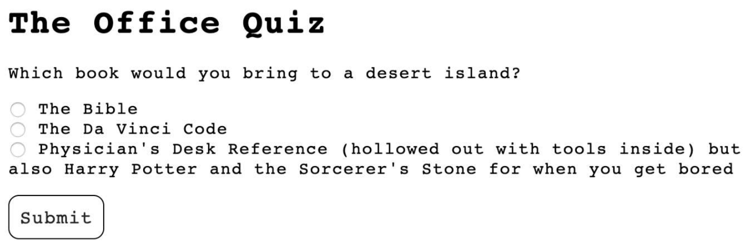
Build a Python Flask App on Glitch!

Tired of making text-based choose-your-own-adventure games in Python? Ready to take your Python programs to the next level by incorporating an HTML/CSS graphical user interface?

Flask is the perfect framework to build an HTML/CSS web app with a Python backend. In this tutorial, we're going to build a *The Office*-themed Flask application… on Glitch!

Yes, you heard me right — no frustrating downloads or setup, no banging your head against the wall. You can build this Flask app directly in your browser using Glitch!

We're going to make this app which asks for your desert island book choice…



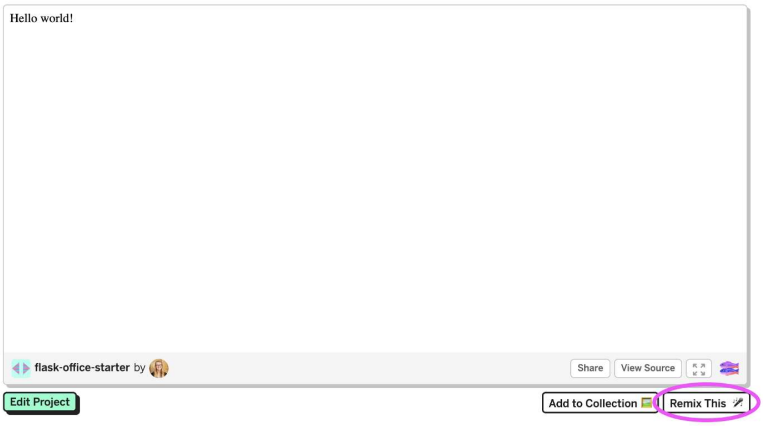
The Office Quiz

Which book would you bring to a desert island?

○ The Bible
○ The Da Vinci Code
○ Physician's Desk Reference (hollowed out with tools inside) but also Harry Potter and the Sorcerer's Stone for when you get bored

… and then tells you which *The Office* character you are based on your answer!

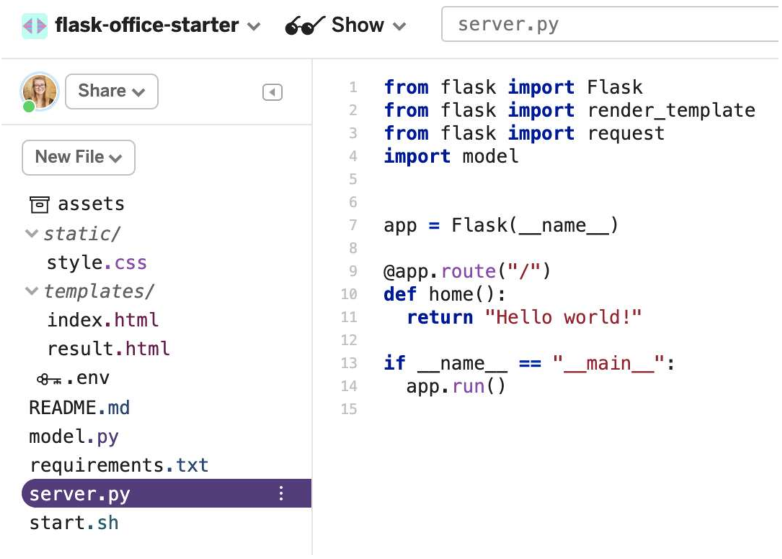
If you haven't already, head over to Glitch and make an account. Then remix the Flask Office starter code by pressing the microphone button at the bottom right hand corner of the Preview screen.


Hello world!

Note: this tutorial assumes familiarity with HTML/CSS and Python programming. If you're not familiar with these topics, check out some of my other tutorials or follow along by copy-pasting if you're undeterred!

