

# Creating and deploying a language translation app using Python, Flask, GitHub, and Heroku

Programming in Python is fun, but as they say, *sharing is caring*. One of the most attractive options is to create a web application from your code so that people can simply access it online.

Say, you want to create a little language app like Duolingo to help you with your language learning skills. You've created a list already with some translations on it — now all you want to do is create a little app out of this. That's what we'll do in this post.

*This post is aimed at those who have some basic knowledge of Python, but have no experience using Flask, and want to learn the basics. I'm assuming you're working on a Unix-type system (Mac, Linux).*

## Introduction

[Flask](#) is a micro web framework. That means it provides you with the tools and libraries to build a web application. In this post, we'll look at a small example: creating a simple language translation web app, which we'll create using Python and Flask. We'll use the GitHub working environment in conjunction with [Heroku](#), a free cloud platform, to deploy our app.

We are going to create a super basic app that looks as follows. It provides a word or sentence to translate from a text file, allows the user to submit a translation, checks that translation, and scores the user if the answer is correct.

# Pardon my Spanish

Translate: maybe

Your score: 3

What we're gonna build

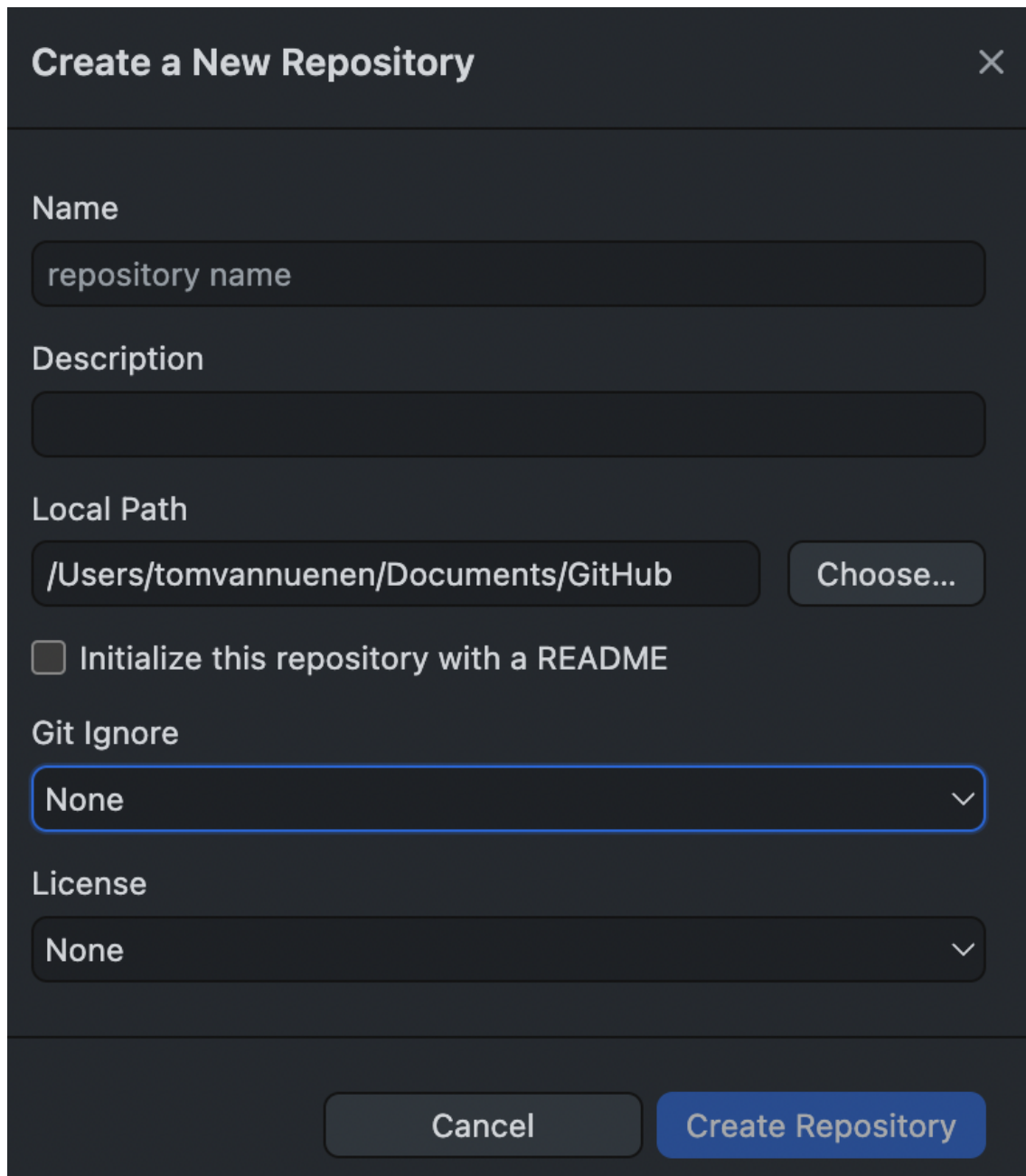
## Setting up a project

Arguably the easiest way to deploy a project like this is to create a repository on GitHub, which we'll later easily connect to Heroku. This GitHub repository is essentially a folder that will contain all of our files and dependencies. Read more [here](#) if you want to learn about GitHub.

If you haven't worked with GitHub or Git, I suggest installing [GitHub Desktop](#), as it gives you a nice user interface to create and modify repositories. If you want to use the example app I'm using in this post, download it from GitHub here: <https://github.com/tomvannuenen/pardon-my-spanish>.

You can clone this repository by clicking on the green Code button, and then download the repository as a ZIP file, or let GitHub Desktop download it for you (if you installed it).

As you clone or create the repository, GitHub Desktop allows you to create a `.gitignore` file, which essentially means it will exclude particular files when uploading and managing them on GitHub. In the dropdown menu, choose `Python` to avoid unnecessary Python files to be checked into Git. Give your repository a name, and let's start.

