fox valley girls coding club

Python Flask Project in Glitch

Objectives:

- Understand Client and Server Communication on the Web
- Understand the difference between Client-side code and Server-side code
- Learn about **Python Flask Web Development** by creating your own Web site

We will create a Web site with code in Python, HTML, CSS, and JavaScript using Glitch, an on-line development tool. Before starting the coding project in Glitch, we will set up a code repository in GitHub, so we will have a place to store our code

Web Client (Browser) / Server Communication --> Request/Response

Communication Protocols:

- Standards are defined for how machines on the Web communicate with each other
- **HTTP**: HyperText Transfer Protocol
- **HTTPS:** Secure HTTP; uses certificates to validate and encryption to protect data

Browser (Client Software): Sends Requests and Presents HTML from Response

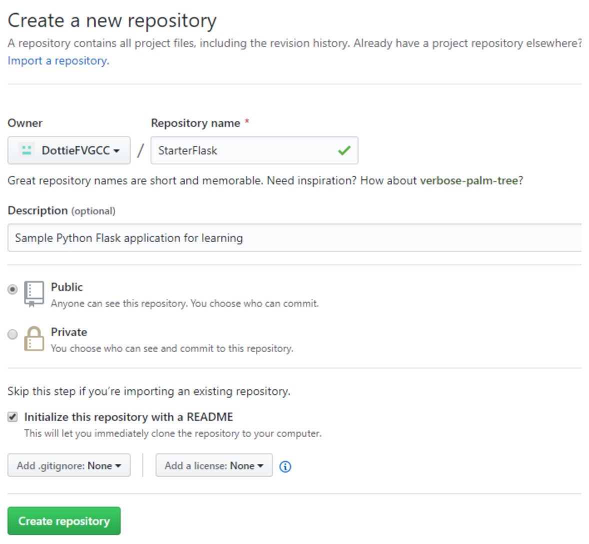
- Browser software is on computers and mobile devices
  Chrome, Safari, Firefox, Edge, Opera, IE, etc.
- Sends Request for a URL (Uniform Resource Locator) – expects to get HTML back
- Browsers all know how to read HTML
- Executes Client-Side code: JavaScript is widely used and understood by Browsers

Server: Receives Requests and Sends Response:

- **Runs Web Server software** (IIS, Apache, etc.) that "Serves" up the Web pages
- **Sends Response** as HTML to the Browser
- Hosts **Web application frameworks**: e.g., Django or Flask for Python
- Hosts and executes **Server-side code** modules/files
  - Code can be in Python, PHP, Java, C#, etc.
  - Image files
- May host complete HTML pages but all or part of the HTML Response could also be generated "on the fly" by the code

Set up GitHub Code Repository for our Python Flask Project

- Go to https://github.com and sign in or create a new account if you don't have one
- Go to the **Create a new repository page**: by clicking the + in the upper right and choosing New repository or by going to github.com/new
- Fill in the Repository name as "StarterFlask", write a brief description, check the box to initialize with a README, and click Create Repository


Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: DottieFVGCC / Repository name *: StarterFlask

Great repository names are short and memorable. Need inspiration? How about verbose-palm-tree?

Description (optional): Sample Python Flask application for learning

● Public — Anyone can see this repository. You choose who can commit.

○ Private — You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☑ Initialize this repository with a README — This will let you immediately clone the repository to your computer.

Stay logged into GitHub, and follow the steps below to create your Python Flask project in Glitch

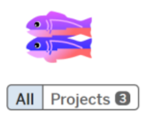
Create Python Flask Project in Glitch

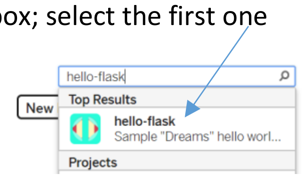
Log into Glitch

- While logged in to GitHub, go to Glitch.com
- Click the Sign In button --> Sign In with GitHub
- If this is the first time you sign in to Glitch with GitHub, you will be asked to authorize FogCreek (the original name of Glitch's development company) to access your GitHub account
  - When you click the Authorize button, you will receive an email from GitHub notifying you that a third-party OAuth application has been added to your account
  - You will also receive a welcoming email from Glitch with some ideas to get started

