**FLASK DOCUMENTATION:**

Flask is a web application framework written in Python.

**Flask Installation:**

The installation of the flask package is easy.

pip install flask

pip install flask-user

pip install flask-wtf

pip install flask-sqlalchemy

**WHY IS FLASK BETTER THAN Node JS/PHP based Web Development?**

1. Well-thought-out design:

You will find it much easier to use Python for writing great code than PHP, although it is not impossible to write good code in PHP/JS. Python has the architecture that helps make it a well-thought-out, well-designed and robust language.

1. In the ease of learning, python wins:

One of the reason why Python was created was for ease in readability.

1. Superior Framework

Allows us to make custom applications with easy overriding techniques.

1. Renders itself more readable.

Easy to read a code in python than PHP/Js.

1. Much simpler syntax.
2. Simpler, easily available debugger tools.
3. A clear winner in package management.
4. Python is more versatile than Js/PHP.
5. Vast support and community.

**PREREQUISITES:**

* Python
* HTML/CSS

**SE Workshop**:

**Introduction**: Let’s look at a basic program in Flask :-

**Start with the hello world program for flask:**

from flask import Flask  
 app = Flask(\_\_name\_\_)  
  
 @app.route('/')  
 def hello\_world():  
 return 'Hello World’  
  
 if \_\_name\_\_ == '\_\_main\_\_':  
 app.run()

**Explanation**:

* app = Flask(\_\_name\_\_)

**\_\_name\_\_** is the name of current module. (py file name)

Flask constructor takes the name of **current module (\_\_name\_\_)** as argument.

* @ is a python decorator, used to wrap up a function. : <https://www.geeksforgeeks.org/decorators-in-python/>
* The **route()** function of the Flask class is a decorator, which tells the application which URL should call the associated function.
* Finally the **run()** method of Flask class runs the application on the local development server.

**What is Buzznet**?

It is a common web platform where people can come and share their creative

***strings - with a title and description***. First, the user has to register with us on this platform. For the login part, we have used Flask User extension. After the login, the user has the option to see their feeds, profile and option to put up a string along with caption for it.

**FEATURES:**

1. An authenticated login/signup.
2. Keeping a track on user activity.
3. POST option for string.
4. An option to see your feed.

* **Flask-Login: Why to use it?**

Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users’ sessions over extended periods of time.

* **SQLAlchemy**:

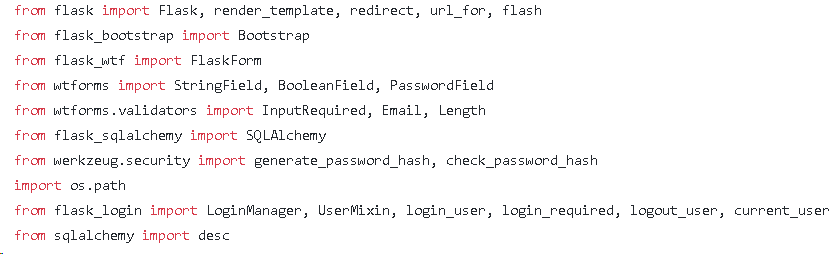
SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

*Brief description about SQL, most SE would be knowing it, as it is taught in RDBMS.*

* Each feature in the website will have an associated HTML template with it. This template would be available on the participants systems.

**MOVING ON WITH THE CODE :)**

**#include < !DEPENDENCIES >:**

****

**flask**: To import Flask web framework.

**flask\_bootstrap**: To work with bootstrap, used for styling the website.

**flask\_wtf**: To work with WTForms, used for user input.

**flask\_sqlalchemy**: To work with the database.

**werkzeug.security**: To generate hash for password.

**os.path**: To work with directories in the local OS.

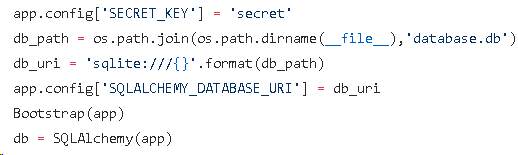
**flask\_login**: Provides us functions necessary to build a robust user signup and login.

**desc**: to order the id in descending order.

**WSGI** Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

**Werkzeug** It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

Setting directory for the database:



The way Flask is designed usually requires the configuration to be available when the application starts up. Independent of how you load your config, there is a config object available which holds the loaded configuration values: ***The config attribute of the Flask object***. This is the place where Flask itself puts certain configuration values and also where extensions can put their configuration values.

Config is a subclass of dictionary and can be modified as a dictionary.

