HW 3 - Exploring Data Collection and Storage

Due May 18 by 11:59pm **Points** 110
Available Apr 27 at 12am - May 19 at 11:59pm 23 days

This assignment was locked May 19 at 11:59pm.

HW 3 - Data Collection, REST Endpoints and Storage

As with the last assignments, read the full writeup before starting!

For max points, start early - while we try to give you the information needed, as we go further in the course we are less step by step - this assignment is more open ended and will require quite a bit of external research. It is purposefully not as structured in its requirements, so the onus of this assignment is on you making framework/library/design decisions!

Also, please make sure you are checking your error logs and Googling error messages before asking for help! Self-sufficiency here is a skill we can hone, let's work on it!

Learning Outcomes

- The student will understand REST and the benefits of mocking.
- The student will be able to create a simple REST style endpoint for data ingestion and retrieval.
- The student will be able to configure a database and manage data in a rudimentary manner.
- The student will be able to build some basic visualization using the library of their choice.

Part 0 - Mocking the Endpoint

There are three primary parts to this assignment:

1. Collecting the data from users
2. Creating a database to store this data
3. Creating a REST endpoint to access/manage this data (The REST endpoint essentially being the interface for our database - you can read more about them here **https://www.sitepoint.com/developers-rest-api/** (https://www.sitepoint.com/developers-rest-api/)).

Before we dive head first into trying to make a full REST endpoint for our project, we should first mock one to understand how they work. To do this, first head over to your website's document root in `/var/www/example.com` or `/var/www/example.com/public_html` (wherever your main HTML files are) and install json-server as instructed here (**https://github.com/typicode/json-server** (https://github.com/typicode/json-server)) under **Getting Started** (only that one section). **Make sure your JSON file is called db.json!**

Once your JSON Server has been installed, use this command to give write access to `db.json` so it can be modified later

```
chmod 666 db.json
```

After the JSON Server is installed, you will need to expose the server on an endpoint that isn't `localhost` (http://localhost) to your droplet. To do that, use the following commands:

```
sudo ufw allow 3000
npm i pm2@latest -g
```

`ufw` allows you to modify the ports exposed on your firewall, so `allow 3000` lets you use port 3000.

Then, `pm2` will let you run node apps in the background so you can run them while not logged into your server. If you have trouble installing pm2 globally, you might need to elevate your permissions.

Now that we have `pm2`, we need to create a `server.js` file for it to run. You can use user **warapitiya**'s simple json-server code on the top post of this link **https://github.com/typicode/json-server/issues/253** (https://github.com/typicode/json-server/issues/253) . Once you have made that file, you can start it with:

```
pm2 start server.js
```

NOTE: The purpose of **pm2** is to be able to run node applications in the background easily. It can slow you down though if you are making and editing node applications and having restart your server all the time. In this case, it might be easier to use the command `node server.js` to run your applications in the foreground so you can see the logs easier, as well as start / stop your server easier. Then when you're finished making the application, you can use **pm2** to run it.

Now, your server should be up and running at ***yourwebsite.com:3000*** ***(http://yourwebsite.com:3000)***. To test it, you can make a simple GET request to ***yourwebsite.com:3000/posts*** ***(http://yourwebsite.com:3000/posts)*** and should get a small JSON packet in return.

While this works, having an exposed port in the URL is kind of ugly, so we should create a proxy route for us to pass this one through. To do that, use the following commands:

```
sudo a2enmod proxy
sudo a2enmod proxy_http
cd /etc/apache2/sites-available
sudo vim yourwebsite.com-le-ssl.conf
```

(if you haven't set up SSL yet, edit **yourwebsite.com.conf** instead)
Underneath DocumentRoot, add the following three lines:

```
ProxyPreserveHost On
ProxyPass /json http://localhost:3000
ProxyPassReverse /json http://localhost:3000
```

After those lines have been added, save and quit from the file. Restart your server using

```
sudo systemctl restart apache2
```

Now your JSON server should be all configured! **The base route for it is **yourwebsite.com/json (http://yourwebsite.com/json).****

So if you wanted to get the same posts like the earlier post, you would simply make a GET request for **yourwebsite.com/json/posts** **(http://yourwebsite.com/json/posts)**. To add data, make a POST request. To delete data, make a DELETE request.

