COSC 331 Lab 5: Load Balancing with Client-Side JavaScript

In this lab, we'll be building a simple **round-robin static load balancer** that uses client-side JavaScript to point the client towards one of several service instances. We'll use a test page that generates many requests, combined with console logging, to see the results of a round-robin load balancer. This lab will combine some of the techniques we learnt in working with **microservice frontends**, while also teaching us about how a round-robin load balancer behaves when there is an **uneven load** being applied to our services.

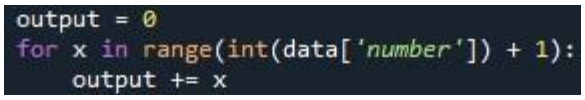
Setting Up our Math Service

In order to demonstrate the round-robin load balancing, we need a function that can take sufficiently long that our servers may start to bog down. We'll calculate **triangle numbers**, which are iterative sums. For a given input number (i.e. 5), a sum is calculated of every normal number such that 0 <= x <= input. For example, the input of 5 results in the calculation 5 + 4 + 3 + 2 + 1 = 15. There are more efficient ways of calculating these numbers than iteration, but we want the function to slow down when performing large calculations, to help with our load simulation.

Inside the **mathservice.py** file, we need to define our **do_GET** function. Inside this function, we will:

- Retrieve a GET parameter named **'number'**
- Print out a message to console with the parameter
- Use a for loop to calculate the **triangle number** of the given input number
- We will then set our headers, and return the triangular number to the client

To get the parameters, we'll use the **self.getParams()** method that we've used in previous labs. To calculate the triangle numbers, we can use a simple for loop to perform the sum:



```
output = 0
for x in range(int(data['number']) + 1):
    output += x
```

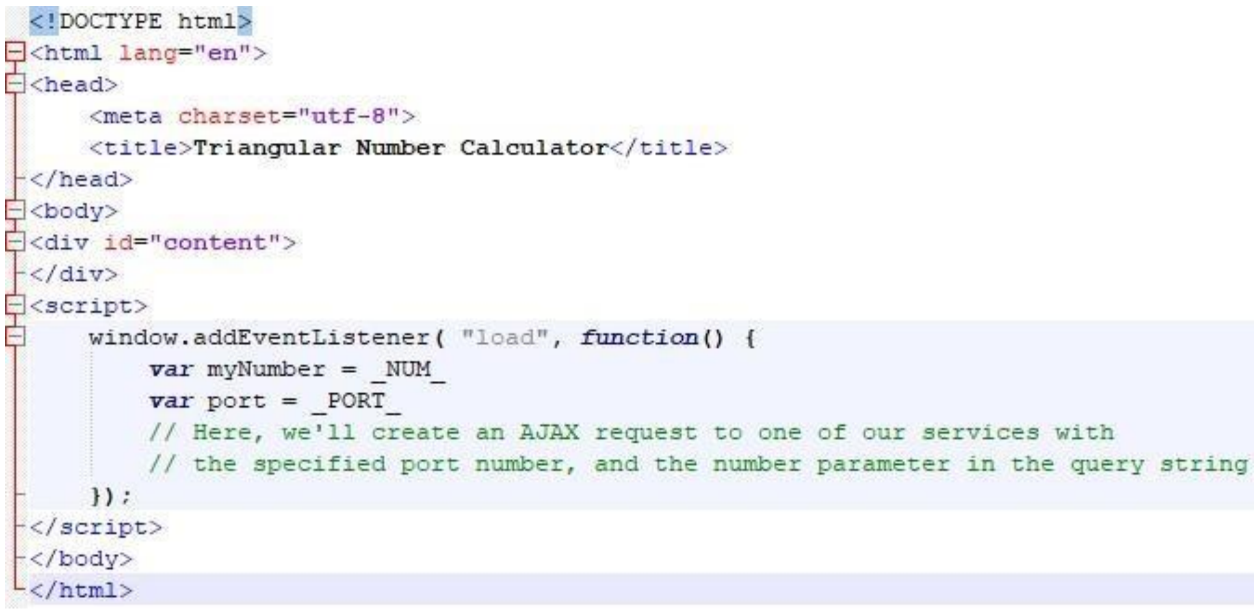
Note the port this service is running on: 8080. You can test the service by visiting your localhost in browser at port 8080, and passing in the number parameter:

http://127.0.0.1:8080?number=25

Once you have a started instance of the Math service, try changing the port number to 8081, and starting a second instance. Verify in your browser that you can now access both instances, **one through port 8080, and one through port 8081**.