Model–view–controller architecture, or why we have all these files

There are so many files, all prepared for you! Why so many?



```python
from flask import Flask
from flask import render_template
from flask import request
import model


app = Flask(__name__)

@app.route("/")
def home():
  return "Hello world!"

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

Why so many files in the file tree?

First, let's go over the basics of how most web apps are broken down.

- **Model**: This is the main Python backend logic of your app. All your classes, functions, etc. live here. Our model is currently contained in **model.py**, though in other places you may see a folder containing multiple Python classes and files.

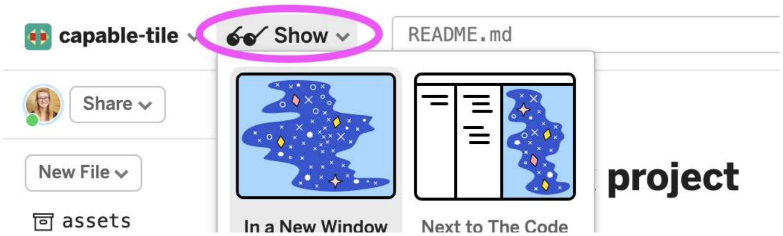
- **View**: This is our beautiful HTML/CSS interface — what the user sees. Our views are currently contained in the **templates** folder, which contains two empty HTML files: **index.html** and **result.html**. There is also the **static** folder, which contains assets such as our **style.css** file.

- **Controller**: The controller goes between the model and view, calling on the model for backend computations and delivering results to the view. Our controller lives in **server.py**.

To conceptualize this architecture, it can help to think of the web app as a restaurant. The model is the kitchen, where all the food is cooked. The view is the menu, which is presented to the guest. The waiter is the controller, running between the guest and the kitchen and delivering the food to the guest.

Previewing the home page

Right now, our home page is very boring. If we press "Show," we just see some very plain Times New Roman text saying, "Hello, world!"

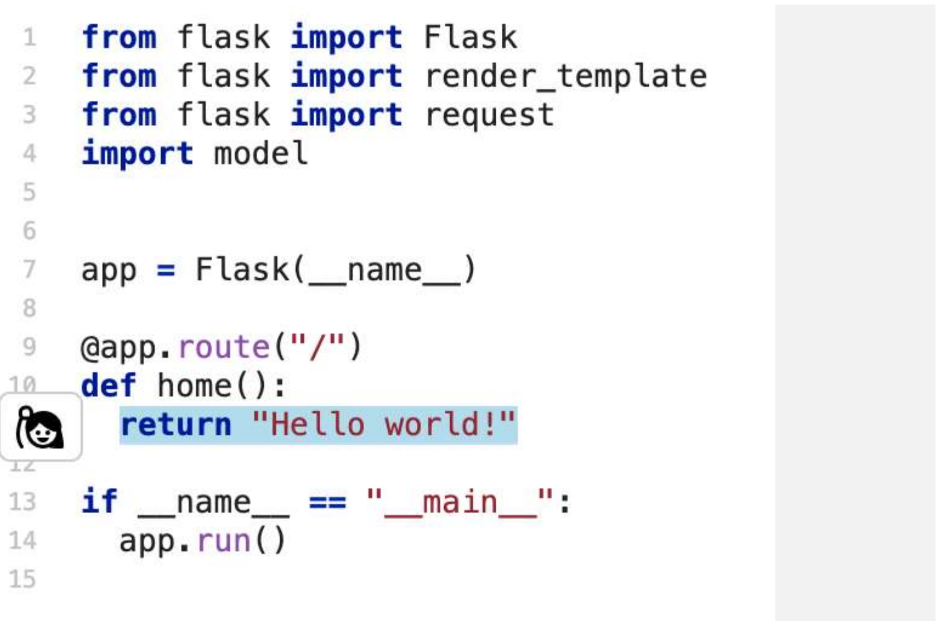


```
> static/
> templates/
  .env
README.md
model.py
```

```
Hello world!
```

BOOOO-RING

Where is this message even coming from? Pop open **server.py** and see if you can find the line.



```python
from flask import Flask
from flask import render_template
from flask import request
import model


app = Flask(__name__)

@app.route("/")
def home():
  return "Hello world!"

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

We're actually going to get fancy here and, instead of just returning a string, return an entire HTML page as our homepage.

You can see from the imports we're already imported `from flask import render_template`, which means we can use the `render_template`function in Flask to render HTML pages.

