How to build your own client-server set up to conduct scientific experiments online

HTML along with JavaScript(JS) provide a powerful platform for writing applications to conduct scientific surveys and studies. With the introduction of WebGL and it's integration with the browsers, 2D and 3D graphics related experiments can also be written in JS + HTML. But conducting an experiment requires collection and analysis of the experiment data and this can be handled by a server. Thus, the browser would run your experiment or 'client' code and the server would manage the collection and storage of the experiment data. There are programs available that provide a platform to write/design experiments, host the experiments online and gather the observations data back. These services often maintain their servers for hosting and maintaining user's experiments and charge a fee depending upon the usage. Sometimes, the user is also limited to use the modules provided by their UI to write their experiments. But the experiment may demand something out of the available options and the researchers may prefer more control over the observations. Here is a detailed approach that may help researchers to create, host and maintain their own experiments.

Creating experiments

JS is the programming language that was written to bring flexibility and add features in any web applications. There are also modules and apis that provide image processing (OpenCV.js) , psychophysics (jspsych) and 3D graphics(threejs) functionality in the web applications. A researcher has all the flexibility to design his/her own experiment in JS using suitable external modules. Once, the experiment is written, it can be sourced in a .html file and run it on any browser. We would call this .html as the client code because this piece of code contains the experiment that each observer or 'client' is going to undertake. The client code must also have written 'POST' calls to communicate the required observation data back to the researcher.

Creating the Server

Once we have our client code, we would need a server to collect and store the data acquired by the client code from different observers. A server is a piece of code that runs a client code on a specific address and collects the data sent back by each client. There are different programming languages that could be used to write the server code. However, we describe building a web server using Python because it is relatively simpler to understand and use.
Flask is a web framework in Python that helps us write web applications. We would use Python along with the Flask module to write our server. Flask can be installed as:

Pip install Flask

A simple python-Flask server can be written as:

```
from flask import render_template, Flask

app = Flask(__name__)

@app.route('/')
def main():
        return render_template('experiment.html')

if __name__ == '__main__':
        app.run()
```

Where, ‘app’ creates an instance of Flask for our server. ‘Render_template’ runs the given client code (.html file) at the specified url (in this example it’s the root ‘/’). ‘app.run()’ starts this server and by default, the client code can be accessed by typing ‘http://localhost:5000’ in the browser. For multiple experiment pages, the researcher can assign different urls for running different code:

```
@app.route('/exp1')
def main():
        return render_template('experiment1.html')

@app.route('/exp2')
def main():
        return render_template('experiment2.html')
```

The researcher can also run the server on any other system and ip in the network by specifying its IP and port:

```
app.run(‘xx.xx.xx.xx’, port:80)
```

There are extensive ways of customizing the code according to each researcher’s need. Many experiments require to show 3D content that uses the client’s device data. The browsers don’t allow these applications if they are not run using secured protocol. Thus, we would need to run our applications over https rather than the default http. The simplest way to do it is to use an on the fly certificate:

```
        app.run(ssl_context='adhoc')
```

This would run the application over https at : ‘https://localhost:5000/’. It would run the graphics and the content that require secure protocol but the browser would always complain the first time anyone runs it. It does so because it does not recognize the on the fly certificate as one of the pre- stored trusted certificates. Also, once we have our experiment and a server to manage our experiment, we would need to host the server somewhere that would be available to any other observer over the Web. Next, we discuss our approach to solve these problems one by one:

1. **Hosting the server**
   As convenient as it appears, the first option to host the server is to do it on our own system. In such a case, we would have to make sure that our IP is accessible over the network. For usability, we might also require to acquire a domain name. And most importantly we would need to keep our systems running 24x7 and handle all the security issues. However achievable, we would not talk much about it as our approach makes use of cloud services. Currently, there are many commercially available computing services for example Microsoft Azure and AWS. While, generally there are some charges for such resources, most of these services provide free resources or credits for students and other education professionals. We used Microsoft Azure as our computing platform to host our server. There are different well documented steps to setup an account with one of the service providers and setup the computing resource. In the outcome, we must have a Virtual Machine on which we can deploy our server. If not present by default, add an inbound port rule in the 'networking' section of your VM to allow SSH service so that we can access the VM from the terminal. After deploying our server and client code on the server, we can run our server and access the application from our local system by typing 'https://<domain_name>:5000' in our browser. The on the fly generated certificate may or may not work on the browser. Even if it worked, it would give a long warning page that may annoy users. In order to make web browsers trust and run our application, we need to generate a certificate from Certificate Authority (CA).

2. **Generate server certificate using CA**
   Lets Encrypt is one of the CAs and has an automated process to generate certificates. It requires us to install a tool called **certbot**. Link https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https explains the process in much more detail. This tool requires a static root directory for verification but Flask does not have one. Hence, we use a reverse proxy server called **Nginx** to map a private directory to be used by certbot for verification of the application.

3. **Serving Flask application using Nginx**
   Here is a detailed description of steps involved in the configuration and setup of Nginx and required supporting tools - https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uswgi-and-nginx-on-ubuntu-18-04. Flask is based on Web Server Gateway Interface(WSGI) standard. When we ran our server and client code locally, the client code directly communicated over a network port to the WSGI based Flask server. After using Nginx as a reverse proxy server, now the client requests would go to Nginx first and Nginx needs to transfer these requests to our server. uWSGI acts as an interface between these requests and our application server. We configure a Unix socket which is used by Nginx to transfer requests to uWSGI. We also create a WSGI entry point to tell uWSGI to run the experiment page from our application Flask server when requested. Lastly, we also need to add network inbound rules on our VM to allow our webserver at port 80 and https service at port 443.

Following all the steps we would have our own server hosting our experiments. The link can be shared to other observers to access our experiments over the web. The server would store or analyze data according to the written code. The stored data can be accessed on the VM or transferred to our local system as and when required.