Once you confirm that it does indeed work, you'll want to go back and remove the rules that open port 3000 for security reasons. To do that, do the following:

```
sudo ufw status
```

This will show all of the current UFW rules. Starting at the top with number 1 and incrementing for each rule down, count which number the 3000 rule is. Ignore any 3000 rule that has a colon after it. There should be two 3000 rules that don't have a colon, the second one having (v6) after.

Then for both, you will need to use the following command:

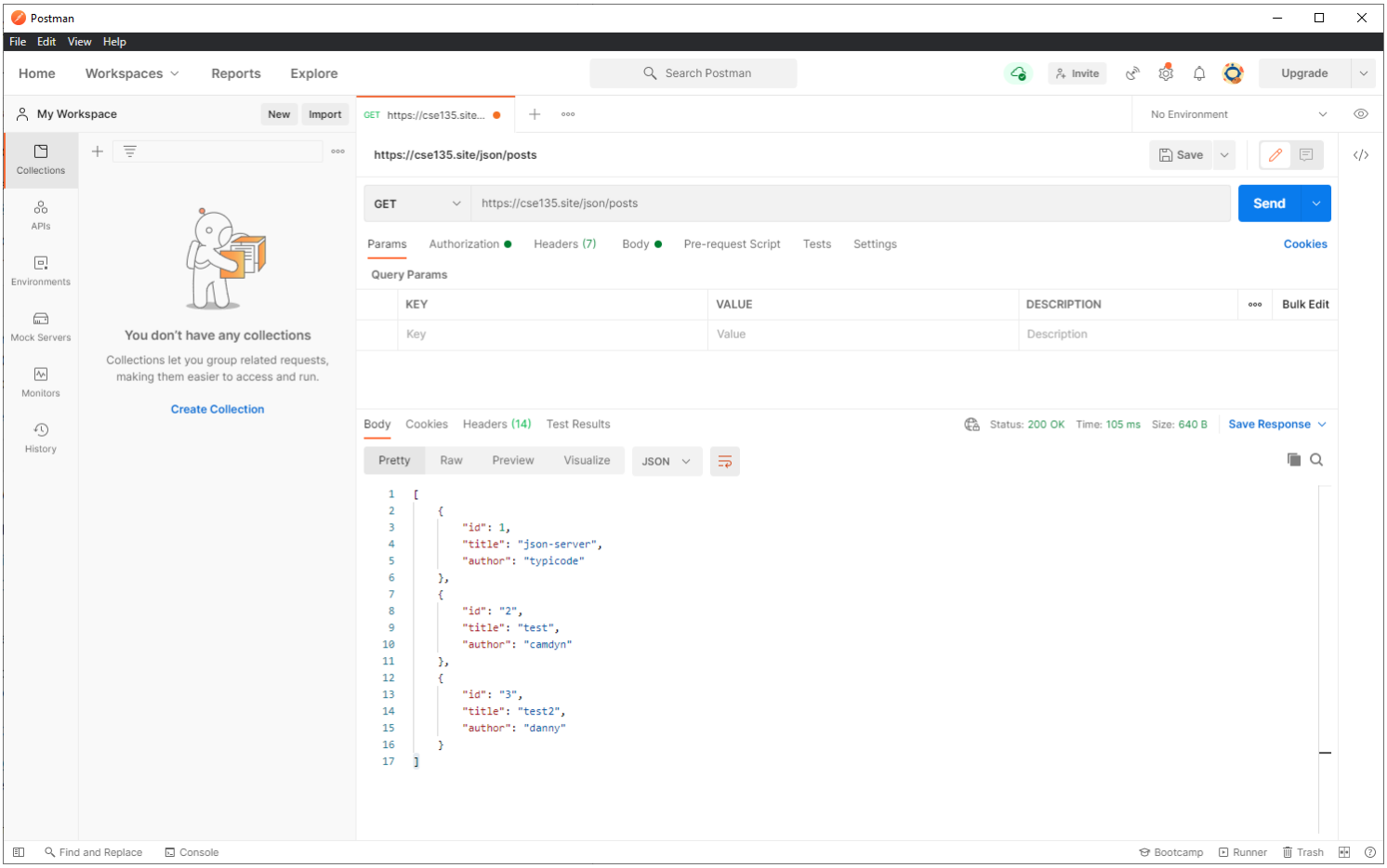
```
sudo ufw delete #
```

Where you replace the # with the number the rule you want to delete is.

