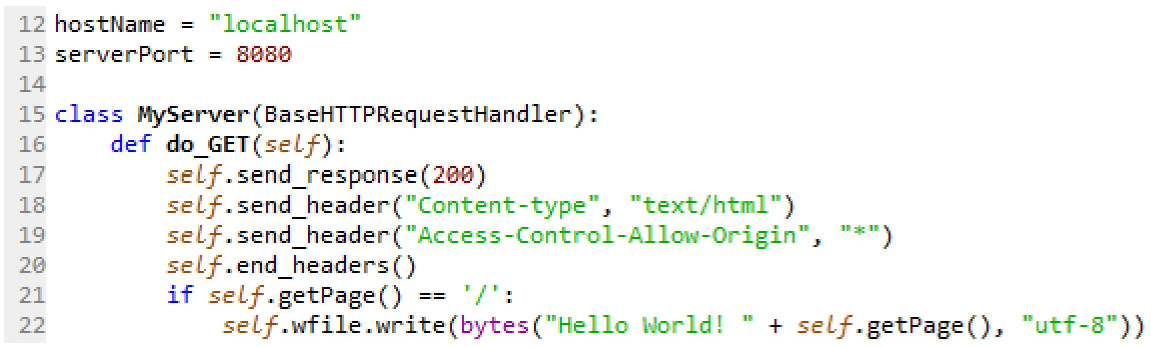
COSC 331
Lab 2: Working with Microservice Frontends in Python

In this lab, we'll be building a new microservice using Python, which will implement a basic microfrontend. Before getting started with this lab, you should download and install Python and an appropriate IDE. You can download the base Python 3 (do not use Python 2) installation here: https://www.python.org/downloads/ . Python comes with a basic console/development application called IDLE which you can use to work on and test your lab. If you prefer a more full-featured IDE, you can use Anaconda, which includes a full Python installation as well as the Spyder IDE, which is similar to Eclipse. You can download Anaconda here: https://www.anaconda.com/products/individual (scroll to the bottom).

Building a Basic Python Application

To begin, you'll want to download the `pythonBase.py` and `encryptionBase.html` files from the course Moodle or GitHub. Make sure they're both kept in the same location, as the Python application will need to read from the HTML file.

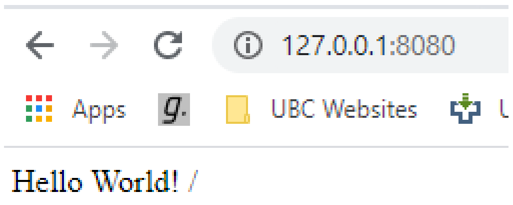
Open up `pythonBase.py` in IDLE, Spyder, or the Python IDE of your choice. The code you'll find inside is a basic Python web server with some additional functions that will be necessary for this lab.



```python
hostName = "localhost"
serverPort = 8080

class MyServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Access-Control-Allow-Origin", "*")
        self.end_headers()
        if self.getPage() == '/':
            self.wfile.write(bytes("Hello World! " + self.getPage(), "utf-8"))
```

At the beginning of the application file, we define our hostname and port. Note that we are using port 8080 for this service, **not** port 80. We then define the **`MyServer`** class, which serves as an HTTP request handler for our web server. The **do_GET** function is called whenever the server receives a GET request.

In the do_GET function, we assign an HTTP response code (`200 – OK`), some headers to describe our content type and access origin control, and then we write a simple Hello World message to our HTTP response body before returning it to the client. If you run this application and visit **`127.0.0.1:8080/`** in your web browser, you should see something like the following:

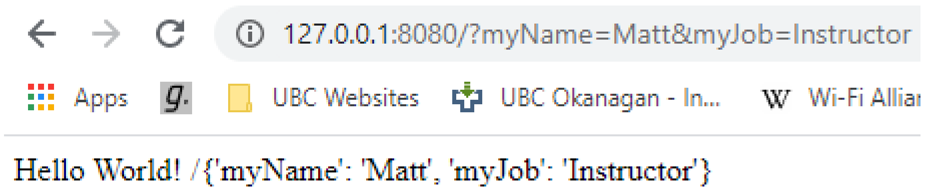


Hello World! /

Several other functions are also defined in the `MyServer` class, which are related to what we intend to do with our lab. The first of these is the **`getParams()`** function, which fetches all parameters in the query string and returns them as key-value pairs in a dictionary object. You can test this out by adding a line to your do_GET method, and then passing a query string to your service:

```
if self.getPage() == '/':
    self.wfile.write(bytes("Hello World! " + self.getPage(), "utf-8"))
    self.wfile.write(bytes(str(self.getParams()), "utf-8"))
```