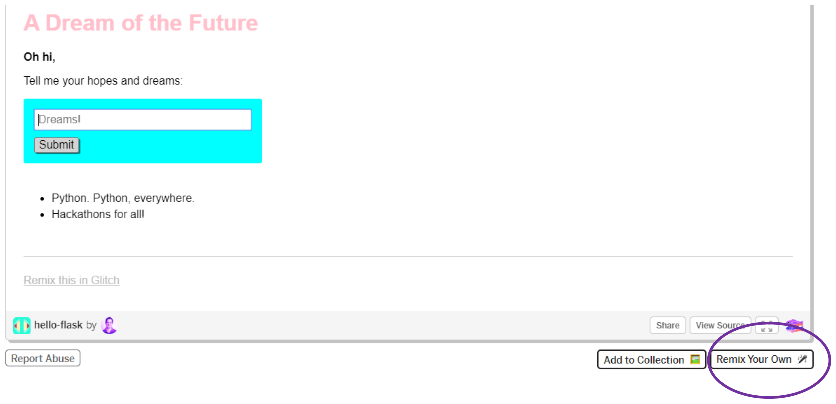
Create a Project based on a sample Flask/Python application:

- Find the hello-flask world project: type *hello-flask* into the search box; select the first one





- Try out the app, by typing a new "dream" in the box and clicking Submit
- Click the Remix your own button:



A Dream of the Future

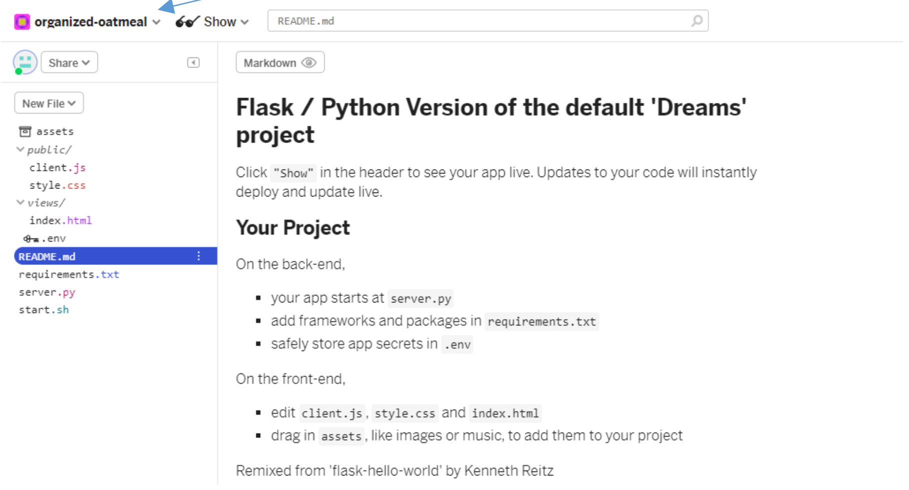
Oh hi,

Tell me your hopes and dreams:

- Python. Python, everywhere.
- Hackathons for all!

Remix this in Glitch

- The editor opens with your own copy of the project! Files are listed in the left column; the README.md file is selected and shown in the main editor window. A unique project name is generated by Glitch. In this example, the project is named *"organized-oatmeal"*. If you prefer a different name, click the down arrow next to the name and type a new one. Take time to read the README.md file as it explains the files in the project



Flask / Python Version of the default 'Dreams' project

Click `"Show"` in the header to see your app live. Updates to your code will instantly deploy and update live.