NOTE: If you delete the top rule before the bottom rule, the number of the bottom rule will change

Before you start playing with the endpoint, download a program called Postman which you can find here ([https://www.postman.com/downloads/ (https://www.postman.com/downloads/)](https://www.postman.com/downloads/)). Postman is a simple app that lets you create/view any kind HTTP request.

To get yourself comfortable with Postman, make a general GET request to your mocked endpoint, as shown below.

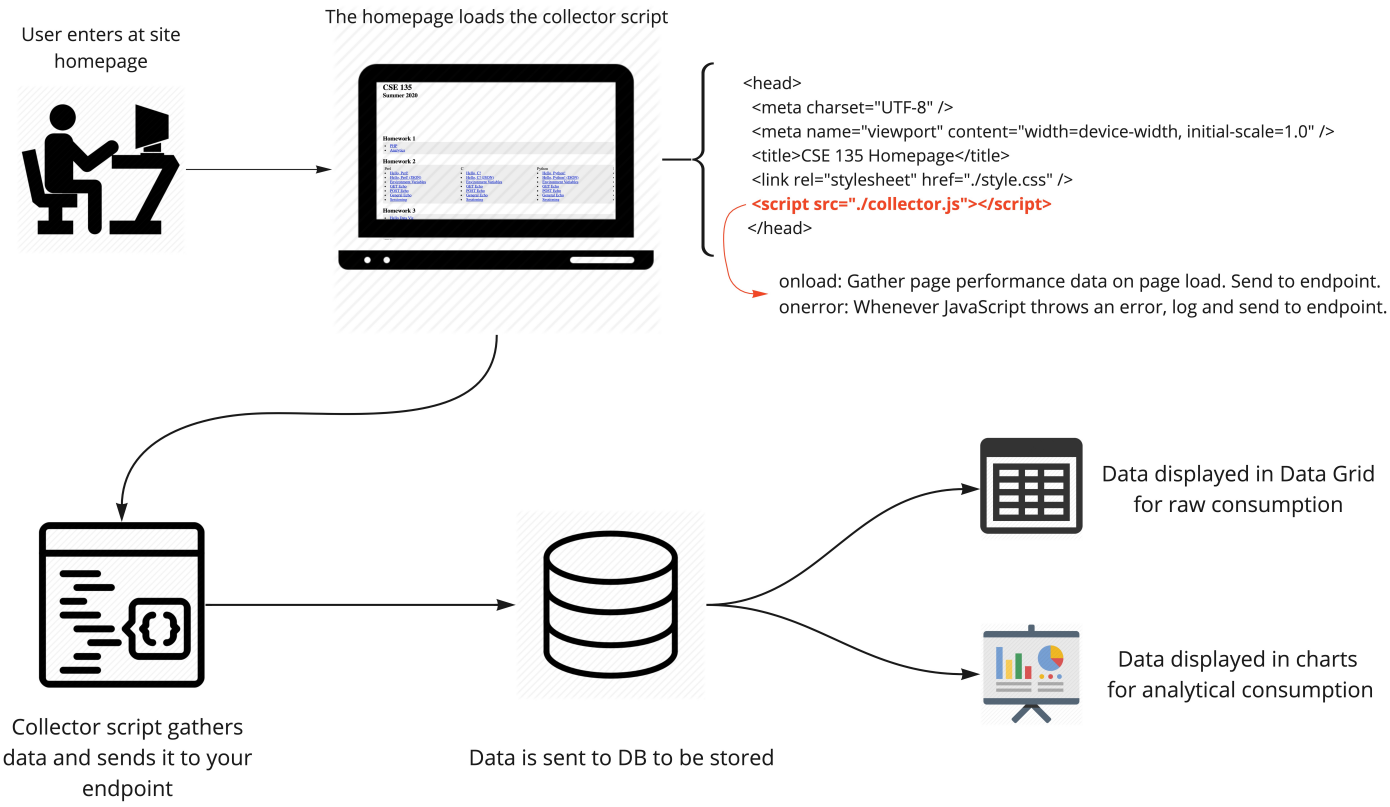


Take a screenshot showing that data is returned and label it postman.png. Include this in your submission.