There is an object called ***session*** which allows you to store information specific to a user from one request to the next. This is implemented on top of cookies for you and signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

**Simple explanation: ‘Session** data is stored on server. Session is the time interval when a client logs into a server and logs out of it.’

The data, which is needed to be held across this session, is stored in a temporary directory on the server.

Anything that requires encryption (for safe - keeping against tampering by attackers) requires the secret key to be set. For *just* Flask itself, that 'anything' is the Session object, but other extensions can make use of the same secret.

If you didn't set a server-side secret for the encryption to use, everyone would be able to break your encryption; it's like the password to your computer. The secret plus the data-to-sign are used to create a signature string, a hard-to-recreate value using a cryptographic hashing algorithm; only if you have the exact same secret *and* the original data can you recreate this value, letting Flask detect if anything has been altered without permission. Since the secret is never included with data Flask sends to the client, a client cannot tamper with session data and hope to produce a new, valid signature.

The OS module in Python provides a way of using operating system dependent functionality. The functions that the OS module provides allows you to interface with the underlying operating system that Python is running on – be that Windows, Mac or Linux.

*os.path.join() joins two parts of a directory.*

*db\_uri stores the directory to the database file in our server.*

*Set the SQLAlchemy Database URI in the config accordingly.*

*Bootstrap() is used to style our app.*

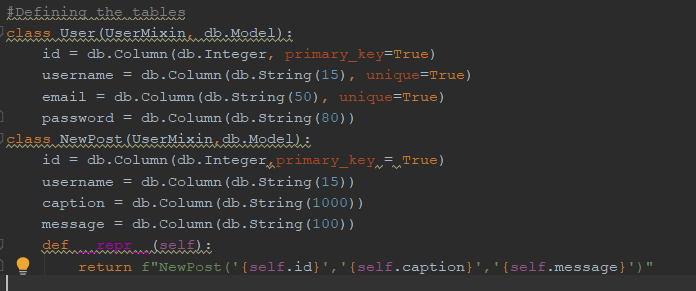
*The SQLAlchemy() object is stored in db for further usage.*

**DQ Querying in CMD**



The databases are defined for two purpose:

1. For Usernames.
2. For Posts made by the User.



Each database is made using class by passing attributes:

1. db.model: This is the base class for all of our models.
2. Usermixin: This provides default implementations for the methods that Flask-Login expects user objects to have.

To add columns in our database, we need to use **db.column** and specify characteristics like key type, datatype and other properties.

**\_\_repr\_\_** is used by Python standalone applications to display class in printable format.

The class that you use to represent users needs to implement these properties and methods:

**is\_authenticated**

This property should return True if the user is authenticated, i.e. they have provided valid credentials.

**is\_active**

This property should return True if this is an active user - in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account.

**is\_anonymous**

This property should return True if this is an anonymous user. (Actual users should return False instead.)

**get\_id()**

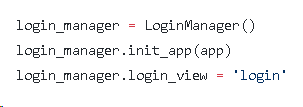
This method must return a unicode that uniquely identifies this user, and can be used to load the user from the user\_loader callback.

This provides default implementations for the methods that Flask-Login expects user objects to have.

To make implementing a user class easier, you can inherit from UserMIxin, which provides default implementations for all of these properties and methods.

The most important part of an application that uses Flask-Login is the LoginManager class.

The login manager contains the code that lets your application and Flask-Login work together, such as how to load a user from an ID, where to send users when they need to log in, and the like.

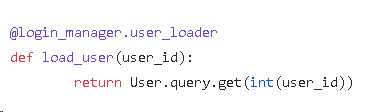


Once the actual application object has been created, you can configure it for login with:

login\_manager.init\_app(app)

The name of the log in view can be set as:

Login\_manager.login\_view = ‘login’



**Login\_manager.user\_loader** is used to load the user object from user ID stored in the session.

load\_user is the function used to load the userID of the user logged in.

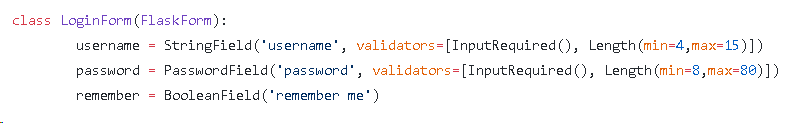
**User.query** stores all the functions necessary for query on db. Here get() is used to get the user\_id.

**GET** is used to request data from a specified resource.

**POST** is used to send data to a server to create/update a resource.

Data entered by a user is submitted in the form of Http request message to the server side script by either GET or POST method.