Your Project

On the back-end,

- your app starts at `server.py`
- add frameworks and packages in `requirements.txt`
- safely store app secrets in `.env`

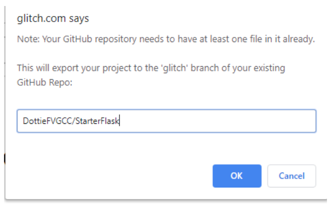
On the front-end,

- edit `client.js`, `style.css` and `index.html`
- drag in `assets`, like images or music, to add them to your project

Remixed from 'flask-hello-world' by Kenneth Reitz

- View your Web Site without even changing anything: Click the Show button next to the project name; try it out; add a dream by typing over the placeholder *Dreams!* Text
- Connect your project to GitHub:
  - If you haven't allowed Glitch to access your repositories previously:
    - At bottom of file list on left, select Tools --> Git, Import, and Export, then click the Connect GitHub button
    - Authorize Access to your GitHub Repositories: If this is the first time connecting, you will get a confirmation popup; click the Authorize FogCreek button
- Export your project to your GitHub Repo:
  - Open Tools --> Git, Import, Export: After you have granted access to your repositories, you will see the Export to GitHub button

  - **Load your current project to GitHub** by clicking Export to GitHub
  - **Change "user/repo"** to your GitHub user name/name of the repo you created; click OK



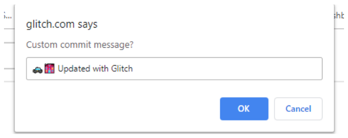
glitch.com says

Note: Your GitHub repository needs to have at least one file in it already.

This will export your project to the 'glitch' branch of your existing GitHub Repo:

DottieFVGCC/StarterFlask

- **Commit with a comment**: a Commit operation in GitHub updates GitHub with your code changes. It requires a comment. Glitch provides a default one that you can change



glitch.com says

Custom commit message?

Updated with Glitch

A Look at Python Flask

Python Flask is a Web Framework

- **Web Framework:** software that runs on the Web Server to provide services, resources, code libraries and API's to run Web applications. ASP.NET is a framework for running Web applications in C# and we are going to look at 2 frameworks for running Python server-side code
- **Python Web Frameworks**:
  - **Django** – a "full stack" framework, often referred to as "batteries included" approach. Includes lots of tools/libraries for administration, authentication, URL routing, database interface, etc.
  - **Flask** - a lightweight, extensible framework, for simple, single application Web sites.
  - Refer to this article for more info on Django vs Flask: https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v

Glitch Simplifies Python Flask Development

What if we wrote the Flask Application on our computer instead of Glitch?

If you were to write a Python Flask application on your computer "from Scratch" you would have to install a bunch of things starting with the Python language itself, and PIP the package manager for Python packages. You would then use PIP to install Flask.

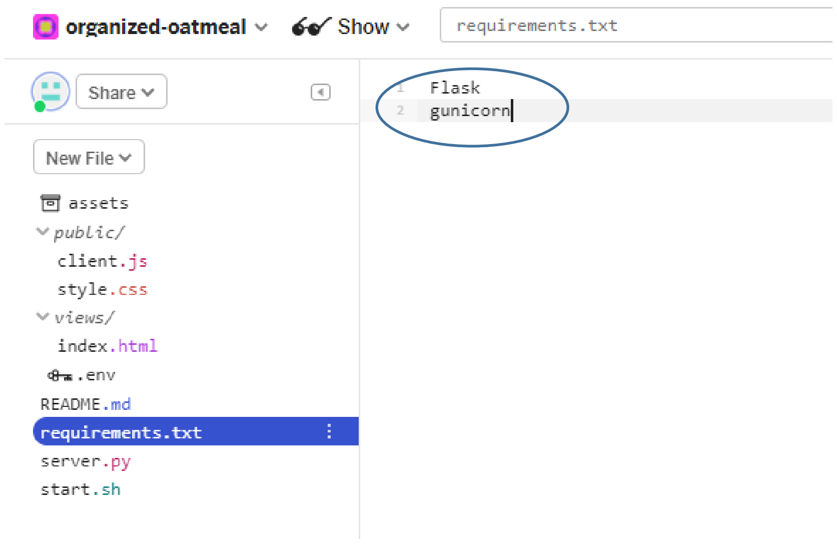
```
pip3 install flask –user
```

Once the software is installed you would create a python file, in which you'd import Flask and create an app that is a Flask instance. In the Python code, call the run method of the app to, hopefully, get your app running on your local host. There is a nice, simple example of this in the article linked above.

Glitch does all the behind the scenes setup for us!