Now head back to the [https://github.com/typicode/json-server#routes (https://github.com/typicode/json-server#routes)](https://github.com/typicode/json-server#routes) JSON server docs to explore how the routes are setup, hit your endpoint with Postman, and get the feel of how a REST endpoint for a CRUD application works. We may not use all aspects of REST/CRUD in this assignment, but we certainly will by the end of the course. Make sure you come back to JSON-server to mock things before writing a bunch of PHP or JS.

Part 1 - Collecting and Sending Analytical Data

Now that we know how to mock out an endpoint, we have to collect user data to send to that endpoint! Before we do that, let's zoom out to a 10,000 foot view of what we want our program to look like by then end of the assignment. The diagram below demonstrates the basic flow of data - a user will enter your site at the homepage, and the request for the homepage file will include the request for **collector.js**. Once the page loads, the script will fire, sending the initial static and performance data to the endpoint. Then, any as the user interacts with the webpage, their activity data will be periodically sent to the endpoint. Your endpoint will store the data in the database, which you will then query to create your data grid and charts.



This diagram represents the flow of data from your user to your analytics pages.

You will be creating your own **collector.js** script yourself. The script must collect the following data (it is allowed to collect more, but this is the minimum)

- Three Types of Data: Static, Performance, and Activity data
  - **Static** (collected after the page has loaded)
    - user agent string
    - the user's language
    - if the user accepts cookies
    - if the user allows JavaScript (you will have to manually figure this one out)

        - if the user allows images (you will have to manually figure this one out)
        - if the user allows CSS (you will have to manually figure this one out)
        - User's screen dimensions
        - User's window dimensions
        - User's network connection type
    - **Performance** (collected after the page has loaded)
        - The timing of the page load
            - The whole timing object
            - Specifically when the page started loading
            - Specifically when the page ended loading
            - The total load time (manually calculated - in milliseconds)
    - **Activity** (continuously collected)
        - All mouse activity
            - Cursor positions (coordinates)
            - Clicks (and which mouse button it was)
            - Scrolling (coordinates of the scroll)
        - All keyboard activity
            - Key down or Key up events
        - **Any idle time where no activity happened for a period of 2 or more seconds**
            - Record when the break ended
            - Record how long it lasted (in milliseconds)
        - When the user entered the page
        - When the user left the page
        - Which page the user was on
- You should be able to tie this data to a specific user session

Do not assume that every network request will work 100% of the time. Save the data locally, then make attempts to send updates to the server.

You can send your data a number of ways, but the easiest is probably one of the following:

- Fetch API: **https://developer.mozilla.org/en/docs/Web/API/Fetch_API (https://developer.mozilla.org/en/docs/Web/API/Fetch_API)**
- sendBeacon: **https://developer.mozilla.org/en-US/docs/Web/API/Navigator/sendBeacon (https://developer.mozilla.org/en-US/docs/Web/API/Navigator/sendBeacon)**
- XHR: **https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)**

REST Endpoint Setup

To set up your REST endpoint you have a choice of either NodeJS or php. In step zero the json-server was built on top of a Node server using Express which is a viable route, but php is also fine. You just need to be able to connect to either MySql or Mongo (the two database options you have in Part 2) and you're good to go.

It is important to know that if you use a separate server, like Node, that you proxy it through your main Apache server as we did above. This will give the Node server a sort of buffer from attacks.

The routes you define should follow the REST pattern, as shown below.

To avoid having to create a different proxy for every single table, you can use a single /api proxy like we did above with /json, to prepend all of your routes.