Using **Flask-WTF**, we can define the form fields in our Python script and render them using an HTML template. The forms to take input from the user is created using FlaskForm.



**BooleanField**: Represents <input type = 'checkbox'> HTML form element.

To keep the user logged in.

By default, when the user closes their browser the Flask Session is deleted and the user is logged out. “Remember Me” prevents the user from accidentally being logged out when they close their browser. This does **NOT** mean remembering or pre-filling the user’s username or password in a login form after the user has logged out.

**PasswordField**: Represents <input type = 'password'> HTML form element.

To extract the password provided by the user.

**Validators** is used to add constraint to the input, like if the input is necessary (InputRequired()) and the min and max length.

**StringField**: Represents <input type = 'string'> HTML form element.

To extract the username of the user.

Similar explanation goes on for the rest two forms.

**SIGNUP**:



Methods is used to specify the property of the function URL.

Our HTML template contains the style and feature design for the website. Each page, has its own template. A navigation bar is provided in order to navigate through different pages of the website.

To provide the links for our navigation bar in the html, we store the corresponding links of each options available for navigation and pass it in render\_template().

form variable stores the RegisterForm()’s object.

validate\_on\_submit() will check if it is a POST request and if it is valid.

User.query: filter\_by() is used to filter out the username from the database. We will check for common usernames in our database.

If username already exists, flash() function is used to display the error message : “Username already exists”.

We also generate hashed password using SHA256 encryption algorithm, in order to keep the user password personal upto him/her.

Else, we create a User object, passing the username, email and hashed password.

db.session.add() help us to add the user into the database

db.session.commit() to finalise our changes to the db.

Once the signup is successful, the user is redirected using redirect() to the login page.

render\_template() is used to render our html template for the web page.

**LOGIN:**



form variable stores the LoginForm()’s object.

validate\_on\_submit will check if it is a POST request and if it is valid.

User.query: filter\_by() is used to filter out the username from the database.

If the user is present in the database, we check the password for it.

Once a user has authenticated, you log them in with the login\_user function by passing the user object to it.

Once the login is successful, the user is redirected using redirect() to the feed.

db.session.add() help us to add the information into the database

db.session.commit() to finalise our changes to the db.

render\_template() is used to render our html template for the web page.

logout\_user(): as its name says, to log out the user. It also terminates the session associated with the user.

NEW POST:



The initial part is similar.

After validation, the caption and message given as an input by the user is passed into NewPost() to create an object and add to the database. The changes are committed and the webpage is rendered using render\_template().

FEED AND PROFILE:

****

The user profile and feed don’t require any communication with the user for any inputs. They only show the data stored in the database, i.e., the getProfile() will show the user details and the getFeeds() will show the feed to the user.

As no validation of any form is required, we simply use render\_template() and pass the options.

For Feeds, one needs to reflect the most recent changes in the database. The data is order in desc (descending) order, since we want to keep the newest post at the top of the user feed.

*~FIN~*

FLASK WORKSHOP (FEs) DRAFT

**Part 1: Introducing flask *(This part should take max 30 mins)***

Ask them what is web development.

Have a **discussion** about frontend, backend and database.

Don’t explicitly teach this.

***{ Database*** *- A Web* ***database*** *is a* ***database*** *application designed to be managed and accessed through the Internet.* ***Website*** *operators can manage this collection of data and present analytical results based on the data in the Web* ***database*** *application.*

***Back-end web developers*** *are responsible for the overall technical construction of the website.*

***Front-end******web developers*** *are responsible for how a* ***website*** *looks. They create the site's layout and integrate graphics, applications (such as a retail checkout tool), and other content. They also write web design programs in a variety of computer languages, such as HTML or JavaScript.* ***}***

Now ask them about the languages used in web development. Ask them which of them are used in frontend/backend/database.

**\* CHEAT SHEET!!! \***

|  |  |  |
| --- | --- | --- |
| **Frontend** | **Backend** | **Database** |
| These all are languages   * HTML * Javascript * CSS * Actionscript * CoffeeScript (compiled to Javascript) * XML-based languages (X3D, SMIL, SVG, DITA, some interpreted by the browser, others transformed using XSL) * VBScript * Silverlight * Java (applets) * .NET * JAVA(Swing apps)   These are frameworks   * Angular * React * Vue * Ember * Backbone | These all are languages   * Java (and other JVM languages like Scala, Groovy, Clojure) * PHP * .NET (C#, VB) * Ruby * Python * Perl * Javascript (Node JS) * Actionscript (Flash Media Server) * CoffeeScript * C (CGI) * Erlang * oh, and SQL for db queries   These are frameworks   * Express * Django * Rails * Laravel * Spring * Flask | * Oracle * MySQL * MS Access * dBase * FoxPro * MongoDB |

Can we build a website like Flipkart using HTML?