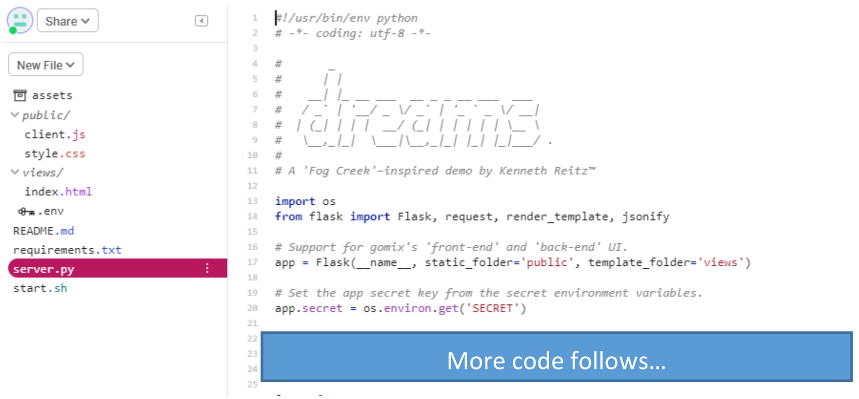
Let's look at what is included in the base project we each created:

Requirements.txt file takes care of telling Glitch what server software to install for our project; in this case, Flask, and a Python Server for UNIX named Green Unicorn, https://gunicorn.org/



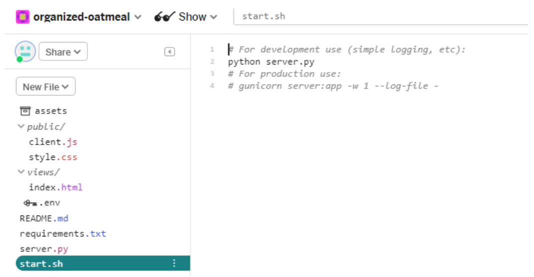
```
Flask
gunicorn
```

Server.py file is the Python code (file extension .py) that runs on the Server; it imports Flask and creates a variable named "app" that is the instance of Flask through which all the requests are processed.



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#
#      _
#     | |
#   __| |_ __ ___  __ _ _ __ ___  ___
#  / _` | '__/ _ \/ _` | '_ ` _ \/ __|
# | (_| | | |  __/ (_| | | | | | \__ \
#  \__,_|_|  \___|\__,_|_| |_| |_|___/ .
#
# A 'Fog Creek'–inspired demo by Kenneth Reitz™

import os
from flask import Flask, request, render_template, jsonify

# Support for gomix's 'front-end' and 'back-end' UI.
app = Flask(__name__, static_folder='public', template_folder='views')

# Set the app secret key from the secret environment variables.
app.secret = os.environ.get('SECRET')
```

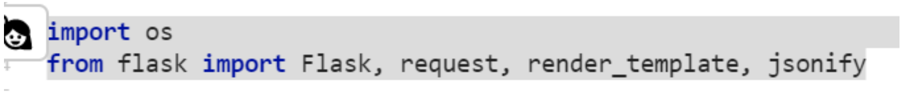
Start.sh file is a "shell script" (file extension .sh) that starts the Web application by running the server.py file



```
# For development use (simple logging, etc):
python server.py
# For production use:
# gunicorn server:app -w 1 --log-file -
```

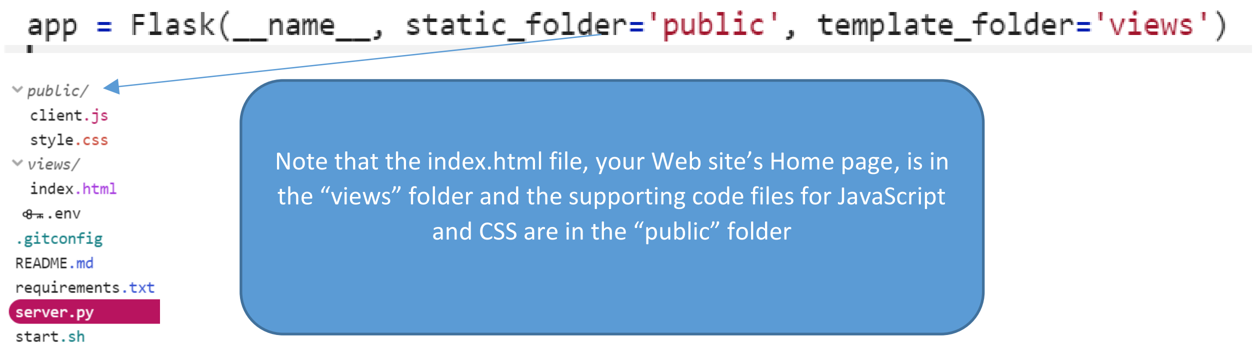
Dive into the Python code (server.py file):

Import Code Libraries: code libraries are SO IMPORTANT in development! Here we import the "os" module to provide access to operating system features, such as environmental variables and we import several functions from the "flask" module



