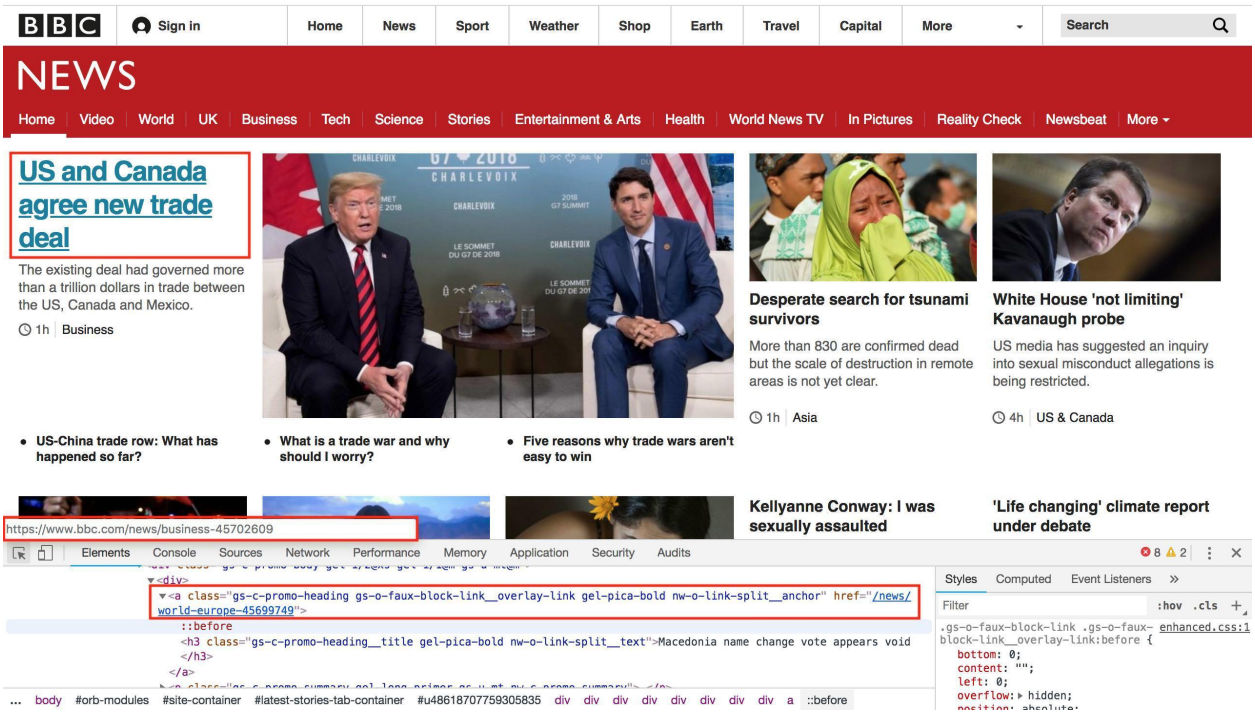
We have now automatically retrieved a web page and we can display parts of the content. We are unlikely to be interested in the full source code dump of a web page (unless we are storing it for archival reasons), so let's extract some interesting parts of the page instead of first 1000 characters.

Using BeautifulSoup to extract all URLs

The world wide web his built from pages that link to each other using Hyperlinks, links, or URLs. (These terms are all used more-or-less interchangably).

Let's asume for now that we want to find all the news articles on the BBC News homepage, and get their URLs. If we look at the main page below, we'll see there are a bunch of stories on the home page and mousing over any of the headlines with the "inspect" tool, we can see that each has a unique URL which takes us to that news story. For example, mousing over the main "US and Canada agree new trade deal" story in the image below is a link to https://www.bbc.com/news/business-45702609.

If we inspect that element using the browser's developer tools, we can see it is a `<a>` element, which is HTML for a link, with an `<href>` component that points to the URL. Note that the `href` section goes only to the last part of the URL, omitting the https://www.bbc.com part. Because we are already on BBC, the site can use *relative URLs* instead of *absolute URLs*. This means that when you click on the link, your browser will figure out that the URL isn't complete and prepend it with https://www.bbc.com. If you look around the source code of the main BBC page, you'll find both relative and absolute URLs, and this already makes scraping all of the URLs on the page more difficult.



Viewing headline links using Developer Tools.

We could try to use Python's built-in text search functions like `find()` or regular expressions to extract all of the URLs from the BBC page, but it is not actually possible to do this reliably. HTML is a complex language which allows web developers to do many unusual things. For an amusing take on why we should avoid a "naive" method of looking for links, see this very famous[8] StackOverflow question and the first answer.

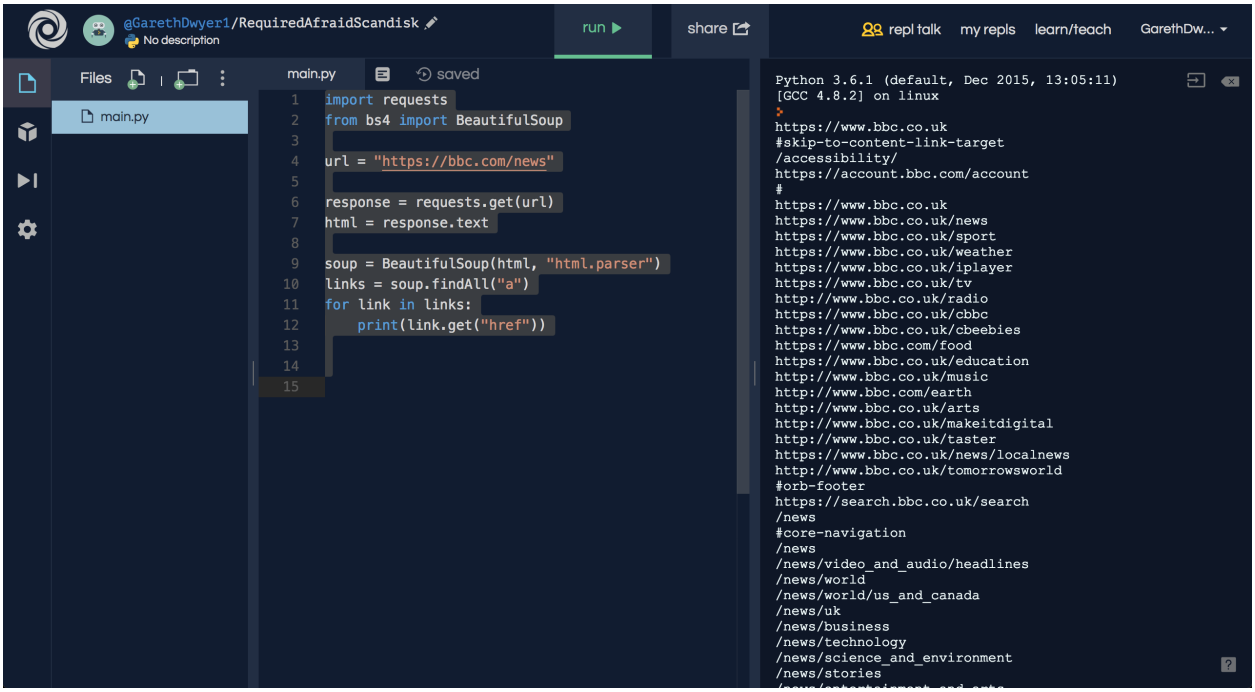
Luckily, there is a powerful and simple-to-use HTML parsing library called BeautifulSoup[9], which

[8] https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags
[9] https://www.crummy.com/software/BeautifulSoup/

will help us extract all the links from a given piece of HTML. We can use it by modifying the code in our Repl to look as follows.

```
1  import requests
2  from bs4 import BeautifulSoup
3
4  url = "https://bbc.com/news"
5
6  response = requests.get(url)
7  html = response.text
8
9  soup = BeautifulSoup(html, "html.parser")
10 links = soup.findAll("a")
11 for link in links:
12     print(link.get("href"))
```

If you run this code, you'll see that it outputs dozens of URLs, one per line. You'll probably notice that the code now takes quite a bit longer to run than before – BeautifulSoup is not built into Python, but is a third-party module. This meant that before running the code, Repl had to go and fetch this library and install it for you. Subsequent runs will be faster.



Extracting all links from BBC News.

The code is simlar to what we had before with a few additions.

- On line 2, we import the BeautifulSoup library, which is used for parsing and processing HTML.

- One line 9, we transform our HTML into "soup". This is BeautifulSoup's representation of a web page, which contains a bunch of useful programatic features to search and modify the data in the page. We use the "html.parser" option to parse HTML which is included by default – BeautifulSoup also allows you specify a custom HTML parser here. For example, you could install and specify a faster parser which can be useful if you need to process a lot of HTML data.
- In line 10, we find all the `a` elements in our HTML and extract them to a list. Remember when we were looking at the URLs using our web browser (Image 7), we noted that the `<a>` element in HTML was used to define links, with the `href` attribute being used to specify where the link should go to. This line finds all of the HTML `<a>` elements.
- In line 11, we loop through all of the links we have, and in line 12 we print out the `href` section.

These last two lines show why BeautifulSoup is useful. To try and find and extract these elements without it would be really difficult, but now we can do it in two lines of readable code!

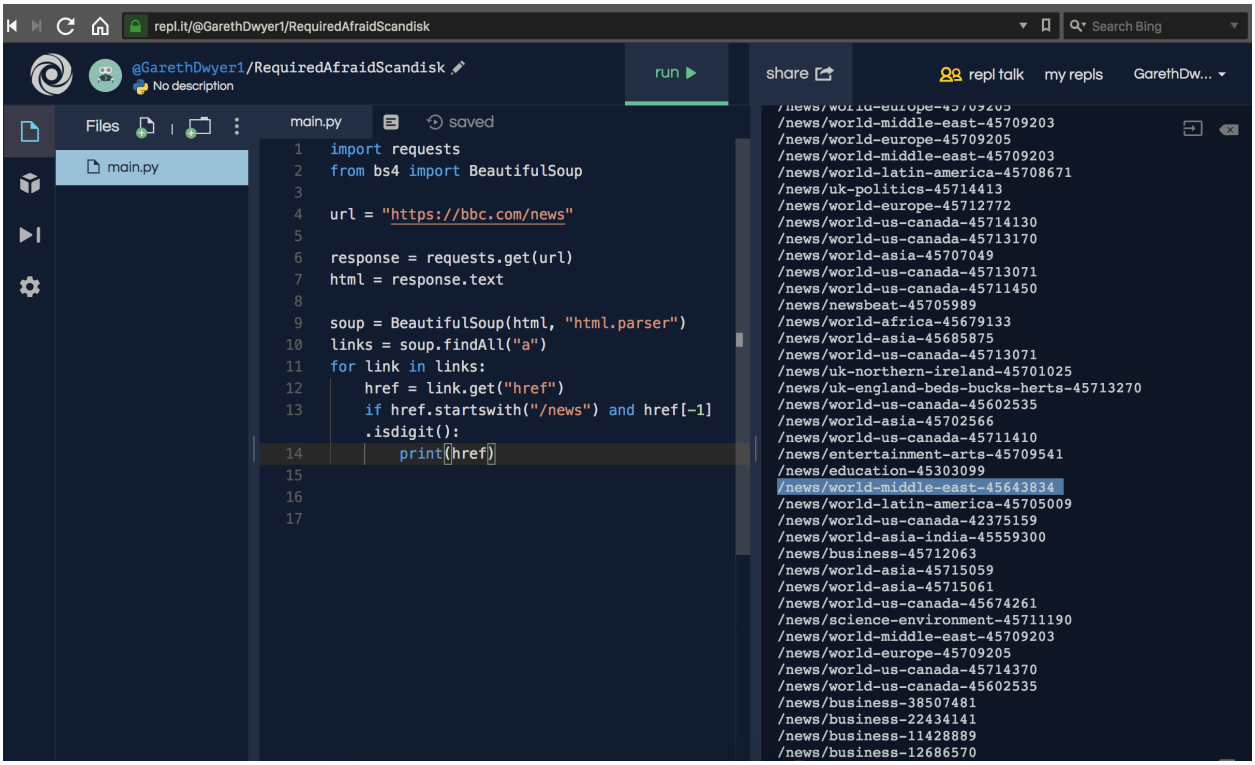
If we look at the URLs in the output pane, we'll see quite a mixed bag of results. We have absolute URLs (starting with "http") and relative ones (starting with "/"). Most of them go to general pages rather than specific news articles. We need to find a pattern in the links we're interested in (that go to news articles) so that we can extract only those.

Again trial and error is the best way to do this. If we go to the BBC News home page and use developer tools to inspect the links that go to news articles, we'll find that they all have a similar pattern. They are relative URLs which start with "/news" and end with a long number, e.g. `/news/newsbeat-45705989`

We can make a small change to our code to only output URLs that match this pattern. Replace the last two lines of our Python code with the following four lines:

```
1  for link in links:
2      href = link.get("href")
3      if href.startswith("/news") and href[-1].isdigit():
4          print(href)
```

Here we still loop through all of the links that BeautifulSoup found for us, but now we extract the `href` to its own variable immediately after. We then inspect this variable to make sure that it matches our conditions (starts with "/news" and ends with a digit), and only if it does then we print it out.



Printing only links to news articles from BBC.

Installing BeautifulSoup through requirements.txt

Repl makes it easy to install third-party Python libraries such as BeautifulSoup through their universal installer[10]. However, as BeautifulSoup is not part of standard Python you would usually have to install it separately.

If you are not using Repl to follow along this tutorial, you'll have to install BeautifulSoup with your favourite package manager, e.g.

```
pip3 install beautifulsoup4
```

You can also explicity specify to Repl which packages to install by creating a `requirements.txt` file in the root of your project. To do this, press the "Add file" button in the top left of your Repl environment, and name the new file `requirements.txt` (the exact name is important). As you add package names to this file, they'll get automatically installed.

BeautifulSoup is called `beautifulsoup4` in the Python Package Index[11], so we need to use that name here. In the new `requirements.txt` file, add a the line `beautifulsoup4`.

You'll see the package get installed as you finish typing the package name (look in the output pane on the right). You should see the phrase "successfully installed" somewhere in the output if all went

[10] https://repl.it/site/blog/python-import
[11] https://pypi.org/

well.

Developers often make changes to their libraries to improve them, but sometimes these improvements can break existing code that relies on the libraries. If you want to make sure that your Repl works well into the future, you can lock a specific version of BeautifulSoup. You'll notice from the output above that the latest version (installed by default) of BeautifulSoup is 4.6.3, so we can modify the line in our `requirements.txt` file to read `beautifulsoup4==4.6.3`. This would lock the current version of BeautifulSoup in place, ensuring that we are unaffected by any updates to the library (but will also stop us from benefiting from any improvements that the BeautifulSoup developers make).

Fetching all of the articles from the homepage

Now that we have the link to every article on the BBC News homepage, we can fetch the data for each one of these individual articles. As a toy project, let's extract the proper nouns (people, places, etc) from each article and print out the most common ones to get a sense on what things are being talked about today.

Adapt our code to look as follows:

```
import requests
import string

from collections import Counter

from bs4 import BeautifulSoup


url = "https://bbc.com/news"


response = requests.get(url)
html = response.text
soup = BeautifulSoup(html, "html.parser")
links = soup.findAll("a")

news_urls = []
for link in links:
    href = link.get("href")
    if href.startswith("/news") and href[-1].isdigit():
        news_url = "https://bbc.com" + href
        news_urls.append(news_url)


```

```
all_nouns = []
for url in news_urls[:10]:
    print("Fetching {}".format(url))
    response = requests.get(url)
    html = response.text
    soup = BeautifulSoup(html, "html.parser")

    words = soup.text.split()
    nouns = [word for word in words if word.isalpha() and word[0] in string.ascii_up\
percase]
    all_nouns += nouns

print(Counter(all_nouns).most_common(100))
```

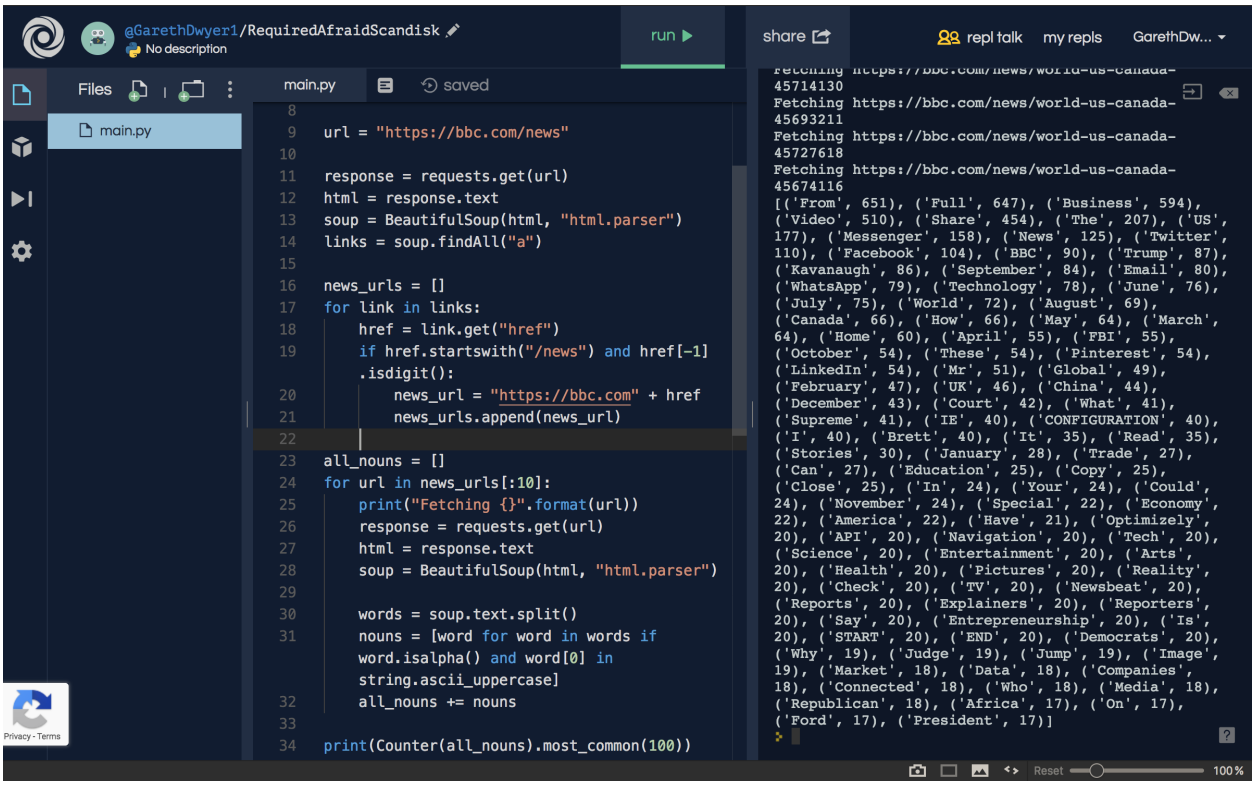
This code is quite a bit more complicated than what we previously wrote, so don't worry if you don't understand all of it. The main changes are

- At the top, we add two new imports. One for `string` which is standard Python module that contains some useful word and letter shortcuts. We'll use it to identify all the capital letters in our alphabet. The second module is a Counter which will lets us find the most common nouns in a list, once we have built a list of all the nouns.
- We've added `news_urls = []` at the top of the first `for` loop. Instead of printing out each URL once we've identified it as a "news URL", we add it to this list so we can download each page later. Inside the for loop two lines down, we combine the root domain ("http://bbc.com") with each href attribute, and then add the complete URL to our `news_urls` list.
- We then go into another for loop, where we loop through the first 10 news URLs (if you have more time, you can remove the `[:10]` part to iterate through all the news pages, but for efficiency we'll just demonstrate with the first 10).
- We print out the URL that we're fetching (as it takes a second or so to download each page, it's nice to display some feedback so we can see that the program is working).
- We then fetch the page and turn it into Soup, as we did before.
- With `words = soup.text.split()` we extract all the text from the page and split this resulting big body of text into individual words. The Python `split()` function splits on white space, which is a horribly crude way to extract words from a piece of text, but it will serve our purpose for now.
- The next line loops through all the words in that given article and keeps only the ones that are made up of numeric characters and which start with a capital letter (`string.ascii_uppercase` is just the uppercase alphabet). This is also an extremely crude way of extracting nouns, and we will get a lot of words (like those at the start of sentences) which are not actually proper nouns, but again it's a good enough approximation for now.
- Finally, we add all the words that look like nouns to our `all_nouns` list and move on to the next article to do the same

Once we've downloaded all the pages, we print out the 100 most common nouns along with a count of how often they appeared using Python's convenience `Counter` object, that's part of the built-in `collections` module.

You should see output similar to that in the image below (though your words will be different, as the news changes every few hours). We have the most common "nouns" followed by a count of how often that noun appeared in all 10 of the articles we looked at.

We can see that our crude extraction and parsing methods are far from perfect – words like "Twitter" and "Facebook" appear in most articles because of the social media links at the bottom of each article, so their presence doesn't mean that Facebook and Twitter themselves are in the news today. Similarly words like "From" aren't nouns, and other words like "BBC", "Optimizely" and "Business" are also included because they appear on each page, outside of the main article text.



The final output of our program, showing the words that appear most often in BBC articles.

Where next?

We've completed the basics of web scraping, and looked at how the web works, how to extract information from web pages, and how to do some very basic text extraction. You will probably want to do something different than extract words from BBC! You can fork this Repl from https://repl.it/@GarethDwyer1/beginnerwebscraping and modify it to change which site it scrapes

and what content it extracts. You can also join the Repl Discord Server[12] to chat with other developers who are working on simlar projects and who will happily exchange ideas with you or help if you get stuck.

We have walked through a very flexible method of web scraping, but it's the "quick and dirty" way. If BBC updates their website and some of our assumptions (e.g. that news URLs will end with a number) break, our web scraper will also break.

Once you've done a bit of web scraping, you'll notice that the same patterns and problems come up again and again. Because of this, there are many frameworks and other tools that solve these common problems (finding all the URLs on the page, extracting text from the other code, dealing with changing web sites, etc), and for any big web scraping project, you'll definitely want to use these instead of starting from scratch.

Some of the best Python web scraping tools are:

- **Scrapy**[13]: A framework used by people who want to scrape millions or even billions of web pages. Scrapy lets you build "spiders" – programmatic robots that move around the web at high speed, gathering data based on rules that you specify.
- **Newspaper**[14]: we touched on how it was difficult to separate the main text of an online news article from all the other content on the page (headers, footers, adverts, etc). This problem is actually an incredibly difficult one to solve. Newspaper uses a combination of manually specified rules and some clever algorithms to remove the "boilerplate" or non-core text from each article.
- **Selenium**[15]: we scraped some basic content without using a web browser, and this works fine for images and text. Many parts of the modern web are dynamic though – e.g. they only load when you scroll down a page far enough, or click on a button to reveal more content. These dynamic sites are very difficult to scrape automatically, but selenium allows you to fire up a real web browser and control it just as a human would (but automatically), which allows you to automatically access this kind of dynamic content.

There is not shortage of other tools, and a lot can be done simply by using them in combination with each other. Web scraping is a large world that we've only just touched on, but we'll explore some more web scraping use cases in the next chapter.

[12] https://discord.gg/QWFfGhy
[13] https://scrapy.org/
[14] https://github.com/codelucas/newspaper
[15] https://www.seleniumhq.org/

Building news word clouds using Python and Repl.it

Word clouds are a popular way to visualise large amounts of text. Word clouds are images showing scattered words in different sizes, where words that appear more frequently in the given text are larger, and less common words are smaller or not shown at all.

In this tutorial, we'll build a web application using Python and Flask that transforms the latest news stories into word clouds and displays them to our visitors.

Our users will see a page similar to the one shown below, but containing the latest news headlines from BBC news. We'll learn some tricks about web scraping, RSS feeds, and building image files directly in memory along the way.

Overview

We'll be building a simple web application step-by-step, and explaining each line of code in detail. To follow, you should have some basic knowledge of programming and web concepts, such as what if statements are and how to use URLs. Specifically, we'll:

- Look at RSS feeds and how to use them in Python
- Show how to set up a basic Flask web application
- Use BeautifulSoup to extract text from online news articles
- Use WordCloud to transform the text into images
- Import Bootstrap and add some basic CSS styling

We'll be using Python, but we won't assume that you're a Python expert and we'll show how to set up a full Python environment online so you won't need to install anything locally to follow along.

Web scraping

We previously looked at basic web scraping in an introduction to web scraping[16]. If you're completely new to the idea of automatically retrieving content from the internet, have a look at that tutorial first.

In this tutorial, instead of scraping the links to news articles directly from the BBC homepage, we'll be using RSS feeds[17] - an old but popular standardised format that publications use to let readers know when new content is available.

[16] https://www.codementor.io/garethdwyer/beginner-web-scraping-with-python-and-repl-it-nzr27jvnq
[17] https://en.wikipedia.org/wiki/RSS

Taking a look at RSS Feeds

RSS feeds are published as XML documents. Every time BBC (and other places) publishes a new article to their home page, they also update an XML machine-readable document at http://feeds.bbci.co.uk/news/world/r
This is a fairly simple feed consisting of a `<channel>` element, which has some meta data and then a list of `<item>` elements, each of which represents a new article. The articles are arranged chronologically, with the newest ones at the top, so it's easy to retrieve new content.

If you click on the link above, you won't see the XML directly. Instead it has some associated styling information so that most web browsers will display something that's a bit more human friendly. For example, opening the page in Google Chrome shows the page below. In order to view the raw XML directly, you can right click on the page and click "view source".

RSS feeds are used internally by software such as the news reader Feedly[18] and various email clients. We'll be consuming these RSS feeds with a Python library to retrieve the latest articles from BBC.

Setting up our online environment (Repl.it)

In this tutorial, we'll be building our web application using repl.it[19], which will allow us to have a consistent code editor, environment, and deployment framework in a single click. Head over there and create an account. Choose to create a Python repl, and you should see an editor where you can write and run Python code, similar to the image below. You can write Python code in the middle pane, run it by pressing the green "run" button, and see the output in the right pane. In the left pane, you can see a list of files, with `main.py` added there by default.

The first thing we want to do to access RSS feeds easily is install the Python feedparser[20] library. To install it in our Repl environment, we'll need to create a new file called `requirements.txt`. To do this, press the small "new file" button in the left pane, call the file exactly `requirements.txt` (Repl will recognise that this is a special file so it's important to name it correctly), and add the single line `feedparser` to this file to tell Repl to install the `feedparser` library.

Pulling data from our feed and extracting URLs

In the previous webscraping tutorial[21] we used BeautifulSoup[22] to look for hyperlinks in a page and extract them. Now that we are using RSS, we can simply parse the feed as described above to find these same URLs.

Let's start by simply printing out the URLs for all of the latest articles from BBC. Switch back to the `main.py` file in the Repl.it IDE and add the following code.

[18] feedly.com
[19] repl.it
[20] https://pythonhosted.org/feedparser/
[21] https://www.codementor.io/garethdwyer/beginner-web-scraping-with-python-and-repl-it-nzr27jvnq
[22] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

```
1  import feedparser
2
3  BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
4  feed = feedparser.parse(BBC_FEED)
5
6  for article in feed['entries']:
7      print(article['link'])
```

Feedparser does most of the heavy lifting for us, so we don't have to get too close to the slightly cumbersome XML format. In the code above, we parse the feed into a nice Python representation (line 4), loop through all of the `entries` (the `<item>` entries from the XML we looked at earlier), and print out the `link` elements.

If you run this code, you should see a a few dozen URLs output on the right pane, as in the image below.

Setting up a web application with Flask

We don't just want to print this data out in the Repl console. Instead our application should return information to anyone who uses a web browser to visit our application. We'll therefore install the lightweight web framework Flask[23] and use this to serve web content to our visitors.

Switch back to the `requirements.txt` file and add a new line `flask` .This should now look as follows.

Navigate back to the `main.py` file and modify our code to look as follows.

```
1  import feedparser
2  from flask import Flask
3
4  app = Flask(__name__)
5
6  BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
7
8  @app.route("/")
9  def home():
10     feed = feedparser.parse(BBC_FEED)
11     urls = []
12
13     for article in feed['entries']:
14         urls.append(article['link'])
15
16     return str(urls)
```

[23] http://flask.pocoo.org/

```
17
18
19 if __name__ == '__main__':
20     app.run('0.0.0.0')
```

Here we still parse the feed and extract all of the latest article URLs, but instead of printing them out, we add them to a list (`urls`), and return them from a function. The interesting parts of this code are

- **Line 2**: we import Flask
- **Line 4**: we initialise Flask to turn our project into a web appliaction
- **Line 8**: we use a decorator to define the homepage of our application (an empty route, or `/`).
- **Lines 19-20**: We run Flask's built-in web server to serve our content.