Suppose there’s a new phone brand that has been launched. Now the company decides to check the overall sales after a day. So, can the flipkart website be updated accordingly in real time (after each phone is bought), using HTML?

Obviously not. This is because HTML is a language that can do nothing much more than displaying plain text on a webpage (or adding a background image for the record). HTML, a static language, cannot be used to incorporate a meaningful functionality into the website.

This is where Dynamic languages and frameworks come into picture.

Wait wait wait. What’s a framework?

Creating a website from scratch uses a huge amount of code.

But as the website becomes more and more complex, code gets repetitive and difficult to manage. A framework is a collection of code that simplifies the web development process.

They take care of the repetitive tasks and help us to focus on the unique features of the project, and makes it a lot more easier for us to build a functional website.

{ Example: baking a cake. You can either bake a cake using ingredients from scratch, or buy a cake mix. The first option will no doubt improve your understanding of cakes, but the second option will help you bake it faster, and let you devote more time to other tasks like frosting and stuff, which will set it apart from other cakes :)

There are a lot of cake mixes you can buy in a supermarket, but it’s important to find the best cake mix that will help you make a great cake! The same applies to web frameworks as well. }

You all must be aware of how dynamic and flexible python is. The only drawback of python was (for until some time ago) that it did not have an interface capable of supporting all (or most) interactions between a Web server and a Web framework.

This is why things like Nodejs and angular gained in popularity. But still they have a setback i.e. they cannot be used in any other applications outside web applications.

Python has recently introduced WSGI *(*Web Server Gateway Interface). It is a specification for a standardized interface between Web servers and Python Web frameworks/applications. So we don’t have to worry about making our application run in multiple Web servers since most servers are WSGI compliant.

Why is such an interface necessary?

Let’s take an example of mobile phones. In order to communicate with each other, there needs to be a 3G/4G connection between the phones. Similarly, there is a need for a protocol to connect the client’s system to the server.

For example, Google. All of its data and applications are on its own server and we don’t download this entire thing every time we search anything, do we?

HTTP stands for "Hypertext Transfer Protocol." HTTP is the protocol used to transfer data over the web. It is part of the Internet protocol suite and defines commands and services used for transmitting web page data.

HTTP uses a server-client model. A client, for example, may be a home computer, laptop, or mobile device. The HTTP server is typically a web host running web server software, such as Apache or IIS. When you access a website, your browser sends a request to the corresponding web server and it responds with an HTTP status code. If the URL is valid and the connection is granted, the server will send your browser the web page and related files.

There are a lot of pcs connected via the internet. Each of them needs to have its own identity. Just like, each house on land has its own address that is unique. An IP address is a collection of numbers that is used to provide this unique identification.

Now each one of us has and aadhar card number. But do we, call the person by his/her aadhar card number? No right. He or she is called by a name. Similarly using IP address every time becomes hectic. Hence, each IP address is provided with a domain name.

The ip address we are using here - 127.0.0.1 is called nothing but localhost.

|  |  |
| --- | --- |
| Flask | Django |
| * Flask provides *simplicity*, *flexibility* and *fine-grained control*. **It is *unopinionated*** (it lets you decide how you want to implement things). |  |

Explain Django VS Flask. (If they ask)

We’ll be dealing with a Python framework called Flask, which is becoming increasingly popular nowadays cos

1. Flask was designed to be easy to use and extend. The idea behind Flask is to build a solid foundation for web applications of different complexity. From then on you are free to plug in any extensions you think you need.
2. Flask gives you more control on the development stage of your project.
3. Flask has a lightweight, modular and flexible design.
4. It’s way easier to learn, and it’s got a great debugger.
5. It’s documentation is comprehensive and well-structured.

To sum up, Flask is one of the most polished and feature-rich micro frameworks available.

Https - draft, example of connecting phones and stuff, about ip addresses and domain names

**Part 2: Getting started (The Hello World of flask)**

\*\*\*Refer the code\*\*\*

Let’s break it down:

1. from flask import Flask

We have the flask module installed on our system. From that module, we are importing a **flask application object** called Flask.

Flask implements a WSGI application and acts as the central object. Our entire program will now revolve around this object.