```
import os
from flask import Flask, request, render_template, jsonify
```

Create the Flask App Instance: this statement **instantiates** Flask, i.e., it creates an instance of Flask that will be referenced by the variable name "app". Note that it includes several parameters including the name of the current module (__name__) and the names of folders where application files are found:



```
app = Flask(__name__, static_folder='public', template_folder='views')
```

```
public/
  client.js
  style.css
views/
  index.html
 .env
.gitconfig
README.md
requirements.txt
server.py
start.sh
```

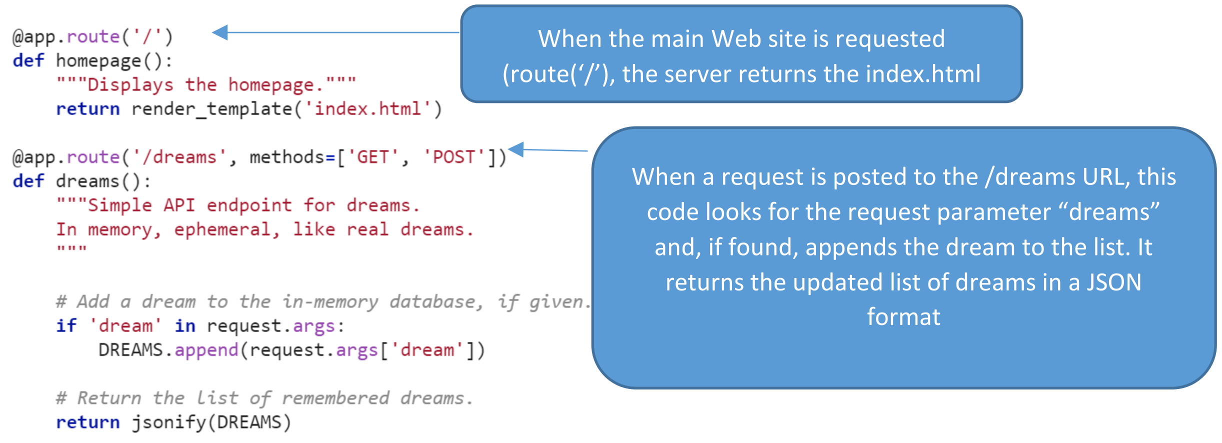
Note that the index.html file, your Web site's Home page, is in the "views" folder and the supporting code files for JavaScript and CSS are in the "public" folder

Create List of Dreams: a List named DREAMS is created and initially populated with one string value; the application will be appending more strings to the comma-separated list:

```
DREAMS = ['Python. Python, everywhere.']
```

Decorators attach functions to routes (i.e., URL paths)

Decorators in Python: a decorator starts with the "@" followed by the variable name and a function name. The decorator executes the function defined directly below it. In Flask, one of the most important decorators is @app.route to define what should happen when specific URLs for the Web site are requested. In this example, there are functions defined for the homepage, represented by "/" and for the /dreams route



```
@app.route('/')
def homepage():
    """Displays the homepage."""
    return render_template('index.html')

@app.route('/dreams', methods=['GET', 'POST'])
def dreams():
    """Simple API endpoint for dreams.
    In memory, ephemeral, like real dreams.
    """

    # Add a dream to the in-memory database, if given.
    if 'dream' in request.args:
        DREAMS.append(request.args['dream'])

