COSC 331
Lab 3: Building a Multi-Service Frontend and Working with JSON

In this lab, we'll be slightly modifying our service from Lab 2 to include some JSON data output. We'll then show our encryption service, our styling service (from lab 1) and our JSON data output all in a single frontend. This will require interfacing with a Java Service (the styling service), a Python service with a microfrontend (our lab 2 code), and parsing JSON data with JavaScript. To get started, download the **frontend.html** file from the course Moodle or GitHub. **I'll be doing a walkthrough for this lab during our lab blocks.**

Setting up JSON on our Python Service

The first thing you should do is open up the Python script (**pythonBase.py**) that you built in Lab 2. We will modify our server script to keep a running history of encrypted strings, which we'll be able to fetch from the **/history** page as a JSON list.

Our first step is going to be importing the **json** library into our Python file. The **json** library is a built-in module in Python and doesn't require installation; it can be directly imported:

```
import json
```

The import statement should go at the top of the Python file along with the rest of your imports. After that, we'll need to create a new variable inside our **MyServer** class called history. We'll initialize it as an empty list using **[]**:

```
class MyServer(BaseHTTPRequestHandler):
    history = []
```

We can then reference the **history** variable within the **MyServer** class by prepending it with the **self** keyword. Inside of your **if** statement for the **/encrypt** page (inside your **do_GET** function), we're going to take our encrypted text and append it to the list as a string. To do so, we'll need to convert it to utf-8 encoding first, as the **MyServer.encode()** function returns a bytes object. We can do this all in one line, like so:

```
self.history.append(encryptedText.decode("utf-8"))
```

In this case, the **encryptedText** variable should be the stored output of the **self.encode()** function, which returns our encrypted text. Now, whenever a user encrypts a string, the encrypted output is appended to the **history** list before it is returned to the client.

Now, we'll add a way to access that **history** list, by converting it to a JSON object. First, we'll go ahead and add another **if** statement to our **do_GET** function:

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

This is in effect adding another page to our microservice, which can be found at http://127.0.0.1/history. Inside of this **if** statement we'll call a couple nested functions. First, we'll use **json.dumps()** to convert our **history** object into a JSON object:

```
json.dumps(self.history)
```

Then, we can wrap it in a **bytes()** function to convert it to utf-8 encoded bytes, and then send it to the client using **self.wfile.write**, like we did in the previous lab when sending data back to the client:

```
self.wfile.write(bytes(json.dumps(self.history), "utf-8"))
```

Once you've done that, try running your server script. Try encrypting several strings, and then go to the history page. You should see output that looks similar to this:

As you can see, our encryption output is being returned as a JSON array of strings. Something to note here is that **restarting your server clears the history**. Because the history variable is always initialized to empty when we start, and we don't store the history in a file or anywhere else that is persistent when our server stops, the history is only kept for the current duration of the service's run time.

Once you've hit this point, we're ready to move on to the next step.

Interfacing with our Java Microservice

First, we'll begin by integrating our Java Microservice into the frontend. This is fairly simple, since all we're really doing is changing the **href** of our stylesheet link, the same as we did in the **demo.html** file in Lab 1 – the only modification is some styling. Open up the **frontend.html** file, and you'll find this HTML to start with:

```
<html id="container" style="height: 95%;">
    <head>
        <meta charset="UTF-8">
        <link id="styleLink" rel="stylesheet" href="http://127.0.0.1/" type="text/css"/>
    </head>
    <body>
        <div id="encryptionService">
        </div>
        <div id="historyService" style="height: 100px; overflow: auto; border: 1px solid
        </div>
        <div style="position:absolute; top:0; right:0; text-align: center;">
            <!-- Your Lab 1 Form block is going to go here -->
        </div>
    </body>
</html>
```

Then, open up your **demo.html** file from Lab 1. Copy and paste the entire **<form>** element block into **frontend.html** where the comment is. Remove any references (inputs, labels, etc) to the **accent** value – we won't be using it:

```
<form id="cssForm" onsubmit="fetchCSS();return false;" action="#">
    <fieldset>
        <legend>Styling Options:</legend>
        <label for="main">Main Colour:</label>
        <input type="color" id="main" name="main" value="#ff0000">
        <br>
        <label for="font">Font:</label>
        <select name="font" id="font">
          <option value="serif">Serif</option>
          <option value="sans-serif">Sans-Serif</option>
          <option value="monospace">Monospace</option>
        </select>
        <br>
        <button type="submit" form="cssForm" value="Submit">Submit</button>
    </fieldset>
</form>
```

Do **not** copy the **div** elements from your **demo.html**, just the **cssForm** block. Then, copy over the **script** element from your **demo.html** file. Remove any references to the **accent** value, as we did with the form:

```
<script>
    function fetchCSS(){
        var main = document.getElementById("main").value.substring(1);
        var font = document.getElementById("font").value;
        var container = document.getElementById("container");
        var styleSheet = document.getElementsByTagName("link");
        container.style.display='none';
        styleSheet[0].href = "http://127.0.0.1/?main="+main+"&font="+font;
        setTimeout(function(){
            container.style.display='initial';
        }, 2000);
    }
</script>
```

Place the script block inside the **<head>** element of your **frontend.html** file, just like it was in **demo.html**. Save your file. Start up your Java styling service from Lab 1 and make sure it's running properly, then try opening your **frontend.html** file in Chrome, Firefox, or another non-Edge web browser. You should see a page that looks similar to the following:

You should see an empty white box, and our styling options form at the upper-right of the page. Your styling form should work, allowing you to change the font and background color of the page by hitting the submit button.