Instead of returning "Hello, world!" I'm going to have my Flask app serve up the **index.html** page from my **templates** folder.

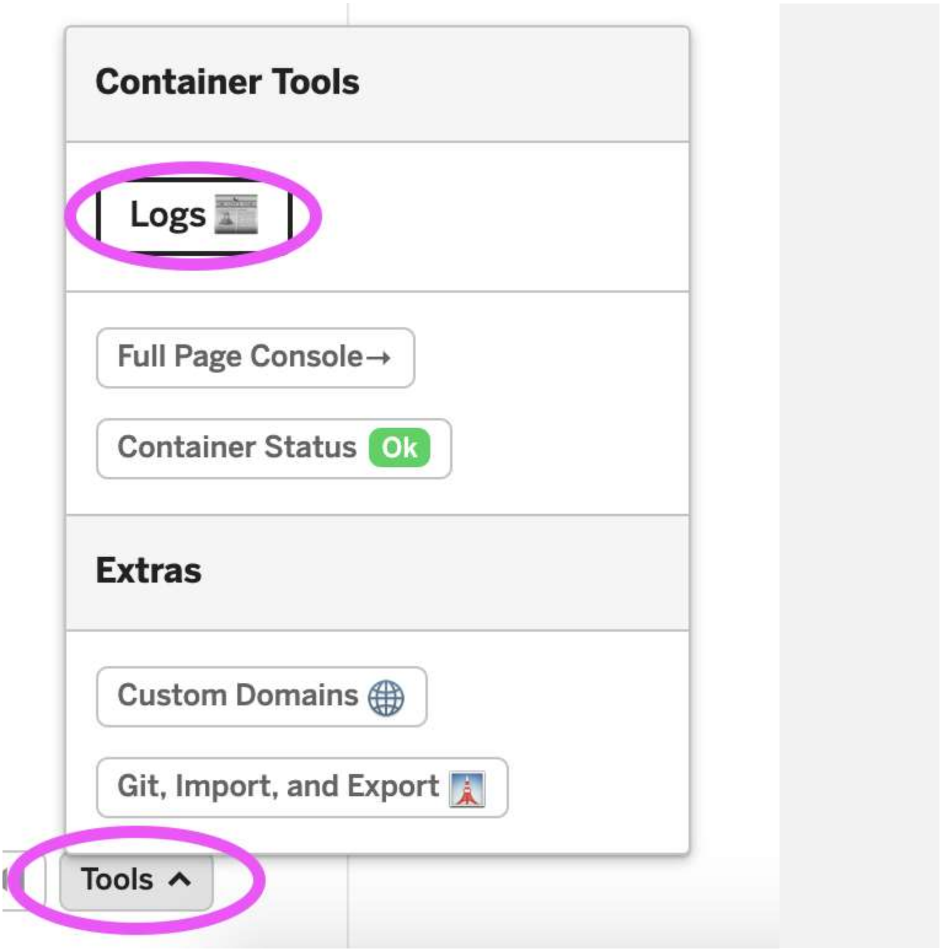
```
@app.route("/")
def home():
  return render_template("index.html")
```

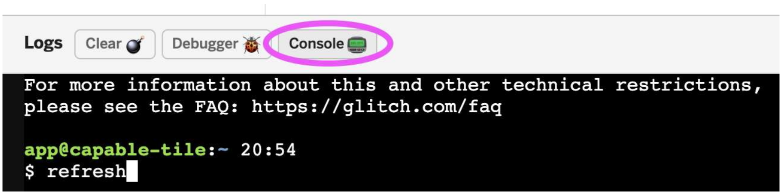
We're just about ready to see what our app does now… except for an **essential piece of information about coding Flask apps on Glitch!**

Using the console on Glitch

Whenever you make a change to your Flask app on Glitch, it can take a little while to take effect. However, you can force Glitch to update the app by opening the console and running the refresh command.

I usually open the console by clicking on “Tools” in the lower left hand corner and then selecting “Logs” at the top of the toolbar. Then I can navigate to “Console” as one of the options and run “refresh.”





```
For more information about this and other technical restrictions,
please see the FAQ: https://glitch.com/faq

app@capable-tile:~ 20:54
$ refresh
```

Type "refresh" and then hit return!

The console, incidentally, is also where you can run commands like `mkdir` and `touch` to make new directories and files.

Using a template on the home page

Okay, you ran "refresh" in the console and then previewed your app again. Was it a totally boring white page?