Press "run" again, and you should see a new window appear in Repl in the top right. Here we can see a basic web page (viewable already to anyone in the world by sharing the URL you see above it), and we see the same output that we previously printed to the console.

Downloading articles and extracting the text

The URLs aren't that useful to us, as we eventaully want to display a summary of the *content* of each URL. The actual text of each article isn't included in the RSS feed that we have (some RSS feeds contain the full text of each article), so we'll need to do some more work to download each article. First we'll add the third-party libraries `requests` and `BeautifulSoup` as dependencies. We'll be using these to download the content of each article from the URL and strip out extra CSS and JavaScript to leave us with plain text.

In `requirements.txt` add the two new dependencies. Your file should now look like the one in the image below (I've kept them ordered alphabetically, which is good practice but not a strict requirement).

Now we're ready to download the content from each article and serve that up to the user. Modify the code in `main.py` to look as follows.

```
1  import feedparser
2  import requests
3
4  from flask import Flask
5  from bs4 import BeautifulSoup
6
7  app = Flask(__name__)
8
9  BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
10 LIMIT = 2
11
12 def parse_article(article_url):
13     print("Downloading {}".format(article_url))
14     r = requests.get(article_url)
15     soup = BeautifulSoup(r.text, "html.parser")
16     ps = soup.find_all('p')
17     text = "\n".join(p.get_text() for p in ps)
18     return text
19
20 @app.route("/")
21 def home():
22     feed = feedparser.parse(BBC_FEED)
23     article_texts = []
24
25     for article in feed['entries'][:LIMIT]:
26         text = parse_article(article['link'])
27         article_texts.append(text)
28     return str(article_texts)
29
30 if __name__ == '__main__':
31     app.run('0.0.0.0')
```

Let's take a closer look at what has changed.

- We import our new libraries on **lines 2 and 5**.
- We create a new global variable `LIMIT` on **line 10** to limit how many articles we want to download.
- **Lines 12-18** define a new function that takes a URL, downloads the article, and extracts the text. It does this using a crude algorithm that assumes anything inside HTML `<p>` (paragraph) tags is interesting content.
- We modify **lines 23, 25, 26, and 27** so that we use the new `parse_article` function to get the actual content of the URLs that we found in the RSS feed and return that to the user instead

of returning the URL directly. Note that we limit this to two articles by truncating our list to `LIMIT` for now as the downloads take a while and Repl's resources on free accounts are limited.

If you run the code now, you should see output similar to that shown in the image below. You can see text from the first article about Trump and the US Trade gap in the top right pane now, and the text for the second article is further down the page. You'll notice that out text extraction algorithm isn't perfect and there's still some extra text about "Share this" at the top that isn't actually part of the article, but this is good enough for us to create word clouds from later.

Returning HTML instead of plain text to the user

Although Flask allows us to return Python `str` objects directly to our visitors, the raw result is ugly compared to how people are used to seeing web pages. In order to take advantage of HTML formatting and CSS styling, it's better to define HTML *templates*, and use Flask's template engine, `jinja`, to inject dynamic content into these. Before we get to creating image files from our text content, let's set up a basic Flask template.

To use Flask's templates, we need to set up a specific file structure. Press the "new folder" button in Repl (next to the "new file" button we used earlier), and name the resulting new folder `templates`. Again, this is a special name recognised by Flask, so make sure you get the spelling exactly correct.

Select the new folder and press the "new file" button to create a new file inside our `templates` folder. Call the file `home.html`. Note below how the `home.html` file is indented one level, showing that it is inside the folder. If yours is not, drag and drop it into the `templates` folder so that Flask can find it.

In the `home.html` file, add the following code, which is a mix between standard HTML and Jinja's templating syntax to mix dynamic content into the HTML.

```
<html>
    <body>
        <h1>News Word Clouds</h1>
        <p>Too busy to click on each news article to see what it's about? Below you \
can see all the articles from the BBC front page, displayed as word clouds. If you w\
ant to read more about any particular article, just click on the wordcloud to go to \
the original article</p>
        {% for article in articles %}
            <p>{{article}}</p>
        {% endfor %}
    </body>
</html>
```

Jinja uses the specials characters `{%` and `{{` (in opening and closing pairs) to show where dynamic content (e.g. variables calculated in our Python code) should be added and to define control structures. Here we loop through a list of `articles` and display each one in a set of `<p>` tags.

We'll also need to tweak our Python code a bit to account for the template. In the `main.py` file, make the following changes.

- Add a new import near the top of the file, below the existing Flask import

```
from flask import render_template
```

- Update the last line of the `home()` function to make a call to `render_template` instead of returning a `str` directly as follows.

```
@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    article_texts = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        article_texts.append(text)
    return render_template('home.html', articles=article_texts)
```

The `render_template` call tells Flask to prepare some HTML to return to the user by combining data from our Python code and the content in our `home.html` template. Here we pass `article_texts` to the renderer as `articles`, which matches the `articles` variable we loop through in `home.html`.

If everything went well, you should see different output now, which contains our header from the HTML and static first paragraph, followed by two paragraphs showing the same article content that we pulled before.

Now it's time to move on to generating the actual wordclouds.

Generating word clouds from text in Python

Once again, there's a nifty Python library that can help us. This one will take in text and return word clouds as images. It's called `wordcloud` and can be installed in the same way as the others by adding it to our `requirements.txt` file. This is the last dependency we'll be adding, so the final `requirements.txt` file should look as follows.

Images are usually served as files living on your server or from an image host like imgur[24]. Because we'll be creating small, short-lived images dynamically from text, we'll simply keep them in memory instead of saving them anywhere permanently. In order to do this, we'll have to mess around a bit with the Python `io` and `base64` libraries, alongside our newly installed `wordcloud` library.

To import all the new libraries we'll be using to process images, modify the top of our `main.py` to look as follows. (These libraries are built into Python so there's no need to add them to requirements.txt.)

[24] imgur.com

```
import base64
import feedparser
import io
import requests

from /bs4 import BeautifulSoup
from wordcloud import WordCloud
from flask import Flask
from flask import render_template
```

We'll be converting the text from each article into a separate word cloud, so it'll be useful to have another helper function that can take text as input and produce the word cloud as output. We can use base64[25] to represent the images, which can then be displayed directly in our visitors' web browsers.

Add the following function to the `main.py` file.

```
def get_wordcloud(text):
    pil_img = WordCloud().generate(text=text).to_image()
    img = io.BytesIO()
    pil_img.save(img, "PNG")
    img.seek(0)
    img_b64 = base64.b64encode(img.getvalue()).decode()
    return img_b64
```

This is probably the hardest part of our project in terms of readibility. Normally, we'd generate the word cloud using the `wordcloud` library and then save the resulting image to a file. However, because we don't want to use our file system here, we'll create a `BytesIO` Python object in memory instead and save the image directly to that. We'll convert the resulting bytes to base64 in order to finally return them as part of our HTML response and show the image to our visitors.

In order to use this function, we'll have to make some small tweaks to the rest of our code.

For our template, in the `home.html` file, change the for loop to read as follows.

```
{% for article in articles %}
        <img src="{{article}}">
{% endfor %}
```

Now instead of displaying our article in `<p>` tags, we'll put it inside an `<img/>` tag so that it can be displayed as an image. We also specify that it is formatted as a png and encoded as base64.

The last thing we need to do is modify our `home()` function to call the new `get_wordcloud()` function and to build and render an array of images instead of an array of text. Change the `home()` function to look as follows.

[25] https://en.wikipedia.org/wiki/Base64

```
1  @app.route("/")
2  def home():
3      feed = feedparser.parse(BBC_FEED)
4      clouds = []
5
6      for article in feed['entries'][:LIMIT]:
7          text = parse_article(article['link'])
8          cloud = get_wordcloud(text)
9          clouds.append(cloud)
10     return render_template('home.html', articles=clouds)
```

We made changes on lines 4, 8, 9, and 10, to change to a `clouds` array, populate that with images from our `get_wordcloud()` function, and return that in our `render_template` call.

If you restart the Repl and refresh the page, you should see something similar to the following. We can see the same content about Trump, the US, tariffs, and trade deficits, but we can now see the important keywords without having to read the entire article.

For a larger view, you can pop out the website in a new browser tab using the button in the top right of the Repl editor (indicated in red above).

The last thing we need to do is add some styling to make the page look a bit prettier and link the images to the original articles.

Adding some finishing touches

Our text looks a bit stark, and our images touch each other which makes it hard to see that they are separate images. We'll fix that up by adding a few lines of CSS and importing the Bootstrap framework.

Adding CSS

Edit the `home.html` file to look as follows

```
<html>
  <head>
    <title>News in WordClouds | Home</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/\
css/bootstrap.min.css" integrity="sha384-HSMxcRTRxnN+Bdg0JdbxYKrThecOKuH5zCYotlSAcp1\
+c8xmyTe9GYg1l9a69psu" crossorigin="anonymous">

    <style type="text/css">
      body {padding: 20px;}
      img{padding: 5px;}
    </style>
  </head>

  <body>
    <h1>News Word Clouds</h1>
      <p>Too busy to click on each news article to see what it's about? Below you ca\
n see all the articles from the BBC front page, displayed as word clouds. If you wan\
t to read more about any particular article, just click on the wordcloud to go to th\
e original article</p>
      {% for article in articles %}
        <a href="{{article.url}}"><img src="{{article.image}}"\
></a>
      {% endfor %}
  </body>
</html>
```

On **line 3** we add a title, which is displayed in the browser tab. On **line 4**, we import Bootstrap[26], which has some nice CSS defaults right out the box (it's probably a bit heavy-weight for our project as we have so little content and won't use most of Bootstrap's features, but it's nice to have if you're planning on extending the project.)

On **lines 6-8**, we add padding to the main body to stop the text going to close to the edges of the screen, and also add padding to our images to stop them touching each other.

On **line 16**, we use an `<a>` tag to add a link to our image. We also change the Jinja templates to `{{article.url}}` and `{{article.image}}` so that we can have images that link back to the original news article.

Now we need to tweak our backend code again to pass through the URL and image for each article, as the template currently doesn't have access to the URL.

[26] https://getbootstrap.com/

Passing through the URLs

To easily keep track of pairs of URLs and images, we'll add a basic Python helper class called `Article`. In the `main.py` file, add the following code before the function definitions.

```
class Article:
    def __init__(self, url, image):
        self.url = url
        self.image = image
```

This is a simple class with two attributes: url and image. We'll store the original URL from the RSS feed in `url` and the final base64 wordcloud in `image`.

To use this class, modify the `home()` function to look as follows.

```
@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    articles = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        cloud = get_wordcloud(text)
        articles.append(Article(article['link'], cloud))
    return render_template('home.html', articles=articles)
```

We changed the name of our `clouds` list to `articles`, and populated it by initialising `Article` objects in the for loop and appending them to this list. We then pass across `articles=articles` instead of `articles=clouds` in the return statement so that the template can access our list of `Article` objects, which each contain the image and the URL of each article.

If you refresh the page again, you'll be able to click any of the images to go to the original article, allowing readers to view a brief summary of the day's news or to read more details about any stories that catch their eye.

Where next?

We've included several features in our web application, and looked at how to use RSS feeds and process and serve images directly in Python, but there are a lot more features we could add. For example:

- Our application only shows two stories at a time as the download time is slow. We could instead look at implementing a threaded solution to downloading web pages so that we could process several articles in parallel. Alternatively or in addition, we could also download the articles on a schedule and cache the resulting images so that we don't have to do the resource heavy downloading and parsing each time a visitor visits our site.
- Our web application only shows articles from a single source (BBC), and only from today. We could add some more functionality to show articles from different sources and different time frames. We could also consider allowing the viewer to choose which category of articles to view (news, sport, politics, finance, etc) by using different RSS feeds as sources.
- Our design and layout is very basic. We could make our site look better and be more responsive by adding more CSS. We could lay out the images in a grid of rows and columns to make it look better on smaller screens such as mobile phones.

If you'd like to keep working on the web application, simply head over to the repl[27] and fork it to continue your own version. Feel free to comment below or contact me on Twitter[28] if you have any questions or comments, or simply want to share what you made using this as a starting point. You can also join the Repl Discord server[29] where there is a community of developers and makers who are happy to help out and discuss ideas. Finally, you might enjoy my other tutorials[30].

[27] https://repl.it/@GarethDwyer1/news-to-wordcloud
[28] https://twitter.com/sixhobbits
[29] https://discord.gg/QWFfGhy
[30] https://dwyer.co.za/writing.html

Building a Discord Bot with Python and Repl.it

In this tutorial, we'll use repl.it[31] and Python to build a Discord Chatbot. The bot will be able to join a Discord server and respond to messages sent by people.

If you prefer JavaScript, the next chapter is the same tutorial using NodeJS instead of Python

You'll find it easier to follow along if you have some Python knowledge and you should have used Discord or a similar app such as Skype or Telegram before. We won't be covering the very basics of Python, but we will explain each line of code in detail, so if you have any experience with programming, you should be able to follow along.

Setting up

We'll be doing all of our coding through the Repl.it web IDE and hosting our bot with Repl.it as well, so you won't need to install any additional software on your machine.

For set up, we'll be walking through the following steps. Skip any that don't apply to you (e.g. if you already have a Discord account, you can skip that section).

- Creating an account on Repl.it[32].
- Creating an account on Discord[33].
- Creating an application and a bot user in your Discord account
- Creating a server on Discord
- Adding our bot to our Discord server

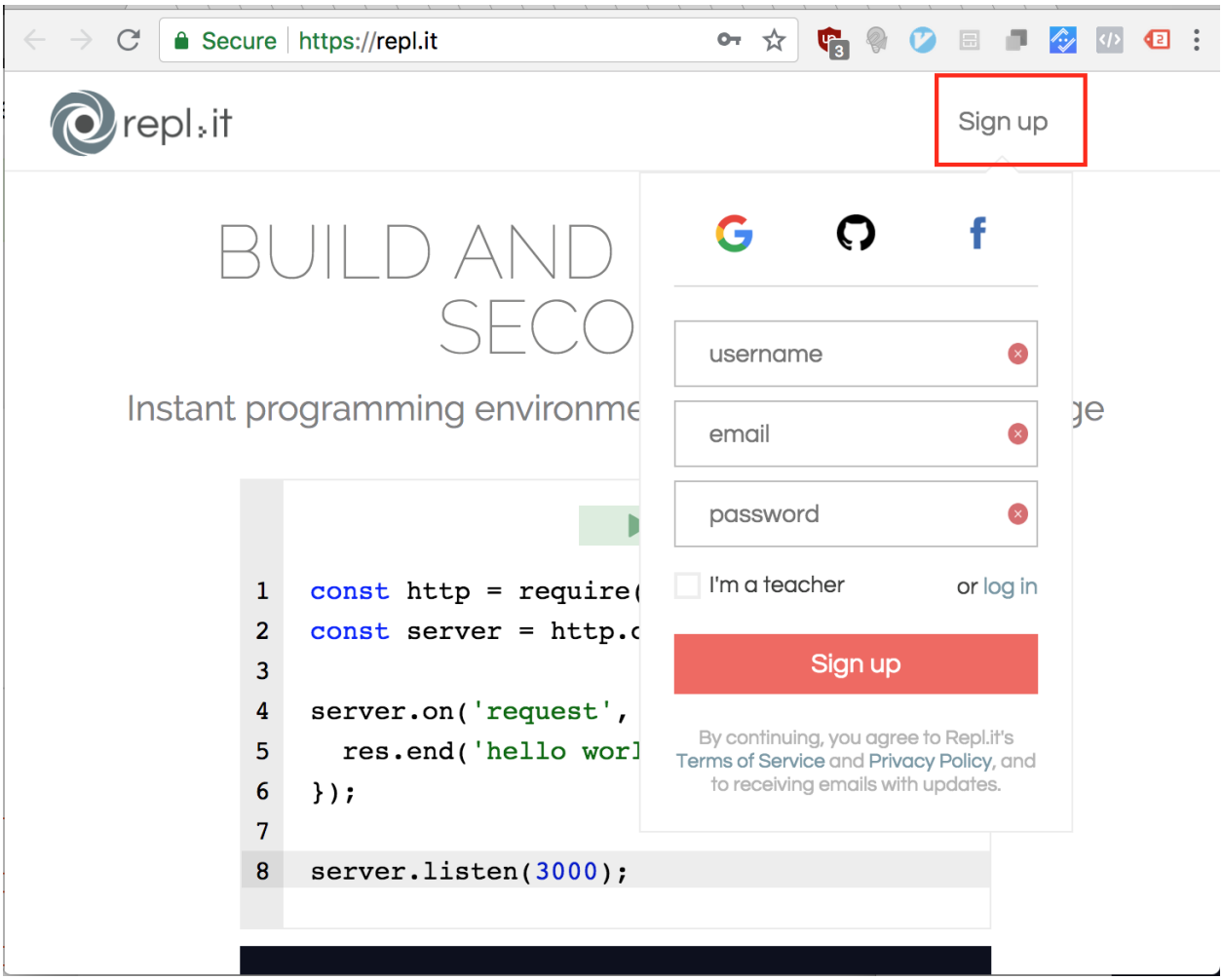
Let's get through these admin steps first and then we can get to the fun part of coding our bot.

Creating an account on Repl.it

Repl.it is an online IDE *and* compute provider. Traditionally, you would write code locally on your machine, and then have to "deploy" the code to a server, so that other people on the internet could interact with it. Repl.it removes one of these steps by combining the two – you can write your code directly through the Repl.it interface and it will automatically be deployed to the public internet.

[31] https://repl.it
[32] https://repl.it
[33] https://discordapp.com/

Visit Repl.it[34] in your web browser and hit the "Sign up" button.



Signing up for Repl

After signing up, press "Start coding now" and choose "Python" from the list of available languages.

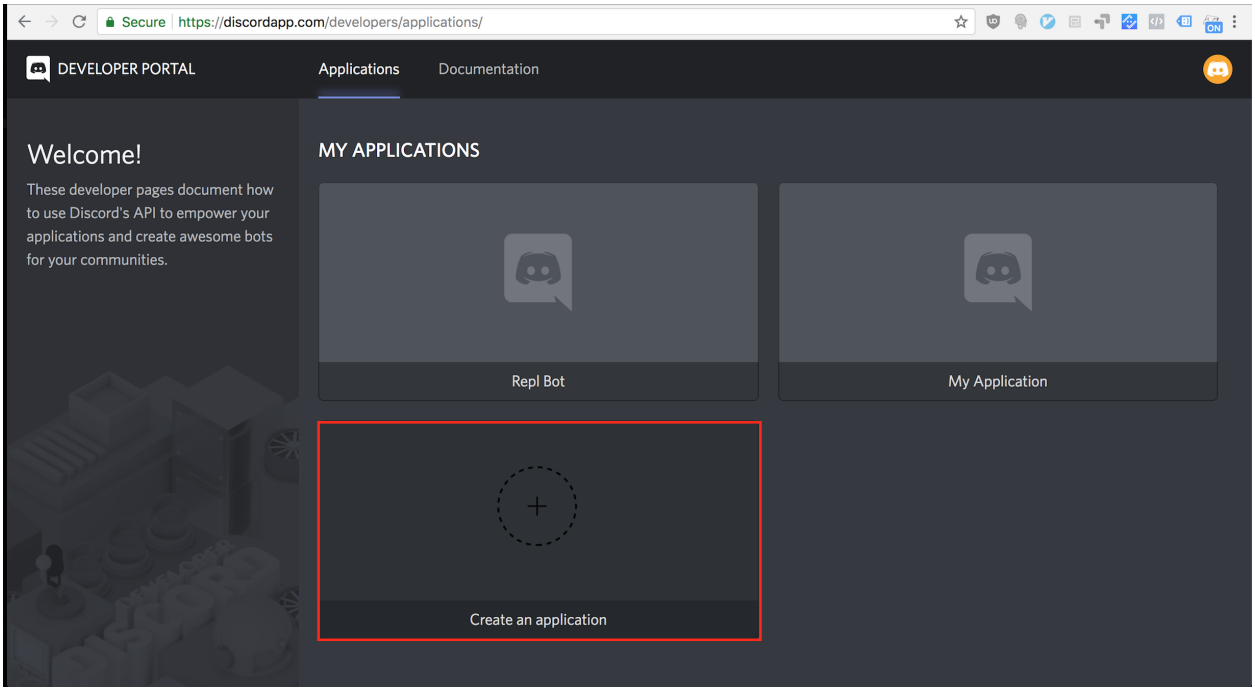
Play around with the interface a bit to see how easy it is to write and run code. We'll be coming back to this interface soon after we've done some of the Discord set up.

Creating a bot in Discord and getting a token

If you're reading this tutorial, you probably have at least heard of Discord and likely have an existing account. If not, Discord is a VoIP and Chat application that is designed to replace Skype for gamers. You can sign up for a free account over at the Discord register page[35], and download one of their desktop or mobile applications from the Discord homepage[36].

[34] https://repl.it
[35] https://discordapp.com/register
[36] https://discordapp.com/

Once you have an account, you'll want to create a Discord application. Visit the Discord developer's page[37] and press the "Create new application" button, as in the image below. (I've already created two applications. If you haven't, you'll only see the button that I've marked in red and not the two above it.)

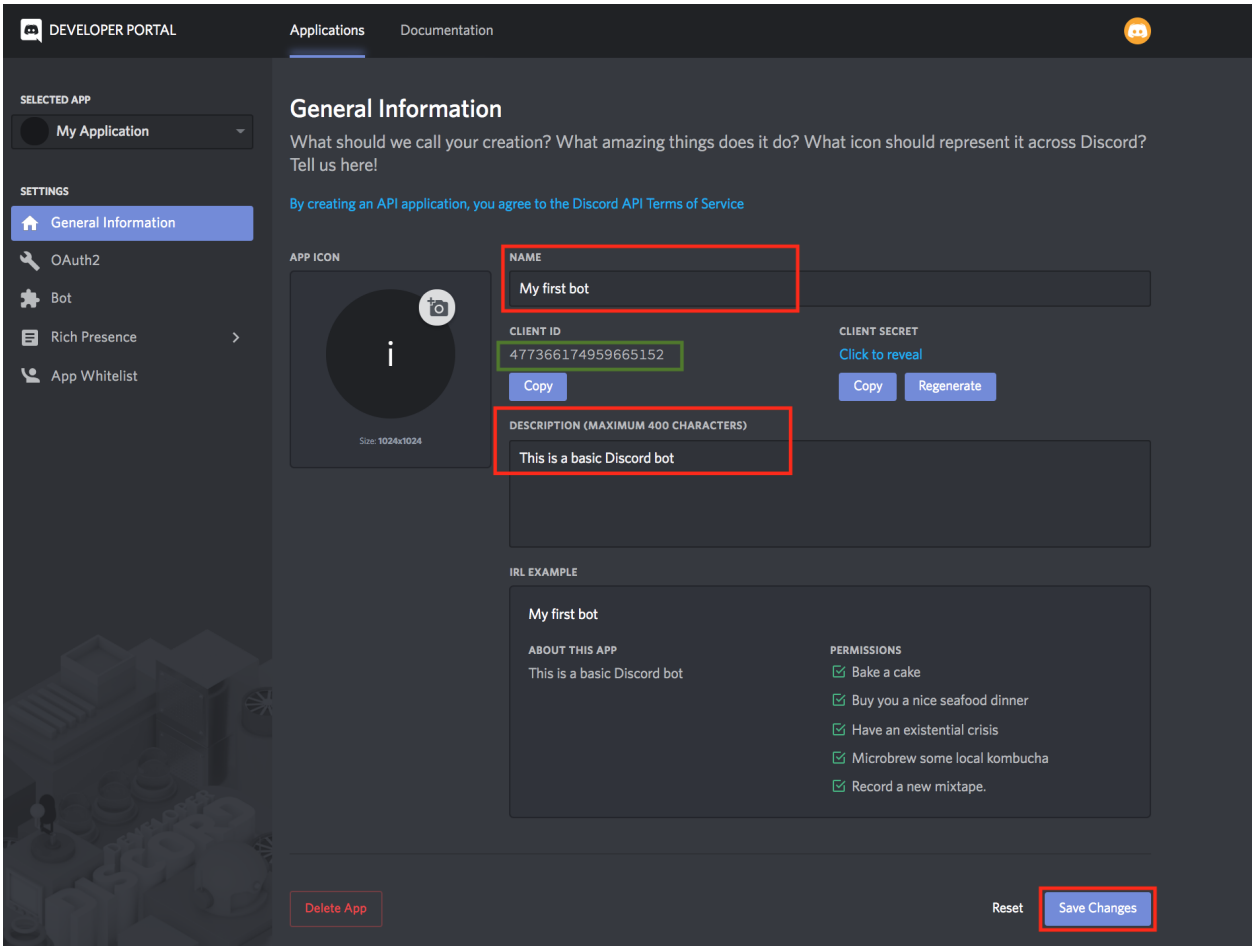


Creating a new Discord application

The first thing to do on the next page is to note your Client ID, which you'll need to add the bot to the server. You can come back later and get if from this page, or copy it somewhere where you can easily find it later.

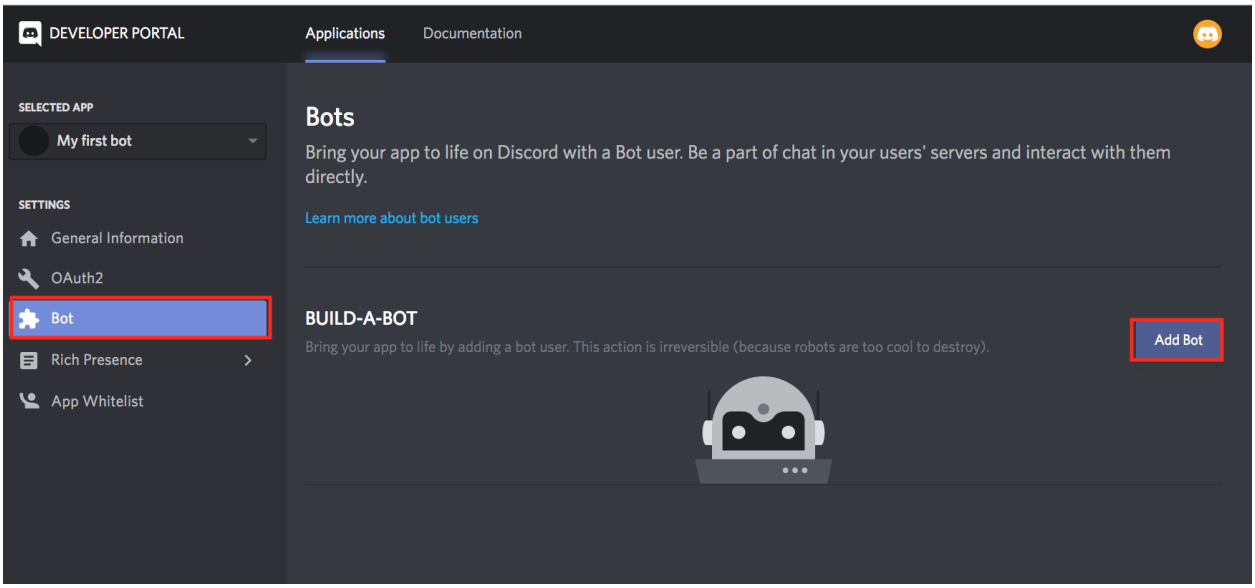
Fill out a name and description for your bot (feel free to be more creative than me) and press "save changes".