    # Return the list of remembered dreams.
    return jsonify(DREAMS)
```

When the main Web site is requested (route('/'), the server returns the index.html

When a request is posted to the /dreams URL, this code looks for the request parameter "dreams" and, if found, appends the dream to the list. It returns the updated list of dreams in a JSON format

Decorators are also used to define event handlers for the app

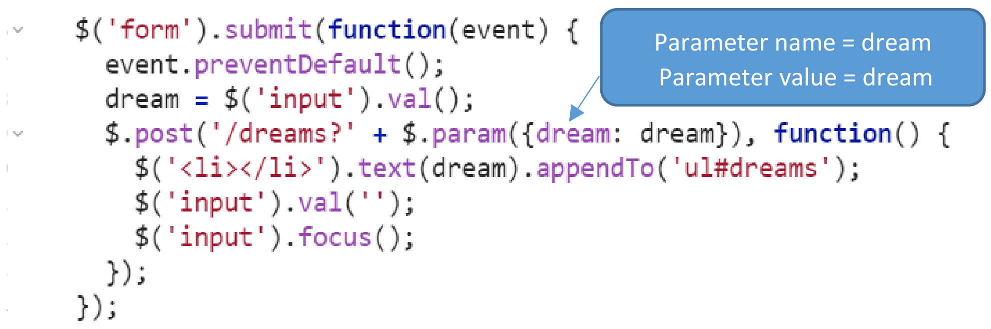
An example in this app is the `@app.after_request` handler that defines a function named "*apply_kr_hello(response)*" to be executed after the server receives a request. It adds information to the headers returned in the response

The app (instance of Flask) is started with the run() statement; when a module is run as a script, its name is __main__:

```
if __name__ == '__main__':

    app.run()
```

Whew! How does the communication flow exactly?

- It starts with clicking the **Submit button** on the form on the **index.html** page
- There is **JavaScript** hooked into the index.html page by this statement just above the closing </body> tag: `<script src="/public/client.js"></script>`
- The client.js file defines a function for the form's submit event:



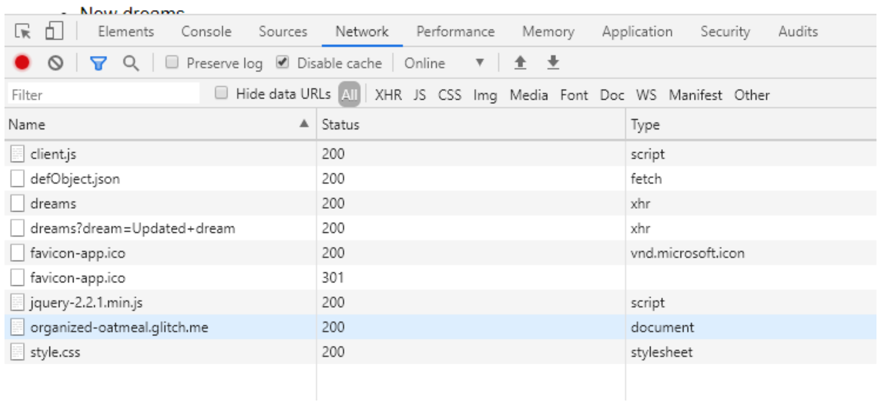
```
$('form').submit(function(event) {
  event.preventDefault();
  dream = $('input').val();
  $.post('/dreams?' + $.param({dream: dream}), function() {
    $('<li></li>').text(dream).appendTo('ul#dreams');
    $('input').val('');
    $('input').focus();
  });
});
```

  - This is Client-side code that does all this:
  - Assign the value in the input textbox to the variable "dream"
  - **POST a request to the server using the /dreams route.** The "?" separator in the query string precedes a list of parameters being passed, an object with one property in format {parameterName:parameterValue}. Name and value are both "dream"
  - Create a new html list item (<li>) element containing the dream and append it to the unordered list (<ul>) html element having the id "dreams":
    - `$('<li></li>').text(dream).appendTo('ul#dreams');`
  - Clear the value of the input text field and set focus there
- **Look again at server.py for what the server-side code does with a request to /dreams:**
  - look in the arguments sent in the request for an argument named 'dream':
  - if found, append the new dream to the list DREAMS, so the in-memory "database" is updated
  - return the updated list in JSON format