Below we have an example routing system just for static data. The example route of /api/static is not the exact routing scheme you have to choose for your table, you may choose to have something like /api/static/useragents or something to get more specific, it just depends on how you structure your database and what makes sense for you. What you **have** to do is follow the rest pattern as shown below. Making a GET request without an ID gets all for the given route, with an ID gets that specific ID. You must not include an ID when making a POST request, and you need to include an ID when making a DELETE and PUT request for a given route. Whatever routes you decide to make, make tables such as the one below explaining all of them, and put them in a .pdf called **routes.pdf**

HTTP Method	**Example Route**	**Description**
GET	/api/static	Retrieve every entry logged in the static table
GET	/api/static/{id}	Retrieve a specific entry logged in the static table (that matches the given id)
POST	/api/static	Add a new entry to the static table
DELETE	/api/static/{id}	Delete a specific entry from the static table (that matches the given id)
PUT	/api/static/{id}	Update a specific entry from the static table (that matches the given id)

To demonstrate that your REST api is working, make a GET request to get your log data. At this point since you don't have a database hooked up yet, it is fine if you pull from a .json file or hard code some data to pull from inside your endpoint server file.

Take a screenshot showing that data is returned and label it **REST.png**. Include this in your submission.

Part 2 - Database Structure and Population

For your Database you have two choices: MySQL or MongoDB. If you'd like to learn some SQL and aren't familiar with it, this might be a nice time to dip your toes into MySQL, otherwise whichever you choose doesn't matter too much. However, later our ability to consume or analyze the data may be affected! For example - MySQL makes relating records between tables easy (handy if you use session IDs as your entry IDs for each table), but Mongo stores the data in a more readable way.

In this part you are going to be keeping the data local on your server, so install your chosen database onto your server (A simple "how to install ___ on DigitalOcean server" google search will lead to some helpful DigitalOcean docs).

Once you've gotten your database up and running, the next question is how you would like to format your data.

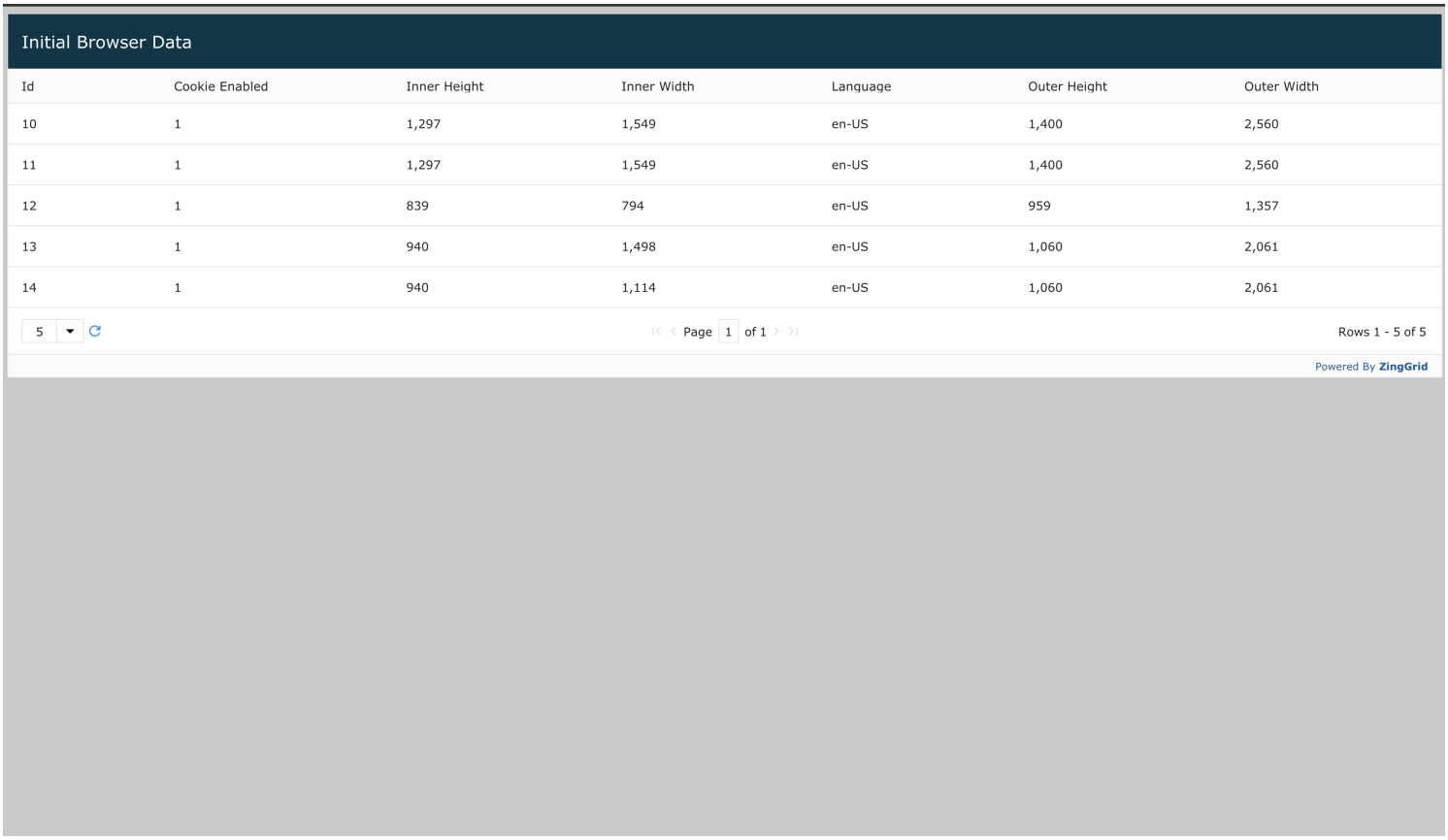
One simple setup might be to create an individual table for Static and Performance data. For Activity data it is up to you as it can get very large very quick, so be careful.