Interfacing with our Python Microfrontend

Interfacing with our Python service is a little trickier. Recall that our microservice returns a microfrontend – a complete HTML object with our included JavaScript for the necessary asynchronous HTTP calls. This means we need to retrieve this HTML and in-line JavaScript, and put it into our current page. What makes this difficult is that you cannot simply inject JavaScript into the DOM and expect it to work; your injected JavaScript won't be recognized, and calls to your functions will return an error that the function is undefined. This means we'll have to use a slightly different strategy to handle our JS.

First, we're going to define a generic asynchronous HTTP call method, which we'll be able to use for two different functions. It will take a URL to send the request to, and a callback function to execute. This is very similar to the asynchronous JS calls that we did in Lab 2, albeit with an extra parameter defined in our function, which we pass to the **xmlHttp.open()** function as our URL:

```
function httpGetAsync(url, callback)
{
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.onreadystatechange = function() {
        if (xmlHttp.readyState == 4 && xmlHttp.status == 200)
            callback(xmlHttp.responseText);
    }
    xmlHttp.open("GET", url, true);
    xmlHttp.send(null);
}
```

The benefit of this is that we don't need to duplicate our asynchronous request code for every different type of request we make: we can just write a custom callback function, and then pass

our URL and the callback to this function to perform our GET request. **This code should go in the <script> after your fetchCSS() function**.

Now we'll make our first asynchronous call to the Python application. This doesn't need a function, as we'll only be running it once – when we first start loading the page. This will fetch the base HTML and the JavaScript from our Python service. This all goes inside the **<head>**

We start by calling our **httpGetAsync()** function that we just created. This also goes inside the **<script>** tags, after your **httpGetAsync()** function. We'll pass it the base URL for our Python service, which is what returns the HTML and JS, as well as an anonymous function which catches a **response** from the asynchronous request:

```
httpGetAsync('http://127.0.0.1:8080/', function(response) {
    // Our callback code here
});
```

Inside this function, we need to do the following things:

- Split the JS (**<script>**) and HTML (** **'s) elements from each other
- Insert our HTML component into the **encryptionService** div
- Create a new **<script>** element and insert our raw JS into it that we separated out
- Append the new **<script>** element to the **<head>** of our page

These steps are necessary to ensure that the JS that we inject into the page can actually be run, otherwise our encryption service won't work properly.

To start, we'll use a **DOMParser()** object to parse the HTML and inline JS that we received as a response from our Python microservice. We can then use the resultant **document** object to fetch pieces of the response, like our **<script>** section:

```
var doc = new DOMParser().parseFromString(response, "text/html");
var code = doc.getElementsByTagName('script')[0].innerHTML;
```

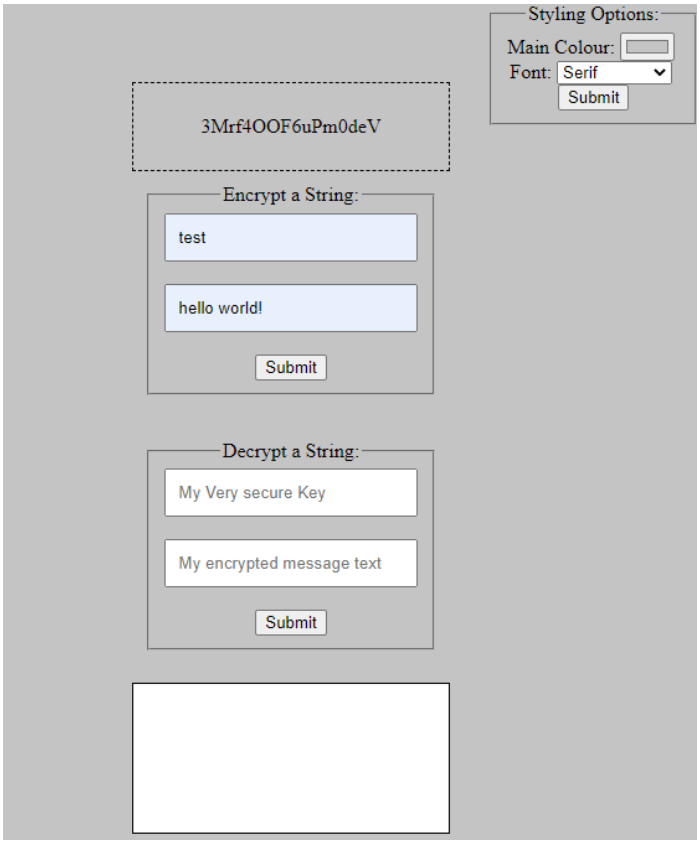
We'll then remove the **<script>** segment from our **doc** object, so that it now only contains the HTML for the encryption service. We'll then insert all of that HTML into the **encryptionService** div:

```
doc.getElementsByTagName('script')[0].remove();
var insertionPoint = document.getElementById("encryptionService");
insertionPoint.innerHTML = doc.body.innerHTML;
```