[37] https://discordapp.com/developers/applications/



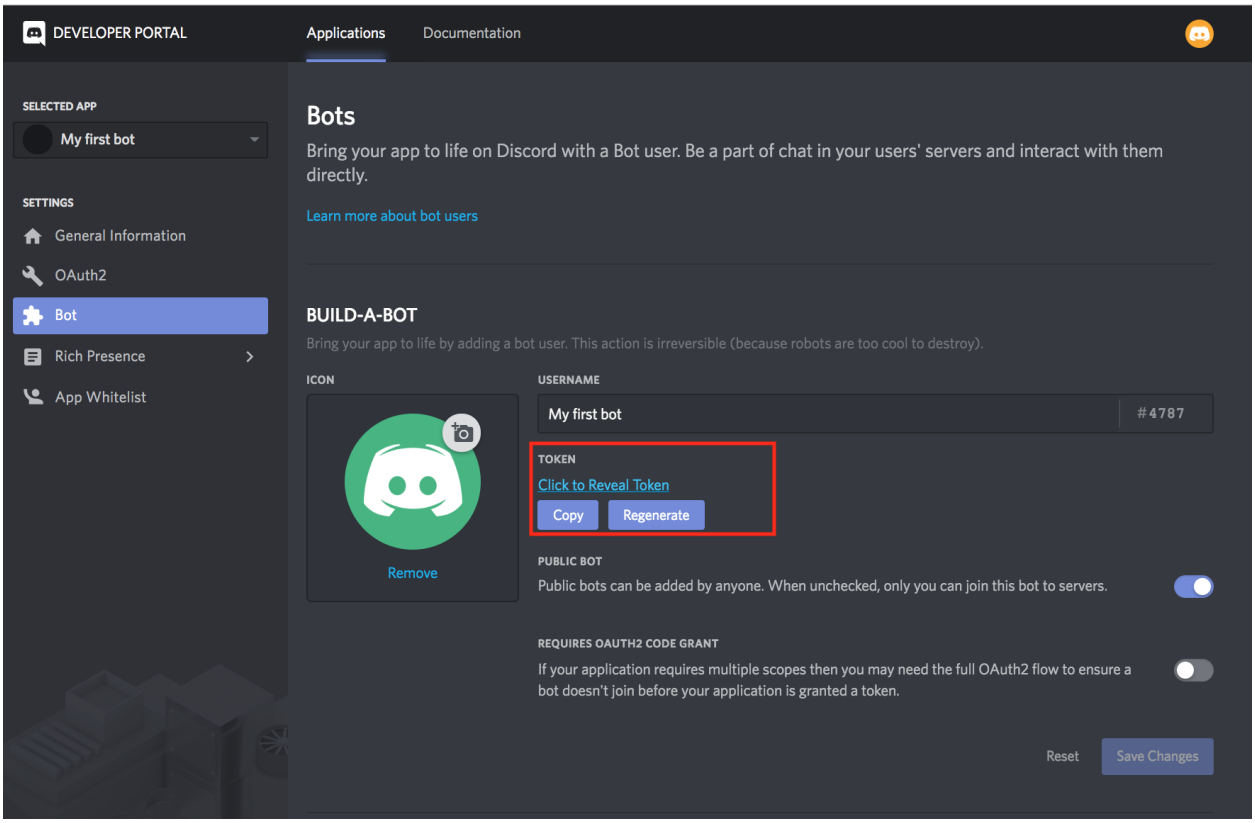
Naming our Discord Application

Now you've created a Discord application. The next step is to add a bot to this application, so head over to the "Bot" tab using the menu on the left and press the "Add Bot" button, as indicated below. Click "yes, do it" when Discord asks if you're sure about bringing a new bot to life.



Adding a bot to our Discord Application

The last thing we'll need from our bot is a Token. Anyone who has the bot's Token can prove that they own the bot, so you'll need to be careful not to share this with anyone. You can get the token by pressing "Click to reveal token", or copy it to your clipboard without seeing it by pressing "Copy".

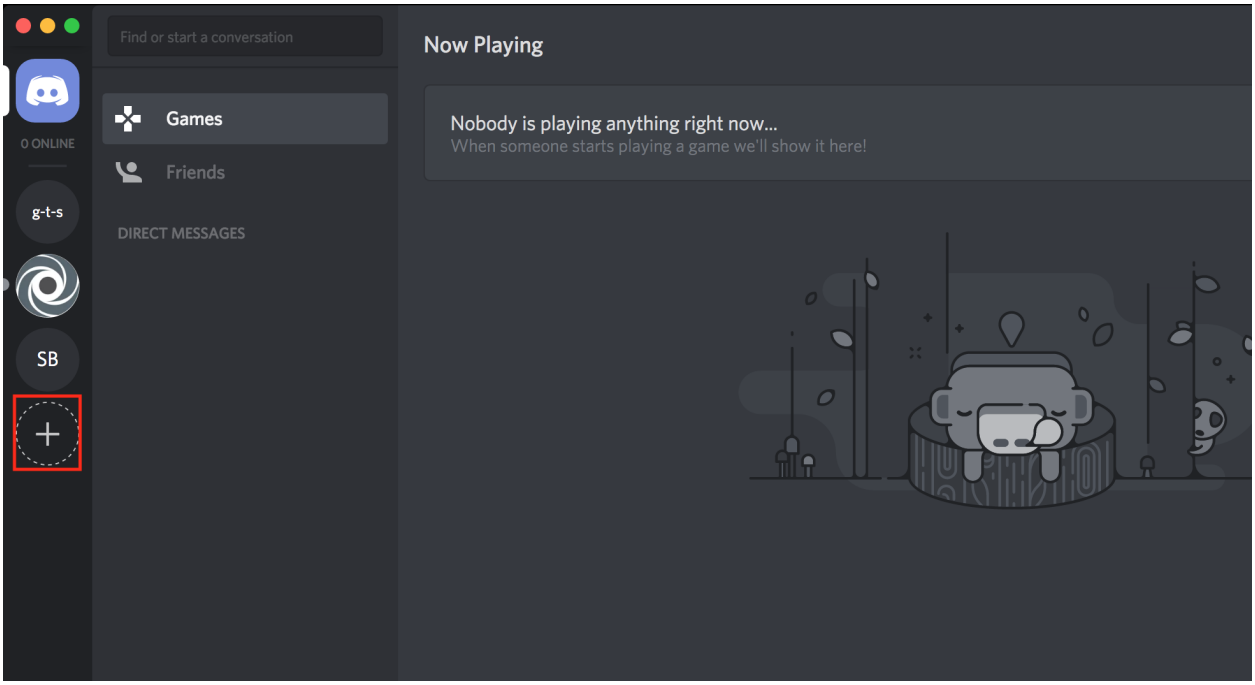


Generating a token for our Discord bot

Take note of your Token or copy it to your clipboard, as we'll need to add it to our code soon.

Creating a Discord server

If you don't have a Discord server to add your bot to, you can create one by opening the desktop Discord application that you downloaded earlier. Press the "+" icon as shown below to create a server.



Creating a Discord server

Press "Create a server" in the screen that follows, and then give your server a name. Once the server is up and running, you can chat to yourself, or invite some friends to chat with you. Soon we'll invite our bot to chat with us as well.

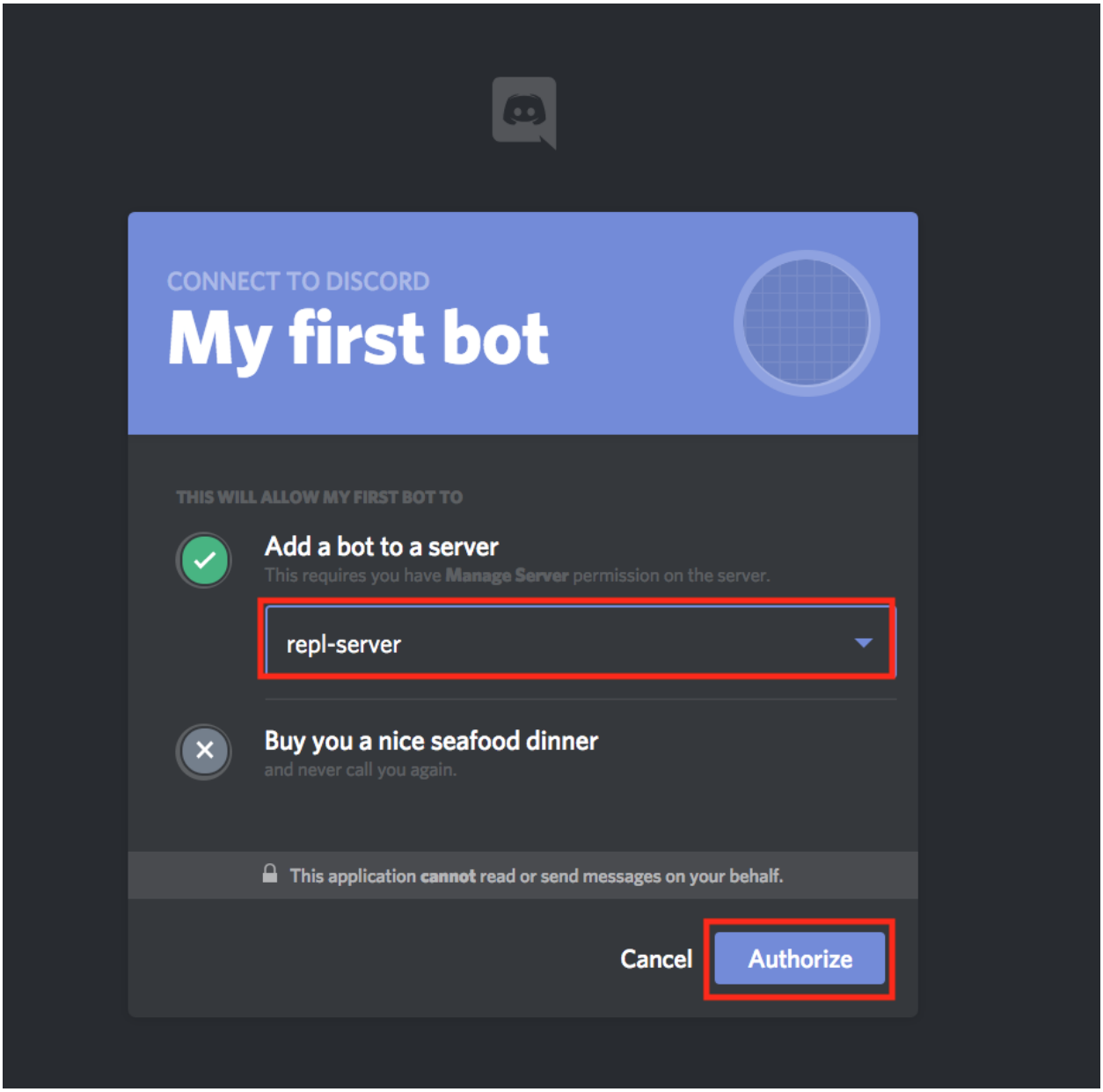
Adding your Discord bot to your Discord server

Our Discord bot is still just a shell at this stage as we haven't written any code to allow him to do anything, but let's go ahead and add him to our Discord server anyway. To add a bot to your server, you'll need the Client ID from the "General Information" page that we looked at before (this is the one outlined in green in Image 3, **not** the Bot Secret from Image 5).

Create a URL that looks as follows, but using your Client ID instead of mine at the end:

https://discordapp.com/api/oauth2/authorize?scope=bot&client_id=477366174959665152.

Visit the URL that you created in your web browser and you'll see a page similar to the following where you can choose which server to add your bot to.



Authorizing our bot to join our server

After pressing "authorize", you should get an in-app Discord notification telling you that your bot has joined your server.

Now we can get to the fun part of building a brain for our bot!

Creating a Repl and installing our Discord dependencies

The first thing we need to do is create a Python Repl to write the code for our Discord bot. Over at repl.it[38], create a new Repl, as you did right at the start of this tutorial, choosing "Python" as your language again.

We don't need to reinvent the wheel as there is already a great Python wrapper for the Discord bot API over on GitHub[39], which makes it a lot faster to get set up with a basic Python discord bot.

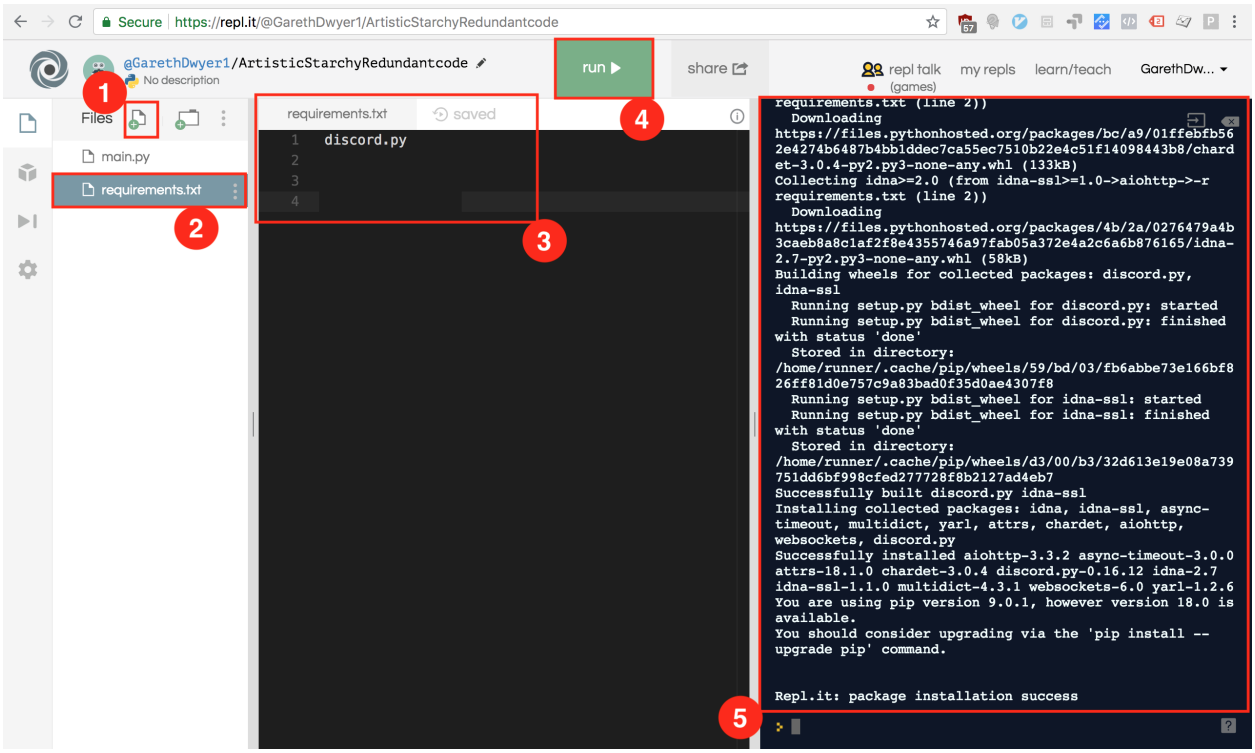
To install the libraries and its dependencies in our Repl, we'll need to create a file called `requirements.txt` at the root of our project. You can create the file by pressing the "Add file" button [Marked `(1)` in the image below] in the Repl interface. Make sure you name the file *exactly* `requirements.txt` `(2)`, which tells Repl to look in this file for any third-party libraries to install, and add the following line to this new file `(3)`:

```
1 discord.py
```

This tells Repl that we want to install the `discord.py` library, which is the same library that we linked to on GitHub above.

Press the "Run" button `(4)` and you should see Repl installing your three libraries in the output pane on the right `(5)`. Note the last line where you can see that the installation went OK.

[38] https://repl.it
[39] https://github.com/Rapptz/discord.py



Adding requirements to our Python application

Our bot is nearly ready to go – but we still need to plug in our secret token. This will authorize our code to control our bot.

Setting up authorization for our bot

By default, Repl code is public. This is great as it encourages collaboration and learning, but we need to be careful not to share our secret bot token (which gives anyone who has access to it full control of our bot).

To get around the problem of needing to give our *code* access to the token while allowing others to access our code but *not* our token, we'll be using environment variables[40]. On a normal machine, we'd set these directly on our operating system, but using Repl we don't have access to this. Repl allows us to set secrets in environment variables through a special `.env` file. Create a file called exactly `.env` in the same way that you created the `requirements.txt` file and add a variable to define your Bot's secret token (note that this is the second token that we got while setting up the Bot – different from the Client ID that we used to add our bot to our server). It should looks something like:

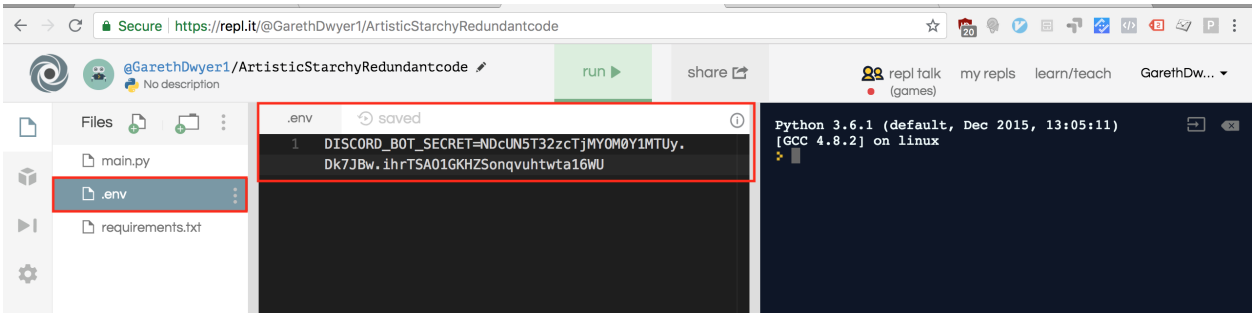
[40] https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps

```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

You'll need to:

- **Replace** the token below (after the = sign) with the token that Discord gave you when creating your own bot.
- Be careful about **spacing**. Unlike in Python, if you put a space on either side of the = in your `.env` file, these spaces will be part of the variable name or the value, so make sure you don't have any spaces around the = or at the end of the line.
- Run the code again. Sometimes you'll need to refresh the whole page to make sure that your environment variables are successfully loaded.


```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

Creating our .env file

Let's make a slightly Discord bot that repeats everything we say but in reverse. We can do this in only a few lines of code. In your `main.py` file, add the following:

```
import discord
import os

client = discord.Client()

@client.event
async def on_ready():
    print("I'm in")
    print(client.user)

@client.event
async def on_message(message):
    if message.author != client.user:
        await client.send_message(message.channel, message.content[::-1])
```

```
15
16 token = os.environ.get("DISCORD_BOT_SECRET")
17 client.run(token)
```

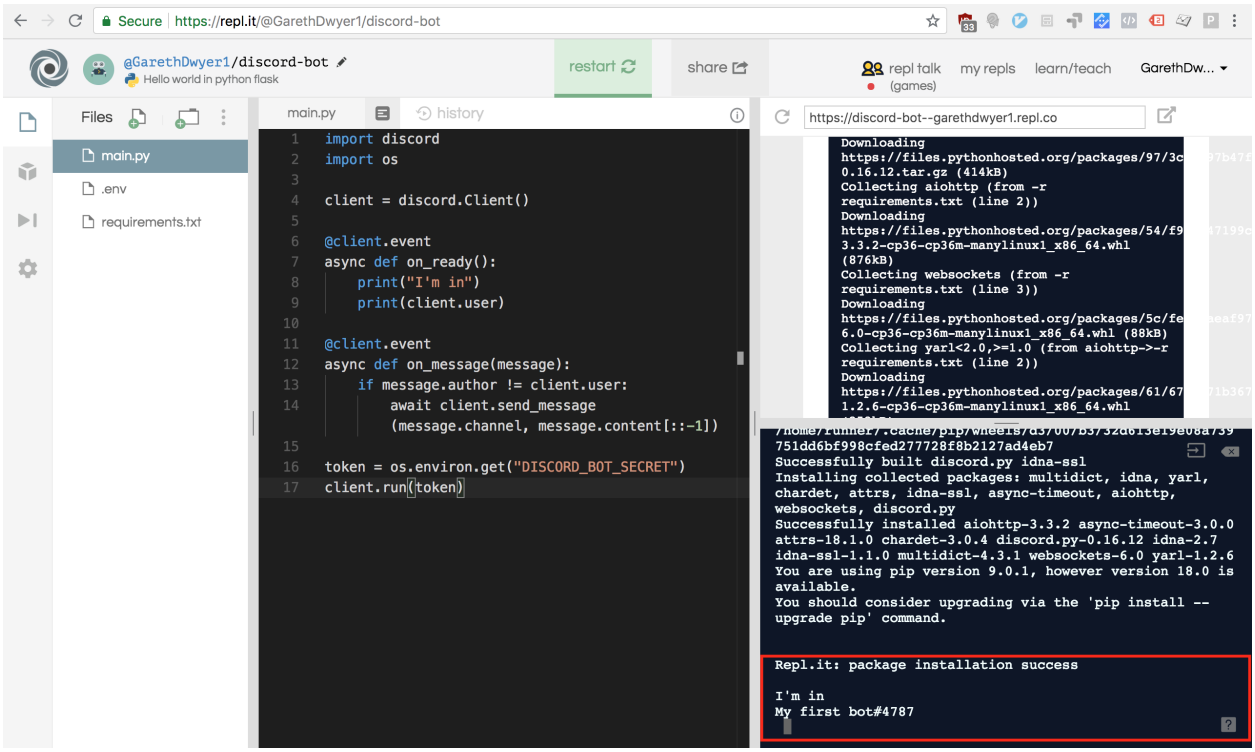
Let's tear this apart line by line to see what it does.

- **Lines 1-2** import the discord library that we installed earlier and the built-in operating system library, which we'll need to access our Bot's secret token.
- In **line 4**, we create a Discord `Client`. This is a Python object that we'll use to send various commands to Discord's servers.
- In **line 6**, we say we are defining an `event` for our client. This line is a Python decorator, which will take the function directly below it and modify it in some way. The Discord bot is going to run *asynchronously*, which might be a bit confusing if you're used to running standard Python. We won't go into asynchronous Python in depth here, but if you're interested in what this is and why it's used, there's a good guide over at FreeCodeCamp[41]. In short, instead of running the code in our file from top to bottom, we'll be running pieces of code in response to specific events.
- In **lines 7-9** we define what kind of event we want to respond to, and what the response should be. In this case, we're saying that in response to the `on_ready` event (when our bot joins a server successfully), we should output some information server side (i.e. this will be displayed in our Repl's output, but not sent as a message through to Discord). We'll print a simple `I'm in` message to see that the bot is there and print our bot's user id (if you're running multiple bots, this will make it easier to work out who's doing what).
- **Lines 11-14** are similar, but instead of responding to an `on_ready` event, we tell our bot how to handle new messages. **Line 13** says we only want to respond to messages that aren't from us (otherwise our bot will keep responding to himself – you can remove this line to see why that's a problem), and **line 15** says we'll send a new message to the same channel where we received a message (`message.channel`) and the content we'll send will be the same message that we received, but backwards (`message.content[::-1]` - `::-1` is a slightly odd but useful Python idiom to reverse a string or list).

The last two lines get our secret token from the environment variables that we set up earlier and then tell our bot to start up.

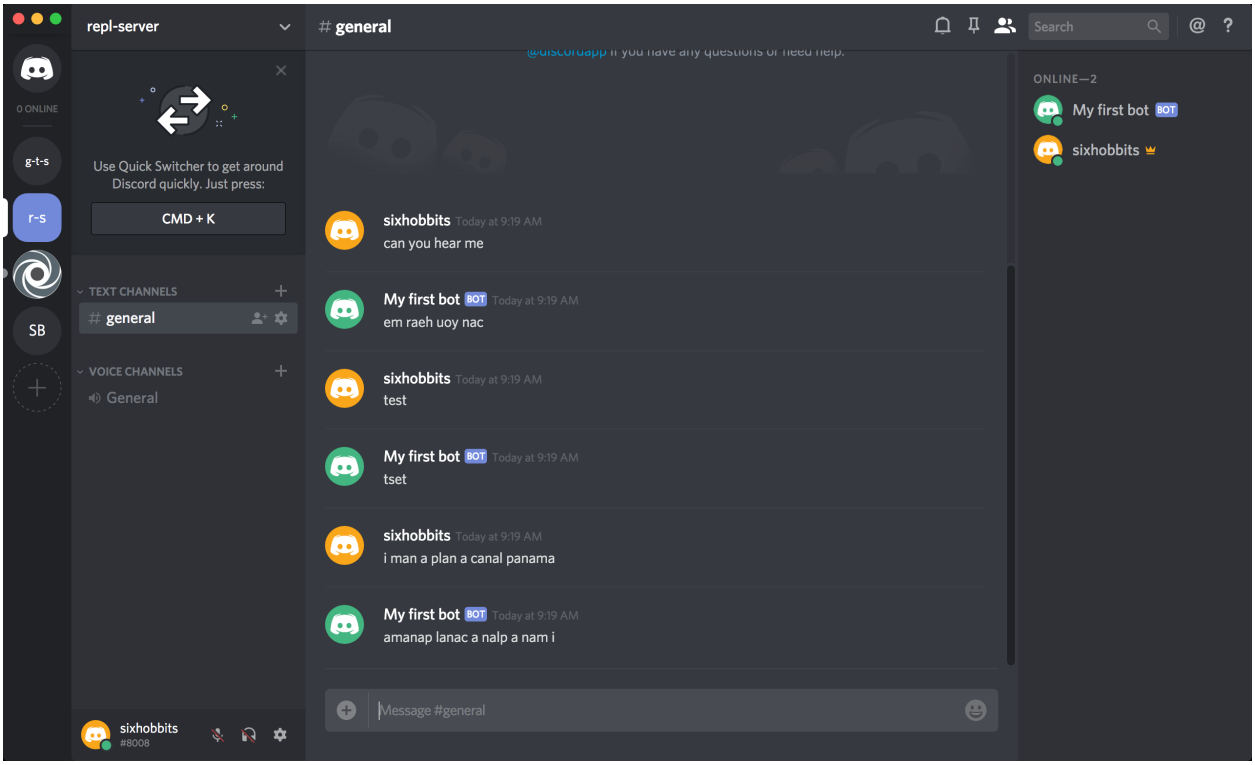
Press the big green "Run" button again and you should see your bot reporting a successful channel join in the Repl output.

[41] https://medium.freecodecamp.org/a-guide-to-asynchronous-programming-in-python-with-asyncio-232e2afa44f6



RSeeing our bot join our server

Over in your Discord app, send a message and see your Bot respond!



MOur bot can talk!

.

Keeping our bot alive

Your bot can now respond to messages, but only for as long as your Repl is running. If you close your browser tab or shut down your computer, your bot will stop and no longer respond to messages on Discord.

Repl will keep your code running after you close the browser tab only if you are running a web server. Because we are using the Python discord.py library, our bot doesn't require an explicit web server, but we can create a server and run it in a separate thread just to keep our Repl alive. We'll do this using the Flask[42] framework, so the first thing we'll need to do is add that in our `requirements.txt` file. Edit `requirements.txt` to look as follows:

```
discord.py
flask
```

Now create a new file in your project called `keep_alive.py` and add the following code:

[42] http://flask.pocoo.org/

```
from flask import Flask
from threading import Thread

app = Flask('')

@app.route('/')
def home():
    return "I'm alive"

def run():
  app.run(host='0.0.0.0',port=8080)

def keep_alive():
    t = Thread(target=run)
    t.start()
```

We won't go over this in detail as it's not central to our bot, but here we start a web server that will return "I'm alive" if anyone visits it, and we'll provide a method to start this in a new thread (leaving the main thread for our Repl bot).