One important thing to note is that the REST pattern relies on unique IDs for each entry. This could be especially important if you want to sessionize your data. *Without* sessionizing, all of your logs are anonymous and not correlated to one another, which can make some data analysis difficult.

Once your database is populated with your data, figure out how to pull all that data back out (using a simple /GET). Find a data grid library or use ZingGrid (**www.zinggrid.com (http://www.zinggrid.com)**).

You can use any library you want, but the teaching staff will only provide support for ZingGrid.

Create a page called `database.html` that has a read-only grid that displays all of your data. (Your data MUST be pulled from your REST endpoints, which MUST be hooked up to your database. You cannot hard code your data in the grid.) It might look something like this (likely with more data / columns):



Initial Browser Data

Id	Cookie Enabled	Inner Height	Inner Width	Language	Outer Height	Outer Width
10	1	1,297	1,549	en-US	1,400	2,560
11	1	1,297	1,549	en-US	1,400	2,560
12	1	839	794	en-US	959	1,357
13	1	940	1,498	en-US	1,060	2,061
14	1	940	1,114	en-US	1,060	2,061

5 Page 1 of 1 Rows 1 - 5 of 5

Powered By ZingGrid

NOTE: This is the minimum acceptable grid and should not be exactly duplicated as is. Be creative!

Part 3 - DBaaS

DBaaS - **D**ata**b**ase **a**s **a** **S**ervice. Now that in Part 2 we explored how to host our own Database on our own server, we can appreciate that it's not the most straightforward thing to do. On top of this, there are a few notable downsides: When collecting massive amounts of data it can be quite difficult to scale up the database by hand, you have to manually backup your data, and you are in charge of keeping up the "health" of your database with updates, patches, and service monitoring. Swapping to a database service that a third party offers can save a large amount of effort as a developer (usually at a monetary cost).

MySql - DBaaS

To explore DBaaS, choose one and start uploading your data there as well. There are many to choose from, but we recommend DigitalOcean (if you chose MySql) for two reasons:

- You can select the same server cluster your droplet is on, increasing performance in terms of communication speed between the two instances.
- You already have DigitalOcean credits, so although their servers start at $15/mo, it will be free to you

To connect to a remote MySql database, you cannot simply ssh into it like you can with a droplet. While you can install MySql through the terminal on your local machine and log in that way, it may be beneficial to check out a MySql application which provides a nice GUI. Some we recommend:

MySql Workbench (Windows): **https://www.mysql.com/products/workbench/ (https://www.mysql.com/products/workbench/)**

Sequel Pro (Mac): **https://www.sequelpro.com/ (https://www.sequelpro.com/)**

MongoDB - DBaaS

If you chose MongoDB, their own website **www.mongodb.com (http://www.mongodb.com/)** offers a free tier DBaaS plan for those learning how to use Mongo which is perfect for our purposes.

For Mongo, during the set up process they will show you how to connect with the mongo shell or with MongoDB Compass.