```
@app.route('/dreams', methods=['GET', 'POST'])
def dreams():
    # Add a dream to the in-memory database, if given.
    if 'dream' in request.args:
        DREAMS.append(request.args['dream'])
    # Return the list of remembered dreams.
    return jsonify(DREAMS)
```

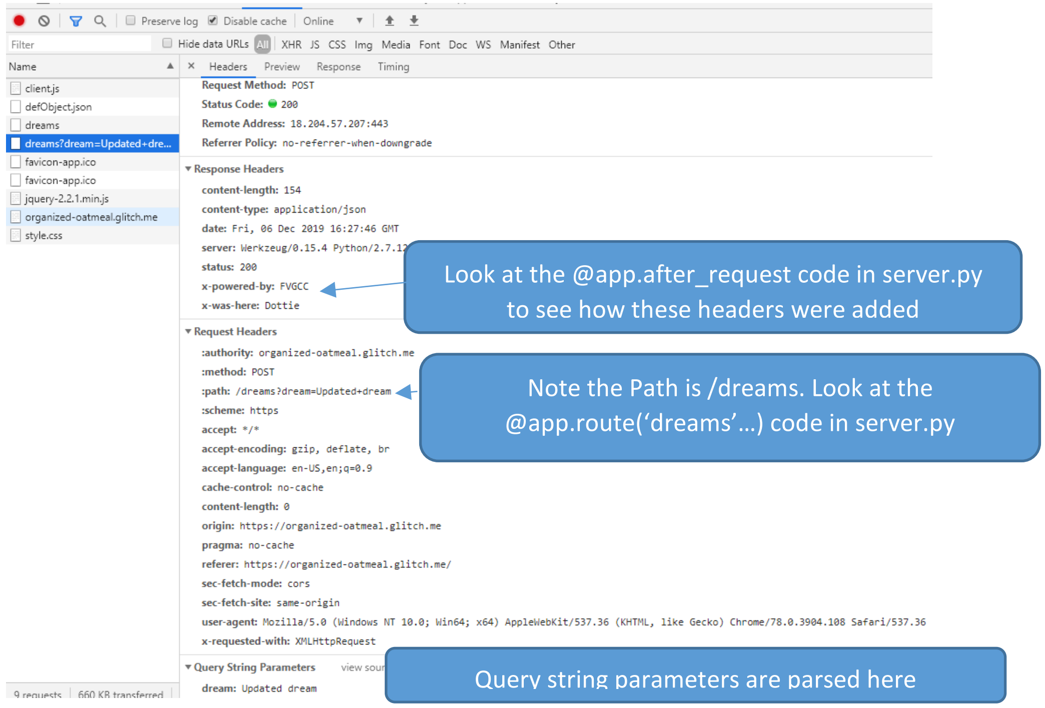
Use Dev Tools to see Request/Response Flow

- Show your project in its own Window by selecting that option under Show to the right of project name
- In Chrome, open Dev Tools by using the Shortcut key F12 in Windows, Command Option I in Mac: https://developers.google.com/web/tools/chrome-devtools/shortcuts
- Select the Network tab in Dev tools
- The Network panel will list each Request/Response as well as resources such as images or files, but will not populate until you make a request by submitting a dream or refreshing the page
- After entering a new dream with value "Updated dream" the network panel looks like this:



Name	Status	Type
client.js	200	script
defObject.json	200	fetch
dreams	200	xhr
dreams?dream=Updated+dream	200	xhr
favicon-app.ico	200	vnd.microsoft.icon
favicon-app.ico	301	
jquery-2.2.1.min.js	200	script
organized-oatmeal.glitch.me	200	document
style.css	200	stylesheet

- The items with type xhr are XMLHttpRequest instances and we can examine the Request and Response details for each
- Select the one that has "dreams?dream=…."; that is the communication where we sent a new dream to the server with a POST request. Look at the Headers tab first.