That's because, if you open **templates** and then **index.html**, we haven't actually coded any HTML in here!

Let's fix that, and add an HTML skeleton along with a header:

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Office Quiz</title>
    <link rel="stylesheet" type="text/css"
href="static/style.css">
  </head>
```

```
    <body>
      <h1>
        The Office Quiz
      </h1>
    </body>
  </html>
```

Now, after running “refresh,” you should see some text on your page!

The Office Quiz

Beautiful.

Making another route

We have the bare bones of our home page set up. We just need one other page, for the result the user gets when they find out what *The Office*character they are.

Let’s hop back to **server.py** and add this route.

```
@app.route("/result")
def result():
  return render_template("result.html")
```

The route can be accessed by adding /result after the app URL that appears when you hit “Show”— for example, flask-office-starter.glitch.me/result. In the code, we define a Python function that runs when the user reaches that page. Here, we’re returning a template again, but this time it’s the **result.html** file.

Let’s head over to **templates/result.html** and spruce that template up a bit! Add an HTML skeleton along with a header:

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Office Quiz</title>
    <link rel="stylesheet" type="text/css"
href="static/style.css">
  </head>
  <body>
    <h1>
      You're {{ character }}!
    </h1>
  </body>
</html>
```

The `{{ character }}` line is a placeholder that we'll come back to. Soon, this will print the character from the *The Office* that the user is!

Now, if you run "refresh" in the console, press "Show" on Glitch, and add "/result" to the URL, you should arrive at your result page!

You're !

Existential delight in the "/result" route

Notice that `{{ character }}` doesn't print anything yet. That will change soon!

Sending data in an HTML form

Now, we're going to give the user a little quiz that will carry them from the home page to the results page, where they can find out which character they are!

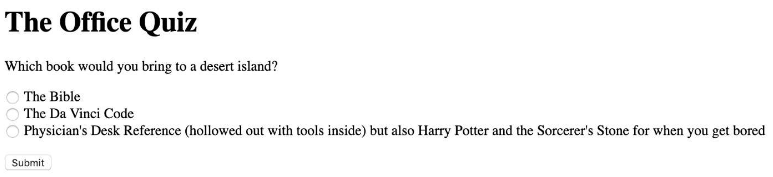
For simplicity, our quiz will just be one question: the desert island book question. We'll ask it using radio buttons in an HTML form, in the body of our **index.html** template:

```
<form method="post" action="/result">
      <p>
        Which book would you bring to a desert
island?
      </p>
      <input type="radio" name="book"
value="bible"> The Bible<br>
      <input type="radio" name="book"
value="davinci"> The Da Vinci Code<br>
      <input type="radio" name="book"
value="hp"> Physician's Desk Reference
(hollowed out with tools inside) but also
Harry Potter and the Sorcerer's Stone for when
you get bored
      <p>
        <input type="submit" />
      </p>
</form>
```

The form is using the POST method to send data from one webpage (index.html) to another (result.html). The form will be going through the "/result" route.

Notice that all the radio buttons have the name "book."

Now this form should appear on the homepage! (Remember, run "refresh" and then hit "Show.")


The Office Quiz

Which book would you bring to a desert island?

◯ The Bible
◯ The Da Vinci Code
◯ Physician's Desk Reference (hollowed out with tools inside) but also Harry Potter and the Sorcerer's Stone for when you get bored

Programming the model

All right, we're going to take a trip to the backend. When our user sends their data through the form, we're going to need to process it. Specifically, I need a function that takes in the book the user selected and returns the *The Office* character they are.

Head into **model.py** and write the following function:

```
def get_character(book):
  if book == "bible":
    return "Angela"
  elif book == "davinci":
    return "Phyllis"
  elif book == "hp":
    return "Dwight"
  else:
    return None
```

Notice that each book in Python is the same as the radio button values in the HTML.

Now, because I'm fancy, I'd also like a function that takes in a character from *The Office* and returns the URL to a funny gif of them. Add this function to **model.py**:

```
def get_gif(character):
  if character == "Angela":
    return
"https://media.giphy.com/media/Vew93C1vI2ze/giphy
  elif character == "Phyllis":
    return
"https://media.giphy.com/media/1oGnY1ZqXb123IDO4a
  elif character == "Dwight":
    return
"https://media.giphy.com/media/Cz1it5S65QGuA/giph
  else:
    return None
```

Connecting the model and view with the controller

Okay, we have our (kinda) beautiful form. We have our (definitely) beautiful functions. Now we need to connect them!