Note that because this function is defined within the same class, we'll need to use the **`self`** prefix when we call the function, i.e. **`self.getParams()`**. When you visit your service in a browser and give it a query string, it will return a string representation of the dictionary object containing the parameters:



Hello World! /{'myName': 'Matt', 'myJob': 'Instructor'}

Similarly, the **`getPage()`** function will return the current page path that the user has requested. An example of this is already included with the Hello World line, which returns "/" when the user is visiting the "home page" or root path of the service. We can use this function to tell which specific path a client is requesting from our service by performing a string comparison on it, as we see in the example code.

The remaining two functions included in the MyServer class are called **`encode()`** and **`decode()`**. These functions take two inputs – a key and a text, both of type string. The encode function will take an input string and encrypt it with the key using a basic addition cipher, and return a base-64 encoded encrypted string as a result. The decode function will take an encrypted string, and use the key to decode it back to it's original text content, assuming the key used matches the key used to encrypt. Together, these form a symmetric encryption system, in which an identical key is used to both encrypt and decrypt our text.

Our microservice will perform three major tasks:

- Return a basic HTML form to the client, a form of micro-frontend, which will include JS required to make AJAX calls back to the service
- Take an AJAX call from the HTML form with a key and a plaintext, and then return to the client the encrypted ciphertext
- Take an AJAX call from the HTML form with a key and a ciphertext, and then return to the client the decrypted plaintext

Implementing Request Handling

In order to handle all three tasks for our service, we'll need to be able to handle three different kinds of requests – the basic request, which doesn't pass any data from the client and just returns the base HTML form, and two types of GET request with parameters, one for encryption, and one for decryption.

The base request is made to the root page, '/', and returns our HTML stored in the encryptionBase.html file. We can use our 'Hello World' response as a template, and make only minor changes. First, we'll want to open our HTML file and read the contents into a string:

```
html = open("encryptionBase.html")
htmlString = html.read()
html.close()
```

If you look at the encryptionBase.html file, you'll notice it has a _PLACEHOLDER_ value near the top. We'll replace this with a rather generic message using a string replacement:

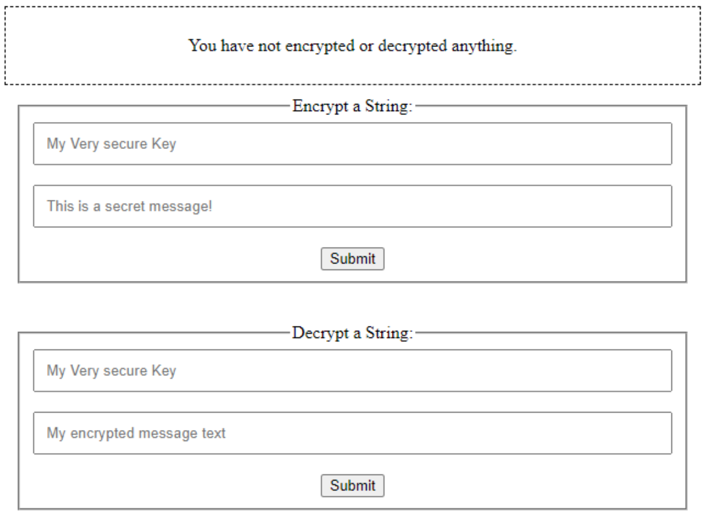
```
htmlString.replace("_PLACEHOLDER_", "You have not encrypted or decrypted anything.")
```

You can then either assign this to a variable, or have it go directly within the **`self.wfile.write()`** function, which takes bytes and writes them to the HTTP response body. We'll need to use the **`bytes()`** function to cast our string HTML to bytes using the **`utf-8`** encoding:

```
self.wfile.write(bytes(htmlString.replace("_PLACEHOLDER_", "You have not encrypted or decrypted anything."), "utf-8"))
```

All of this code should go under the if statement that checks if the user has requested the root page ('/'), to make sure that we only return this HTML when the root is requested.