The image shows a dark-themed dialog box titled "Create a New Repository" with a close button (X) in the top right corner. The dialog contains several input fields and a checkbox. The "Name" field has the placeholder text "repository name". The "Description" field is empty. The "Local Path" field contains the text "/Users/tomvannuenen/Documents/GitHub" and a "Choose..." button to its right. Below this is a checkbox labeled "Initialize this repository with a README", which is currently unchecked. The "Git Ignore" field is a dropdown menu showing "None". The "License" field is also a dropdown menu showing "None". At the bottom of the dialog are two buttons: "Cancel" and "Create Repository".

Create a New Repository

Name

repository name

Description

Local Path

/Users/tomvannuenen/Documents/GitHub

Choose...

☐ Initialize this repository with a README

Git Ignore

None

License

None

Cancel

Create Repository

Creating a repository using GitHub Desktop

After clicking “Create Repository”, you will now have a folder on your machine in which we are going to build our app. Make sure to check the Local Path box above, which shows you where this folder is.

## Creating a virtual environment

Let's start from scratch. The first thing we must do is create a virtual environment in Python. This means we create an isolated environment for our Python project, in which we can specify which packages and dependencies we need without interfering with the global Python settings on our machine. This is also necessary if we want to deploy our web app later (as it needs all of these dependencies to be specified clearly).

We do this by going to Terminal, navigating to the folder we just created using GitHub Desktop, and typing the following two lines:

```
> python3 -m venv venv  
> source venv/bin/activate
```

If, after running this, you see the name of this new virtual environment before the command prompt, you have successfully activated it!

The first line will create the virtual environment in the folder we are in; the second will activate it. This means the work in Python we'll be doing from now on will be inside this virtual environment.

The next thing to do is install the dependencies our app needs. These are `flask` and `gunicorn`. The latter is a Python web server interface that we'll need to run our application. We'll use `pip`:

```
(venv) > pip install flask gunicorn
```

Okay, that's the setup. Let's get coding.