Once you get your database up and running and get it populated with some data, take a screenshot to prove that you've completed this. Acceptable screenshots include:

- A database GUI application showing tables populated with data
- A shell command showing tables populated with data

Name this `database.png` and include it with your submission.

Part 4 - Data Viz Hello World

Now that you've gotten the opportunity to see different logging methods and analytics providers, we are going to try to do some preliminary data visualization as the next step towards our own analytics application. You can use *any* charting library you'd like, but please note that the teaching staff will provide support for ZingChart only (**www.zingchart.com (http://www.zingchart.com)**) given familiarity.

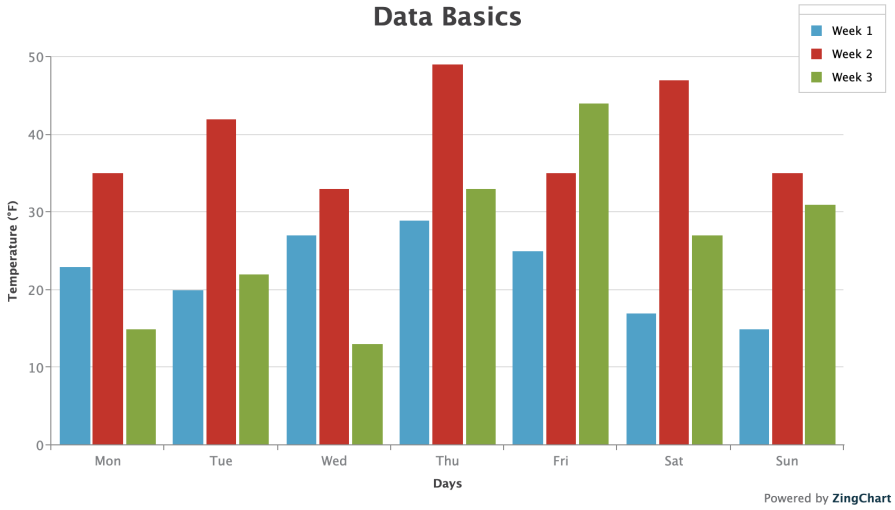
We have many different types of data at this point between the browser and performance data collected with collector.js. Using the data you've collected, create the following charts to display on a page called `hellodataviz.html`:

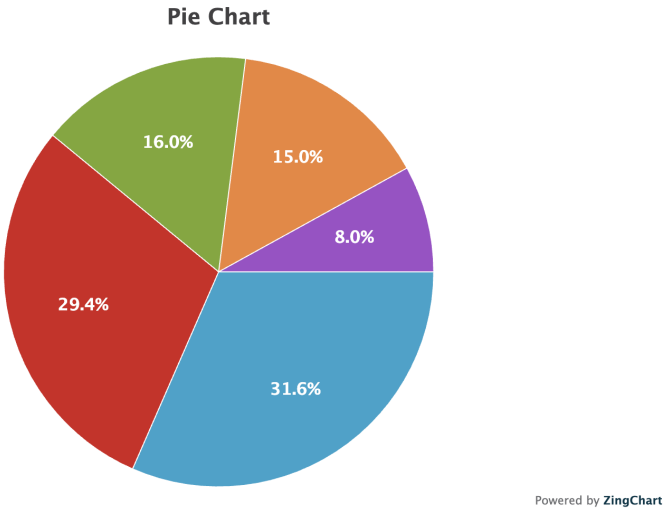
- A 3-series line chart

- A 2-series bar chart
- A pie chart

Get your charts running with dummy data first! *Then* figure out how to populate with real data.

Below are some basic charts on a page. They are default, unstyled, and have no CSS used on them. Do not copy them, we expect more effort. It's just an example of what some basic zing grid charts look like.