Once done, you should be able to visit the home page in your web browser, and see the following form:


You have not encrypted or decrypted anything.

Encrypt a String:

My Very secure Key

This is a secret message!

Submit

Decrypt a String:

My Very secure Key

My encrypted message text

Submit

This is our basic microfrontend – not an entire application, but just the basic HTML (and eventually JavaScript) which is necessary for our microservice application. It consists of two forms – one for encryption, and one for decryption. This form won't do anything right now, however.

Next, let's implement our request handling for when a user wants to encrypt something with our microservice. We'll begin by adding another if statement, this time to catch requests when a user wants to get the **`/encrypt`** page:

```
if self.getPage() == '/encrypt':
```

The first thing we should do inside this **if** block is fetch the GET query string parameters of the request using the `getParams()` function:

```
myParams = self.getParams()
```

We'll then pass two of the parameters, our **key** and our **`plaintext`**, into the **`encode()`** function. Note that you can name these parameters whatever you want (since we'll be defining them in the JavaScript we'll write to make these calls), but for example purposes we'll be assuming they'll be using the parameter names "key" and "plaintext". We can then write the output of this function to our HTTP response:

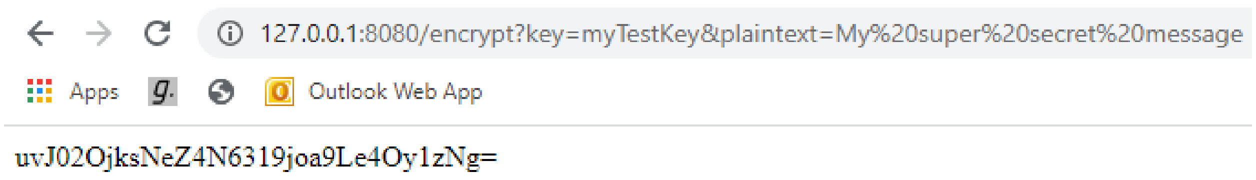
```
encryptedText = self.encode(myParams['key'], myParams['plaintext'])
self.wfile.write(encryptedText)
```

Note that we do not have to use the **`bytes()`** function in this case, as the encode() function returns the encrypted string as bytes already.

I leave it as an exercise to implement the same mechanics for the **`/decrypt`** page. This type of request should take a **key** and a **ciphertext** from the client, and then run it through the **decode()** function. **NOTE: The decode() function returns a string, not bytes, so you will need to**

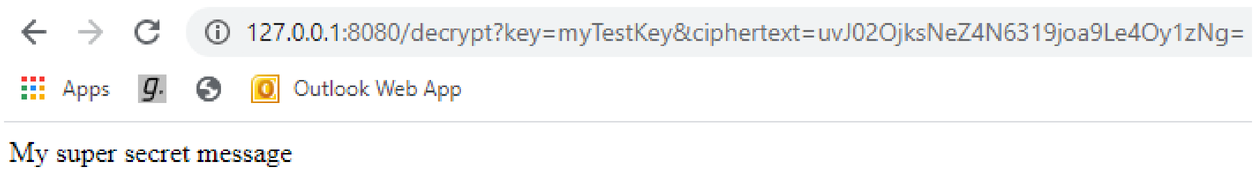
use the bytes() function to convert the string to UTF-8 encoded bytes before you write it to the HTTP response.

When you've completed this task, you should be able to go to test it in your browser, manually passing the correct query string parameters to the service and getting your encrypted/decrypted results back from the service:



```
127.0.0.1:8080/encrypt?key=myTestKey&plaintext=My%20super%20secret%20message
```

```
uvJ02OjksNeZ4N6319joa9Le4Oy1zNg=
```



```
127.0.0.1:8080/decrypt?key=myTestKey&ciphertext=uvJ02OjksNeZ4N6319joa9Le4Oy1zNg=
```