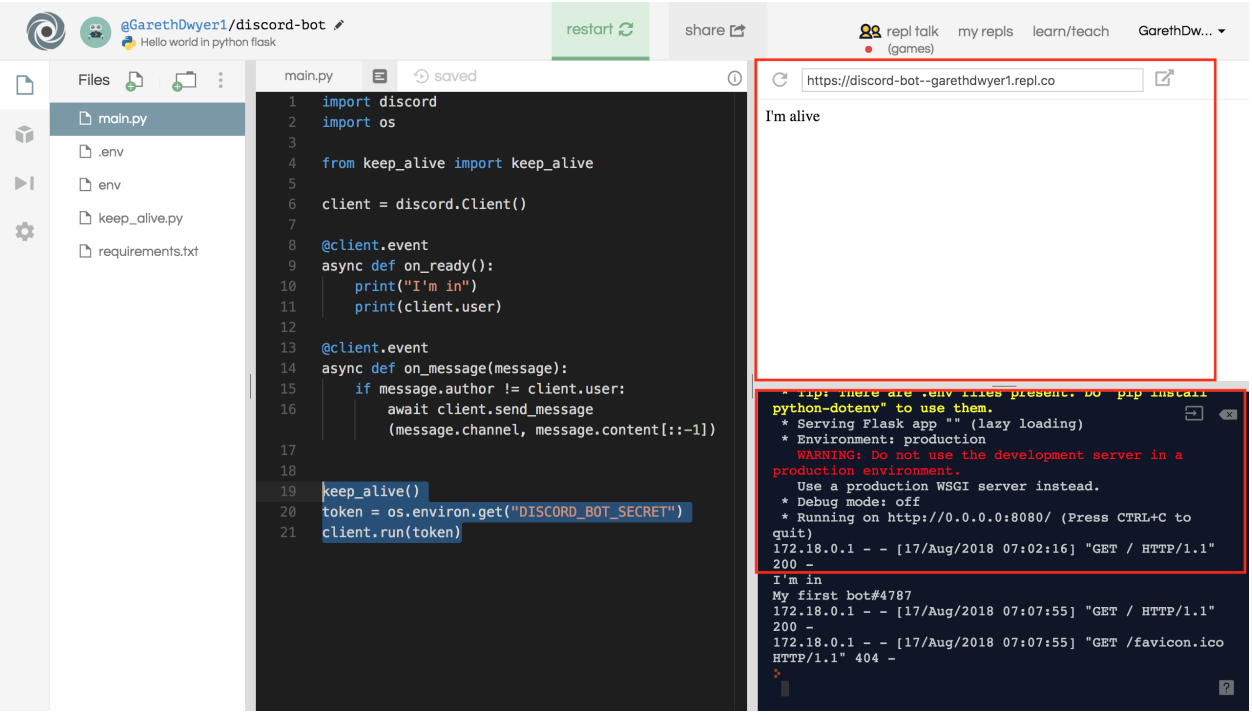
In our `main.py` file, we need to add an import for this server at the top. Add the following line near the top of `main.py`.

```
from keep_alive import keep_alive
```

And at the bottom of `main.py` start up the web server just before you start up the bot. The last three lines of `main.py` should be:

```
keep_alive()
token = os.environ.get("DISCORD_BOT_SECRET")
client.run(token)
```

After doing this and hitting the green "Run" button again, you should see some changes to your Repl. For one, you'll see a new pane in the top right which shows the web output from your server. We can see that visiting our Repl now returns a basic web page showing the "I'm alive" string that we told our web server to return by default. In the bottom-right pane, you can also see some additional output from Flask starting up and running continuously, listening for requests.



Output from our Flask server

Now your bot will stay alive even after closing your browser or shutting down your development machine. Repl will still clean up your server and kill your bot after about one hour of inactivity, so if you don't use your bot for a while, you'll have to log into Repl and start the bot up again. Alternatively, you can set up a third-party (free!) service like Uptime Robot[43]. Uptime Robot pings your site every 5 minutes to make sure it's still working – usually to notify you of unexpected downtime, but in this case the constant pings have the side effect of keeping our Repl alive as it will never go more than an hour without receiving any activity.

Forking and extending our basic bot

This is not a very useful bot as is, but the possibilities are only limited by your creativity now! You can have your bot receive input from a user, process the input, and respond in any way you choose. In fact, with the basic input and output that we've demonstrated, we have most of the components of any modern computer, all of which are based on the Von Neumann architecture[44] (we could easily add the missing memory by having our bot write to a file, or with a bit more effort link in a SQLite database[45] for persistent storage).

If you followed along this tutorial, you'll have your own basic Repl bot to play around with and extend. If you were simply reading, you can fork my bot at https://repl.it/@GarethDwyer1/discord-

[43] https://uptimerobot.com/
[44] https://en.wikipedia.org/wiki/Von_Neumann_architecture
[45] https://www.sqlite.org/index.html

bot[46] and extend it how you want (you'll need to add your own token and recreate the `.env` file still). Happy hacking!

If you're stuck for ideas, why not link up your Discord bot to the Twitch API[47] to get notified when your favourite streamers are online, or build a text adventure[48]. Also join Repl's Discord server by using this invite link https://discord.gg/QWFfGhy[49] - you can test your bot, share it with other bot builders to get feedback, and see what Discord bots people are building on Repl.

In the next chapter, we'll build exactly the same bot again but using NodeJS instead of Python. Even if you prefer Python, it's often a good idea to build the same project in two languages so that you can better appreciate the differences and similarities.

[46] https://repl.it/@GarethDwyer1/discord-bot
[47] https://dev.twitch.tv/
[48] https://en.wikipedia.org/wiki/Interactive_fiction
[49] https://discord.gg/QWFfGhy

Building a Discord bot with Node.js and Repl.it

In this tutorial, we'll use repl.it[50] and Node.js to build a Discord Chatbot. The bot will be able to join a Discord server and respond to messages sent by people.

If you don't like JavaScript, there's also a Python version of this tutorial[51].

You'll find it easier to follow along if you have some JavaScript knowledge and you should have used Discord or a similar app such as Skype or Telegram before. We won't be covering the very basics of Node.js, but we will explain each line of code in detail, so if you have any experience with programming, you should be able to follow along.

Setting up

We'll be doing all of our coding through the Repl.it web IDE and hosting our bot with Repl.it as well, so you won't need to install any additional software on your machine.

For set up, we'll be walking through the following steps. Skip any that don't apply to you (e.g. if you already have a Discord account, you can skip that section).

- Creating an account on Repl.it[52].
- Creating an account on Discord[53].
- Creating an application and a bot user in your Discord account
- Creating a server on Discord
- Adding our bot to our Discord server

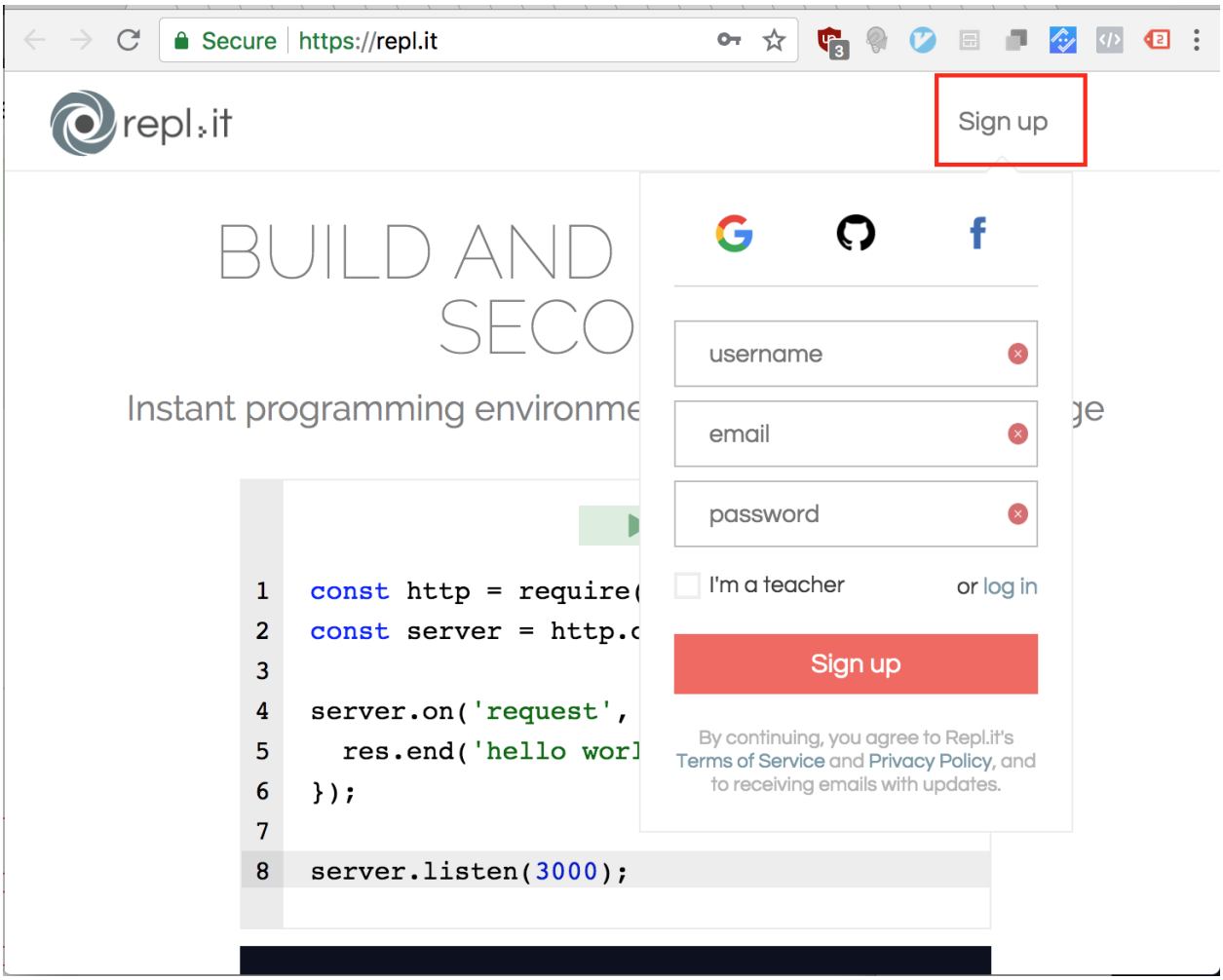
Let's get through these admin steps first and then we can get to the fun part of coding our bot.

Creating an account on Repl.it

Repl.it is an online IDE *and* compute provider. Traditionally, you would write code locally on your machine, and then have to "deploy" the code to a server, so that other people on the internet could interact with it. Repl.it removes one of these steps by combining the two – you can write your code directly through the Repl.it interface and it will automatically be deployed to the public internet.

[50] https://repl.it
[51] https://www.codementor.io/garethdwyer/building-a-discord-bot-with-python-and-repl-it-miblcwejz
[52] https://repl.it
[53] https://discordapp.com/

Visit Repl.it[54] in your web browser and hit the "Sign up" button.



Repl.it sign up button

Image 1: Signing up for Repl

After signing up, press "Start coding now" and choose "Node.js" from the list of available languages.

Play around with the interface a bit to see how easy it is to write and run code. We'll be coming back to this interface soon after we've done some of the Discord set up.

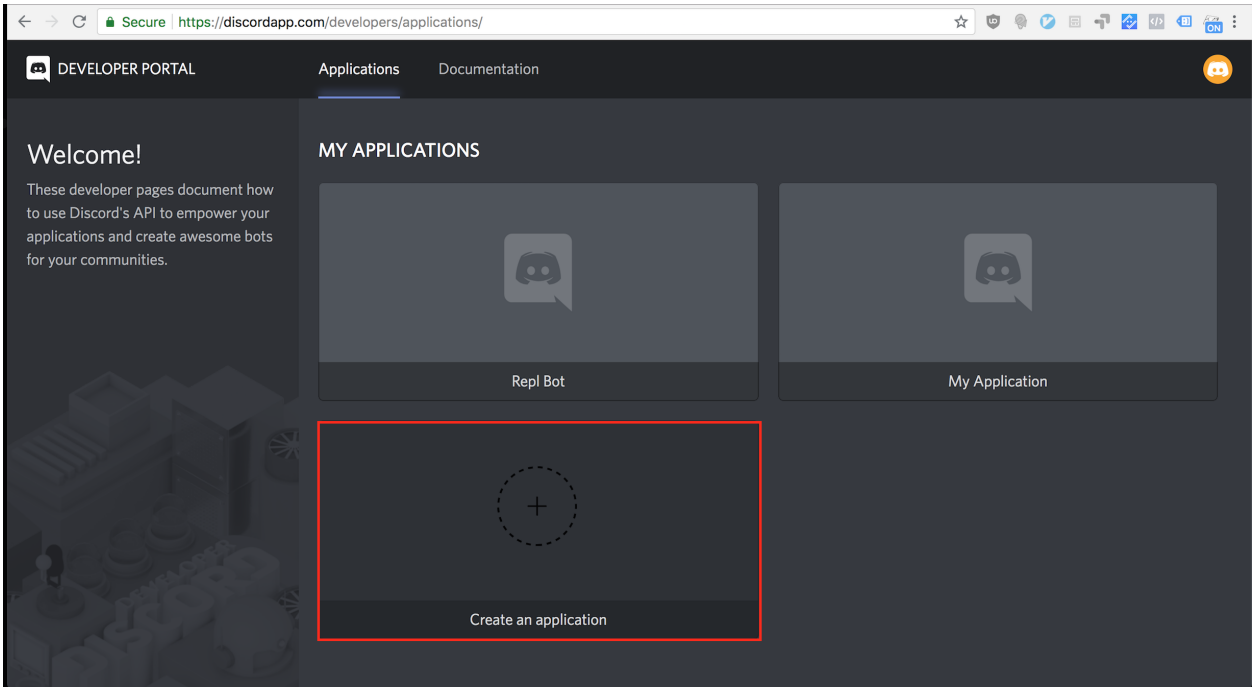
Creating a bot in Discord and getting a token

If you're reading this tutorial, you probably have at least heard of Discord and likely have an existing account. If not, Discord is a VoIP and Chat application that is designed to replace Skype for gamers. You can sign up for a free account over at the Discord register page[55], and download one of their

[54] https://repl.it
[55] https://discordapp.com/register

desktop or mobile applications from the Discord homepage[56].

Once you have an account, you'll want to create a Discord application. Visit the Discord developer's page[57] and press the "Create new application" button, as in the image below. (I've already created two applications. If you haven't, you'll only see the button that I've marked in red and not the two above it.)



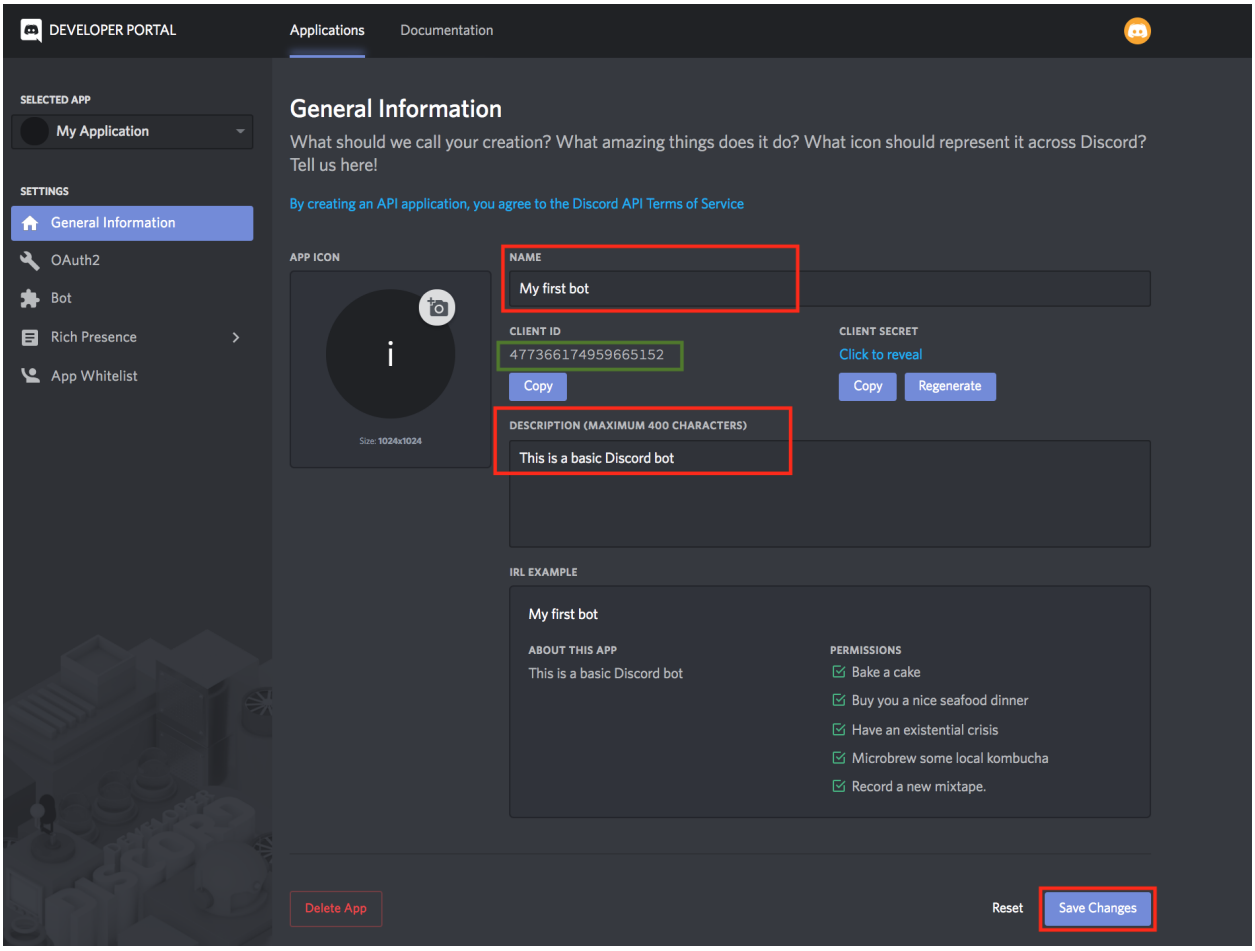
Discord create application page

Image 2: Creating a new Discord application

The first thing to do on the next page is to note your Client ID, which you'll need to add the bot to the server. You can come back later and get if from this page, or copy it somewhere where you can easily find it later.

Fill out a name and description for your bot (feel free to be more creative than me) and press "save changes".

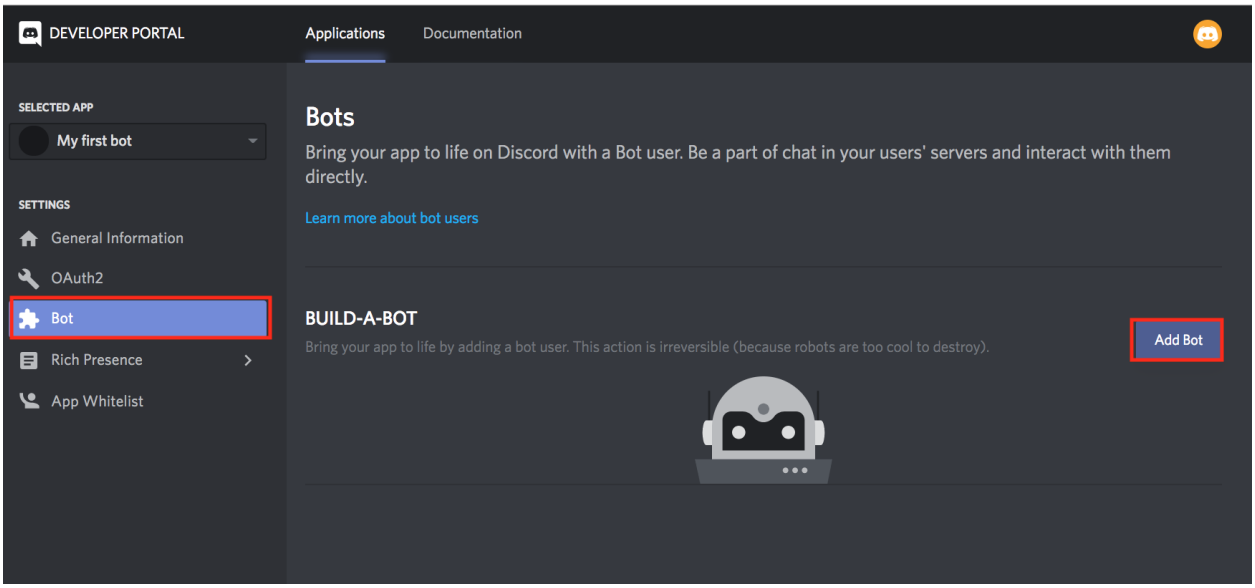
[56] https://discordapp.com/
[57] https://discordapp.com/developers/applications/



Naming our Discord bot

Image 3: Naming our Discord Application

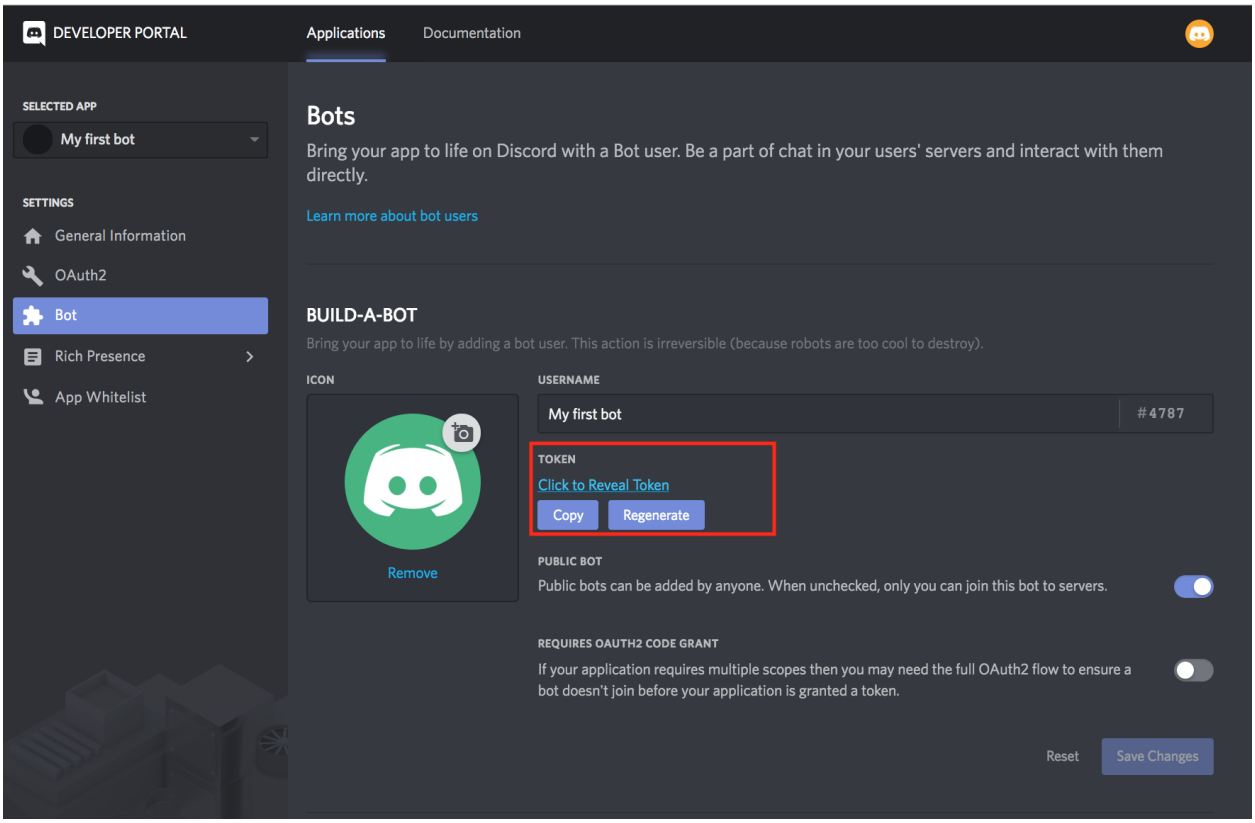
Now you've created a Discord application. The next step is to add a bot to this application, so head over to the "Bot" tab using the menu on the left and press the "Add Bot" button, as indicated below. Click "yes, do it" when Discord asks if you're sure about bringing a new bot to life.



Adding a bot to our Discord application

Image 4: Adding a bot to our Discord Application

The last thing we'll need from our bot is a Token. Anyone who has the bot's Token can prove that they own the bot, so you'll need to be careful not to share this with anyone. You can get the token by pressing "Click to reveal token", or copy it to your clipboard without seeing it by pressing "Copy".



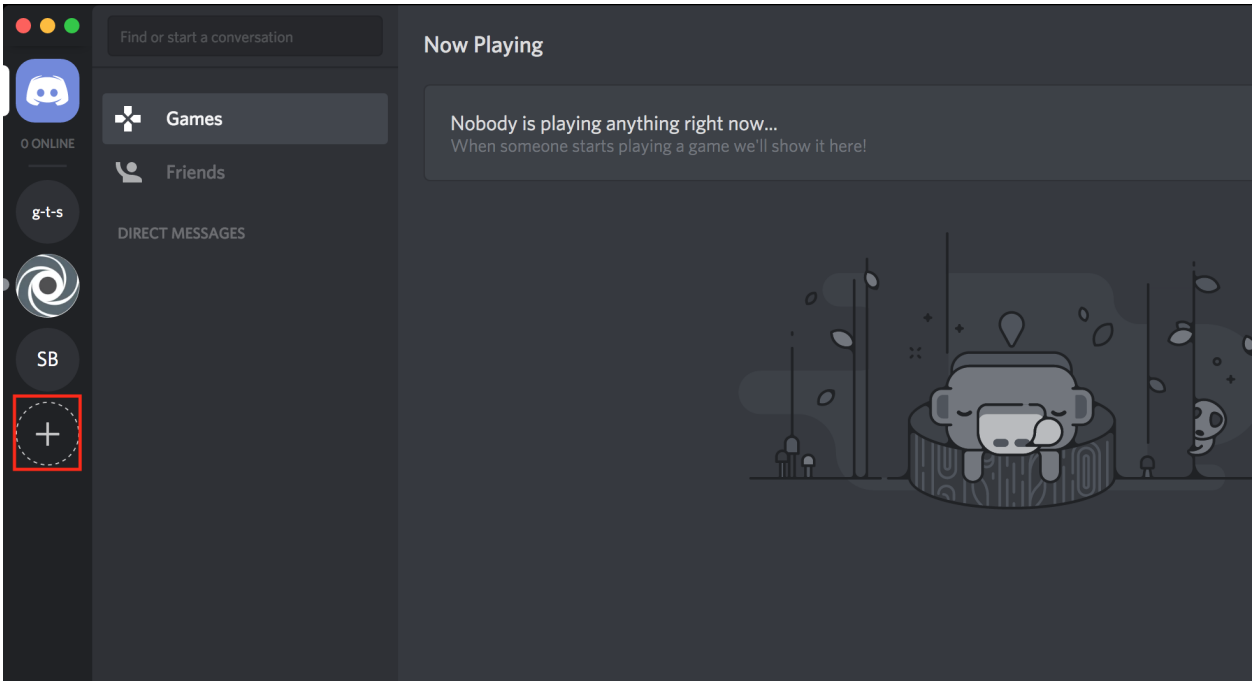
Getting a Token for your bot

Image 5: Generating a token for our Discord bot

Take note of your Token or copy it to your clipboard, as we'll need to add it to our code soon.

Creating a Discord server

If you don't have a Discord server to add your bot to, you can create one by opening the desktop Discord application that you downloaded earlier. Press the "+" icon as shown below to create a server.



Creating a discord server

Image 6: Creating a Discord server

Press "Create a server" in the screen that follows, and then give your server a name. Once the server is up and running, you can chat to yourself, or invite some friends to chat with you. Soon we'll invite our bot to chat with us as well.

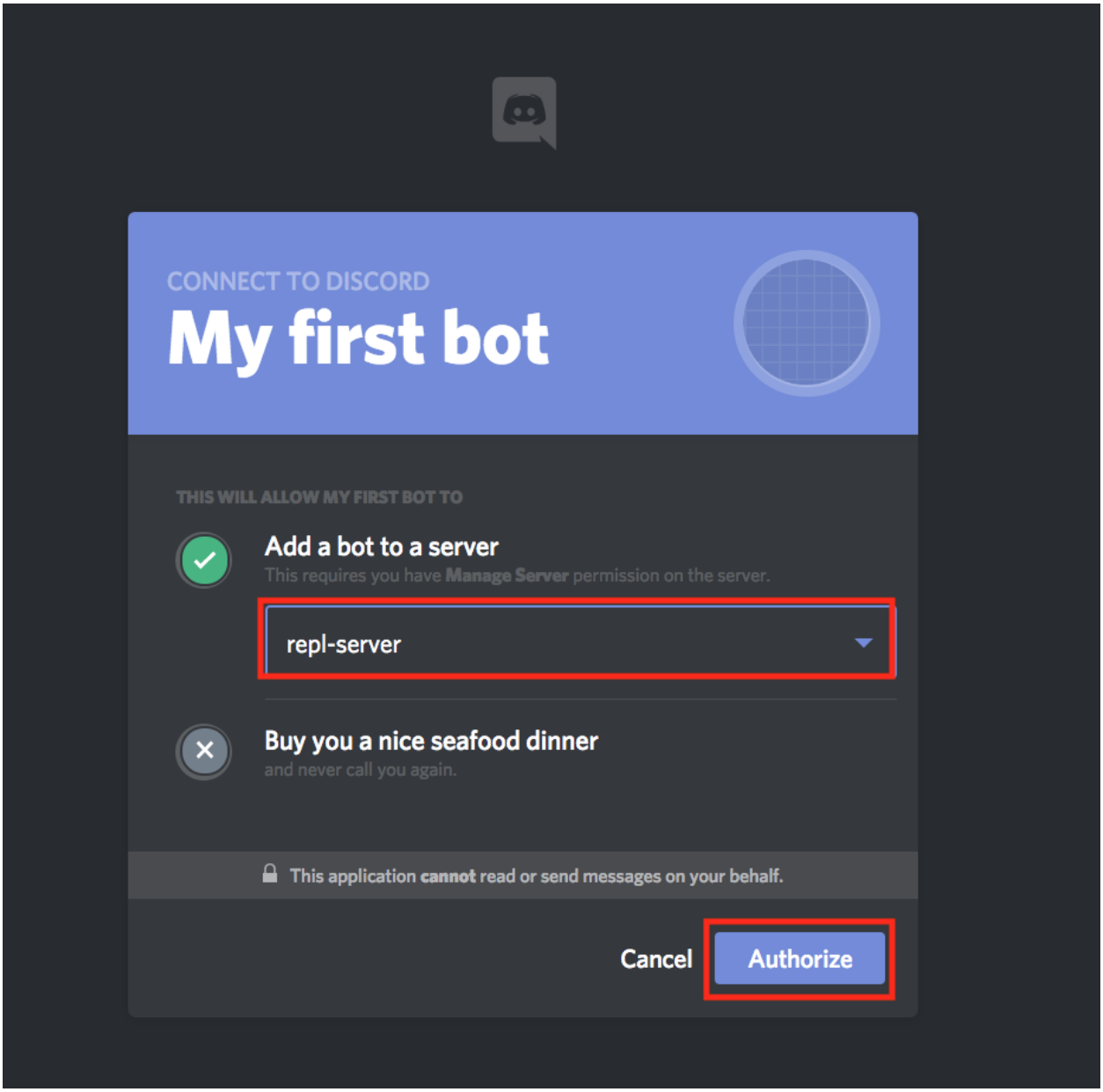
Adding your Discord bot to your Discord server

Our Discord bot is still just a shell at this stage as we haven't written any code to allow him to do anything, but let's go ahead and add him to our Discord server anyway. To add a bot to your server, you'll need the Client ID from the "General Information" page that we looked at before (this is the one outlined in green in Image 3, **not** the Bot Secret from Image 5).