Head over to the heart of your controller, **server.py**.

You can see by the line `import model` that we've imported all the contents of model.py, and those functions are available to us now.

The line `from flask import request` means that we have some functionality for dealing with HTTP GET and POST requests now. (GET is when we're just asking for webpages, like the homepage, whereas POST will be used for sending data around, like for the result page.)

Because we're sending data to the "/result" route, we need to modify the header to include POST requests. Change the first line to incorporate functionality for both GET and POST requests:

```
@app.route("/result", methods=["GET", "POST"])
```

Now, we can make our function do different things depending if the user is getting or posting to the page. Modify your function to read:

```
def result():
  if request.method == "POST":
    return render_template("result.html")
  else:
    return "Sorry, there was an error."
```

It’s always good practice to catch errors (like sending a GET request when we need a POST request) with an else statement.

So this is great, but we’re still not interacting with or processing the user data.

Inside the if branch of your function, you can grab all the data sent by a form like so. Put this line inside the if branch, but before the return statement:

```
userdata = dict(request.form)
```

If you print the data, “refresh” on Glitch and run your app, you’ll see something like this in the **Logs**when you submit the form:

```
{'book': [u'bible']}
```

It’s a dictionary where “book” is one of the keys, and the value is a one-item list! You can drill down and isolate the exact book like so:

```
book = userdata["book"][0]
```

Now we can feed the `book` variable into our `get_character` function from the model, and find out what character the user is:

```
character = model.get_character(book)
```

And finally, because I'm fancy, I'm going to feed the `character` variable into the `get_gif` function and get a URL of the user's gif!

```
gif_url = model.get_gif(character)
```

Okay… but how do we print the user's character to the results page?!

Remember how in the h1 in results.html, we wrote the following placeholder:

```
<h1>You're {{ character }}!</h1>
```

Well, this placeholder was actually in Jinja, a templating language for Python. We can actually define what value `{{ character }}` should have when we render our template.

```
return render_template("result.html",
character=character)
```

If we hop over to **result.html**, we can even add in a placeholder for a gif:

```
<p>
  <img src="{{ gif_url }}" />
</p>
```

And we can send the gif_url value along with the character value when we render the template.

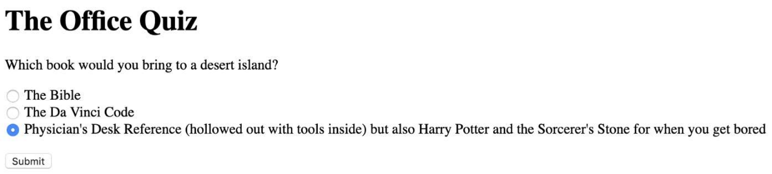
```
return render_template("result.html",
character=character, gif_url=gif_url)
```

Now, in **server.py**, your finished “/result” route should look like this:

```
@app.route("/result", methods=["GET", "POST"])
def result():
  if request.method == 'POST' and
len(dict(request.form)) > 0:
    userdata = dict(request.form)
    print(userdata)
    book = userdata["book"][0]
    character = model.get_character(book)
    gif_url = model.get_gif(character)
    return render_template("result.html",
character=character, gif_url=gif_url)
  else:
    return "Sorry, there was an error."
```

Can you find where I sneakily added in a check to make sure that the user actually filled out the form before sending it?!

Run “refresh,” press “Show,” and test out your quiz!


The Office Quiz

Which book would you bring to a desert island?

- ○ The Bible
- ○ The Da Vinci Code
- ● Physician's Desk Reference (hollowed out with tools inside) but also Harry Potter and the Sorcerer's Stone for when you get bored

Styling with CSS

Last thing: head over to **static/style.css** and add your styles. Here’s the simple stylesheet I used:

```
body {
  font-family: courier;
  margin: 2em;
  letter-spacing: 1px;
}

input[type="submit"] {
  border: 1px solid black;
  font-family: courier;
  padding: 10px;
  border-radius: 10px;
  font-size: 1.1em;
  letter-spacing: 1px;
}

input[type="submit"]:hover {
  background-color: #eee;
  cursor: pointer;
}
```

Run “refresh,” hit “Show,” and admire your work! I find that the CSS styles take a while to take effect, so don’t be afraid to clear your cache with a hard refresh.

Congratulations, you’ve built a Flask app on Glitch!