## File structure

Right now, your repository only has a `.git` folder and a `.gitattributes` file in it. We first need to create the file structure needed by Flask. First, we create a new directory we'll call `templates/`. Inside it, we will create a file called `index.html`. This is the file we'll be looking at when accessing our web application.

*(By the way, creating and editing files like these is easier using a source code editor, such as Atom or Brackets. Install either if you don't have a code editor yet.)*

Let's open this `index.html` file and write some basic HTML. If you don't know anything about HTML, check out the great introduction by [W3Schools](https://www.w3schools.com/html/) first. But really, we don't need much in this HTML file yet. Let's just start with the title.

```
<head>
  <title>Pardon my Spanish</title>
</head><body>
<div>
  <h1>Pardon my Spanish</h1>
  <h2>{{ q }}</h2>
</div>
</body>
```

If we now open `index.html` with a web browser, we'll see the following:

# Pardon my Spanish

{{ q }}

The one thing that may look new is this `q` in between double curly brackets. These brackets mean that Python code will be running in that block: `q` is a variable we'll connect to our Python script in a minute.

First, let's create some prettier fonts. We can use [Google Fonts](#) for this, a library of open source font families we can access. Just go to [Google Fonts](#), select a font, and find the button that says "Select this style". Clicking it will open up a window that has code you can copy/paste into your HTML to access that font.

I'm also going to use some simple CSS to style the page (e.g. putting the `div` in the center of the page). Here's

what `index.html` looks like now.

```
html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Pardon my Spanish</title>
    <script
      src="https://code.jquery.com/jquery-
1.12.4.min.js"
      integrity="sha384-
nvAa0+6Qg9clwYCGGPpDQLVpLNn0fRaROjHqs13t4Ggj3Ez50XnGQqc/r8Mhn
```

```

RDZ" crossorigin="anonymous"></script>
</head><link rel="preconnect"
href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com"
crossorigin>
<link
href="https://fonts.googleapis.com/css2?family=Raleway&displa
y=swap" rel="stylesheet"><style>
    * {
        font-family: "Raleway";
    }
    div {
        text-align: center;
        margin-top: 50px;
    }
    h1 {
        color: red;
        font-size: 40pt;
    }
    p {
        font-size: 24pt;
    }
</style><body>
<div>
    <h1>Pardon my Spanish</h1>
    <h2>{{ q }}</h2>
</div>
</body>

```

# Pardon my Spanish

{{ q }}

Okay, let's save and exit the file for now.

## A simple language translation script

The next thing we need is our Python script. We want to create a simple piece of code that can take in a list of translations, and



then use Flask to create an interactive scenario in which the user can try to translate a word or sentence.

Say you are learning Spanish and have already created a list with some chunks and phrases you're trying to learn, using the following notation:

```
mantener la cordura - staying sane  
engreído - conceited  
Sacando raíces/echando raíces - growing roots
```

Suppose these phrases live in a text file called `spanishdict.txt` on your machine somewhere. The first thing we need is a little code to pull them into Python, separate them, and pass them on for our usage in the app we're building.

So, in the main GitHub folder (so *not* `templates/` where our HTML file lives) we'll paste our `spanishdict.txt` file. We'll also use our code editor to create a new Python file. Simply create a new file and save it. Let's call it `app.py` to save it as Python file.

We want to create an app that serves the English text and lets the user write the Spanish translation. Our code will be quite simple: all we need is to pull in the text line by line and use `.split()` to split up each sentence based on some conditions. The `-` symbol will be used to separate the Spanish from the English translation, and if we encounter a `/` symbol, it means we have two translations. We save our variables in a new list each time, with the English phrase first and the Spanish translations after.

Because `spanishdict.txt` lives in the same folder as this Python file, we can open it in the way described above. We now have a very simple format: a list of all our translation phrases, consisting of lists with a length of 2 or more (one English phrase, and one or multiple Spanish translations).

Time to start using Flask!