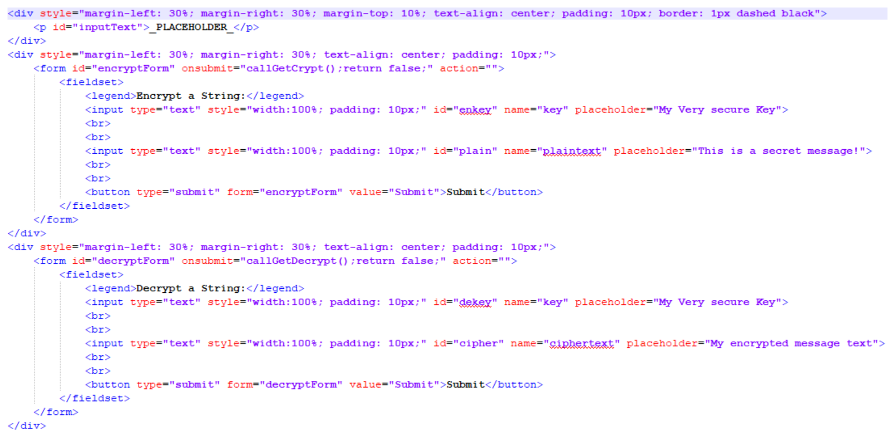
```
My super secret message
```

Once you've done that, we've finished the server-side of our Python microservice. Now, we can move on to adding in the client-side logic required for our microfrontend, which will actually interface with our microservice.

Using Asynchronous Calls

Now you should open your encryptionBase.html file. I would recommend using an editor like Notepad++ for this section of the lab, or some other editor that provides syntax highlighting, although this can be done in Notepad or other basic editors.

Initially, our file will look like this:



```html
<div style="margin-left: 30%; margin-right: 30%; margin-top: 10%; text-align: center; padding: 10px; border: 1px dashed black">
    <p id="inputText">_PLACEHOLDER_</p>
</div>
<div style="margin-left: 30%; margin-right: 30%; text-align: center; padding: 10px;">
    <form id="encryptForm" onsubmit="callGetCrypt();return false;" action="">
        <fieldset>
            <legend>Encrypt a String:</legend>
            <input type="text" style="width:100%; padding: 10px;" id="enkey" name="key" placeholder="My Very secure Key">
            <br>
            <br>
            <input type="text" style="width:100%; padding: 10px;" id="plain" name="plaintext" placeholder="This is a secret message!">
            <br>
            <br>
            <button type="submit" form="encryptForm" value="Submit">Submit</button>
        </fieldset>
    </form>
</div>
<div style="margin-left: 30%; margin-right: 30%; text-align: center; padding: 10px;">
    <form id="decryptForm" onsubmit="callGetDecrypt();return false;" action="">
        <fieldset>
            <legend>Decrypt a String:</legend>
            <input type="text" style="width:100%; padding: 10px;" id="dekey" name="key" placeholder="My Very secure Key">
            <br>
            <br>
            <input type="text" style="width:100%; padding: 10px;" id="cipher" name="ciphertext" placeholder="My encrypted message text">
            <br>
            <br>
            <button type="submit" form="decryptForm" value="Submit">Submit</button>
        </fieldset>
    </form>
</div>
```

Consisting of just some HTML for our encryption and decryption forms, and the placeholder box that will serve to hold the output of our microservice.

The first thing we'll need to do is add a set of `<script></script>` tags at the end of our file, which will hold the JavaScript we use to communicate with our microservice. Note that since this file is served by our microservice (when we visit the root '/' page), we're also serving this JavaScript to the client at the same time.

Now, we'll design our first asynchronous JavaScript HTTP request. We'll begin by defining a function called **`getEncrypt()`**, which will take a parameter called **`callback`**, which is the function we'll run when our request is complete and we've received a response:

```
function getEncrypt(callback){

}
```

The first thing we'll need to do in our function is fetch the current values that the user has entered in the key and plaintext form fields. The encryption key input has an ID of **`enkey`**, while the plaintext input has an ID of **`plain`**. We can use the **`getElementById()`** function to fetch their values:

```
function getEncrypt(callback){
        var key = document.getElementById("enkey").value;
        var plaintext = document.getElementById("plain").value;
```

Next, we'll prepare our request. We'll create a new **`XMLHttpRequest`** object, and then we'll define what we want to do when that request has completed. We'll use the `readyState` variable to see if the request is complete – a readyState of 4 means the request is finished. We'll also check the HTTP status code to make sure it's 200, meaning our request was returned OK with no errors. If the request is done and returned an OK status, we'll pass the response content to our callback function:

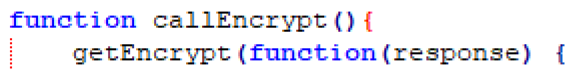
```
xmlHttp.onreadystatechange = function(){

    if (xmlHttp.readyState == 4 && xmlHttp.status == 200){
        callback(xmlHttp.responseText);
    }
}
```

Once we've defined what to do when the request is done, we'll move on to actually opening the request and sending it. We'll need to specify the type of request we're making, which is **GET**. We'll also need to give it a URL to send the request to – for the encryption form, this will be our **/encrypt** page, plus a query string we'll build using the key and plaintext variables. **Note that the names you use in your query string here need to match the parameter names your Python server code is using with the `getParams()` function**.

```
xmlHttp.open("GET", "http://127.0.0.1:8080/encrypt?key=" + key + "&plaintext=" + plaintext, true);
xmlHttp.send('null');
```

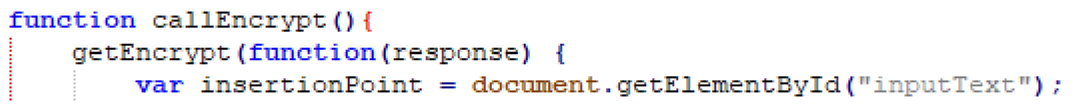
The URL is built first with our local IP and port (**127.0.0.1:8080**), the page we're trying to reach (**`/encrypt`**), and then our query string. In this case, we're using the parameter names **key** and **plaintext**, which match what we used in our Python code. We use simply string concatenation to place the content of our JavaScript variables into the query string. The final parameter passed to the **open()** function is whether this is an asynchronous request or not. We want to use an asynchronous request, so we'll pass in **true**. We then call the **send()** function to send our request to the microservice.

Next, we'll define a second function called **callEncrypt()**. This function will actually call our **getEncrypt()** function, and also pass it our callback function. We create this callback function as an anonymous function without a name, which takes the response content in as a parameter:



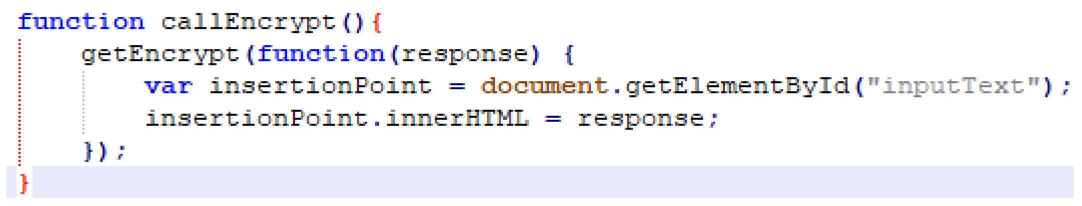
```
function callEncrypt(){
    getEncrypt(function(response) {
```

Within the callback function, we'll want to fetch a reference to the **inputText** paragraph object, which contains the **_PLACEHOLDER_** value, and will be used to write our service output to. We don't use **value** on it though, because we want a reference to the actual object itself, not it's contents:



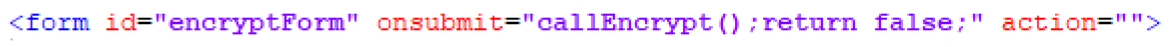
```
function callEncrypt(){
    getEncrypt(function(response) {
        var insertionPoint = document.getElementById("inputText");
```

Finally, we'll take our response text (which is simply our now-encrypted string returned from the microservice), and we'll use **innerHTML** to write it as the content of the inputText paragraph:



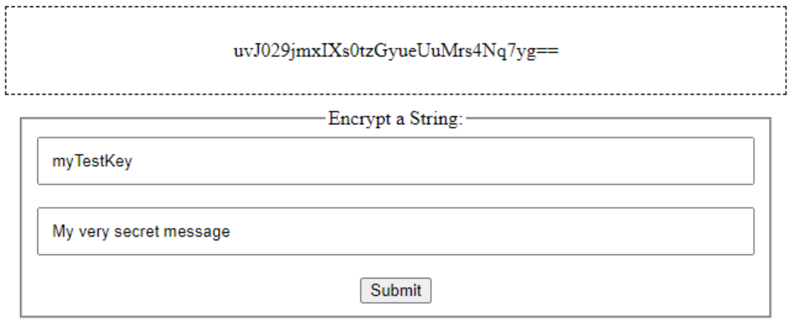
```
function callEncrypt(){
    getEncrypt(function(response) {
        var insertionPoint = document.getElementById("inputText");
        insertionPoint.innerHTML = response;
    });
}
```