Create a URL that looks as follows, but using your Client ID instead of mine at the end:

https://discordapp.com/api/oauth2/authorize?scope=bot&client_id=477366174959665152.

Visit the URL that you created in your web browser and you'll see a page similar to the following where you can choose which server to add your bot to.



Authorizing your bot to join your server

Image 7: Authorizing our bot to join our server

After pressing "authorize", you should get an in-app Discord notification telling you that your bot has joined your server.

Now we can get to the fun part of building a brain for our bot!

Creating a Repl and installing our Discord dependencies

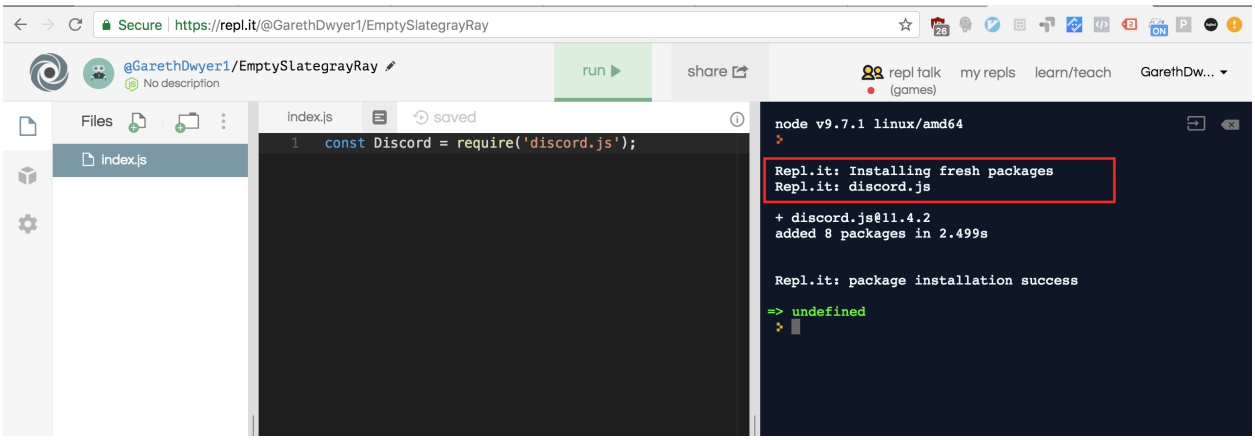
The first thing we need to do is create a Node.js Repl to write the code for our Discord bot. Over at repl.it[58], create a new Repl, as you did right at the start of this tutorial, choosing "Node.js" as your language again.

We don't need to reinvent the wheel as there is already a great Node wrapper for the Discord bot API called discord.js[59]. Normally we would install this third-party library through npm[60], but because we're using Repl.it, we can skip the installation. Our Repl will automatically pull in all dependencies.

In the default `index.js` file that is included with your new Repl, add the following line of code.

```
1  const Discord = require('discord.js');
```

Press the "Run" button and you should see Repl installing the Discord library in the output pane on the right, as in the image below.



installing Discord.js

Image 8: Installing Discord.js in our Repl.

Our bot is nearly ready to go – but we still need to plug in our secret token. This will authorize our code to control our bot.

Setting up authorization for our bot

By default, Repl code is public. This is great as it encourages collaboration and learning, but we need to be careful not to share our secret bot token (which gives anyone who has access to it full control of our bot).

[58] https://repl.it
[59] https://discord.js.org/
[60] https://www.npmjs.com/

To get around the problem of needing to give our *code* access to the token while allowing others to access our code but *not* our token, we'll be using environment variables[61]. On a normal machine, we'd set these directly on our operating system, but using Repl we don't have access to this. Repl allows us to set secrets in environment variables through a special `.env` file. Create a new file called exactly `.env` by using the new file button in the left pane and add a variable to define your Bot's secret token (note that this is the second token that we got while setting up the Bot – different from the Client ID that we used to add our bot to our server). It should looks something like:

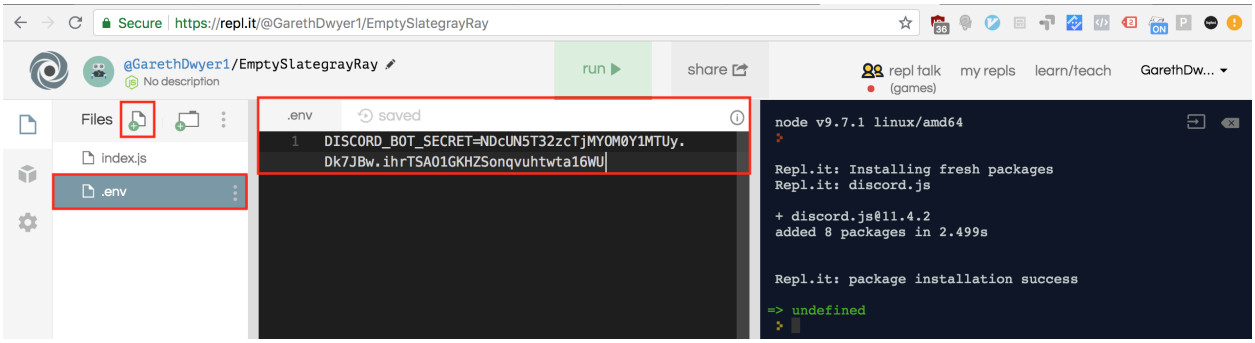
```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

You'll need to:

- **Replace** the token below (after the = sign) with the token that Discord gave you when creating your own bot.
- Be careful about **spacing**. Unlike in Python, if you put a space on either side of the = in your `.env` file, these spaces will be part of the variable name or the value, so make sure you don't have any spaces around the = or at the end of the line.
- Run the code again. Sometimes you'll need to refresh the whole page to make sure that your environment variables are successfully loaded.

```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

In the image below you we've highlighted the "Add file" button, the new file (`.env`) and how to deine the secret token for our bot's use.



Creating our .env file

Image 9: Creating our .env file

Let's make a slightly Discord bot that repeats everything we say but in reverse. We can do this in only a few lines of code. In your `index.js` file, add the following:

[61] https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps

```
1  const Discord = require('discord.js');
2  const client = new Discord.Client();
3  const token = process.env.DISCORD_BOT_SECRET;
4
5  client.on('ready', () => {
6    console.log("I'm in");
7    console.log(client.user.username);
8  });
9
10 client.on('message', msg => {
11     if (msg.author.id != client.user.id) {
12         msg.channel.send(msg.content.split('').reverse().join(''));
13     }
14 });
15
16 client.login(token);
```

Let's tear this apart line by line to see what it does.

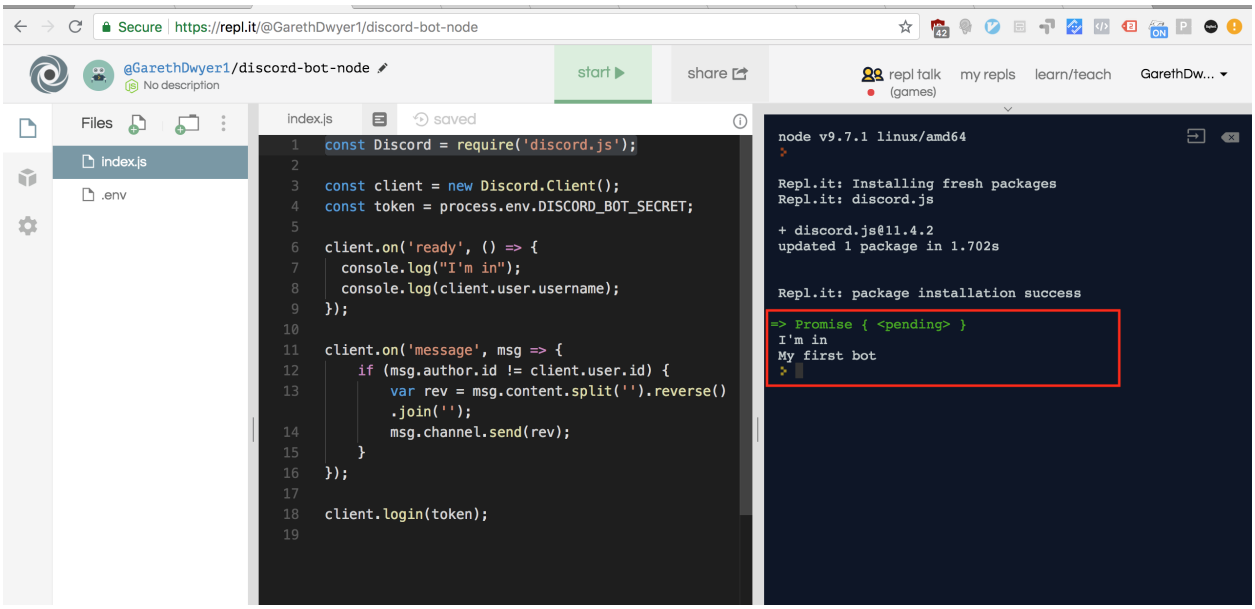
- **Line 1** is what we had earlier. This line both tells Repl to install the third party library and brings it into this file so that we can use it.
- In **line 2**, we create a Discord `Client`. We'll use this client to send commands to the Discord *server* to control our bot and send it commands.
- In **line 3** we retrieve our secret token from the environment variables (which Repl set from our `.env` file).
- In **line 5**, we define an `event` for our client, which defines how our bot should react to the "ready" event. The Discord bot is going to run *asynchronously*, which might be a bit confusing if you're used to running standard synchronous code. We won't go into asynchronous coding in depth here, but if you're interested in what this is and why it's used, there's a good guide over at RisingStack[62]. In short, instead of running the code in our file from top to bottom, we'll be running pieces of code in response to specific events.
- In **lines 6-8** we define how our bot should respond to the "ready" event, which is fired when our bot successfully joins a server. We instruct our bot to output some information server side (i.e. this will be displayed in our Repl's output, but not sent as a message through to Discord). We'll print a simple `I'm in` message to see that the bot is there and print our bot's username (if you're running multiple bots, this will make it easier to work out who's doing what).
- **Lines 10-14** are similar, but instead of responding to an "ready" event, we tell our bot how to handle new messages. **Line 11** says we only want to respond to messages that aren't from us (otherwise our bot will keep responding to himself – you can remove this line to see why that's a problem), and **line 12** says we'll send a new message to the same channel where we received a message (`msg.channel`) and the content we'll send will be the same message that we

[62] https://blog.risingstack.com/node-hero-async-programming-in-node-js/

received, but backwards. To reverse a string, we split it into its individual characters, reverse the resulting array, and then join it all back into a string again.

The last line fires up our bot and uses the token we loaded earlier to log into Discord.

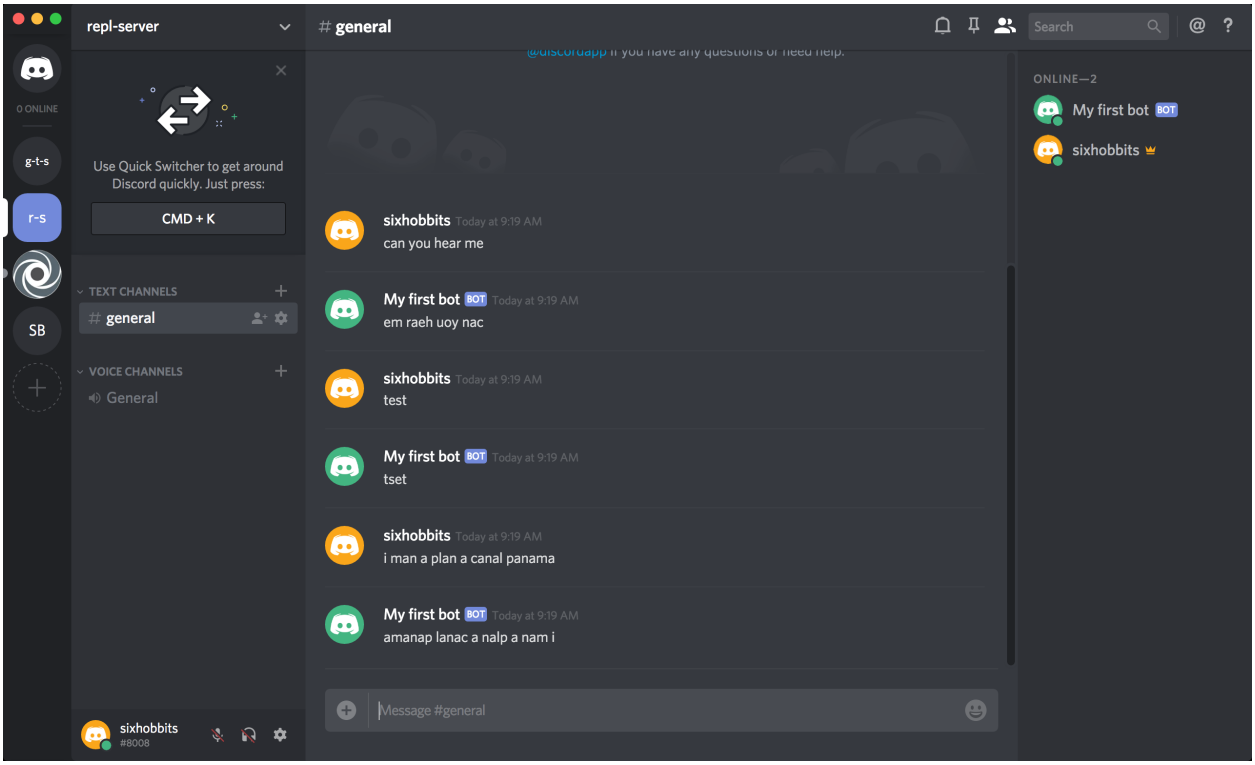
Press the big green “Run” button again and you should see your bot reporting a successful channel join in the Repl output.



Repl output showing channel join

Image 10: Seeing our bot join our server

Over in your Discord app, send a message and see your Bot respond!



Messages from our bot

Image 11: Our bot can talk!

Keeping our bot alive

Your bot can now respond to messages, but only for as long as your Repl is running. If you close your browser tab or shut down your computer, your bot will stop and no longer respond to messages on Discord.

Repl will keep your code running after you close the browser tab only if you are running a web server. Our bot doesn't require an explicit web server to run, but we can create a server and run it in the background just to keep our Repl alive.

Create a new file in your project called `keep_alive.js` and add the following code:

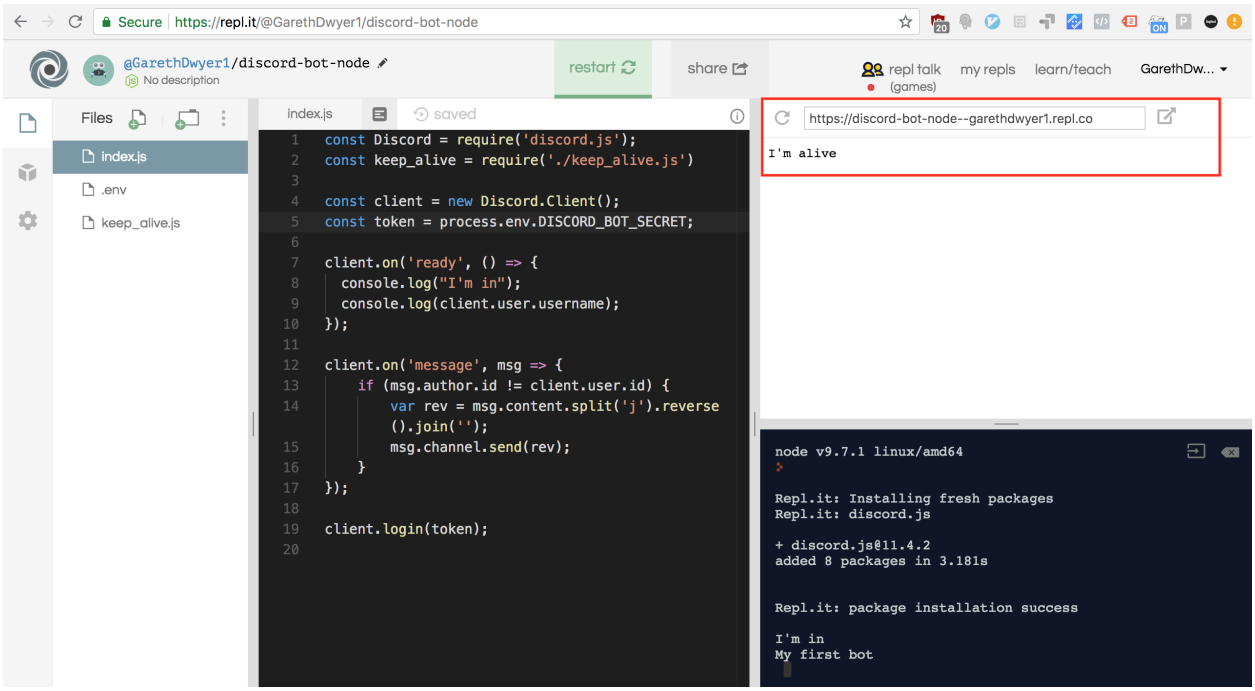
```
var http = require('http');

http.createServer(function (req, res) {
  res.write("I'm alive");
  res.end();
}).listen(8080);
```

We won't go over this in detail as it's not central to our bot, but here we start a web server that will return "I'm alive" if anyone visits it.

In our `index.js` file, we need to add a require statement for this server at the top. Add the following line near the top of `index.js`.

```
1  const keep_alive = require('./keep_alive.js')
```

After doing this and hitting the green "Run" button again, you should see some changes to your Repl. For one, you'll see a new pane in the top right which shows the web output from your server. We can see that visiting our Repl now returns a basic web page showing the "I'm alive" string that we told our web server to return by default.



Running a Node server in the background

Image 12 Output from our Node server

Now your bot will stay alive even after closing your browser or shutting down your development machine. Repl will still clean up your server and kill your bot after about one hour of inactivity, so if you don't use your bot for a while, you'll have to log into Repl and start the bot up again. Alternatively, you can set up a third-party (free!) service like Uptime Robot[63]. Uptime Robot pings your site every 5 minutes to make sure it's still working – usually to notify you of unexpected downtime, but in this case the constant pings have the side effect of keeping our Repl alive as it will never go more than an hour without receiving any activity. Note that you need to select the HTTP option instead of the Ping option when setting up Uptime Robot as repl.it requires regular HTTP requests to keep your chatbot alive.

[63] https://uptimerobot.com/

Forking and extending our basic bot

This is not a very useful bot as is, but the possibilities are only limited by your creativity now! You can have your bot receive input from a user, process the input, and respond in any way you choose. In fact, with the basic input and output that we've demonstrated, we have most of the components of any modern computer, all of which are based on the Von Neumann architecture[64] (we could easily add the missing memory by having our bot write to a file, or with a bit more effort link in a SQLite database[65] for persistent storage).

If you followed along this tutorial, you'll have your own basic Repl bot to play around with and extend. If you were simply reading, you can easily fork my bot at https://repl.it/@GarethDwyer1/discord-bot-node[66] and extend it how you want (you'll need to add your own token and recreate the `.env` file still). Happy hacking!

If you're stuck for ideas, why not link up your Discord bot to the Twitch API[67] to get notified when your favourite streamers are online, or build a text adventure[68]. Also join Repl's Discord server by using this invite link https://discord.gg/QWFfGhy[69] - you can test your bot, share it with other bot builders to get feedback, and see what Discord bots people are building on Repl.

If you enjoyed this tutorial, you might also enjoy my tutorial on building a chatbot for Telegram[70] or my book Flask by Example[71] where I show how to build Python applications using the Flask framework. If you have any questions or comments about this tutorial, feel free to reach out on Twitter[72].

[64] https://en.wikipedia.org/wiki/Von_Neumann_architecture
[65] https://www.sqlite.org/index.html
[66] https://repl.it/@GarethDwyer1/discord-bot-node
[67] https://dev.twitch.tv/
[68] https://en.wikipedia.org/wiki/Interactive_fiction
[69] https://discord.gg/QWFfGhy
[70] https://www.codementor.io/garethdwyer/building-a-telegram-bot-using-python-part-1-goi5fncay
[71] https://www.packtpub.com/web-development/flask-example
[72] https://twitter.com/sixhobbits

Creating and hosting a basic web application with Django and Repl.it

In this tutorial, we'll be using Django to create an online service that shows visitors their current weather and location. We'll develop the service and host it using repl.it[73].

To work through this tutorial, you should ideally have basic knowledge of Python and some knowledge of web application development. However, we'll explain all of our reasoning and each line of code thoroughly , so if you have any programming experience you should be able to follow along as a complete Python or web app beginner too. We'll also be making use of some HTML, JavaScript, and jQuery, so if you have been exposed to these before you'll be able to work through more quickly. If you haven't, this will be a great place to start.

To display the weather at the user's current location, we'll have to tie together a few pieces. The main components of our system are:

- A Django[74] application, to show the user a webpage with dynamic data
- Ipify[75] to get our visitors' IP address so that we can guess their location
- ip-api[76] to look up our visitors' city and country using their IP address
- Open Weather Map[77] to get the current weather at our visitors' location.

The main goals of this tutorial are to:

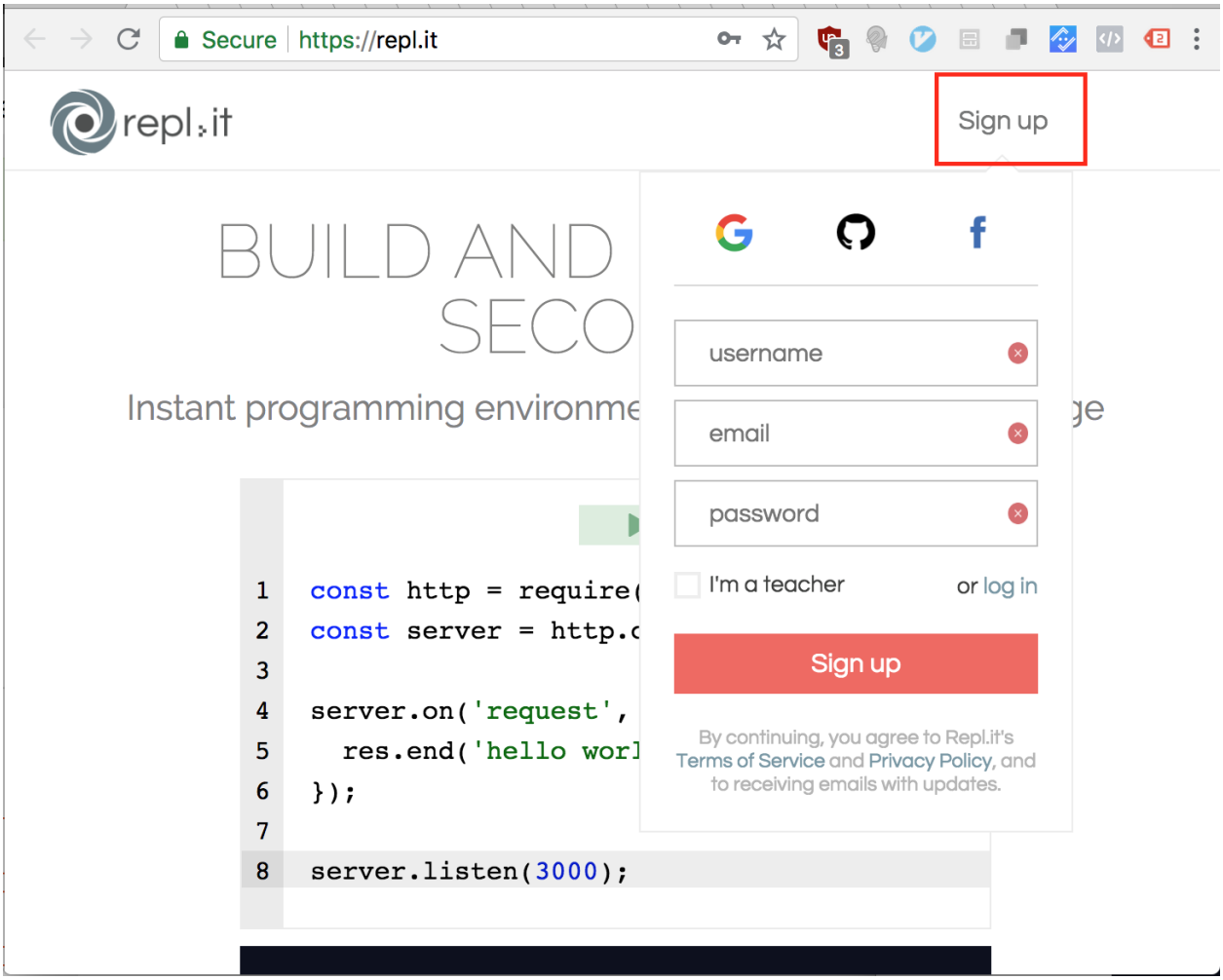
- Show how to set up and host a Django application using repl.it.
- Show how to join existing APIs together to create a new service.

By using this tutorial as a starting point, you can easily create your own bespoke web applications. Instead of showing weather data to your visitors, you could, for example, pull and combine data from any of the hundreds of APIs found at this list of public APIs[78].