## Using Flask

We need to add some import statements to our `app.py` code first:

```
from flask import Flask, render_template, request
from random import choice
import stringapp = Flask(__name__)
```

In the code above, we are importing `Flask` to allow us to instantiate our app, and `render_template` to connect our HTML to our application. We also import `choice` and `string` for reasons shown later. Finally, we create the Flask app and save it in a variable named `app` by doing `Flask(__name__)`.

Next, we'll create a little text file called `user_score.txt` that will keep track of our user's score, and we'll also create a function that we'll call `basic()` that takes in our previously created list.

Here's the first part of it:

```
loaded_q = []
with open('user_score.txt', 'w') as f:
    f.write(str(0))
def basic():
    with open('user_score.txt', 'r') as f:
        score = int(f.read())
    loaded_q.append(choice(spanish_list))
    # getting the current English phrase in the list
    eng = loaded_q[len(loaded_q)-1][0]
```

```
q = f"Translate: {eng}"  
return q
```

This function accesses our list, takes out a random item from it (using `choice`), and adds it to a new list with all the used phrases so far (for reasons we'll discuss later). We then access the English part of that new list by doing `loaded_q[len(loaded_q) - 1][0]`. Note that we're slicing our new list *using its own length* meaning we need to subtract one, as Python is zero-indexed. We then get the 0th index of that list, which is the English part of the translation.

Okay, that's nice, but how do we integrate this into Flask? Have a look at the following:

Looks almost the same, but with some notable differences!

First, we're creating a new `user_score.txt` file, which can save the score the user has (keep in mind that when we deploy our app, variables we want to remember need to be saved somehow). For now, we just set a `0` in there.

Next, the `@` symbol above our function is what's called a "decorator" used by Flask. A decorator essentially adds functionality to existing code. Here, it creates a so-called "route" to our HTML. The `/` symbol simply means Flask will be using the *main folder* of our app. If we would've added HTML files in other folders, we need to change the `/` part to that path, e.g. `/some_path`.

We also specify that this function is allowed to use so-called [HTTP methods](#), namely GET and POST. Essentially, this will allow us to both retrieve and send information to the webserver that will serve our app.

Note that when we `return` something from this Flask function, we are using `render_template`. What does this do? Well, it automatically looks for the `templates/` folder in the main folder of our app, and can then use variables we specify here in the HTML file in that folder. The `q=q` argument we set means that the variable `q` in our Python code will be equal to a variable, also called `q`, that we referred to in the HTML file.

On the bottom, we are running `app.run(debug=True)` which will give us a debug screen if our code is somehow wrong, so we can make adjustments (make sure to comment this out when deploying later!). Finally, we added:

```
if __name__ == "__main__":  
    app.run()
```

This starts the application web server when we call the Python script from the command line. If we want to run our code on a local server (i.e., our own machine), we need to navigate to the folder we're working in from our Terminal, and type:

```
(venv) > python app.py* Serving Flask app 'app' (lazy  
loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a  
production deployment.  
  Use a production WSGI server instead.  
* Debug mode: on  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)  
* Restarting with watchdog (fsevents)
```

```
* Debugger is active!  
* Debugger PIN: 979-828-401
```

As the output shows, we are running our web server on <http://127.0.0.1:5000> (or, synonymously, `localhost:5000`). If we now enter this into our web browser, here's what we'll see:

# Pardon my Spanish

Translate: I'm having a great time

Using the `<h2>{{ q }}</h2>` statement we created in our HTML, Flask is taking `q` from our Python script and putting it into the HTML.

## POSTing something

Finally, we need to add the interactive bit: the user must be able to enter some text, which can then be checked against the right answer. We'll also score the user if they did a good job.

First, let's add the following to our `index.html`:

```
<form action="/" method="POST">  
<input id="user_input" name="text">  
<input id="user_submit" type="submit">  
</form>  
<p>{{response}}</p>  
<p>Your score: {{score}}</p>
```

We are creating an [HTML form](#) with two fields: a text input and a submit button. We are also expecting a few more variables: a `response` from our script based on the user input, and a `score`.