```
Request Method: POST
Status Code: ● 200
Remote Address: 18.204.57.207:443
Referrer Policy: no-referrer-when-downgrade
▼ Response Headers
content-length: 154
content-type: application/json
date: Fri, 06 Dec 2019 16:27:46 GMT
server: Werkzeug/0.15.4 Python/2.7.12
status: 200
x-powered-by: FVGCC
x-was-here: Dottie
▼ Request Headers
:authority: organized-oatmeal.glitch.me
:method: POST
:path: /dreams?dream=Updated+dream
:scheme: https
accept: */*
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
cache-control: no-cache
content-length: 0
origin: https://organized-oatmeal.glitch.me
pragma: no-cache
referer: https://organized-oatmeal.glitch.me/
sec-fetch-mode: cors
sec-fetch-site: same-origin
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
x-requested-with: XMLHttpRequest
▼ Query String Parameters    view sour
dream: Updated dream
```

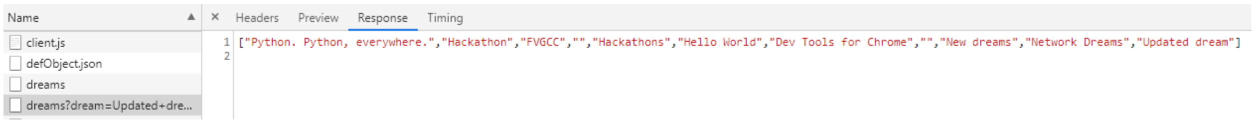
Look at the @app.after_request code in server.py to see how these headers were added

Note the Path is /dreams. Look at the @app.route('dreams'…) code in server.py

Query string parameters are parsed here

-

- Now select the Response tab to see what was returned from the server:

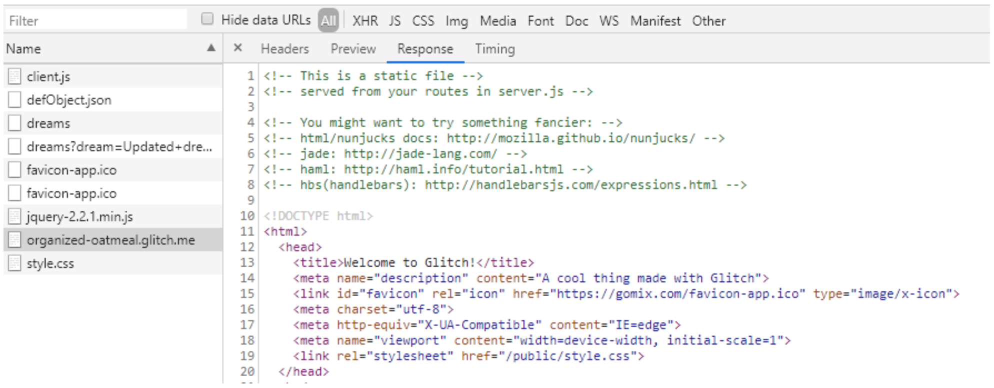


```
["Python. Python, everywhere.","Hackathon","FVGCC","","Hackathons","Hello World","Dev Tools for Chrome","","New dreams","Network Dreams","Updated dream"]
```

- That Response comes from the last line of the function for the /dreams route

```
# Return the list of remembered dreams.

    return jsonify(DREAMS)
```

- Next select the other request from the list, the one without the dreams? Appended. In my example, it is organized-oatmeal.glitch.me:
    - Look at the Headers and note that there are no query parameters and that the Request Method is GET
    - Look at the Response and note that it is the HTML returned from the server after being updated by the server-side code



```
<!-- This is a static file -->
<!-- served from your routes in server.js -->

<!-- You might want to try something fancier: -->
<!-- html/nunjucks docs: http://mozilla.github.io/nunjucks/ -->
<!-- jade: http://jade-lang.com/ -->
<!-- haml: http://haml.info/tutorial.html -->
<!-- hbs(handlebars): http://handlebarsjs.com/expressions.html -->

<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to Glitch!</title>
    <meta name="description" content="A cool thing made with Glitch">
    <link id="favicon" rel="icon" href="https://gomix.com/favicon-app.ico" type="image/x-icon">
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="/public/style.css">
  </head>
```