[73] https://repl.it
[74] https://www.djangoproject.com/
[75] https://www.ipify.org/
[76] http://ip-api.com/
[77] https://openweathermap.org
[78] https://github.com/toddmotto/public-apis

Setting up

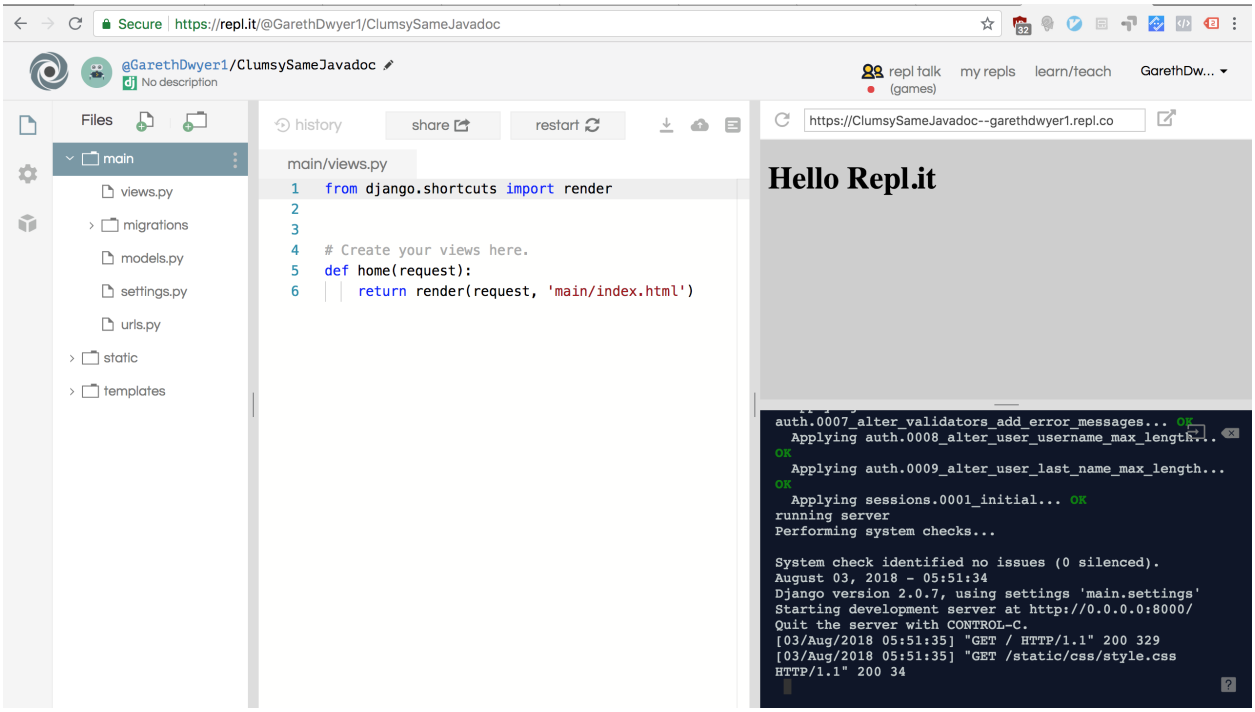
You won't need to install any software or programming languages on your local machine, as we'll be developing our application directly through repl.it[79]. Head over there and create an account by hitting "Sign up" in the top right-hand corner.



Repl sign up page

Press the "Start coding now" button on the next page, and search for "Django" from the list of available languages and frameworks. Repl.it will go ahead and set up a full Django project for you with some sensible defaults. The project will include a single page saying "Hello Repl.it". It should look very similar to the page below.

[79] repl.it



The repl.it IDE on a new Django project

In the left-most panel, you can see all the files that make up your Django project, organized into directories.

We won't explain what all these different components are for and how they tie together in this tutorial. If you want to understand Django better, you should definitely go through their official tutorial[80]. Normally I recommend Flask over Django[81] for beginners and for projects this simple, but Repl.it makes the Django setup easy by providing this initial set up. We'll be modifying only a few of these files to get the results that we want.

In the middle Repl.it pane, you can see your code. In the top-right pane, you can see your web application, as a visitor would see it. In the bottom-right pane, you can see any errors or other console output from your Repl.it webserver.

Changing our static content

Viewing the default website isn't that interesting. Let's make a small change to make the website our own. The first thing we'll change is the static "front end" of the website – the text that is displayed to all visitors.

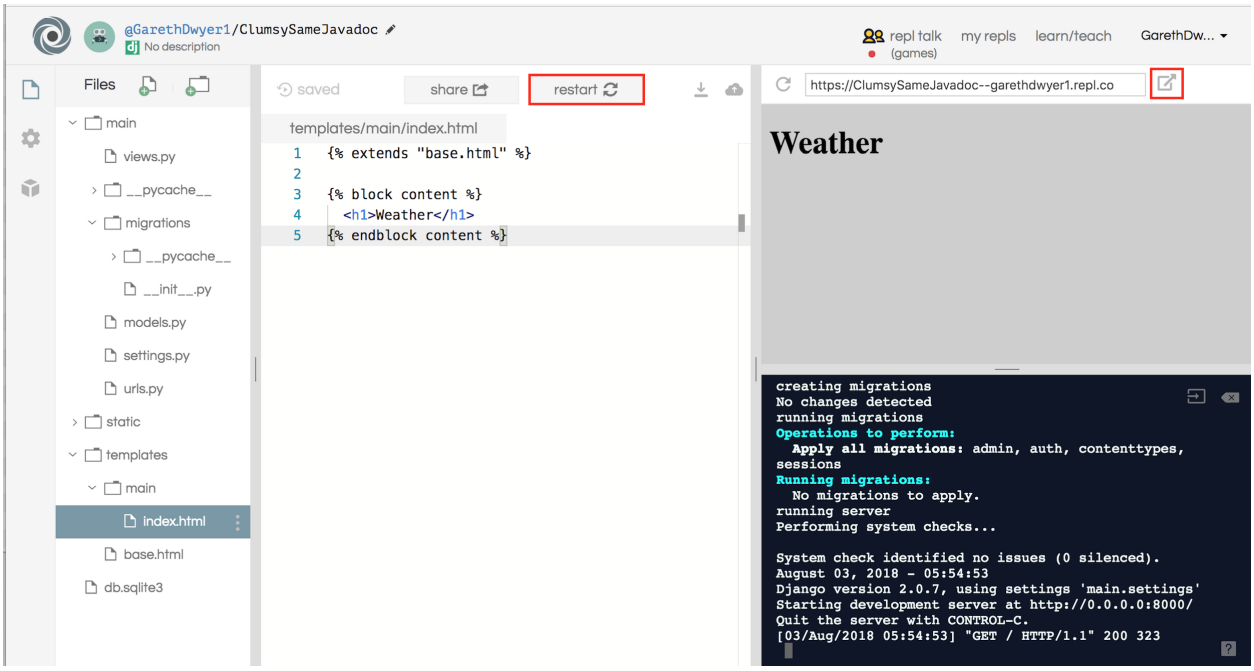
The file that we'll need to make changes to is the main template at `templates/main/index.html`. You can access it by selecting it from the left file bar. It should look like this:

[80] https://docs.djangoproject.com/en/2.0/intro/tutorial01/
[81] https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v

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

{% block content %}
  <h1>Hello Repl.it</h1>
{% endblock content %}
```

This is a file written in Django's template language[82], which often looks very much like HTML (and is usually found in files with a `.html` extension), but which contains some extra functionality to help us load dynamic data from the back end of our web application into the front end for our users to see.

Change the code where it says "Hello Repl.it" to read "Weather", and press the restart button as indicated below to see the result change from "Hello Repl.it" to "Weather".



Repl.it Restart and pop-out buttons

You can also press the pop-out button to the right of the the URL bar to open only the resulting web page that we're building, as a visitor would see it. You can share the URL with anyone and they'll be able to see your Weather website already!

Changing the static text that our visitors see is a good start, but our web application still doesn't *do* anything. We'll change that in the next step by using JavaScript to get our user's IP Address.

[82] https://docs.djangoproject.com/en/2.0/topics/templates/

Calling IPIFY from JavaScript

An IP address is like a phone number. When you visit "google.com", your computer actually looks up the the name google.com to get a resulting IP address that is linked to one of Google's servers. While people find it easier to remember names like "google.com", computers work better with numbers. Instead of typing "google.com" into your browser toolbar, you could type the IP address 216.58.223.46, with the same results. Every device connecting to the internet, whether to serve content (like google.com) or to consume it (like you, reading this tutorial) has an IP address.

IP addresses are interesting to us because it is possible to guess a user's location based on their IP address. (In reality, this is an imprecise and highly complicated[83] process, but for our purposes it will be more than adequate). We will use the web service ipify.org[84] to retrieve our visitors' IP addresses.

In the Repl.it files tab, navigate to `templates/base.html`, which should look as follows.

```
1  {% load staticfiles %}
2  <!DOCTYPE html>
3
4  <html lang="en">
5  <head>
6      <meta charset="UTF-8">
7      <title>Hello Django</title>
8      <meta charset="UTF-8"/>
9      <meta name="viewport" content="width=device-width, initial-scale=1"/>
10     <link rel="stylesheet" href="{% static "css/style.css" %}">
11 </head>
12 <body>
13     {% block content %}{% endblock content %}
14 </body>
15 </html>
```

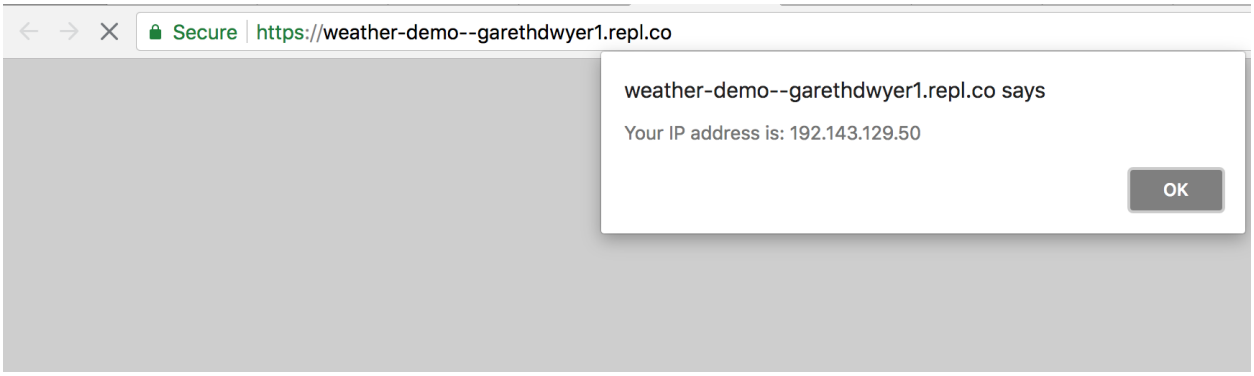
The "head" section of this template is between lines 5 and 11 – the opening and closing `<head>` tags. We'll add our scripts directly below the `<link ...>` on line 10 and above the closing `</head>` tag on line 11. Modify this part of code to add the following lines:

[83] https://dyn.com/blog/finding-yourself-the-challenges-of-accurate-ip-geolocation/
[84] https://www.ipify.org/

```
1  <script>
2    function use_ip(json) {
3      alert("Your IP address is: " + json.ip);
4    }
5    </script>
6
7  <script src="https://api.ipify.org?format=jsonp&callback=use_ip"></script>
```

These are two snippets of JavaScript. The first (lines 1-5) is a function that when called will display a pop-up box (an "alert") in our visitor's browser showing their IP address from the json object that we pass in. The second (line 7) loads an external script from ipify's API and asks it to pass data (including our visitor's IP address) along to the `use_ip` function that we provide.

If you open your web app again and refresh the page, you should see this script in action (if you're running an adblocker, it might block the ipify scripts, so try disabling that temporarily if you have any issues).



This code doesn't do anything with the IP address except display it to the user, but it is enough to see that the first component of our system (getting our user's IP Address) is working. This also introduces the first *dynamic* functionality to our app – before, any visitor would see exactly the same thing, but now we can show each visitor something related specifically to them (no two people have the same IP address).

Now instead of simply showing this IP address to our visitor, we'll modify the code to rather pass it along to our Repl webserver (the "backend" of our application), so that we can use it to fetch location information through a different service.

Adding a new route and view, and passing data

Currently our Django application only has one route, the default (`/`) route which is loaded as our home page. We'll add another route at `/get_weather_from_ip` where we can pass an IP address to our application to detect the location and get a current weather report.

To do this, we'll need to modify the files at `main/views.py` and `main/urls.py`.

Edit `urls.py` to look as follows (add line 8, but you shouldn't need to change anything else).

```
1  from django.conf.urls import url
2  from django.contrib import admin
3  from main import views
4
5  urlpatterns = [
6    url(r'^admin/', admin.site.urls),
7    url(r'^$', views.home, name='home'),
8    url(r'^get_weather_from_ip/', views.get_weather_from_ip, name="get_weather_from_ip\
9  "),
10 ]
```

We've added a definition for the `get_weather_from_ip` route, telling our app that if anyone visits http://weather-demo-garethdwyer1.repl.co/get_weather_from_ip[85] then we should trigger a function in our `views.py` file that is also called `get_weather_from_ip`. Let's write that function now.

In your `views.py` file, add a `get_weather_from_ip()` function beneath the existing `home()` one, and add an import for JsonResponse on line 2. Your whole `views.py` file should now look like this:

```
1  from django.shortcuts import render
2  from django.http import JsonResponse
3
4
5  # Create your views here.
6  def home(request):
7      return render(request, 'main/index.html')
8
9  def get_weather_from_ip(request):
10   print(request.GET.get("ip_address"))
11   data = {"weather_data": 20}
12   return JsonResponse(data)
```

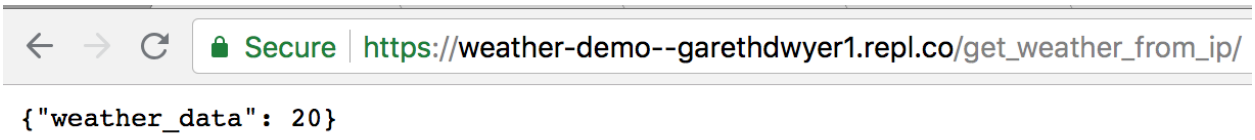
By default, Django passes a `request` argument to all views. This is an object that contains information about our user and the connection, and any additional arguments passed in the URL. As our application isn't connected to any weather services yet, we'll just make up a temperature (20) and pass that back to our user as JSON.

In line 10, we print out the IP address that we will pass along to this route from the `GET` arguments (we'll look at how to use this later). We then create the fake data (which we'll later replace with real data) and return a JSON response (the data in a format that a computer can read more easily, with no formatting). We return JSON instead of HTML because our system is going to use this route

[85] http://weather-demo--garethdwyer1.repl.co/get_weather_from_ip

internally to pass data between the front and back ends of our application, but we don't expect our users to use this directly.

To test this part of our system, open your web application in a new tab and add `/get_weather_from_ip?ip_address=123` to the URL. Here, we're asking our system to fetch weather data for the IP address `123` (not a real IP address). In your browser, you'll see the fake weather data displayed in a format that can easily be programmatically parsed.



Viewing the fake JSON data

In our Repl's output, we can see that the backend of our application has found the "IP address" and printed it out, between some other outputs telling us which routes are being visited and which port our server is running on:

Django print output of fake IP

The steps that remain now are to:

- pass the user's real IP address to our new route in the background when the user visits our main page
- add more backend logic to fetch the user's location from the IP address
- add logic to fetch the user's weather from their location

- display this data to the user.

Let's start by using Ajax[86] to pass the user's IP address that we collected before to our new route, without our user having to explicitly visit the `get_weather_from_ip` endpoint or refresh their page.

Calling a Django route using Ajax and jQuery

We'll use Ajax through jQuery[87] to do a "partial page refresh" – that is, to update part of the page the user is seeing by sending new data from our backend code without the user needing to reload the page.

To do this, we need to include jQuery as a library.'

> Note: usually you wouldn't add JavaScript directly to your base.html template, but to keep things simpler and to avoid creating too many files, we'll be diverging from some good practices. See the Django documentation[88] for some guidance on how to structure JavaScript properly in larger projects.

In your `templates/base.html` file, add the following script above the line where we previously defined the `use_ip()` function.

```
<script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
  integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
  crossorigin="anonymous"></script>
```

This loads the entire jQuery library from a CDN[89], allowing us to complete certain tasks using fewer lines of JavaScript.

Now, modify the `use_ip()` script that we wrote before to call our backend route using Ajax. The new `use_ip()` function should be as follows:

[86] https://en.wikipedia.org/wiki/Ajax_(programming)
[87] http://api.jquery.com/jquery.ajax/
[88] https://docs.djangoproject.com/en/2.0/howto/static-files/
[89] https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

```
1   function use_ip(json) {
2     $.ajax({
3       url: {% url 'get_weather_from_ip' %},
4       data: {"ip": json.ip},
5       dataType: 'json',
6       success: function (data) {
7         document.getElementById("weatherdata").innerHTML = data.weather_data
8       }
9     });
10  }
```

Our new `use_ip()`function makes an asynchronous[90] call to our `get_weather_from_ip` route, sending along the IP address that we previously displayed in a pop-up box. If the call is successful, we call a new function (in the `success:` section) with the returned data. This new function (line 7) looks for an HTML element with the ID of `weatherdata` and replaces the contents with the `weather_data` attribute of the response that we received from `get_weather_from_ip` (which at the moment is still hardcoded to be "20").

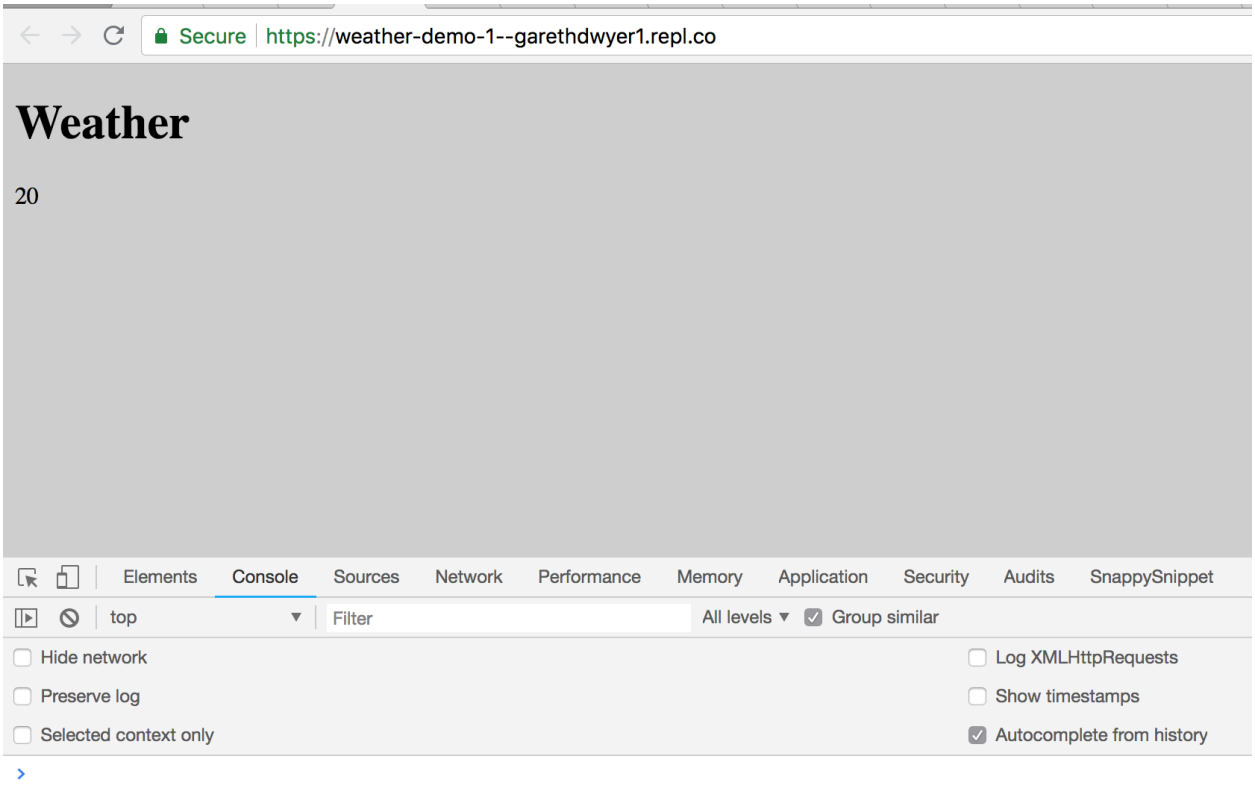
To see the results, we'll need to add an HTML element as a placeholder with the id `weatherdata`. Do this in the `templates/main/index.html` file as follows.

```
1  {% extends "base.html" %}
2
3  {% block content %}
4    <h1>Weather</h1>
5    <p id=weatherdata></p>
6  {% endblock %}
```

This adds an empty HTML *paragraph* element which our JavaScript can populate once it has the required data.

Now reload the app and you should see our fake `20` being displayed to the user. If you don't see what you expect, open up your browser's developer tools for Chrome[91] and Firefox[92]) and have a look at the Console section for any JavaScript errors. A clean console (with no errors) is shown below.

[90] http://api.jquery.com/jquery.ajax/
[91] https://developers.google.com/web/tools/chrome-devtools/
[92] https://developer.mozilla.org/son/docs/Tools



Now it's time to change out our mock data for real data by calling two services backend – the first to get the user's location from their IP address and the second to fetch the weather for that location.

Using ip-api.com for geolocation

The service at ip-api.com[93] is very simple to use. To get the country and city from an IP address we only need to make one web call. We'll use the python `requests` library for this, so first we'll have to add an import for this to our `views.py` file, and then write a function that can translate IP addresses to location information. Add the following import to your`views.py` file:

```
1 import requests
```

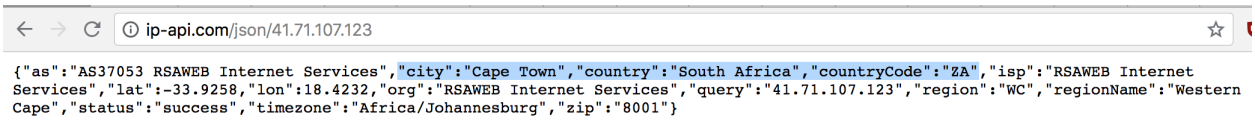
and above the `get_weather_from_ip()` function, add the`get_location_from_ip()` function as follows:

[93] http://ip-api.com

```
def get_location_from_ip(ip_address):
    response = requests.get("http://ip-api.com/json/{}".format(ip_address))
    return response.json()
```

> Note: again we are diverging from best practice in the name of simplicity. Usually whenever you write any code that relies on networking (as above), you should add exception handling[94] so that your code can fail more gracefully if there are problems.

You can see the response that we'll be getting from this service by trying it out in your browser. Visit http://ip-api.com/json/41.71.107.123[95] to see the JSON response for that specific IP address.



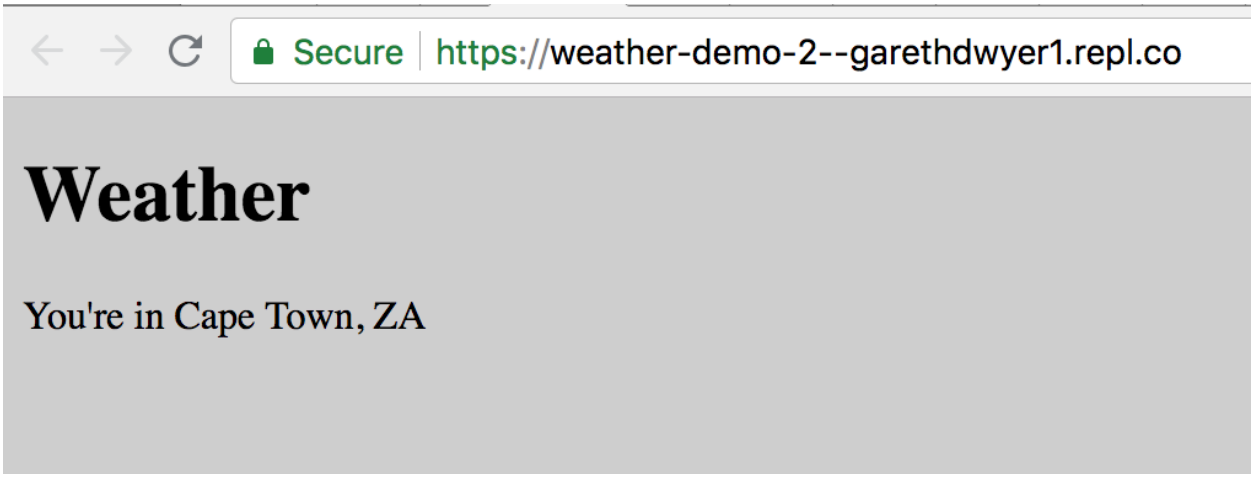
Take a look specifically at the highlighted location information that we'll need to extract to pass on to a weather service.

Before we set up the weather component, let's display the user's current location data instead of the hardcoded temperature that we had before. Change the `get_weather_from_ip()` function to call our new function and pass along some useful data as follows:

```
def get_weather_from_ip(request):
  ip_address = request.GET.get("ip")
  location = get_location_from_ip(ip_address)
  city = location.get("city")
  country_code = location.get("countryCode")
  s = "You're in {}, {}".format(city, country_code)
  data = {"weather_data": s}
  return JsonResponse(data)
```

Now, instead of just printing the IP address that we get sent and making up some weather data, we use the IP address to guess the user's location, and pass the city and country code back to the template to be displayed. If you reload your app again, you should see something similar to the following (though hopefully with your location instead of mine).

[94] https://docs.python.org/3/tutorial/errors.html
[95] http://ip-api.com/json/41.71.107.123



weather app, location showing

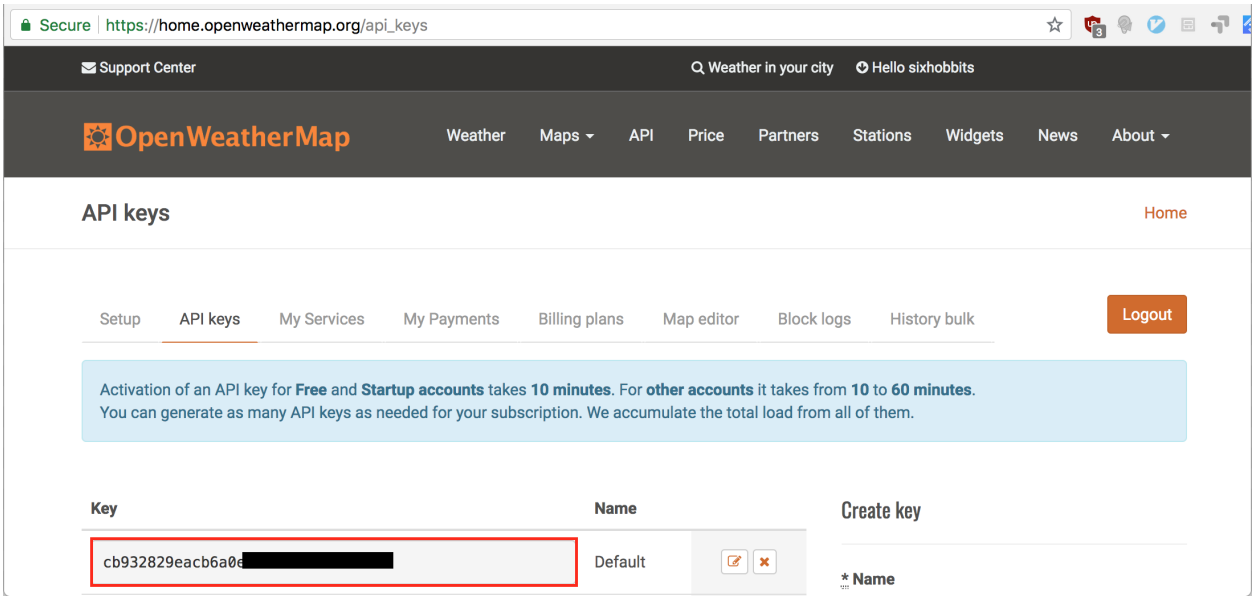
That's the location component of our app done and dusted – let's move on to getting weather data for that location now.

Getting weather data from OpenWeatherMap

To get weather data automatically from OpenWeatherMap[96], you'll need an API Key. This is a unique string that OpenWeatherMap gives to each user of their service and it's used mainly to restrict how many calls each person can make in a specified period. Luckily, OpenWeatherMap provides a generous "free" allowance of calls, so we won't need to spend any money to build our app. Unfortunately, this allowance is not quite generous enough to allow me to share my key with every reader of this tutorial, so you'll need to sign up for your own account and generate your own key.

Visit openweathermap.org[97], hit the "sign up" button, and register for the service by giving them an email address and choosing a password. Then navigate to the API Keys[98] section and note down your unique API key (or copy it to your clipboard).

[96] https://openweathermap.org/
[97] https://openweathermap.org/
[98] https://home.openweathermap.org/api_keys

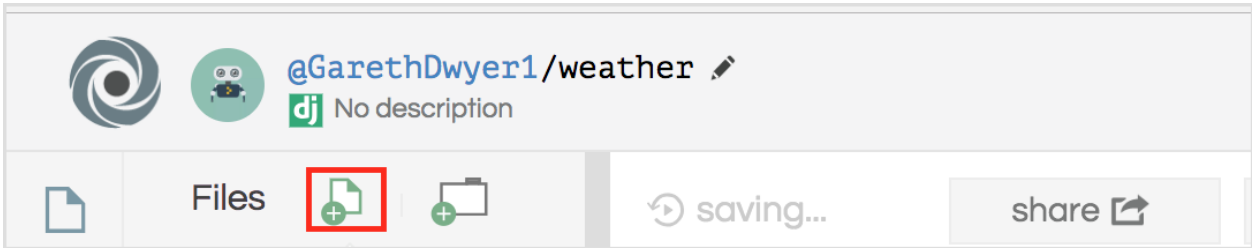


OpenWeatherMap API Key page

This key is a bit like a password – when we use OpenWeatherMap's service, we'll always send along this key to indicate that it's us making the call. Because Repl.it's projects are public by default, we'll need to be careful to keep this key private and prevent other people making too many calls using our OpenWeatherMap quota (potentially making our app fail when OpenWeatherMap starts blocking our calls). Luckily Repl.it provides a neat way of solving this problem using `.env` files[99].

In your project, create a new file using the "New file" button as shown below. Make sure that the file is in the root of your project and that you name the file `.env` (in Linux, starting a filename with a `.` usually indicates that it's a system or configuration file). Inside this file, define the `OPEN_WEATHER_TOKEN` variable as follows, but using your own token instead of the fake one below. Make sure not to have a space on either side of the `=` sign.