This object contains data about the application and also different functions that tell the application to do certain actions.

*Note:*

* *Don’t get confused between a flask application object and a flask application. Think of a flask application object as a box of paints and the flask application as the painting.*
* *WSGI*

2. app = Flask(\_\_name\_\_)

Now that we’ve made our flask application object, it is time to use its features and functions to make our application. Let’s call our application app.

Every module in python has a special attribute called \_\_name\_\_, which (duh) contains the name of the module. Remember, the value of the \_\_name\_\_ attribute is set to '\_\_main\_\_' when the module **runs** as the main program. This name information is used to find resources on the filesystem, it can be used by extensions to improve debugging and a lot more.

3. @app.route('/')

Flask has its own web server at http://127.0.0.1:5000/ . We can define various functions (That will ultimately result in web pages) at either this url or at its extensions (say, for example, http://127.0.0.1:5000/about or http://127.0.0.1:5000/contact ). The @app.route() function defines the urls that are needed for our web pages. Since we gave the argument as ‘/’, the url is defined at the default- http://127.0.0.1:5000/ .

(If we want to define a path to our about page i.e. http://127.0.0.1:5000/about then we can give ‘/about’ as the argument to this function.)

4. def hello\_world():

return 'Hello World!'

We have defined a function called hello\_world that will create a web page containing “Hello World” at the url http://127.0.0.1:5000/ .

5. if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Time to run our application! The function app.run() has the job of kickstarting our web app :)

Now we need to see whether our web page has been created properly or not. For this, open the command prompt and follow these steps:  
1. Navigate to the folder where the program is saved.

2. set FLASK.APP=program\_name.py

3. set FLASK.DEBUG=1

4. flask run

This will generate an IP address to be entered into the browser.

If we set FLASK.DEBUG=1, then the flask debugger is activated. Suppose there’s a bug in our program. If the debugger would’ve been inactive, we’d have to repeat all of the above steps to make the changes reflect on our webpage. But with the debugger active, we just need to refresh our page to incorporate the changes. How cool is that?!

**Part 3: The Weather App**

Now that we know what the basics of flask are, let’s start building something more fancy. For that we require website templates. These are programs containing lines and lines of html/css/bootstrap that will design a website along with its basic functionalities like buttons, forms, etc. In flask, the render\_template module is a great tool to help us manage and integrate different templates for different pages of the website.

return render\_template('template\_name')

\*\*\*be sure to demonstrate render.py\*\*\*

Before getting into building our weather app, let’s first understand what APIs are.

An **Application Programming Interface (API)** refers to a part of a computer program designed to be used or manipulated by another program. Computer programs frequently need to communicate amongst themselves or with the underlying operating system, and APIs are one way they do it.

A **web API** allows for information or functionality to be manipulated by other programs via the internet. For example, with Twitter’s web API, you can write a program in a language like Python or Javascript that can perform tasks such as favoriting tweets or collecting tweet metadata.

Every time you make a call to a server in name of an application using an API, it counts as an API request. Logins, saves, queries are examples of operations counted as API requests among other types of operations. Consider you downloaded an application on your smartphone, you opened it and the application asked your Email and Password. At the moment you press Register and send your data to the API it is counted as one API request.

When you ping a website or portal for information, even that is counted as a request. That is exactly what the requests library has been designed to do.

If we want the data from the pre-existing Weather app API (We’re using the **weather** data corresponding to a particular **city** from openweathermap.org) , we’ll use the following line of code:

req = requests.get('http://api.openweathermap.org/data/2.5/weather?q='+ city +'&units=celsius&appid=271d1234d3f497eed5b1d80a07b3fcd1').json()

Now, we have a **response** **object** called req . We can get all the information we need from this object.

*Note: JSON (JavaScript Object Notation) is a text-based data storage format that is designed to be easy to read for both humans and machines. JSON is generally the most common format for returning data through an API, XML being the second most common.*

\*\*\*Refer wcity.py\*\*\*

Step 1 : We have imported libraries and modules that we’ve described in the earlier sections, except for the one called request.

The data entered from a client’s web page is sent to the server as a **global request object** (We will need this for giving the city name as an input). In order to work with request objects, we need to import the request module.

Step 2 : We have created a flask application object like before. We’ve also activated the debug mode.

Step 3 : Define a function called city that will generate a webpage at the default flask IP address.

Step 4 : Input the city using the ‘GET’ method in the form tag in HTML. The input will be of ‘NoneType’ in python until the user inputs the city name. By default we are taking the city name as Mumbai.