Back to our Python code. We need to add the following `if`-statement to account for our user input. Have a look:

First, we use the `requests` method we imported earlier. If the user posts something in the textbox of our site (which, as you can see above, has `name="text"` , meaning we can access in Python using the same `text` denominator), we will do a few things with that input. First, we'll lowercase it and get rid of all the punctuation the user might have accidentally inputted using `string.punctuation` (remember we imported it earlier).

Then, we access the *previous entry* in the `loaded_q` list we made. We do this because when the page refreshes (and a new phrase is pulled in), we want to check whether the *previous entry* of the list matches the user input. If it does, we'll set a `response` variable (which we need to define first in our code, but more on that later) to "Good!".

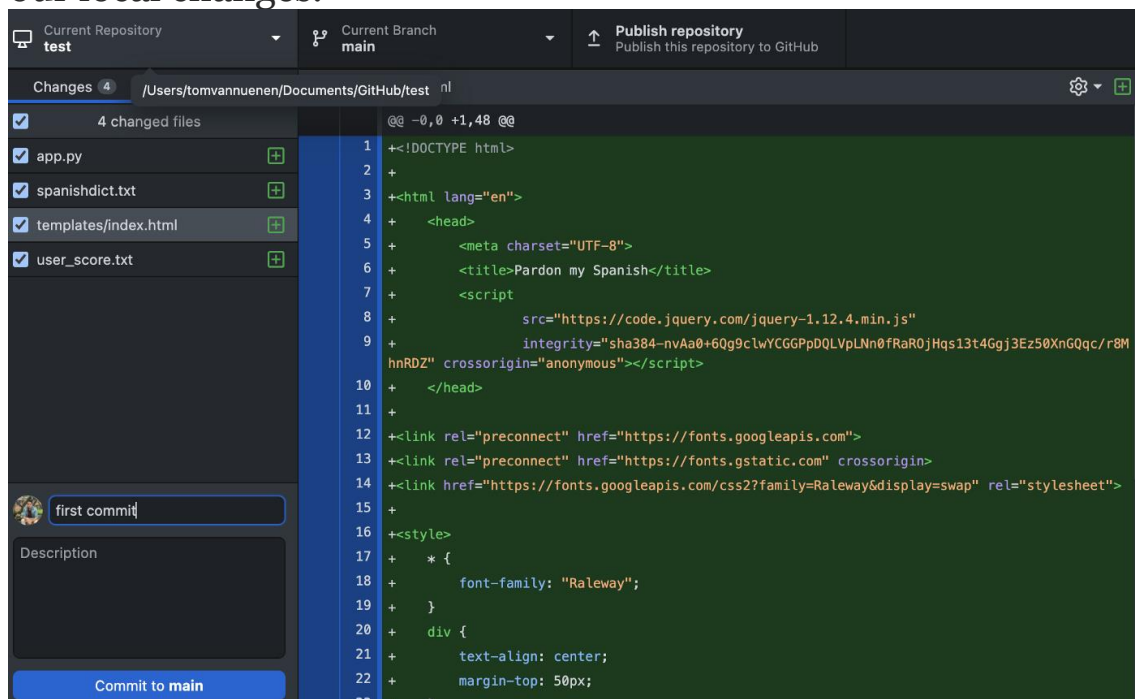
We also read our `user_score.txt` file, turn the string we find there into an integer, and, if the user's answered correctly, add one to the score and write it back to our file. If the user has answered incorrectly, we provide the right answer in `response` (*by the way, if you don't understand the `f` in front of the string there, check out *f-strings* in Python*).

## Putting it all together

Almost there! Let's see all the code we have now in our `index.html` and `app.py` files.

Note that in the Python file we are adding our new `response` and `score` variables to the `return` statement. Note also we are creating variables for `response` because the first time the website loads, those need to be returned as well. We also set our `good` variable, which determines the `response`, to `False`, only changing it to `True` if the user gave the right answer.

That's it! Make sure to *push* our web app to GitHub using GitHub Desktop: this means updating the remote branch with our local changes.



Press “Commit to main” to commit, then click “Publish repository” in the box that will pop up.