```
1 OPEN_WEATHER_TOKEN=1be9250b94bf6803234b56a87e55f
```



Creating a new file

Repl.it will load the contents of this file into our server's environment variables[100]. We'll be able to access this using the `os` library in Python, but when other people view or fork our Repl, they won't

[99] https://repl.it/site/docs/secret-keys
[100] https://wiki.archlinux.org/index.php/environment_variables

see the `.env` file, keeping our API key safe and private.

To fetch weather data, we need to call the OpenWeatherMap api, passing along a search term. To make sure we're getting the city that we want, it's good to pass along the country code as well as the city name. For example, to get the weather in London right now, we can visit (again, you'll need to add your own API key in place of the string after `appid=`) https://api.openweathermap.org/data/2.5/weather?q=Lond

To test this, you can visit the URL in your browser first. If you prefer Fahrenheit to Celsius, simply change the `unit=metric` part of the url to `units=imperial`.

Json response from OpenWeatherMap

Let's write one last function in our `views.py` file to replicate this call for our visitor's city which we previously displayed.

First we need to add an import for the Python `os` (operating system) module so that we can access our environment variables. At the top of `views.py` add:

```
1  import os
```

Now we can write the function. Add the following to `views.py`:

```
1  def get_weather_from_location(city, country_code):
2      token = os.environ.get("OPEN_WEATHER_TOKEN")
3      url = "https://api.openweathermap.org/data/2.5/weather?q={},{}&units=metric&appi\
4  d={}".format(
5          city, country_code, token)
6      response = requests.get(url)
7      return response.json()
```

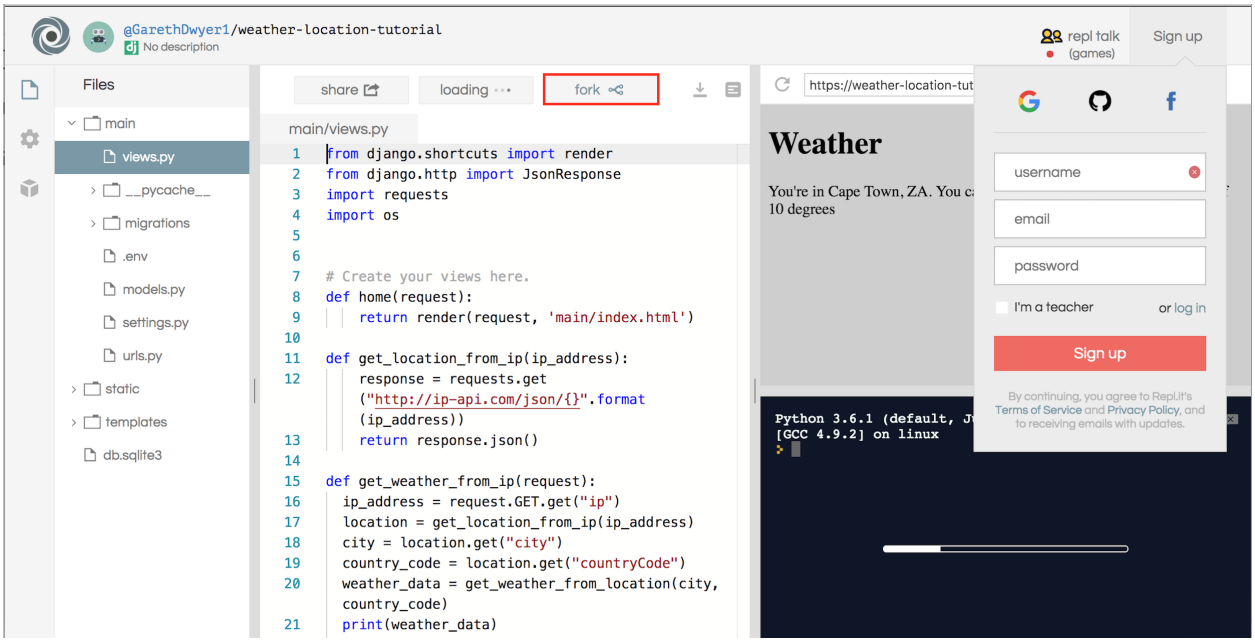
In line 2, we get our API key from the environment variables (note, you sometimes need to refresh the repl.it page with your repl in to properly load in the environment variables), and we then use this to format our URL properly in line 3. We get the response from OpenWeatherMap and return it as json.

We can now use this function in our `get_weather_from_ip()` function by modifying it to look as follows:

```
1  def get_weather_from_ip(request):
2    ip_address = request.GET.get("ip")
3    location = get_location_from_ip(ip_address)
4    city = location.get("city")
5    country_code = location.get("countryCode")
6    weather_data = get_weather_from_location(city, country_code)
7    description = weather_data['weather'][0]['description']
8    temperature = weather_data['main']['temp']
9    s = "You're in {}, {}. You can expect {} with a temperature of {} degrees".format(\
10 city, country_code, description, temperature)
11   data = {"weather_data": s}
12   return JsonResponse(data)
```

We now get the weather data in line 6, parse this into a description and temperature in lines 7 and 8, and add this to the string we pass back to our template in line 9. If you reload the page, you should see your location and your weather.

and hit the "Fork" button. If you didn't create an account at the beginning of this tutorial, you'll be prompted to create one again. (You can even use a lot of Repl functionality without creating an account.)



Forking a Repl

If you're stuck for ideas, some possible extensions are:

- Make the page look nicer by using Bootstrap[101] or another CSS framework in your template files.

[101] https://getbootstrap.com/

- Make the app more customizable by allowing the user to choose their own location if the IP location that we guess is wrong
- Make the app more useful by showing the weather forecast along with the current weather. (This data is also available[102] from Open Weather Map).
- Add other location-related data to the web app such as news, currency conversion, translation, postal codes. See https://github.com/toddmotto/public-apis#geocoding[103] for a nice list of possibilities.

If you liked this tutorial, you might be interested in Building a Chatbot using Telegram and Python[104] where I show how to build chatbots, or my book Flask by Example[105] where I give a more thorough introduction to web application development using Python. Feel free to ping me on Twitter[106] for any questions about this tutorial, to point out mistakes, or to ask about web development in general.

[102] https://openweathermap.org/forecast5
[103] https://github.com/toddmotto/public-apis#geocoding
[104] https://www.codementor.io/garethdwyer/building-a-telegram-bot-using-python-part-1-goi5fncay
[105] https://www.packtpub.com/web-development/flask-example
[106] https://twitter.com/sixhobbits

Building a CRM app with NodeJS, Repl.it, and MongoDB

In this tutorial we'll use NodeJS on Repl.it, along with a MongoDB database to build a basic CRUD[107] (Create, Read, Update, Delete) CRM[108] (Customer Relationship Management) application. A CRM lets you store information about customers to help you track the status of every customer relationship. This can help businesses keep track of their clients and ultimately increase sales. The application will be able to store and edit customer details, as well as keep notes about them.

This tutorial won't be covering the basics of Node.js, but each line of code will be explained in detail.

Setting up

All of the code will be written and hosted in Repl.it, so you won't need to install any additional software on your computer.

For setup, we'll be walking you through the following steps. Skip any that don't apply to you (e.g. if you already have a Repl.it account, you don't have to make a new one).

- Creating an account on Repl.it[109]
- Creating an account on MongoDB Atlas[110]

Creating an account on Repl.it

The first thing you need to do is create an account on Repl.it. You can find instructions on how to do so here[111]. Once you're done, head back here and continue the tutorial.

Creating an account on MongoDB Atlas

MongoDB Atlas is a fully managed Database-as-a-Service. It provides a document database (often referred to as NoSQL), as opposed to a more traditional relational database like PostgreSQL.

Head over to MongoDB Atlas[112] and hit the "Start free" button.

[107] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
[108] https://en.wikipedia.org/wiki/Customer_relationship_management
[109] https://repl.it/
[110] https://www.mongodb.com/cloud/atlas
[111] https://www.codementor.io/garethdwyer/building-a-discord-bot-with-node-js-and-repl-it-mm46r1u8y#creating-an-account-on-replit
[112] https://www.mongodb.com/cloud/atlas

After signing up, under “Starter Clusters”, press the “Create a Cluster” button.

You now have to select a provider and a region. For the purposes of this tutorial we chose Google Cloud Platform as the provider and Iowa (us-central1) as the region, although it should work regardless of the provider and region.

Cluster Region

Under “Cluster Name” you can change the name of your cluster. Note that you can only change the name now - it can’t be changed once the cluster is created. After you’ve done that, click “Create Cluster”.

After a bit of time, your cluster will be created. Once it’s available, click on “Database Access” under the Security heading in the left-hand column and then click “Add New User” to create a new database user. You need a database user to actually store and retrieve data. Enter a username and password for the user and make a note of those details - you’ll need them later. Select “Read and write to any database” as the user privilege.

Next, you need to allow network access to the database. Click on “Network Access” in the left-hand column, and “Add an IP Address”. Because we won’t have a static IP from Repl.it, we’re just going to allow access from anywhere - don’t worry, the database is still secured with the username and password you created earlier. In the popup, click “Allow Access From Anywhere”.

Allow Access From Anywhere

Navigate back to “Clusters”, click on “Connect” and select “Connect Your Application”. Copy the Connection String as you will need it shortly to connect to your database from Repl.it. Ours looked like this: `mongodb+srv://<username>:<password>@cluster0-zrtwi.gcp.mongodb.net/test?retryWrites=true&w=m`

Creating a Repl and connecting to our Database

First, we need to create a new Node.js Repl to write the code necessary to connect to our shiny new Database. On Repl.it, create a new Repl and select “Node.js” as the language.

A great thing about Repl is that it makes projects public by default. This makes it easy to share and is great for collaboration and learning, but we have to be careful not to make our database credentials available on the open Internet.

To solve this problem we’ll be using `environment variables`. We’ll create a special file that Repl.it recognizes and keeps private for you, and in that file we declare variables that become part of our Repl.it development environment and are accessible in our code.

Select your Repl by clicking "My Repls" in the left-hand pane, followed by clicking on the Repl's name. Now create a file called `.env` by selecting "Files" in the left-hand pane and then clicking the "Add File" button. Note that the spelling has to be exact or the file will not be recognized. Add your MongoDB database username and password (not your login details to MongoDB Atlas) into the file in the below format:

```
MONGO_USERNAME=username
MONGO_PASSWORD=password
```

- Replace `username` and `password` with your database username and password
- **Spacing matters**. Make sure that you don't add any spaces before or after the = sign

Now that we have credentials set up for the database, we can move on to connecting to it in our code.

MongoDB is kind enough to provide a client that we can use. To test out our database connection, we're going to insert some customer data into our database. In your `index.js` file (created automatically and found under the Files pane), add the following code:

```
1 const MongoClient = require('mongodb').MongoClient;
2 const mongo_username = process.env.MONGO_USERNAME
3 const mongo_password = process.env.MONGO_PASSWORD
4
5 const uri = `mongodb+srv://${mongo_username}:${mongo_password}@cluster0-zrtwi.gcp.mo\
6 ngodb.net/crmdb?retryWrites=true&w=majority`;
7 const client = new MongoClient(uri, { useNewUrlParser: true });
```

Here's what's going on:

- **Line 1** adds the dependency for the MongoDB Client. Repl.it makes things easy by installing all the dependencies for us, so we don't have to use something like npm to do it manually.
- **Line 2 & 3** we retrieve our MongoDB username and password from the environment variables that we set up earlier.
- **Line 5** has a few very important details that we need to get right.
  - replace the section between the `@` and the next `/` with the same section of your connection string from MongoDB that we copied earlier. You may notice the `${mongo_username}` and `${mongo_password}` before and after the colon near the beginning of the string. These are called Template Literals. Template Literals allow us to put variables in a string, which Node.js will then helpfully replace with the actual values of the variables.
  - Note `crmdb` after the `/` and before the `?`. This will be the name of the database that we will be using. MongoDB creates the database if it doesn't exist for us. You can change this to whatever you want to name the database, but remember what you changed it to for future sections of this tutorial.
- **Line 6** creates the client that we will use to connect to the database.

Making a user interface to insert customer data

We're going to make an HTML form that will capture the customer data, send it to our Repl.it code, which will then insert it into our database.

In order to actually present and handle an HTML form, we need a way to process HTTP GET and POST requests. The easiest way to do this is to use a web application framework. A web application framework is designed to support the development of web applications - it gives you a standard way to build your application and lets you get to building your application fast without having to do the boilerplate code.

A really simple and flexible Node.js web application framework is Express[113]. Express is a fast and flexible web application framework that provides a robust set of features for the development of web applications.

The first thing we need to do is add the dependencies we need. Right at the top of your index.js file, add the following lines:

```
1 let express = require('express');
2 let app = express();
3 let bodyParser = require('body-parser');
4 let http = require('http').Server(app);
5
6 app.use(bodyParser.json())
7 app.use(bodyParser.urlencoded({ extended: true }));
```

Let's break this down.

- **Line 1** adds the dependency for Express. Repl.it will take care of installing it for us.
- **Line 2** creates a new Express app that will be needed to handle incoming requests.
- **Line 3** adds a dependency for 'body-parser'. This is needed for the Express server to be able to handle the data that the form will send, and give it to us in a useful way in code.
- **Line 4** adds a dependency for a basic HTTP server.
- **Line 6 & 7** tell the Express app which parsers to use on incoming data. This is needed to handle form data.

Next, we need to add a way for the Express to handle an incoming request and give us the form that we want. Add the following lines of code below your dependencies:

[113] https://expressjs.com/

```
app.get('/', function (req, res) {
  res.sendFile('/index.html', {root:'.'});
});

app.get('/create', function (req, res) {
  res.sendFile('/create.html', {root:'.'});
});
```

- `app.get` tells Express that we want it to handle a GET request.
- `'/'` tells Express that it should respond to GET requests sent to the root URL. A root URL looks something like 'https://crm.hawkiesza.repl.co' - note that there are no slashes after the URL.
- `'/create'` tells Express that it should respond to GET requests to /create after the root URL i.e. 'https://crm.hawkiesza.repl.co/create'
- `res.sendFile` tells Express to send the given file as a response.

Before the server will start receiving requests and sending responses, we need to tell it to run. Add the following code below the previous line.

```
app.set('port', process.env.PORT || 5000);
http.listen(app.get('port'), function() {
    console.log('listening on port', app.get('port'));
});
```

- **Line 1** tells Express to set the port number to either a number defined as an environment variable, or 5000 if no definition was made.
- **Line 2-4** tells the server to start listening for requests.

Now we have an Express server listening for requests, but we haven't yet built the form that it needs to send back if it receives a request.

Make a new file called `index.html` and paste the following code into it:

```
<!DOCTYPE html>
<html>
<body>
<form action="/create" method="GET">
  <input type="submit" value="Create">
</form>

</body>
</html>
```

This is just a simple bit of HTML that puts a single button on the page. When this button is clicked it sends a GET request to `/create`, which the server will then respond to according to the code that we wrote above - in our case it will send back the `create.html` file which we will define now.

Make a new file called `create.html` and paste the following into it:

```
<!DOCTYPE html>
<html>
<body>

<h2>Customer details</h2>

<form action="/create" method="POST">
  <label for="name" >Customer name *</label><br>
  <input type="text" id="name" name="name" class="textInput" placeholder="John Smith\
" required>
  <br>
  <label for="address" >Customer address *</label><br>
  <input type="text" name="address" class="textInput" placeholder="42 Wallaby Way, S\
ydney" required>
  <br>
  <label for="telephone" >Customer telephone *</label><br>
  <input type="text" name="telephone" class="textInput" placeholder="+275554202" req\
uired>
  <br>
  <label for="note" >Customer note</label><br>
  <input type="text" name="note" class="textInput" placeholder="Needs a new pair of \
shoes">
  <br><br>
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

We won't go in depth into the above HTML. It is a very basic form with 4 fields (name, address, telephone, note) and a Submit button, which creates an interface that will look like the one below.

Customer Details

When the user presses the submit button a POST request is made to `/create` with the data in the form - we still have to handle this request in our code as we're currently only handling a GET request to `/`.

If you now start up your application (click the "run" button) a new window should appear on the right that displays your form. You can also navigate to `https://<repl_name>.<your_username>.repl.co` (replace <repl_name> with whatever you named your Repl (but with no underscores or spaces) and <your_username> with your Repl username) to see the form. You will be able to see this URL in your Repl itself.

If you fill in the form and click submit, you'll get a response back that says `Cannot POST /create`. We haven't added the code that handles the form POST request, so let's do that. Add the following code into your `index.js` file, below the `app.get` entry that we made above.

```
1  app.post('/create', function (req, res, next) {
2    client.connect(err => {
3      const customers = client.db("crmdb").collection("customers");
4
5      let customer = { name: req.body.name, address: req.body.address, telephone: req.\
6  body.telephone, note: req.body.note };
7      customers.insertOne(customer, function(err, res) {
8        if (err) throw err;
9        console.log("1 customer inserted");
10     });
11   })
12   res.send('Customer created');
13 })
```

- **Line 1** defines a new route that listens for an HTTP 'POST' request at `/create`.
- **Line 2** connects to the database. This happens asynchronously, so we define a callback function that will be called once the connection is done.
- **Line 3** creates a new collection of customers. Collections in MongoDB are similar to Tables in SQL.
- **Line 5** defines customer data that will be inserted into the collection. This is taken from the incoming request. The form data is parsed using the parsers that we defined earlier and is then placed in the `req.body` variable for us to use in the code.

- **Line 6** inserts the customer data into the collection. This also happens asynchronously, and so we define another callback function that will get an error if an error occurred, or the response if everything happened successfully.
- **Line 7** throws an error if the above insert had a problem.
- **Line 8** gives us some feedback that the insert happened successfully.

If you now fill in the form and click submit, you'll get a message back that says "Customer created". If you then go and look in your MongoDB collection, you'll see a document has been created with the details that we submitted in the form.

Updating and deleting database entries

As a final step in this tutorial, we want to be able to update and delete database documents in our collection. To make things simpler we're going to make a new HTML page where we can request a document and then update or delete it.

First, let's make the routes to our new page. In your `index.js`, add the following code below the rest of your routing code:

```
1  app.get('/get', function (req, res) {
2    res.sendFile('/get.html', {root:'.'});
3  });
4
5  app.get('/get-client', function (req, res) {
6      client.connect(err => {
7          client.db("crmdb").collection("customers").findOne({name: req.query.name}, f\
8  unction(err, result) {
9            if (err) throw err;
10           res.render('update', {oldname: result.name, oldaddress: result.address, ol\
11 dtelephone: result.telephone, oldnote: result.note, name: result.name, address: resu\
12 lt.address, telephone: result.telephone, note: result.note});
13         });
14       });
15 });
```

- **Line 1-3** as before, this tells Express to respond to incoming GET requests on `/get` by sending the `get.html` file which we will define below.
- **Line 5-12** this tells Express to respond to incoming GET requests on `/get-client`.
  - **Line 7** makes a call to the database to fetch a customer by name. If there are more than 1 with the same name, then the first one found will be returned.

- **Line 9** tells Express to render the `update` template, replacing variables with the given values as it goes. Important to note here is that we are also replacing values in the hidden form fields we created earlier with the current values of the customer details. This is to ensure that we update or delete the correct customer.

In your `index.html` file, add the following code after the `</form>` tag:

```
<br>
<form action="/get" method="GET">
  <input type="submit" value="Update/Delete">
</form>
```

This adds a new button that will make a GET request to `/get`, which will then return `get.html`.

Index

Make a new file called `get.html` with the following contents:

```
<!DOCTYPE html>
<html>
<body>
  <form action="/get-client" method="GET">
    <label for="name" >Customer name *</label><br>
    <input type="text" id="name" name="name" class="textInput" placeholder="John Smi\
th" required>
    <input type="submit" value="Get customer">
  </form>
</body>
</html>
```

This makes a simple form with an input for the customer's name and a button.

Get Customer

Clicking this button will then make a GET call to `/get-client` which will respond with the client details where we will be able to update or delete them.

To actually see the customer details on a form after requesting them, we need a templating engine to render them onto the HTML page and send the rendered page back to us. With a templating engine, you define a template - a page with variables in it - and then give it the values you want to fill into

the variables. In our case, we're going to request the customer details from the database and tell the templating engine to render them onto the page.

We're going to use a templating engine called Pug[114]. Pug is a simple templating engine that integrates fully with Express. The syntax that Pug uses is very similar to HTML. One important difference in the syntax is that spacing is very important as it determines your parent/child hierarchy.

First, we need to tell Express which templating engine to use and where to find our templates. Put the following line above your route definitions (i.e. after the other app. lines in index.js):

```
app.engine('pug', require('pug').__express)
app.set('views', '.')
app.set('view engine', 'pug')
```

Now create a new file called `update.pug` with the following content:

```
html
  body
    p #{message}
    h2= 'Customer details'
    form(method='POST' action='/update')
      input(type='hidden' id='oldname' name='oldname' value=oldname)
      input(type='hidden' id='oldaddress' name='oldaddress' value=oldaddress)
      input(type='hidden' id='oldtelephone' name='oldtelephone' value=oldtelephone)
      input(type='hidden' id='oldnote' name='oldnote' value=oldnote)
      label(for='name') Customer name:
      br
      input(type='text', placeholder='John Smith' name='name' value=name)
      br
      label(for='address') Customer address:
      br
      input(type='text', placeholder='42 Wallaby Way, Sydney' name='address' value=a\
ddress)
      br
      label(for='telephone') Customer telephone:
      br
      input(type='text', placeholder='+275554202' name='telephone' value=telephone)
      br
      label(for='note') Customer note:
      br
      input(type='text', placeholder='Likes unicorns' name='note' value=note)
```

[114] https://pugjs.org/api/getting-started.html

```
26        br
27        button(type='submit' formaction="/update") Update
28        button(type='submit' formaction="/delete") Delete
```

This is very similar to the HTML form we created previously for `create.html`, however this is written in the Pug templating language. We're creating a hidden element to store the "old" name, telephone, address, and note of the customer - this is for when we want to do an update.

Using the old details to update the customer is an easy solution, but not the best solution as it makes the query cumbersome and slow. If you add extra fields in your database you would have to remember to update your query as well, otherwise it could lead to updating or deleting the wrong customer if they have the same information. A better, but more complicated way is to use the unique ID of the database document as that will only ever refer to one customer.

We have also put in placeholder variables for name, address, telephone, and note, and we have given the form 2 buttons with different actions.

If you now run the code, you will have an index page with 2 buttons. Pressing the 'Update/Delete' button will take you to a new page that asks for a Customer name. Filling the customer name and pressing 'Get customer' will, after a little time, load a page with the customer's details and 2 buttons below that say 'Update' and 'Delete'.

Update-Delete

Our next step is to add the 'Update' and 'Delete' functionality. Add the following code below your routes in `index.js`:

```
1  app.post('/update', function(req, res) {
2    client.connect(err => {
3      if (err) throw err;
4      let query = { name: req.body.oldname, address: req.body.oldaddress, telephone: r\
5  eq.body.oldtelephone, note: req.body.oldnote };
6      let newvalues = { $set: {name: req.body.name, address: req.body.address, telepho\
7  ne: req.body.telephone, note: req.body.note } };
8      client.db("crmdb").collection("customers").updateOne(query, newvalues, function(\
9  err, result) {
10         if (err) throw err;
11         console.log("1 document updated");
12         res.render('update', {message: 'Customer updated!', oldname: req.body.name, \
13 oldaddress: req.body.address, oldtelephone: req.body.telephone, oldnote: req.body.no\
14 te, name: req.body.name, address: req.body.address, telephone: req.body.telephone, n\
15 ote: req.body.note});
16       });
```

```
17    });
18  })
19
20  app.post('/delete', function(req, res) {
21    client.connect(err => {
22      if (err) throw err;
23      let query = { name: req.body.name, address: req.body.address ? req.body.address \
24  : null, telephone: req.body.telephone ? req.body.telephone : null, note: req.body.no\
25  te ? req.body.note : null };
26      client.db("crmdb").collection("customers").deleteOne(query, function(err, obj) {
27        if (err) throw err;
28        console.log("1 document deleted");
29        res.send(`Customer ${req.body.name} deleted`);
30      });
31    });
32  })
```

This introduces 2 new 'POST' handlers - one for `/update`, and one for `/delete`.

- **Line 2** connects to our MongoDB database.
- **Line 3** throws an error if there was a problem connecting to the database.
- **Line 4** defines a query that we will use to find the document to update. In this case, we are using the details of the customer *before* it was updated. We saved this name earlier in a hidden field in the HTML. Trying to find the customer by its updated name obviously won't work because it hasn't been updated yet. Also, note that we are setting some of the fields to null if they are empty. This is so that the database returns the correct document when we update or delete - if we search for a document that has no address with an address of '' (empty string), then our query won't return anything.
- **Line 5** defines the new values that we want to update our customer with.
- **Line 6** updates the customer with the new values using the query
- **Line 7** throws an error if there was a problem with the update.
- **Line 8** logs that a document was updated.
- **Line 9** re-renders the update page with a message saying that the customer was updated, and displays the new values.
- **Line 15** connects to our MongoDB database.
- **Line 16** throws an error if there was a problem connecting to the database.
- **Line 17** defines a query that we will use to find the document to delete. In this case we are using all the details of the customer *before* any changes were made on the form to make sure we delete that specific customer.
- **Line 18** we connect to the database and delete the customer.
- **Line 19** throws an error if there was a problem with the delete.
- **Line 20** logs that a document was deleted.
- **Line 21** sends a response to say that the customer was deleted.

Putting it all together

If you run your application now, you'll be able to create, update, and delete documents in a MongoDB database. This is a very basic CRUD application, with a very basic and unstyled UI, but it should give you the foundation to build much more sophisticated applications.

For instance, you could add fields to the database to classify customers according to which stage they are in your sales pipeline[115] so that you can track if a customer is potentially stuck somewhere and contact them to re-engage.

You could then integrate some basic marketing automation with a page allowing you to send an email or SMS to customers (though don't spam clients!).

You could also add fields to keep track of customer purchasing information so that you can see which products do well with which customers.

If you want to start from where this tutorial leaves off, simply fork the Repl at https://repl.it/@GarethDwyer1/nodejs-crm[116].
To get additional guidance from the Repl community, also join Repl's Discord server by using this invite link https://discord.gg/QWFfGhy[117].

This article was contributed by Gerrit Vermeulen and edited by Katherine James.

[115] https://www.bitrix24.com/glossary/what-is-pipeline-management-definition-crm.php
[116] https://repl.it/@GarethDwyer1/nodejs-crm
[117] https://discord.gg/QWFfGhy

Introduction to Machine Learning with Python and repl.it

In this tutorial, we're going to walk through how to set up a basic Python repl[118] that can learn the difference between two categories of sentences, positive and negative. For example, if you had the sentence "I love it!", we want to train a machine to know that this sentence is associated with happy and positive emotions. If we have a sentence like "it was really terrible", we want the machine to label it as a negative or sad sentence.

The maths, specifically calculus and linear algebra, behind machine learning gets a bit hairy. We'll be abstracting this away with the Python library scikit-learn[119], which makes it possible to do advanced machine learning in a few lines of Python.

At the end of this tutorial you'll understand the fundamental ideas of automatic classification and have a program that can learn by itself to distinguish between different categories of text. You'll be able to use the same code to learn new categories (e.g. spam/not-spam, or clickbait/non-clickbait).

Prerequisites

To follow along this tutorial, you should have at least basic knowledge of Python or a similar programming language. Ideally, you should also sign up for a repl.it[120] account so that you can modify and extend the bot we build, but it's not completely necessary.