Finally, we'll need to create a new **<script>** element node and an accompanying text node, which will contain the raw code within the **<script>** element. We'll then append these together, before append the entire thing to the **<head>** of our document:

```
var scriptNode = document.createElement("script");
var scriptContent = document.createTextNode(code);
scriptNode.appendChild(scriptContent);
document.head.appendChild(scriptNode);
```

Our code for interfacing with our Python microfrontend is now done. Save and make sure that your Python service is running, and then try visiting your page again. You should now see a page that looks like this, with our encryption service now available alongside our styling:



Verify that you are able to encrypt and decrypt a string using the encryption service, and that you are able to style the page using the styling options. Note that the font styling option will **not** change the font of the form inputs and placeholders, depending on which browser you are using (confirmed it does **not** in Google Chrome) – that is fine. You should be able to use both services on this page without them interfering with each other or any errors occurring.

Fetching Our History Automatically

Now we'll write the last part of our frontend: the bit that interfaces with the history function of our Python microservice. This means we'll need to fetch and parse some JSON using JavaScript.

To start, we'll want our history to update automatically. It doesn't have to update immediately, but can instead do what is known as "long-polling", which means it automatically makes a request on a timer, usually using a fairly long time (we'll use every 30 seconds).

We can do this easily using the **setInterval()** function, which we pass an anonymous function (which contains the code we want to run), and a time in milliseconds (1000ms = 1s):

```
const interval = setInterval(function() {
        // Code to run every 30 seconds
}, 30000);
```

As with the rest of the code in this frontend, this should go in the **<script>** element in the **<head>**, after the rest of your script. Inside of anonymous function, we'll use the **httpGetAsync()** function again to make a call to **http://127.0.0.1/history**, where our microservice is serving our history data as a JSON array:

```
httpGetAsync("http://127.0.0.1:8080/history", function(response) {
    // Our history callback processing goes here
});
```

Inside of this function, we're going to first grab the **div** object that our history data will be printed to, which has the id **historyService**:

```
var myDiv = document.getElementById("historyService");
```

Then, we're going to parse our response data from the asynchronous request, using the **JSON.parse()** function that is built into JavaScript:

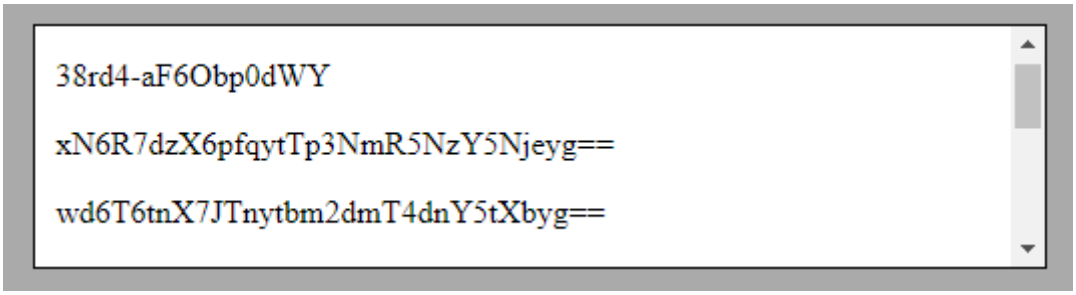
```
var jsonobj = JSON.parse(response);
```

This will parse the raw string JSON response into an array of strings that we can then iterate over to "build" our history HTML. This is simply done using string concatenation, and then setting the **innerHTML** value of our **div** element to the string that we have generated:

```
var myString = "";
for (obj in jsonobj){
    myString += "<p>" + jsonobj[obj] + "</p>";
}
myDiv.innerHTML = myString;
```

So, to recap: We have our **setInterval()**, which is in turn performing a call to the **httpGetAsync()** function every 30 seconds. Inside of our **httpGetAsync()** anonymous callback function, we're taking the response, parsing it with the **JSON.parse()** function, and then iterating over the resulting array to produce our output, which we then write to our page.

Once you've finished this, save the file and refresh the page in your browser. You should notice that the empty white box at the bottom of the page, after about 30 seconds, will be populated with any encrypted strings that have been generated since the server started.



Since this div is updated every thirty seconds, you will notice a delay between when you encrypt a new string, and when it appears in the div.

Submission

When you are done, submit your finished **Python service code, Java service code, and frontend HTML file on Moodle**. Marks will be given according to the Lab 2 marking schema (as seen in the Lab 2 document), as well as this schema for Lab 3:

Marks	Item
2 Marks	Implement the history function in your Python service
1 Mark	Styling service is properly integrated into frontend
2 Marks	Python encrypt/decrypt service is properly integrated into frontend
1 Mark	Python history service is properly integrated into frontend