## Deploying to Heroku

We have our Flask app deployed locally — but what if we want to share it with others?

[Heroku](#) is a service allowing us to deploy our web app online. First, we need to create a free Heroku account. Once we've done so, from our Dashboard, we can click the button saying “Create New App”. Give your application a name and create it.

Make sure to comment out `app.run(debug=True)` in your Python code before continuing.

Before we can deploy our web app, we need **two more things**. The first is a `requirements.txt` file, which will consist of all the Python dependencies that the Heroku server needs to install in order to run our app. Happily, because we've been working from a virtual environment, we can access these dependencies really easily via the Terminal. Make sure you're still in the main folder of our app, and run:

```
(venv) > pip freeze > requirements.txt
```

This creates a `requirements.txt` file in the folder we're in, with all the dependencies we need. Opening it, it will look something like this:

```
certifi==2021.10.8
charset-normalizer==2.0.12
click==8.0.4
Flask==2.0.3
gunicorn==20.1.0
idna==3.3
itsdangerous==2.1.2
Jinja2==3.1.1
MarkupSafe==2.1.1
requests==2.27.1
```



```
urllib3==1.26.9
Werkzeug==2.0.3
```

These are all the dependencies we installed when running `pip install flask gunicorn` in our local environment.

The second thing we need is a so-called `Procfile` : it includes a list of instructions for Heroku on how to run our app. It will be a very small file including just one line of code, specifying the web server interface to use (`heroku` ), and the name of the app in the Python file we created (`app` ).

Create a new file in your code editor, and enter the following:

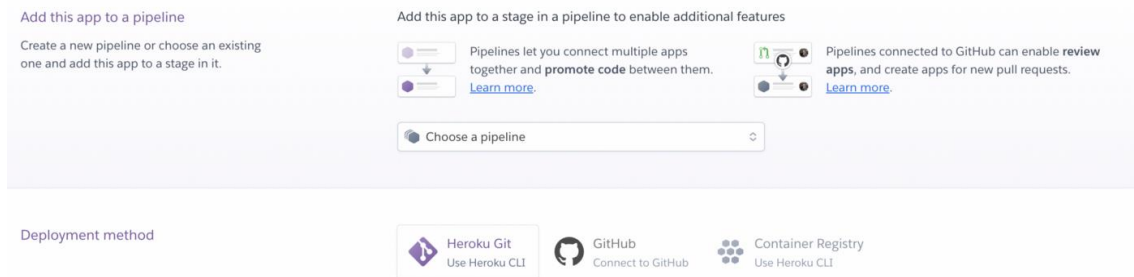
```
web: gunicorn app:app
```

Save it as `Procfile` (without an extension!), and Push the updates to GitHub.

This is what the final file structure looks like. Note how `index.html` is in the `templates` folder, and `Procfile` (without extension name) and `requirements.txt` are in the main folder. We also have our `spanishdict.txt` and `userscore.txt` files in the main folder.

```
/users/tomvannunen/Documents/GitHub/pardon-my-spanish
├── templates
│   └── index.html
├── .gitattributes
├── .gitignore
├── LICENSE
├── Procfile
├── README.me
├── app.py
├── requirements.txt
├── spanishdict.txt
└── user_score.txt
```

Next, from our Heroku dashboard, we can easily connect to GitHub by pressing the “GitHub” button next to “Deployment method”. We can then choose the repository we created and updated.



Next, you’ll be able to set “Automated deploy” — this will automatically re-deploy the application when there’s a new push to the GitHub repository. If you don’t want that, click “Manual deploy”. Deploying will take a few minutes. Check the output for errors, and if your app was deployed successfully, click on “View App” on the top right to see your application!

Keep in mind that if the app doesn’t receive traffic it will go in hibernation mode (meaning it will take a few seconds to spin up when you try to access it again). Also, Heroku servers are reset once per day, so that's when you’ll lose the settings to `user_score.txt` as well. If you want those to remain unaffected you should put them on a cloud storage system like [Amazon S3](#).