Setting up

If you're following along using repl.it, then visit the homepage and login or create a new account. Follow the prompts to create your first Repl, and choose "Python". You'll be taken to a new Repl project where you can run Python code and immediately see the output, which is great for rapid development.

The first thing we need to do is install scikit-learn, which is a really nice Python library to get started with machine learning. Create a new file using the "add file" button at the top left, call the file `requirements.txt` (the exact name is important – it's a special file that repl will look for dependencies in and install them automatically), and add the line `scikit-learn` to the top of the new file that gets created.

[118] https://repl.it
[119] https://scikit-learn.org/
[120] https://repl.it

You should see the library installing through the output produced in the right-hand panel, as in the image below.

Now you have the powerful and simple scikit-learn available! Let's learn how to use it. Open the `main.py` file that Repl created for you automatically and add the following two imports to the top.

```
1 from sklearn import tree
2 from sklearn.feature_extraction.text import CountVectorizer
```

In line 1, we import the `tree` module, which will give us a Decision Tree classifier that can learn from data. In line 2, we import a vectoriser – something that can turn text into numbers. We'll describe each of these in more detail soon!

Throughout the next steps, you can hit the big green "run" button to run your code, check for bugs, and view output along the way (you should do this every time you add new code).

Creating some mock data

Before we get started with the exciting part, we'll create a very simple dataset – too simple in fact. You might not see the full power of machine learning at first as our task will look so easy, but once we've walked through the concepts, it'll be a simple matter of swapping the data out for something bigger and more complicated.

On the next lines of main.py add the following lines of code.

```
1  positive_texts = [
2      "we love you",
3      "they love us",
4      "you are good",
5      "he is good",
6      "they love mary"
7  ]
8
9  negative_texts =  [
10     "we hate you",
11     "they hate us",
12     "you are bad",
13     "he is bad",
14     "we hate mary"
15 ]
16
17 test_texts = [
18     "they love mary",
```

```
    "they are good",
    "why do you hate mary",
    "they are almost always good",
    "we are very bad"
]
```

We've created three simple datasets of five sentences each. The first one contains positive sentences; the second one contains negative sentences; and the last contains a mix of positive and negative sentences.

It's immediately obvious to a human which sentences are positive and which are negative, but can we teach a computer to tell them apart?

We'll use the two lists `positive_texts` and `negative_texts` to *train* our model. That is, we'll show these examples to the computer along with the correct answers for the question "is this text positive or negative?". The computer will try to find rules to tell the difference, and then we'll test how well it did by giving it `test_texts` without the answers and ask it to guess whether each example is positive or negative.

Understanding vectorization

The first step in nearly all machine learning problems is to translate your data from a format that makes sense to a human to one that makes sense to a computer. In the case of language and text data, a simple but effective way to do this is to associate each unique word in the dataset with a number, from `0` onwards. Each text can then be represented by an array of numbers, representing how often each possible word appears in the text.

Let's go through an example to see how this works. If we had the two sentences

`["nice pizza is nice"], ["what is pizza"]`

then we would have a dataset with four unique words in it. The first then we'd want to do is create a vocabulary mapping to map each unique word to a unique number. We could do this as follows:

```
{
    "nice": 0,
    "pizza": 1,
    "is": 2,
    "what": 3
}
```

To create this, we simply go through both sentences from left to right, mapping each new word to the next available number and skipping words that we've seen before. Now we can again convert our sentences into bag of words vectors as follows

```
[
    [2, 1, 1, 0], # two "nice", one "pizza", one "is", zero "what"
    [0, 1, 1, 1]  # zero "nice", one "pizza", one "is", one "what"
]
```

Each sentence vector is always the same length as the *total* vocabulary size. We have four words in total (across all of our sentences), so each sentence is represented by an array of length four. Each position in the array represents a word, and each value represents how often that word appears in that sentence.

The first sentence contains the word "nice" twice, while the second sentence does not contain the word "nice" at all. According to our mapping, the zeroth element of each array should indicate how often the word nice appears, so the first sentence contains a `2` in the beginning and the second sentence contains a `0` there.

This representation is called "bag of words" because we lose all of the information represented by the *order* of words. We don't know, for example, that the first sentence starts and ends with "nice", only that it contains the word "nice" twice.

With real data, these arrays get *very* long. There are millions of words in most languages, so for a big dataset containing most words, each sentence needs to be represented by a very long array, where nearly all values are set to zero (all the words not in that sentence). This could take up a lot of space, but luckily scikit-learn uses a clever sparse-matrix implementation that doesn't quite look like the above, but the overall concept remains the same.

Let's see how to achieve the above using scikit-learn's optimised vectoriser.

First we want to combine all of our "training" data (the data that we'll show the computer along with the correct labels of "positive" or "negative" so that it can learn), so we'll combine our positive and negative texts into one array. Add the following code below the datasets you created.

```
training_texts = negative_texts + positive_texts
training_labels = ["negative"] * len(negative_texts) + ["positive"] * len(positive_t\
exts)
```

Our dataset now looks like this:

```
['we hate you', 'they hate us', 'you are bad', 'he is bad', 'we hate mary', 'we love\
 you', 'they love us', 'you are good', 'he is good', 'they love mary']
['negative', 'negative', 'negative', 'negative', 'negative', 'positive', 'positive',\
 'positive', 'positive', 'positive']
```

The two arrays (texts and labels) are associated by index. The first text in the first array is negative, and corresponds to the first label in the second array, and so on.

Now we need a vectoriser to transform the texts into numbers. We can create one in scikit-learn with

```
vectorizer = CountVectorizer()
```

Before we can use our vectorizer, it needs to run once through all the data we have so it can build the mapping from words to indices. This is referred to as "fitting" the vectoriser, and we can do it like this:

```
vectorizer.fit(training_texts)
```

If we want, we can see the mapping it created (which might not be in order, as in the examples we walked through earlier, but each word will have its own index). We can inspect the vectoriser's vocabulary by adding the line

```
print(vectorizer.vocabulary_)
```

(Note the underscore at the end. Scikit-learn uses this as a convention for "helper" attributes. The mapping is explicit only for debugging purposes and you shouldn't need to use it in most cases). My vocabulary mapping looked as follows:

```
{'we': 10, 'hate': 3, 'you': 11, 'they': 8, 'us': 9, 'are': 0, 'bad': 1, 'he': 4, 'i\
s': 5, 'mary': 7, 'love':6, 'good': 2}
```

Behind the scenes, the vectoriser inspected all of our texts, did same basic preprocessing like making everything lowercase, split the text into words using a built-in *tokenization* method, and produced a vocabulary mapping specific to our dataset.

Now that we have a vectorizer that knows what words are in our dataset, we can use it to transform our texts into vectors. Add the following lines of code to your Repl:

```
training_vectors = vectorizer.transform(training_texts)
testing_vectors = vectorizer.transform(test_texts)
```

The first line creates a list of vectors which represent all of the training texts, still in the same order, but now each text is a vector of numbers instead of a string.

The second line does the same with the test vectors. The machine learning part isn't looking at our test texts (that would be cheating) – it's just mapping the words to numbers so that it can work with them more easily. Note that when we called `fit()` on the vectoriser, we only showed it the training texts. Because there are words in the test texts that don't appear in the training texts, these words will simply be ignored and will not be represented in `testing_vectors`.

Now that we have a vectorised representation our problem, let's take a look at how we can solve it.

Understanding classification

A classifier is a statistical model that tries to predict a label for a given input. In our case, the input is the text and the output is either "positive" or "negative", depending on whether the classifier thinks that the input is positive or negative.

A machine learning classifier can be "trained". We give it labelled data and it tries to learn rules based on that data. Every time it gets more data, it updates its rules slightly to account for the new information. There are many kinds of classifiers, but one of the simplest is called Decision Tree.

Decision trees learn a set of yes/no rules by building decisions into a tree structure. Each new input moves down the tree, while various questions are asked one by one. When the input filters all the way to a leaf node in the tree, it acquires a label.

If that's confusing, don't worry! We'll walk through a detailed example with a picture soon to clarify. First, let's show how to get some results using Python.

```
classifier = tree.DecisionTreeClassifier()
classifier.fit(training_vectors, training_labels)
predictions = classifier.predict(testing_vectors)
print(predictions)
```

Similarly to the vectoriser, we first create a classifier by using the module we imported at the start. Then we call `fit()` on the classifier and pass in our training vectors and their associated labels. The decision tree is going to look at both and attempt to learn rules that separate the two kinds of data.

Once our classifier is trained, we can call the `predict()` method and pass in previously unseen data. Here we pass in `testing_vectors` which is the list of vectorized test data that the computer didn't look at during training. It has to try and apply the rules it learned from the training data to this new "unseen" data. Machine learning is pretty cool, but it's not magic, so there's no guarantee that the rules we learned will be any good yet.

The code above produces the following output:

```
['positive' 'positive' 'negative' 'positive' 'negative']
```

Let's take a look at our test texts again to see if these predictions match up to reality.

```
"they love mary"
"they are good"
"why do you hate mary"
"they are almost always good"
"we are very bad"
```

The output maps to the input by index, so the first output label ("positive") matches up to the first input text ("they love mary"), and the last output label ("negative") matches up to the last input text ("we are very bad").

It looks like the computer got every example right! It's not a difficult problem to solve. The words "bad" and "hate" appear only in the negative texts and the words "good" and "love", only in the positive ones. Other words like "they", "mary", "you" and "we" appear in both good and bad texts. If our model did well, it will have learned to ignore the words that appear in both kinds of texts, and focus on "good", "bad", "love" and "hate".

Decision Trees are not the most powerful machine learning model, but they have one advantage over most other algorithms: after we have trained them, we can look inside and see exactly how they work. More advanced models like deep neural networks are nearly impossible to make sense of after training.

Scikit-learn contains a useful "graphviz" helper to inspect tree-based models. Add the following code to the end of your repl.

```
tree.export_graphviz(
    classifier,
    out_file='tree.dot',
    feature_names=vectorizer.get_feature_names(),
)
```

This will create an export of the trained model which we can visualise. Look for the new `tree.dot` file in the left-most pane that should have been created after running the above code.

Copy the contents of this file (shown in the middle pane above) to your clipboard and navigate to http://www.webgraphviz.com/. Paste the Tree representation into the big input box on the page you see and press "Generate Graph"

You should see a tree graph that looks as follows.

The above shows a decision tree that only learned two rules. The first rule (top square) is about the word "hate". The rule is "is the number of times 'hate' occurs in this sentence less than or equal to 0.5". None of our sentences contain duplicate words, so each rule will really be only about whether the word appears or not (you can think of the `<= 0.5` rules as `< 1` in this case).

For each question in our training dataset, we can ask if the first rule is True or False. If the rule is True for a given sentence, we'll move that sentence down the tree left (following the "True" arrow). If not, we'll go right (following the "False" arrow).

Once we've asked this first question for each sentence in our dataset, we'll have three sentences for which the answer is "False", because three of our training sentences contain the word "hate". These three sentences go right in the decision tree and end up at first leaf node (an end node with no arrows coming out the bottom). This leaf node has `value = [3, 0]` in it, which means that three samples reach this node, and three belong to the negative class and zero to the positive class.

For each sentence where the first rule is "True" (the word "hate" appears less than 0.5 times, or in our case 0 times), we go down the left of the tree, to the node where `value = [2,5]`. This isn't a leaf node (it has more arrows coming out the bottom), so we're not done yet. At this point we have two negative sentences and all five positive sentences still.

The next rule is "bad <= 0.5". In the same way as before, we'll go down the right path if we have more than 0.5 occurrences of "bad" and left if we have fewer than 0.5 occurrences of "bad". For the last two negative sentences that we are still evaluating (the two containing "bad"), we'll go *right* and end up at the node with `value=[2,0]`. This is another leaf node and when we get here we have two negative sentences and zero positive ones.

All other data will go left, and we'll end up at `[0,5]`, or zero negative sentences and five positive ones.

As an exercise, take each of the test sentences (not represented in the annotated tree above) and try to follow the set of rules for each one. If it ends up in a bucket with more negative sentences than positive ones (either of the right branches), it'll be predicted as a negative sentence. If it ends up in the left-most leaf node, it'll be predicted as a positive sentence.

Building a manual classifier

When the task at hand is this simple, it's often easier to write a couple of rules manually rather than using Machine Learning. For this dataset, we could have achieved the same result by writing the following code.

```
def manual_classify(text):
    if "hate" in text:
        return "negative"
    if "bad" in text:
        return "negative"
    return "positive"

predictions = []
for text in test_texts:
    prediction = manual_classify(text)
    predictions.append(prediction)
print(predictions)
```

Here we have replicated the decision tree above. For each sentence, we check if it contains "hate" and if it does we classify it as negative. If it doesn't, we check if it contains "bad", and if it does, we classify it as negative. All other sentences are classified as positive.

So what's the difference between machine learning and traditional rule-based models like this one? The main advantage of learning the rules directly is that it doesn't really get more difficult as the

dataset grows. Our dataset was trivially simple, but a real-world dataset might need thousands or millions of rules, and while we could write a more complicated set of if-statements "by hand", it's much easier if we can teach machines to learn these by themselves.

Also, once we've perfected a set of manual rules, they'll still only work for a single dataset. But once we've perfected a machine learning model, we can use it for many tasks, simply by changing the input data!

In the example we walked through, our model was a perfect student and learned to correctly classify all five unseen sentences, this is not usually the case for real-world settings. Because machine learning models are based on probability, the goal is to make them as accurate as possible, but in general you will not get 100% accuracy. Depending on the problem, you might be able to get higher accuracy by hand-crafting rules yourself, so machine learning definitely isn't the correct tool to solve all classification problems.

Try the code on bigger datasets to see how it performs. There is no shortage of interesting data sets to experiment with. I wrote another machine learning walkthrough here[121] that shows how to use a larger clickbait dataset to teach a machine how to classify between clickbait articles and real ones, using similar methods to those described above. That tutorial uses an SVM classifier and a more advanced vectorisation method, but see if you can load the dataset from there into the classifier we built in this tutorial and compare the results.

You can fork this repl here: https://repl.it/@GarethDwyer1/machine-learning-intro[122] to keep hacking on it (it's the same code as we walked through above but with some comments added.) If you prefer, the entire program is shown below so you can copy paste it and work from there.

If you want to try your hand at machine learning, join the repl.it AI competition[123] that is running until 11 February. You can get general help by joining the Repl discord channel[124], or if you have questions specifically about this tutorial feel free to leave a comment below or to tweet at me[125].

```
from sklearn import tree
from sklearn.feature_extraction.text import CountVectorizer

positive_texts = [
    "we love you",
    "they love us",
    "you are good",
    "he is good",
    "they love mary"
]

negative_texts =  [
```

[121] https://www.codementor.io/garethdwyer/introduction-to-machine-learning-with-python-s-scikit-learn-czha398p1
[122] https://repl.it/@GarethDwyer1/machine-learning-intro
[123] https://repl.it/talk/challenge/AI/10058
[124] https://discord.gg/QWFfGhy
[125] https://twitter.com/sixhobbits

```
    "we hate you",
    "they hate us",
    "you are bad",
    "he is bad",
    "we hate mary"
]

test_texts = [
    "they love mary",
    "they are good",
    "why do you hate mary",
    "they are almost always good",
    "we are very bad"
]

training_texts = negative_texts + positive_texts
training_labels = ["negative"] * len(negative_texts) + ["positive"] * len(positive_t\
exts)

vectorizer = CountVectorizer()
vectorizer.fit(training_texts)
print(vectorizer.vocabulary_)

training_vectors = vectorizer.transform(training_texts)
testing_vectors = vectorizer.transform(test_texts)

classifier = tree.DecisionTreeClassifier()
classifier.fit(training_vectors, training_labels)

print(classifier.predict(testing_vectors))

tree.export_graphviz(
    classifier,
    out_file='tree.dot',
    feature_names=vectorizer.get_feature_names(),
)

def manual_classify(text):
    if "hate" in text:
        return "negative"
    if "bad" in text:
        return "negative"
    return "positive"
```

```
56
57 predictions = []
58 for text in test_texts:
59     prediction = manual_classify(text)
60     predictions.append(prediction)
61 print(predictions)
```

Quicksort tutorial: Python implementation with line by line explanation

In this tutorial, we'll be going over the Quicksort[126] algorithm with a line-by-line explanation. We're going to assume that you already know at least something about sorting algorithms[127], and have been introduced to the idea of Quicksort, but understandably you find it a bit confusing and are trying to better understand how it works.

We're also going to assume that you've covered some more fundamental computer science concepts, especially recursion[128], which Quicksort relies on.

To recap, Quicksort is one of the most efficient and most commonly used algorithms to sort a list of numbers. Unlike its competitor Mergesort, Quicksort can sort a list in place, saving the need to create a copy of the list, and therefore saving on memory requirements.

The main intuition behind Quicksort is that if we can efficiently *partition* a list, then we can efficiently sort it. Partitioning a list means that we pick a *pivot* item in the list, and then modify the list to move all items larger than the pivot to the right and all smaller items to the left.

Once the pivot is done, we can do the same operation to the left and right sections of the list recursively until the list is sorted.

Here's a Python implementation of Quicksort. Have a read through it and see if it makes sense. If not, read on below!

```
1  def partition(xs, start, end):
2      follower = leader = start
3      while leader < end:
4          if xs[leader] <= xs[end]:
5              xs[follower], xs[leader] = xs[leader], xs[follower]
6              follower += 1
7          leader += 1
8      xs[follower], xs[end] = xs[end], xs[follower]
9      return follower
10
11 def _quicksort(xs, start, end):
```

[126] https://en.wikipedia.org/wiki/Quicksort
[127] https://en.wikipedia.org/wiki/Sorting_algorithm
[128] https://en.wikipedia.org/wiki/Recursion#In_computer_science

```
12      if start >= end:
13          return
14      p = partition(xs, start, end)
15      _quicksort(xs, start, p-1)
16      _quicksort(xs, p+1, end)
17
18  def quicksort(xs):
19      _quicksort(xs, 0, len(xs)-1)
```

The Partition algorithm

The idea behind partition algorithm seems really intuitive, but the actual algorithm to do it efficiently is pretty counter-intuitive.

Let's start with the easy part – the idea. We have a list of numbers that isn't sorted. We pick a point in this list, and make sure that all larger numbers are to the *right* of that point and all the smaller numbers are to the *left*. For example, given the random list:

```
1  xs = [8, 4, 2, 2, 1, 7, 10, 5]
```

We could pick the *last* element (5) as the pivot point. We would want the list (after partitioning) to look as follows:

```
1  xs = [4, 2, 2, 1, 5, 7, 10, 8]
```

Note that this list isn't sorted, but it has some interesting properties. Our pivot element, `5`, is in the correct place (if we sort the list completely, this element won't move). Also, all the numbers to the left are smaller than `5`and all the numbers to the right are greater.

Because `5` is the in the correct place, we can ignore it after the partition algorithm (we won't need to move it again). This means that if we can sort the two smaller sublists to the left and right of `5`() `[4, 2, 2, 1]` and `[7, 10, 8]`) then the entire list will be sorted. Any time we can efficiently break a problem into smaller sub-problems, we should think of *recursion* as a tool to solve our main problem. Using recursion, we often don't even have to think about the entire solution. Instead, we define a base case (a list of length `0` or `1` is always sorted), and a way to divide a larger problem into smaller ones (e.g. partitioning a list in two), and almost by magic the problem solves itself!

But we're getting ahead of ourselves a bit. Let's take a look at how to actually implement the partition algorithm on its own, and then we can come back to using it to implement a sorting algorithm.

A bad partition implementation

You could probably easily write your own `partition` algorithm that gets the correct results without referring to any textbook implementations or thinking about it too much. For example:

```
def bad_partition(xs):
    smaller = []
    larger = []
    pivot = xs.pop()
    for x in xs:
        if x >= pivot:
            larger.append(x)
        else:
            smaller.append(x)
    return smaller + [pivot] + larger
```

In this implementation, we set up two temporary lists (`smaller` and `larger`). We then take the `pivot` element as the last element of the list (`pop` takes the last element and removes it from the original `xs` list).

We then consider each element `x` in the list `xs`. The ones that are smaller than are partition we store in the `smaller` temporary list, and the others go to the `larger` temporary list. Finally, we combine the two lists with the pivot item in the middle, and we have partitioned our list.

This is much easier to read than the implementation at the start of this post, so why don't we do it like this?

The primary advantage of Quicksort is that it is an *in place* sorting algorithm. Although for the toy examples we're looking at, it might not seem like much of an issue to create a few copies of our list, if you're trying to sort terrabytes of data, or if you are trying to sort any amount of data on a very limited computer (e.g a smartwatch), then you don't want to needlessly copy arrays around.

In Computer Science terms, this algorithm has a space-complexity of `O(2n)`, where `n` is the number of elements in our `xs` array. If we consider our example above of `xs = [8, 4, 2, 2, 1, 7, 10, 5]`, we'll need to store all 8 elements in the original `xs` array as well as three elements (`[7, 10, 8]`] in the `larger` array and four elements (`[4, 2, 2, 1]`) in the `smaller` array. This is a waste of space! With some clever tricks, we can do a series of swap operations on the original array and not need to make any copies at all.

Overview of the actual partition implementation

Let's pull out a few key parts of the good `partition` function that might be especially confusing before getting into the detailed explanation. Here it is again for reference.

```
1  def partition(xs, start, end):
2      follower = leader = start
3      while leader < end:
4          if xs[leader] <= xs[end]:
5              xs[follower], xs[leader] = xs[leader], xs[follower]
6              follower += 1
7          leader += 1
8      xs[follower], xs[end] = xs[end], xs[follower]
9      return follower
```

In our good `partition` function, you can see that we do some swap operations (lines 5 and 8) on the `xs` that is passed in, but we never allocate any new memory. This means that the storage remains constant to the size of `xs`, or `O(n)` in Computer Science terms. That is, this algorithm has *half* the space requirement of the "bad" implementation above, and should therefore allow us to sort lists that are twice the size using the same amount of memory.

The confusing part of this implementation is that although everything is based around our pivot element (the last item of the list in our case), and although the pivot element ends up somewhere in the middle of the list at the end, we don't actually touch the pivot element *until the very last swap*.

Instead, we have two other counters (`follower` and `leader`) which move around the smaller and bigger numbers in a clever way and implicitly keep track of where the pivot element should end up. We then switch the pivot element into the correct place at the end of the loop (line 8).

The `leader` is just a loop counter. Every iteration it increments by one until it gets to the pivot element (the end of the list). The follower is more subtle, and it keeps count of the number of swap iterations we do, moving up the list more slowly than the leader, tracking where our pivot element should eventually end up.

The other confusing part of this algorithm is on line 4. We move through the list from left to right. All numbers are currently to the *left* of the pivot but we eventually want the "big" items to end up on the *right*.

Intuitively then you would expect us to do the swapping action when we find an item that is *larger* than the pivot, but in fact, we do the opposite. When we find items that are *smaller* than the pivot, we swap the leader and the follower.

You can think of this as pushing the small items further to the left. Because the leader is always ahead of the follower, when we do a swap, we are swapping a small element with one further left in the list. The follower only looks at "big" items (ones that the leader has passed over without action), so when we do the swap, we're swapping a small item (leader) with a big one (follower), meaning that small items will move towards the left and large ones towards the right.

Line by line examination of partition

We define `partition` with three arguments, `xs` which is the list we want to sort, `start` which is the index of the first element to consider and `end` which is the index of the last element to consider.

We need to define the `start` and `end` arguments because we won't always be partitioning the entire list. As we work through the sorting algorithm later, we are going to be working on smaller and smaller sublists, but because we don't want to create new copies of the list, we'll be defining these sublists by using indexes to the original list.

In line 2, we start off both of our pointers – `follower`, and `leader` – to be the same as the beginning of the segment of the list that we're interested in. The leader is going to move faster than the follower, so we'll carry on looping until the leader falls off the end of the list segment (`while leader < end`).

We could take any element we want as a pivot element, but for simplicity, we'll just choose the last element. In line 4 then, we compare the `leader` element to the pivot. The leader is going to step through each and every item in our list segment, so this means that when we're done, we'll have compared the partition with every item in the list.

If the `leader` element is smaller or equal to the pivot element, we need to send it further to the left and bring a larger item (tracked by `follower`) further to the right. We do this in lines 4-5, where if we find a case where the `leader` is smaller or equal to the pivot, we swap it with the `follower`. At this point, the follower is pointing at a small item (the one that was `leader` a moment ago), so we increment `follower` by one in order to track the next item instead. This has a side effect of counting how many swaps we do, which incidentally tracks the exact place that our pivot element should eventually end up.

Whether or not we did a swap, we want to consider the next element in relation to our pivot, so in line 7 we increment `leader`.

Once we break out of the loop (line 8), we need to swap the pivot item (still on the end of the list) with the `follower` (which has moved up one for each element that was smaller than the pivot). If this is still confusing, look at our example again:

```
1  xs = [8, 4, 2, 2, 1, 7, 10, 5]
```

In `xs`, there are 4 items that are smaller than the pivot. Every time we find an item that is smaller than the pivot, we increment `follower` by one. This means that at the end of the loop, follower will have incremented 4 times and be pointing at index 4 in the original list. By inspection, you can see that this is the correct place for our pivot element (5).

The last thing we do is return the follower index, which now points to our pivot element in its *correct* place. We need to return this as it defines the two smaller sub-problems in our partitioned list - we now want to sort`xs[0:4]` (the first 4 items, which form an unsorted list) and the `xs[5:]` (the last 3 items, which form an unsorted list).

```
1  xs = [4, 2, 2, 1, 5, 7, 10, 8]
```

If you want another way to visualise exactly how this works, going over some examples by hand (that is, writing out a short randomly ordered list with a pen and paper, and writing out the new

list at each step of the algorithm) is very helpful. You can also watch this detailed YouTube video[129] where KC Ang demonstrates every step of the algorithm using paper cups in under 5 minutes!

The Quicksort function

Once we get the partition algorithm right, sorting is easy. We'll define a helper `_quicksort` function first to handle the recursion and then implement a prettier public function after.

```
1  def _quicksort(xs, start, end):
2      if start >= end:
3          return
4      p = partition(xs, start, end)
5      _quicksort(xs, start, p-1)
6      _quicksort(xs, p+1, end)
```

To sort a list, we partition it (line 4), sort the left sublist (line 5: from the start of the original list up to the pivot point), and then sort the right sublist (line 6: from just after the pivot point to the end of the original list). We do this recursively with the `end` boundary moving left, closer to `start`, for the left sublists and the `start` boundary moving right, closer to `end`, for the right sublists. When the start and end boundaries meet (line 2), we're done!

The first call to Quicksort will always be with the entire list that we want sorted, which means that `0` will be the start of the list and `len(xs)-1` will be the end of the list. We don't want to have to remember to pass these extra arguments in every time we call Quicksort from another program (e.g. in any case where it is not calling itself), so we'll make a prettier wrapper function with these defaults to get the process started.

```
1  def quicksort(xs):
2      return _quicksort(xs, 0, len(xs)-1)
```

Now we, as users of the sorting function, can call `quicksort([4,5,6,2,3,9,10,2,1,5,3,100,23,42,1])`, passing in only the list that we want sorted. This will in turn go and call the `_quicksort` function, which will keep calling itself until the list is sorted.

Testing our algorithm

We can write some basic driver code to take our newly implemented Quicksort out for a spin. The code below generates a random list of 100 000 numbers and sorts this list in around 5 seconds.

[129] https://www.youtube.com/watch?v=MZaf_9IZCrc

```
from datetime import datetime
import random

# create 100000 random numbers between 1 and 1000
xs = [random.randrange(1000) for _ in range(100000)]

# look at the first few and last few
print(xs[:10], xs[-10:])

# start the clock
t1 = datetime.now()
quicksort(xs)
t2 = datetime.now()
print("Sorted list of size {} in {}".format(len(xs), t2 - t1))

# have a look at the results
print(xs[:10], xs[-10:])
```

If you want to try this code out, visit my Repl at https://repl.it/@GarethDwyer1/quicksort[130]. You'll be able to run the code, see the results, and even fork it to continue developing or testing it on your own.

Also have a look at https://repl.it/@GarethDwyer1/sorting[131] where I show how Quicksort compares to some other common sorting algorithms.

If you need help, the folk over at the Repl discord server[132] are very friendly and keen to help people learn. Also feel free to drop a comment below, or to follow me on Twitter[133] and ask me questions there.

[130] https://repl.it/@GarethDwyer1/quicksort?language=python3
[131] https://repl.it/@GarethDwyer1/sorting
[132] https://discordapp.com/invite/QWFfGhy
[133] https://twitter.com/sixhobbits