Building the Skeleton HTML

Our load balancer will return a skeleton HTML page, which will in turn execute JavaScript to actually make a request to the **math service**. The **skeleton.html** template looks like this:

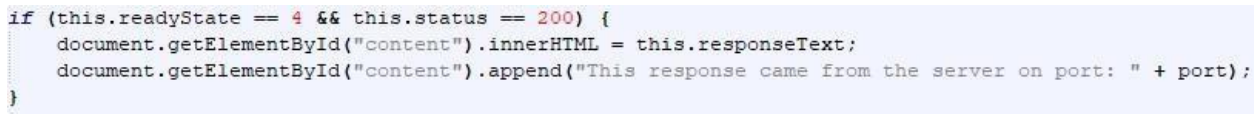


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Triangular Number Calculator</title>
</head>
<body>
<div id="content">
</div>
<script>
    window.addEventListener( "load", function() {
        var myNumber = _NUM_
        var port = _PORT_
        // Here, we'll create an AJAX request to one of our services with
        // the specified port number, and the number parameter in the query string
    });
</script>
</body>
</html>
```

Note the **_PORT_** and **_NUM_** flags. These indicate where we will be inserting data into our HTML (or more specifically, our JavaScript) when we send the HTML to the client. The **myNumber** variable will represent the number the user wants to send to the math service, and the **port** variable is the port of the service that the load balancer wants us to connect to.

With this in mind, we need to build an AJAX request that will make a request to 127.0.0.1, on the specified port, with the **myNumber** as a GET parameter named **number**. This should be the same kind of URI signature as you tested earlier to verify your math service was working.

When our AJAX request is complete, we'll want to append the response content (the message from the math service) to our **content** div, along with a little message specifying which service instance (port) we retrieved it from:



```
if (this.readyState == 4 && this.status == 200) {
    document.getElementById("content").innerHTML = this.responseText;
    document.getElementById("content").append("This response came from the server on port: " + port);
}
```

We need to send this request to the port the will be specified by the **port** variable, while making sure the **myNumber** variable is included as a GET request parameter. I leave this task up to you – we have worked with asynchronous requests a number of times before, so you can check your prior labs (or even the old testing pages) for examples.

Once this is done, make sure you save your skeleton HTML file. It should be in the same folder (or nearby) to your **balancer** file.

Building the Balancer Service

The **balancer.py** service will load our **skeleton.html** into a string variable, and then replace the **_PORT_** and **_NUM_** flags with the correct service port and number parameter. We'll fetch the number from the **number** parameter in the GET request, and then we'll open up our **skeleton.html** file. We'll read it into a string.

We'll then use the **replace()** function to replace the **_PORT_** and **_NUM_** flags:

```
html = html.replace("_NUM_", data['number'])
html = html.replace("_PORT_", ports[0])
```

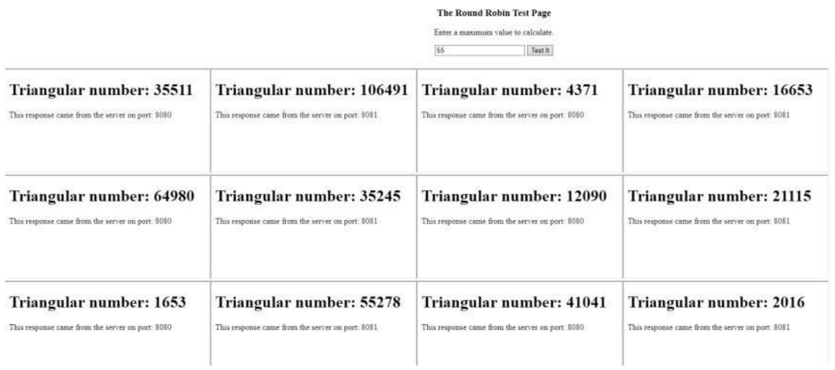
The **ports[]** list is defined at the top of the **balancer.py** server file, above the class declaration. This is a list of valid ports (service instances) that traffic can be directed to. If you have your math service running on both **8080** and **8081**, add those ports to your port list, as strings:

```
ports = ["8080", "8081"]
```

We use the first port in our list, because we'll treat our list as a **queue** of sorts. Immediately after we replace the **_PORT_**, we'll rotate the list by **pop()**'ing off the zero-index element, and then re-appending it, moving it to the back of the list. This is what provides the **round-robin** behaviour – each server instance port is used, then sent to the back of the line to wait it's turn again. Afterwards, we just send our modified **html** back to the client, setting our headers as before.

Testing out the Load Balancer

The **loadTester.html** page is a basic page that allows you to test out your load balancing service. It takes a number input. When the "test" button is pressed, it will open up twelve new iframes, each making a request to your load balancer:



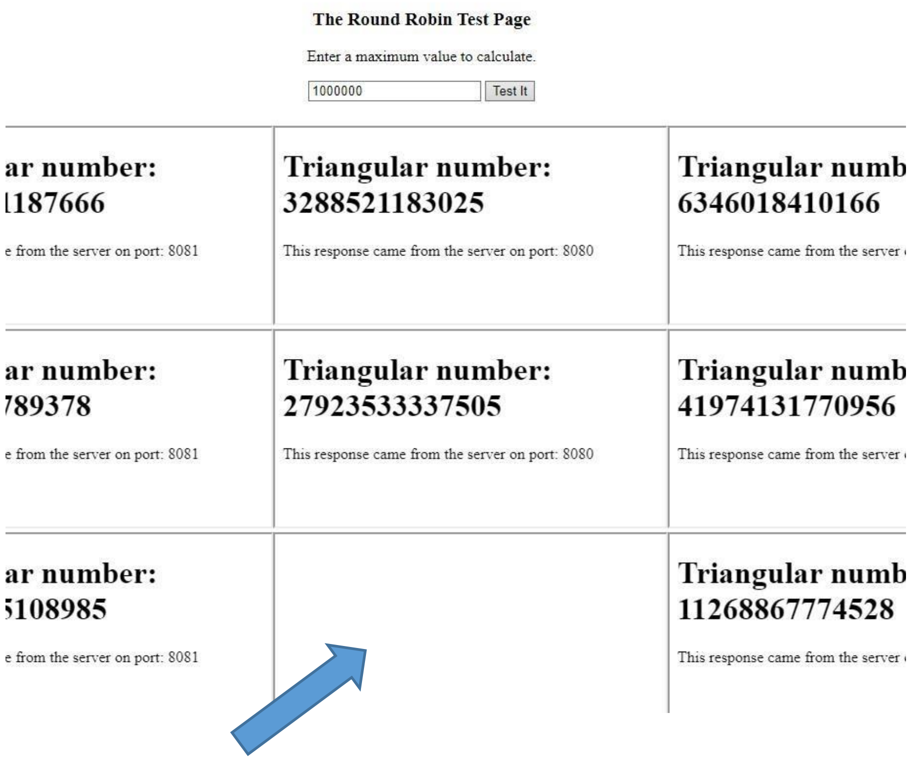
The Round Robin Test Page

Enter a maximum value to calculate.

55 [Test It]

Triangular number: 35511 This response came from the server on port: 8080	**Triangular number: 106491** This response came from the server on port: 8081	**Triangular number: 4371** This response came from the server on port: 8080	**Triangular number: 16653** This response came from the server on port: 8081
Triangular number: 64980 This response came from the server on port: 8080	**Triangular number: 35245** This response came from the server on port: 8081	**Triangular number: 12090** This response came from the server on port: 8080	**Triangular number: 21115** This response came from the server on port: 8081
Triangular number: 1653 This response came from the server on port: 8080	**Triangular number: 55278** This response came from the server on port: 8081	**Triangular number: 41041** This response came from the server on port: 8080	**Triangular number: 2016** This response came from the server on port: 8081

The load tester generates a different request each time, requesting a random number between 0 and the user-inputted value. This means each request will take a varying amount of time to complete, depending on how many iterations of calculation are required. As the maximum number size is increased, requests will take longer to complete, and you may notice a "patchwork" effect, as one or more of the service instances lags behind for a particularly "big" calculation:


The Round Robin Test Page

Enter a maximum value to calculate.

1000000 Test It

ar number: 1187666 e from the server on port: 8081	Triangular number: 3288521183025 This response came from the server on port: 8080	Triangular numb 6346018410166 This response came from the server
ar number: 789378 e from the server on port: 8081	Triangular number: 27923533337505 This response came from the server on port: 8080	Triangular numb 41974131770956 This response came from the server
ar number: 5108985 e from the server on port: 8081		Triangular numb 11268867774528 This response came from the server

Try to capture this behaviour in a **screenshot**.

Submission

Submit your **skeleton.html, balancer.py**, **mathservice.py,** and your **screenshot** files on Moodle. Marking schema is provided below:

Marks	Item
2 Marks	Complete Math Service – calculates triangle number correctly
3 Marks	Working load balancer service – redirects traffic to multiple

	ports, passes data into HTML
1 Marks	Working skeleton asynchronous request
1 Mark	Screenshot of nonconsistent load on services (one ahead of other)