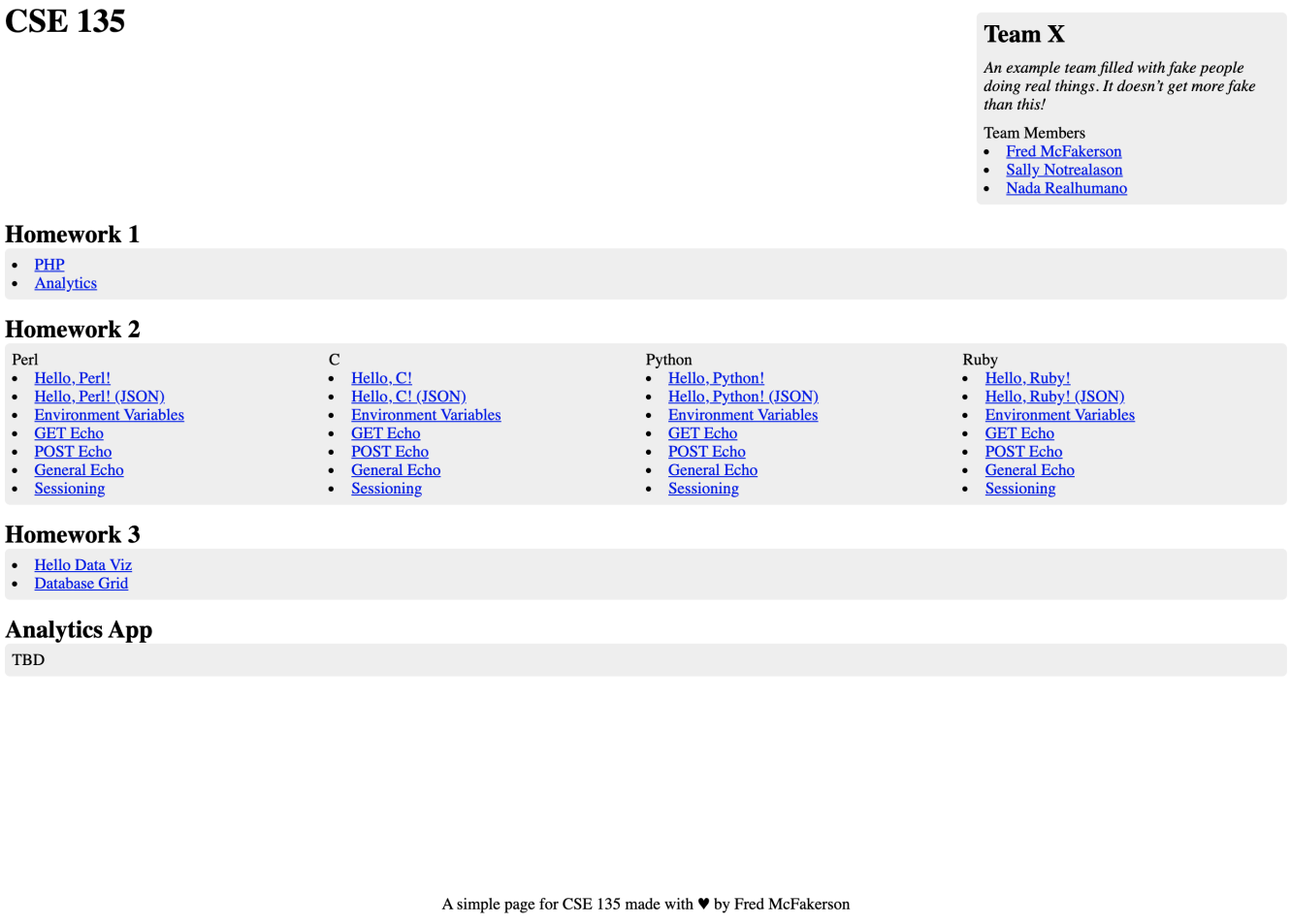


Submission Details

You will be submitting everything to Gradescope. The assignment will be created on Gradescope shortly.

As with the previous assignments, create a section on your site homepage for homework 3. Add links to the following:

- **database.html**
- **hellodataviz.html**



Submit your source code for the following:

- **database.html**
- **hellodataviz.html**
- **Any JavaScript/PHP/other files** that you wrote and included on these pages

Submit the following media:

- **postman.png** (request to mock endpoint)
- **REST.png** (REST endpoint setup)
- **database.png** (screenshot of database setup)
- **routes.pdf** (Export a table/spreadsheet to pdf of the routing scheme you chose. Make sure that it follows the REST pattern)

Finally, submit a **README.md** **(http://readme.bd)** with the following:

- A link to your site
- Any necessary notes to the graders to make grading easier
- **team member names**
- **your IP address of server, ssh key, grader log in information to the site and server**
- Any changes you made to collector.js