The **callEncrypt()** function will be called when the user submits the encryption form on the page. We can do this by specifying the function name in an **onsubmit** parameter in our form. The `encryptionBase` file already has these parameters, but you'll need to modify them: replace the callGetCrypt() function with our new `callEncrypt()` function:



```
<form id="encryptForm" onsubmit="callEncrypt();return false;" action="">
```

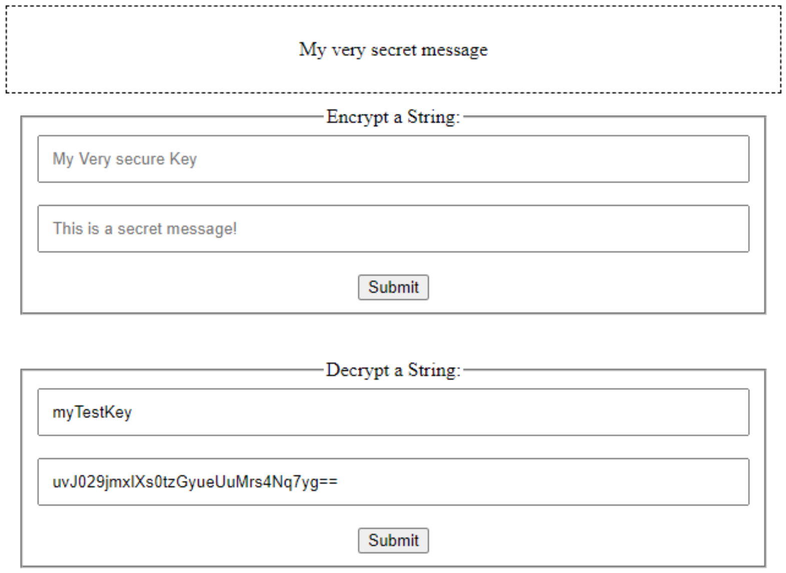
Once you've done that, make sure you save your HTML file, and then try going to the root page ('/') of 127.0.0.1 in your browser again. You should now be able to enter a key and a plaintext into the encryption form, and when submitted, the encrypted text should appear in the placeholder box above:


uvJ029jmxIXs0tzGyueUuMrs4Nq7yg==

If you are encountering errors or issues, you'll want to check both your Python console (for server-side issues) and the inspector console in your browser (I recommend Chrome or FireFox for this lab, do not use Edge) to check for JavaScript errors.

Once you've got the encryption side working, your exercise is to do the same for the decryption side. The JavaScript functions are largely the same, but you will need to change the **ids** you're pulling from as appropriate (for example, fetching the value of **`dekey`** rather than **`enkey`**), and you'll want to send your request to the **`/decrypt`** page rather than the **`/encrypt`** page. Again, make sure that the parameters you use in your query string match those expected by the Python service, and don't forget to edit the **`onsubmit`** parameter for the decryption form.

When complete, the decryption form should take an encrypted string and a key, and the decrypted output should be returned in the `inputText` box, just like the encrypted output:


My very secret message

Once you've made sure you're encryption and decryption forms are working appropriately, you've completed this lab. If you want a full walkthrough, I have recorded the L02 session where I did a complete walkthrough of the lab, which is available in the course Moodle.

Submission

There is no required submission for this lab. We will be continuing with this code base for lab 3, and marking will be performed when the code base is complete. The rubric below applies, but will be integrated with the lab 3 marking:

Marks	Item
1 Mark Each	Request handlers for ‘/’, ‘/encrypt’, and ‘/decrypt’
1 Mark Each	Completed JS for the getEncrypt() and getDecrypt() functions
0.5 Marks Each	Completed JS for callEncrypt() and callDecrypt()
2 Marks	Working service and frontend with no errors