Step 5: Request the API to generate the weather data by passing the city name to the API. Convert it into a JSON object. An example of weather data in JSON format is

{"coord":{"lon":-0.13,"lat":51.51},"weather":[{"id":500,"main":"Rain","description":"light rain","icon":"10d"}],"base":"stations","main":{"temp":282.22,"pressure":1000,"humidity":87,"temp\_min":280.93,"temp\_max":283.71},"visibility":10000,"wind":{"speed":10.3,"deg":250,"gust":16.5},"clouds":{"all":90},"dt":1552550914,"sys":{"type":1,"id":1468,"message":0.0082,"country":"GB","sunrise":1552544217,"sunset":1552586602},"id":2643743,"name":"London","cod":200}

Step 6: Locate the important details that we want to show. All we need is the city name, the temperature, the description of the weather and the icon. The details are highlighted in the JSON object below

{"coord":{"lon":-0.13,"lat":51.51},"**weather**":[{"id":500,"main":"Rain",**"description":"light rain"**,**"icon":"10d"**}],"base":"stations","**main**":{**"temp":282.22**,"pressure":1000,"humidity":87,"temp\_min":280.93,"temp\_max":283.71},"visibility":10000,"wind":{"speed":10.3,"deg":250,"gust":16.5},"clouds":{"all":90},"dt":1552550914,"sys":{"type":1,"id":1468,"message":0.0082,"country":"GB","sunrise":1552544217,"sunset":1552586602},"id":2643743,"name":"London","cod":200}

Step 7: Store these values in variables. Tell them how to access these data from the JSON object. For example, to store the description, first locate to which object it is mapped. It is present as “description” itself. The description itself is present in a list “weather”. So to access the description we will have to use

Req['weather'][0]['description']

Step 8: Store each value in a variable. The icon is an image stored in the database of openweather. So to access that image, append the name of the image icon to the url to get the full url of the icon.

Step 9: Open the city.html file and show them the location where we want to show our details. Show them how do we pass parameters to a HTML template. For example, if we want to display the city name and the name is stored in a variable ‘city’, then in the HTML file {{ city }} will replace the variable ‘city’ with the city name.

Step 10: In order to display a template using Flask we need to use the render\_template function. Show them how to code it in the flask. The render\_template function takes the html filename as input. (Note: In order to render a template, all html files are stored in the templates folder and all css/js files are stored in static folder.) Since we need to pass parameters to the template, we have to pass them as parameters as well. Syntax: render\_template(‘templatename.html’, template\_variable = value)

Part 4 - Weather using current location.

The weather app which we made just now was using the input from the user. Now we want to display the weather of the place using the user’s current location. Our main objective in this app is to find user’s location. The API mentioned above is not going to help us find the location of the user. We need another API to find location. There are many APIs available to us. Google some APIs and show it to them, like, GeoLocation, GeoIpify and many more. We are going to use IP-API for our app.

Step 1 : We have imported libraries and modules that we’ve described in the earlier sections, except for the one called request.

The data entered from a client’s web page is sent to the server as a **global request object** (We will need this for giving the city name as an input). In order to work with request objects, we need to import the request module.

Step 2 : We have created a flask application object like before. We’ve also activated the debug mode.

Step 3 : Define a function called location that will generate a webpage at the default flask IP address.

Step 4: Request the api to provide data for our current location.

URL: <http://ip-api.com/json>

This url will return an JSON object and we shall store it in a variable. The json object looks like this -

{"as":"AS9498 Bharti Airtel Limited","city":"Mumbai","country":"India","countryCode":"IN","isp":"Bharti Airtel Limited","lat":18.9721,"lon":72.8246,"org":"Bharti Airtel","query":"182.73.90.242","region":"MH","regionName":"Maharashtra","status":"success","timezone":"Asia/Kolkata","zip":"400002"}

Step 5: Locate the city name in the JSON object. It is highlighted in the object given below

{"as":"AS9498 Bharti Airtel Limited",**"city":"Mumbai"**,"country":"India","countryCode":"IN","isp":"Bharti Airtel Limited","lat":18.9721,"lon":72.8246,"org":"Bharti Airtel","query":"182.73.90.242","region":"MH","regionName":"Maharashtra","status":"success","timezone":"Asia/Kolkata","zip":"400002"}

Step 6: Store the city name in a variable, how we did it in the previous app.

city = object\_name[“city”]

Step 7: It is now similar like the previous app. We have just figured out the city on our own instead of asking it from the user.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |