University of California, Santa Cruz

Experiment 7:

Data Visualization and Networks

Dominic Berardi & Nathan Prieto

CMPM 169: Creative Coding

Modes

Due Date 3/2/2023

Graph Wars

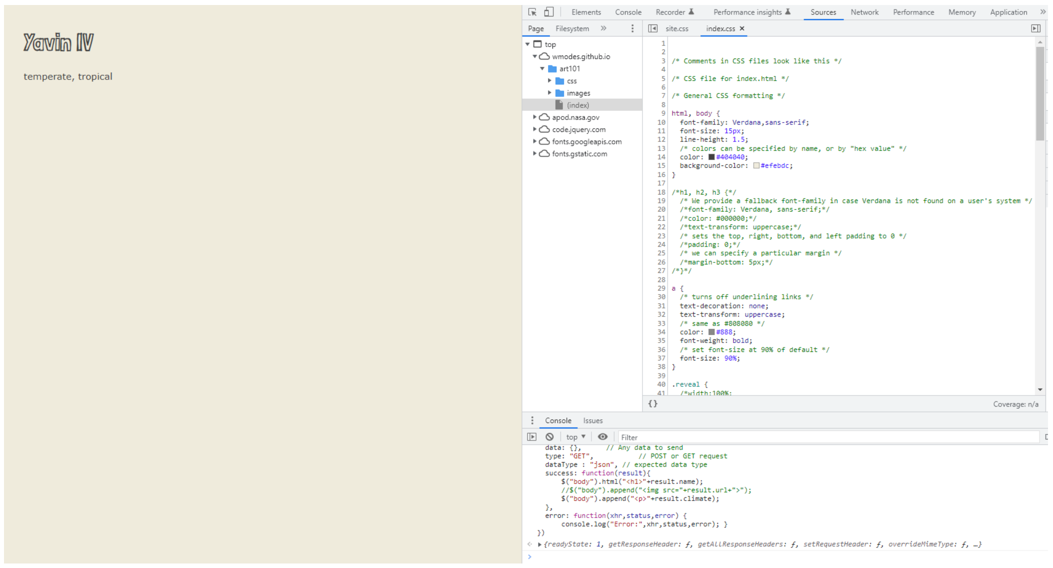
A data visualization that shows various points of data throughout the Star Wars movies.
NOTE: If it does not work at first, wait a bit and refresh. It sometimes has issues with the API.

Playable Link: here
Sketch.js code: https://github.com/domberardi10/cmpm169/blob/master/experiment7/js/sketch.js

Step 1: Imitate

We were completely unsure of what kind of piece we wanted to make coming into this experiment. Most of the experiments shown in class appeared far too complex or not adjacent to p5 to recreate in a reasonable time frame. Though, we both had some interests related to each topic; Dominic wanted to play around with APIs and fetch some interesting data, and Nate has experience with D3.js, and could make some sort of data visualization with that. So, we decided to create a data visualization that uses an API to collect information on a topic.

To start, we had to nail down exactly how to set up connecting to an API. The slides were pretty unclear, but Wes did a demo in class of getting data from an API onto a webpage. We first started by mimicking this process with an entirely different API than the one they used. However, many APIs require some sort of authorization to even access them. This awesome webpage, we found, has many free, open source APIs to utilize (https://mixedanalytics.com/blog/list-actually-free-open-no-auth-needed-apis/) and one that both piqued our interests, as Star Wars fans, was the “Star Wars information” API (https://swapi.dev/), which has a bunch of information on characters, films, planets, and more to retrieve. We were able to use this API following how Wes did, in the console.



Yavin IV

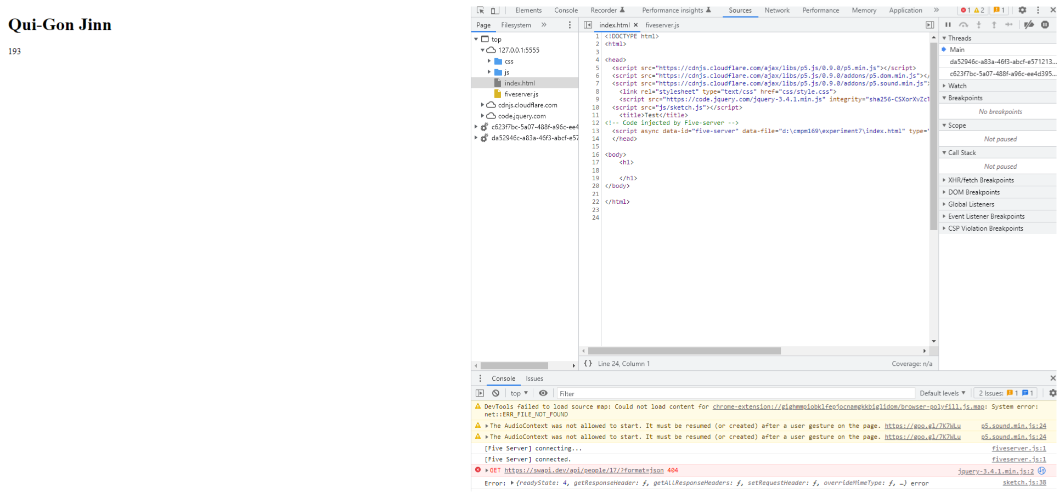
temperate, tropical

```
/* Comments in CSS files look like this */

/* CSS file for index.html */

/* General CSS formatting */

html, body {
  font-family: Verdana,sans-serif;
  font-size: 15px;
  line-height: 1.5;
  /* colors can be specified by name, or by "hex value" */
  color: #404040;
  background-color: #efebdc;
}

/*h1, h2, h3 {*/
  /* We provide a fallback font-family in case Verdana is not found on a user's system */
  /*font-family: Verdana, sans-serif;*/
  /*color: #000000;*/
  /*text-transform: uppercase;*/
  /* sets the top, right, bottom, and left padding to 0 */
  /*padding: 0;*/
  /* we can specify a particular margin */
  /*margin-bottom: 5px;*/
/*}*/

a {
  /* turns off underlining links */
  text-decoration: none;
  text-transform: uppercase;
  /* same as #808080 */
  color: #888;
  font-weight: bold;
  /* set font-size at 90% of default */
  font-size: 90%;
}

.reveal {
  /*width:100%;
```

```
    data: {},      // Any data to send
    type: "GET",           // POST or GET request
    dataType : "json", // expected data type
    success: function(result){
        $("body").html("<h1>"+result.name);
        //$("body").append("<img src="+result.url+">");
        $("body").append("<p>"+result.climate);
    },
    error: function(xhr,status,error) {
        console.log("Error:",xhr,status,error); }
  })
< ▸{readyState: 1, getResponseHeader: ƒ, getAllResponseHeaders: ƒ, setRequestHeader: ƒ, overrideMimeType: ƒ, …}
>
```

We were entirely confused on how to actually *get* API information from not within the console. It appeared that the index.html file we use would have to be set up in a certain way for the AJAX call to append words to it. The error we would run into was one that did not recognize the sketch file and AJAX, but as it turned out, this was simply because the calls were out of order in index.html. Way too much time wasted trying to figure that out!

After fixing that bug, the AJAX call within our sketch.js was working fine. It would simply display the first character's name and height (at https://swapi.dev/api/people/1/). However, we needed more characters' information. The best way to do this, we found, was by iterating through all of the character JSONs (there were 83 total) and displaying a different one every second. Some string concatenation was in order, as well as a variable which stored the current URL. Iterating through all of the characters' names and heights worked great.

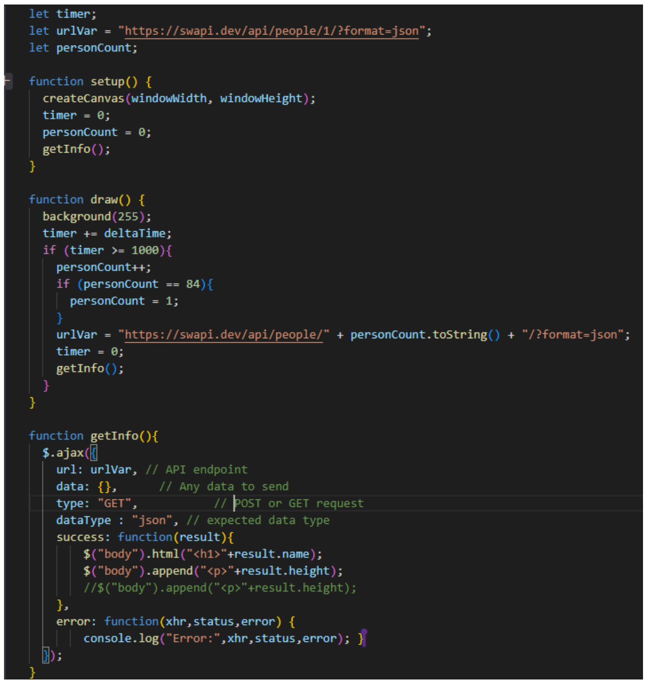


Qui-Gon Jinn

193

```
<!DOCTYPE html>
<html>

<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.9.0/p5.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.9.0/addons/p5.dom.min.js"></
  <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.9.0/addons/p5.sound.min.js">
    <link rel="stylesheet" type="text/css" href="css/style.css">
    <script src="https://code.jquery.com/jquery-3.4.1.min.js" integrity="sha256-CSXorXvZcI
  <script src="js/sketch.js"></script>
    <title>Test</title>
<!-- Code injected by Five-server -->
  <script async data-id="five-server" data-file="d:\cmpm169\experiment7\index.html" type='
  </head>

<body>
    <h1>

    </h1>
</body>

</html>
```

```
DevTools failed to load source map: Could not load content for chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/browser-polyfill.js.map: System error: net::ERR_FILE_NOT_FOUND
▸The AudioContext was not allowed to start. It must be resumed (or created) after a user gesture on the page. https://goo.gl/7K7WLu     p5.sound.min.js:24
▸The AudioContext was not allowed to start. It must be resumed (or created) after a user gesture on the page. https://goo.gl/7K7WLu     p5.sound.min.js:24
[Five Server] connecting...                                                                                    fiveserver.js:1
[Five Server] connected.                                                                                       fiveserver.js:1
▸GET https://swapi.dev/api/people/17/?format=json 404                                                          jquery-3.4.1.min.js:2
Error: ▸{readyState: 4, getResponseHeader: ƒ, getAllResponseHeaders: ƒ, setRequestHeader: ƒ, overrideMimeType: ƒ, …} error    sketch.js:38
```



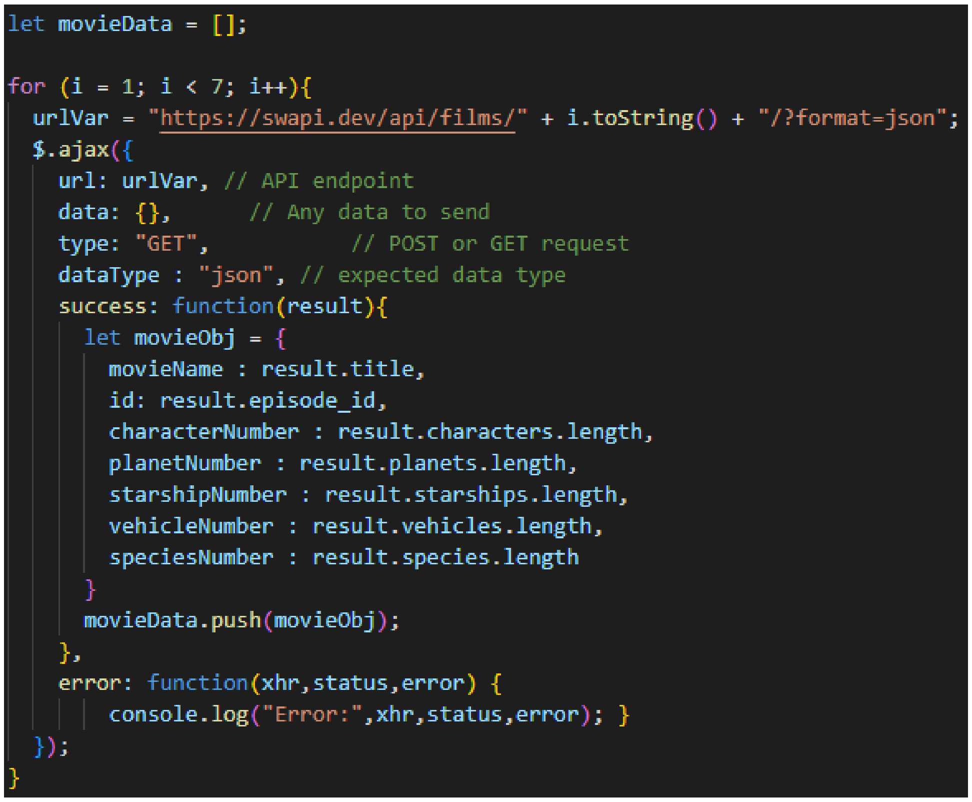
```
let timer;
let urlVar = "https://swapi.dev/api/people/1/?format=json";
let personCount;

function setup() {
  createCanvas(windowWidth, windowHeight);
  timer = 0;
  personCount = 0;
  getInfo();
}

function draw() {
  background(255);
  timer += deltaTime;
  if (timer >= 1000){
    personCount++;
    if (personCount == 84){
      personCount = 1;
    }
    urlVar = "https://swapi.dev/api/people/" + personCount.toString() + "/?format=json";
    timer = 0;
    getInfo();
  }
}

function getInfo(){
  $.ajax({
    url: urlVar, // API endpoint
    data: {},      // Any data to send
    type: "GET",            // POST or GET request
    dataType : "json", // expected data type
    success: function(result){
        $("body").html("<h1>"+result.name);
        $("body").append("<p>"+result.height);
        //$("body").append("<p>"+result.height);
    },
    error: function(xhr,status,error) {
        console.log("Error:",xhr,status,error); }
  });
}
```

Step 2: Integrate

Given the information we could retrieve from the API, we began to brainstorm ways we could display its data visually. We came up with three separate ideas:

1. Plot points of height/mass ratio of each character
2. Bar graph showing number of characters per film
3. Timeline showing when each character was born

Out of these options, we felt that the second one would be the most interesting (who cares about height/weight ratios?!). Each character's JSON did indeed have a "film" parameter that held a list of the films they starred in; we would simply take the length of each film array and add to values that would track each of the films' character counts.

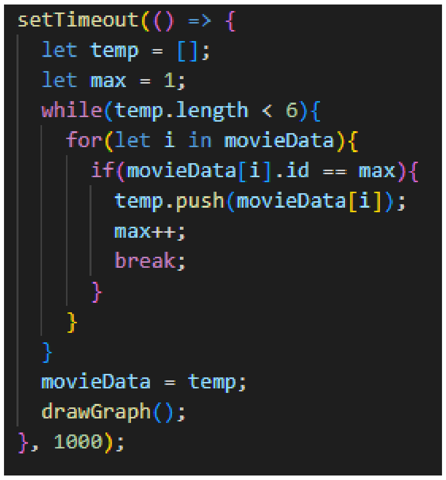


```
let movieData = [];

for (i = 1; i < 7; i++){
  urlVar = "https://swapi.dev/api/films/" + i.toString() + "/?format=json";
  $.ajax({
    url: urlVar, // API endpoint
    data: {},      // Any data to send
    type: "GET",           // POST or GET request
    dataType : "json", // expected data type
    success: function(result){
      let movieObj = {
        movieName : result.title,
        id: result.episode_id,
        characterNumber : result.characters.length,
        planetNumber : result.planets.length,
        starshipNumber : result.starships.length,
        vehicleNumber : result.vehicles.length,
        speciesNumber : result.species.length
      }
      movieData.push(movieObj);
    },
    error: function(xhr,status,error) {
        console.log("Error:",xhr,status,error); }
  });
}
```

Along the way, though, we were finding it to be difficult due to how the JSONs were set up. Luckily, we noticed that the "films" JSONs from the API already had information about characters in each film, as well as information about vehicles, planets, and starships! We shifted our focus to retrieving each of the film JSONs, and then storing information from those we wanted.

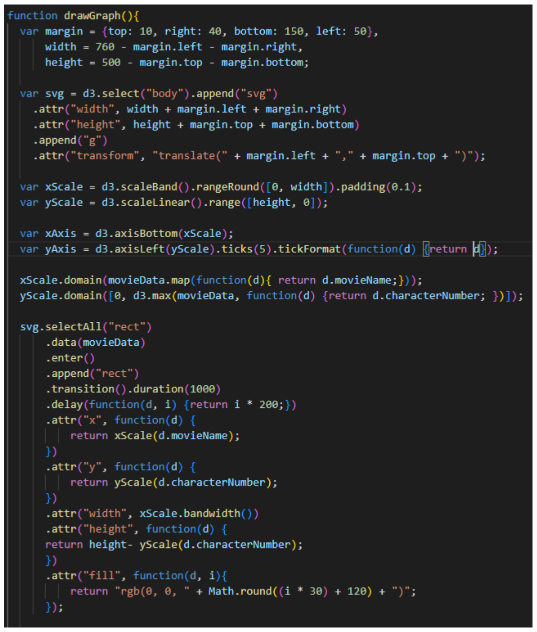
At first, this approach was giving us some issues when attempting to test if the information was retrieved properly. We found this to stem from the fact that JavaScript runs things asynchronously, and thus will execute code even if other pieces of code (that may be dependent on the other) have not completed their execution yet. The AJAX calls to the API do not occur instantaneously, as they must wait for the server to send its messages back to the client– this creates the disconnect between our data execution code blocks and the data retrieval calls.

The solution to this problem involves using JavaScript's "Promise" and "Await" functions, where a dependent piece of code waits for another block to complete. However, we found ourselves simply confused with how to implement these functions into our existing code, and decided to cut our losses by compromising. The "setTimeout()" function forces a block of code to run only after a set time has elapsed. We set this to 1 second to give a good enough buffer for the server to send back its information. Also, the data was sometimes being retrieved out of order, and the Star Wars movies are famously numbered out of order as well. We fix this here.

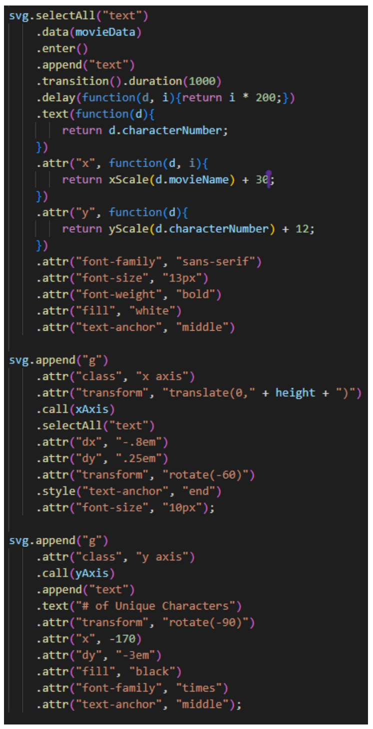


```
setTimeout(() => {
  let temp = [];
  let max = 1;
  while(temp.length < 6){
    for(let i in movieData){
      if(movieData[i].id == max){
        temp.push(movieData[i]);
        max++;
        break;
      }
    }
  }
  movieData = temp;
  drawGraph();
}, 1000);
```

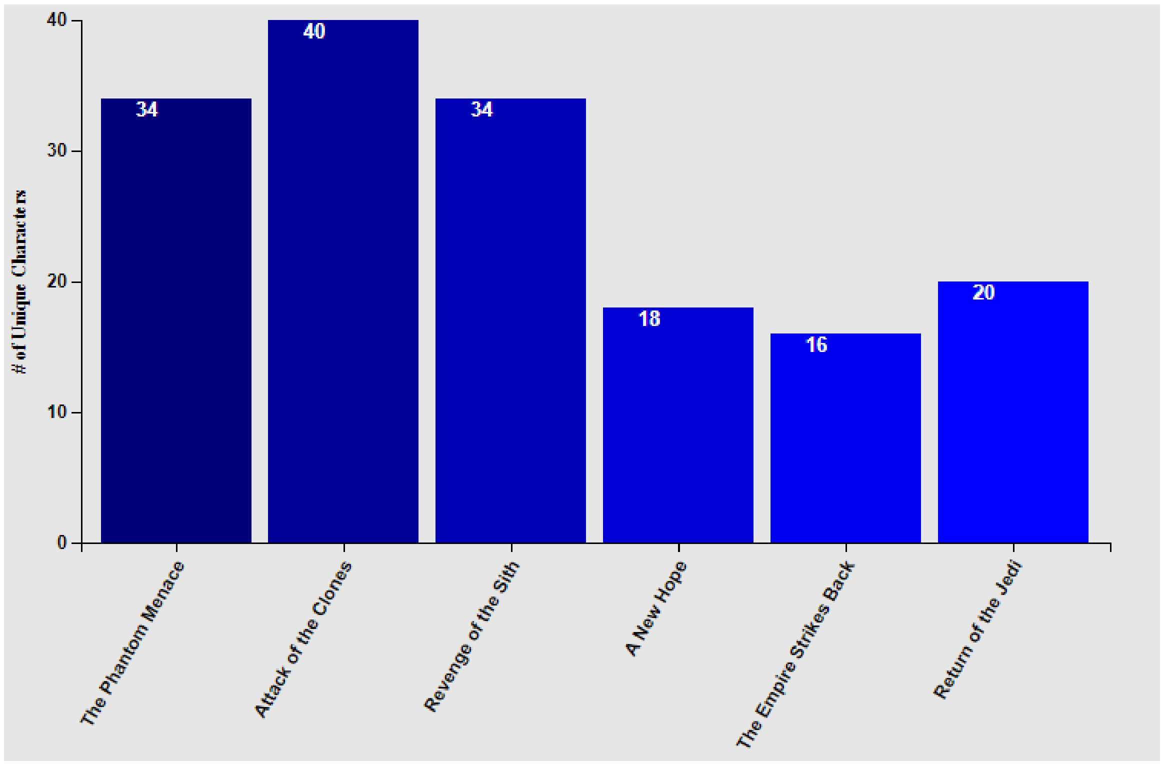
Now, we had our data and could display it visually. For this, we used D3.js. The upper half of the image on the left defines how large the SVG canvas is and then appends that canvas to the body of the html document. It sets the SVG element's width and height and then appends a group to it that will ultimately hold the graph itself. It then creates the scale for the x and y axis using the width and height of the SVG canvas mentioned above. It then defines what orientation each axis is denoted by the .axisBottom() and then .axisLeft() functions. The xScale.domain() and yScale.domain() determine the range of values that could possibly fit within their scale. So the x domain will contain only 6 values, due to the dataset only containing 6 movies, and the y domain will contain values between 0 and the maximum number of characters found within the entire dataset. The lower half of the image defines some attributes of the "bars" of the graph such as the rectangles x and y position in relation to the SVG canvas, its width and height, and its fill color. The upper half of the image on the right places the number of characters on top of the rectangle previously mentioned and determines its x and y position based on the dataset. The block of code underneath places a text element that contains the name of each movie at the bottom of the graph at its appropriate position and applies a -60° rotation. The chunk underneath places another text element that holds the label for the y axis and applies a -90° rotation to it.



```javascript
function drawGraph(){
  var margin = {top: 10, right: 40, bottom: 150, left: 50},
      width = 760 - margin.left - margin.right,
      height = 500 - margin.top - margin.bottom;

  var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

  var xScale = d3.scaleBand().rangeRound([0, width]).padding(0.1);
  var yScale = d3.scaleLinear().range([height, 0]);

  var xAxis = d3.axisBottom(xScale);
  var yAxis = d3.axisLeft(yScale).ticks(5).tickFormat(function(d) {return d});

  xScale.domain(movieData.map(function(d){ return d.movieName;}));
  yScale.domain([0, d3.max(movieData, function(d) {return d.characterNumber; })]);

  svg.selectAll("rect")
      .data(movieData)
      .enter()
      .append("rect")
      .transition().duration(1000)
      .delay(function(d, i) {return i * 200;})
      .attr("x", function(d) {
          return xScale(d.movieName);
      })
      .attr("y", function(d) {
          return yScale(d.characterNumber);
      })
      .attr("width", xScale.bandwidth())
      .attr("height", function(d) {
      return height- yScale(d.characterNumber);
      })
      .attr("fill", function(d, i){
          return "rgb(0, 0, " + Math.round((i * 30) + 120) + ")";
      });
```

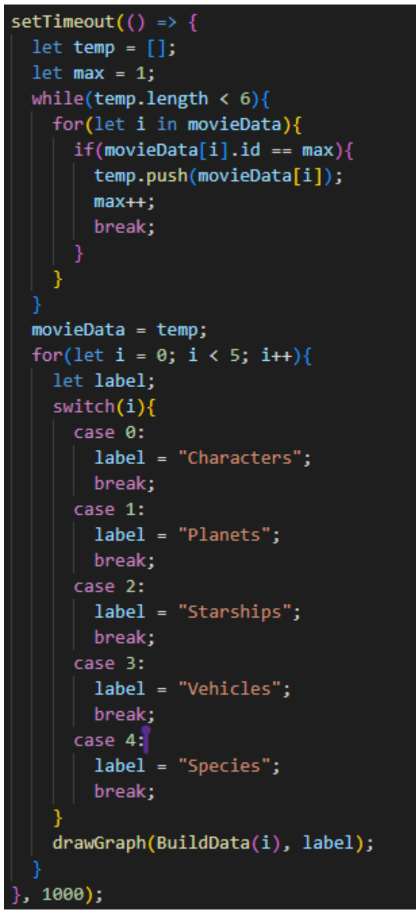


```javascript
svg.selectAll("text")
     .data(movieData)
     .enter()
     .append("text")
     .transition().duration(1000)
     .delay(function(d, i){return i * 200;})
     .text(function(d){
         return d.characterNumber;
     })
     .attr("x", function(d, i){
         return xScale(d.movieName) + 30;
     })
     .attr("y", function(d){
         return yScale(d.characterNumber) + 12;
     })
     .attr("font-family", "sans-serif")
     .attr("font-size", "13px")
     .attr("font-weight", "bold")
     .attr("fill", "white")
     .attr("text-anchor", "middle")

svg.append("g")
     .attr("class", "x axis")
     .attr("transform", "translate(0," + height + ")")
     .call(xAxis)
     .selectAll("text")
     .attr("dx", "-.8em")
     .attr("dy", ".25em")
     .attr("transform", "rotate(-60)")
     .style("text-anchor", "end")
     .attr("font-size", "10px");

svg.append("g")
     .attr("class", "y axis")
     .call(yAxis)
     .append("text")
     .text("# of Unique Characters")
     .attr("transform", "rotate(-90)")
     .attr("x", -170)
     .attr("dy", "-3em")
     .attr("fill", "black")
     .attr("font-family", "times")
     .attr("text-anchor", "middle");
```

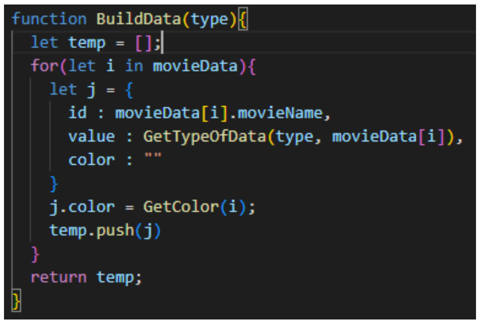


Step 3: Innovate

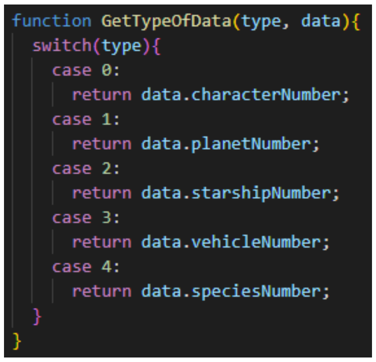
For the final step, we wanted to create more graphs with different information we have collected from the API and make the whole thing look pretty (or at least, more Star Wars-y). We created the function BuildData() that creates arrays filled with different data that is used to inform the values of each graph. This relies on GetTypeOfData() which essentially returns the desired data value based on the current graph to be made. The two functions severely diminished the amount of lines we had within our code and overall made understanding it much easier. We had to edit drawGraph() slightly so that it accepts a set of data as well as a label that uses the name of the y axis of each graph. This allowed us to reuse the D3 code used to create each graph rather than copying and pasting the same chunk of code multiple times each with a slightly different dataset.



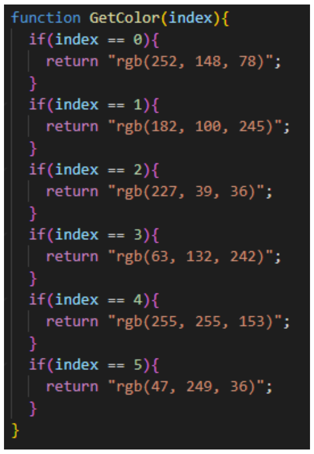
```
setTimeout(() => {
  let temp = [];
  let max = 1;
  while(temp.length < 6){
    for(let i in movieData){
      if(movieData[i].id == max){
        temp.push(movieData[i]);
        max++;
        break;
      }
    }
  }
  movieData = temp;
  for(let i = 0; i < 5; i++){
    let label;
    switch(i){
      case 0:
        label = "Characters";
        break;
      case 1:
        label = "Planets";
        break;
      case 2:
        label = "Starships";
        break;
      case 3:
        label = "Vehicles";
        break;
      case 4:
        label = "Species";
        break;
    }
    drawGraph(BuildData(i), label);
  }
}, 1000);
```



```
function BuildData(type){
  let temp = [];
  for(let i in movieData){
    let j = {
      id : movieData[i].movieName,
      value : GetTypeOfData(type, movieData[i]),
      color : ""
    }
    j.color = GetColor(i);
    temp.push(j)
  }
  return temp;
}
```



```
function GetTypeOfData(type, data){
  switch(type){
    case 0:
      return data.characterNumber;
    case 1:
      return data.planetNumber;
    case 2:
      return data.starshipNumber;
    case 3:
      return data.vehicleNumber;
    case 4:
      return data.speciesNumber;
  }
}
```

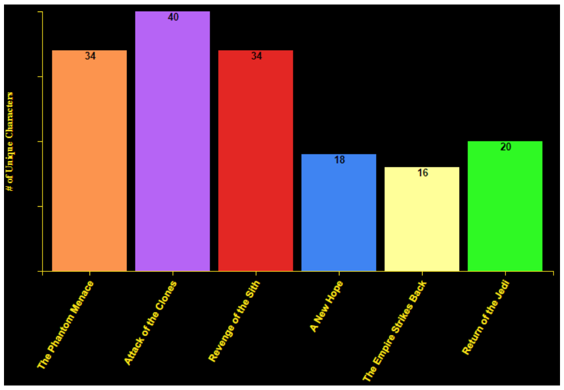


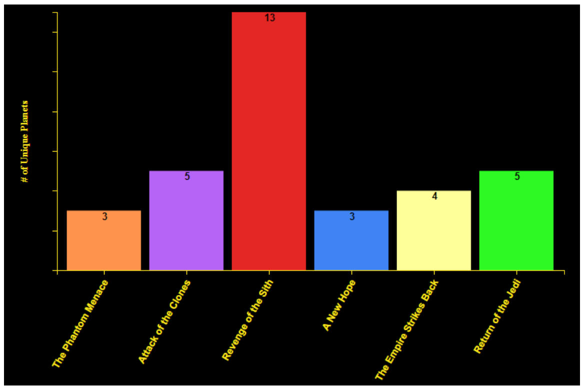
```
function GetColor(index){
  if(index == 0){
    return "rgb(252, 148, 78)";
  }
  if(index == 1){
    return "rgb(182, 100, 245)";
  }
  if(index == 2){
    return "rgb(227, 39, 36)";
  }
  if(index == 3){
    return "rgb(63, 132, 242)";
  }
  if(index == 4){
    return "rgb(255, 255, 153)";
  }
  if(index == 5){
    return "rgb(47, 249, 36)";
  }
}
```

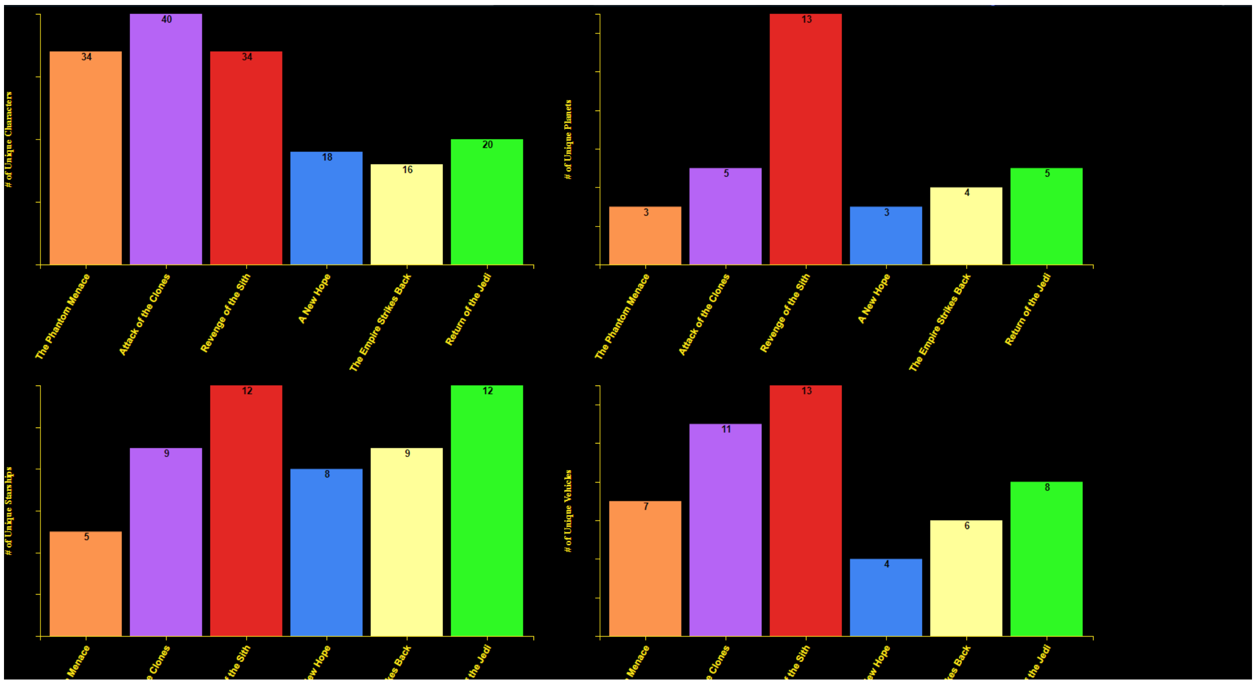
Once we were able to create multiple graphs, it was time to make the aesthetics a little bit nicer (emphasis on a little bit). We figured the best way would be to copy the opening crawl colors of Star Wars (black and yellow) for the background and main text. Then, the bars themselves would be colored after lightsaber colors, with some exceptions (is there an orange saber?). We hoped the result would look at least somewhat reminiscent of the aesthetics of the movies.

But setting the color of each bar on the graph wasn't so simple. You see, the color is decided within the D3 algorithm to create the graph, and that is not directly accessible to change values of. Thus, we created the GetColor() function, as seen in the photos above. This function determines what color should be routed to the D3 color by taking a look at what the current film name is, and associates the color with that film on the graph. This worked perfectly.

Here's a look at the final results!







Reflect

Dominic:

This experiment was a bit of a tough one. Neither of us were very familiar with using APIs, so a significant amount of time was spent trying to wrap our heads around utilizing even the simplest API. Thankfully we found this Star Wars one with a plethora of info to use and it made the next step, creating a data visualization, much easier. I am super thankful that Nate had worked with D3.js before, as the data viz portion went much smoother than the API portion as he spearheaded the data viz. I was the leader during the API coding and it took probably 75% of the time, even though it was only a small chunk of the experiment in the end. I certainly feel like we could have added more to the project, such as making it look even nicer (images) and adding more types of data visualizations (timelines, scatter plots). However I am happy with how it turned out nonetheless and glad we were able to work with an API in any capacity.

Nathan:

This project was an ok introduction to understanding APIs and how to use them. I was mildly aware of what they were and the purpose that they serve but had never actually interacted with one. I felt as if the lecture didn't further that understanding as the examples presented were difficult to understand and implement as a lot of functionally surrounding them relies on specific libraries being downloaded or referenced or something similar. When it came to implementing an API within our project, we were very confused as to which API to choose and how to get it working which was where most of our time was allocated to. Most of the popular APIs out there required complicated authentications, logins, or a fully working application that had to be approved by the developers before even being able to use their API. This was a decent refresher on D3 though as recalling on past knowledge was necessary for implementing the graphs.

Self-Evaluation

Dominic

Self Evaluation Rubric						
Did you complete the assignment and did you complete it on time?	Submitted on time	Up to 1 day late	Up to 2 days late	Up to 3 days late	4 days late or more	Do you need to clarify? No
	X	☐	☐	☐	☐	
Did you collaborate with a partner?	Worked with partner			Worked alone		Do you need to clarify? No

	X			☐		
Did you put in earnest effort and provide an articulate summary of your experience?	Excellent	Pretty good	About average	Could be improved	Not this time	What supports this? I tried to explain my process as thoroughly as possible. Used many pictures.
	X	☐	☐	☐	☐	
Was the assignment complete, with minimal errors, correct output, and good style?	Excellent	Pretty good	About average	Could be improved	Not this time	What supports this? Had two people look over each part of the assignment.
	X		☐	☐	☐	
How much EXTRA effort did you put into the assignment?	A lot of extra effort		Some extra effort		Not this time	What supports this? Just wanted it to be done after a while.
	☐		X		☐	
Reflection in section above.						

Nathan

Self Evaluation Rubric						
Did you complete the assignment and did you complete it on time?	Submitted on time	Up to 1 day late	Up to 2 days late	Up to 3 days late	4 days late or more	Do you need to clarify? No
	X	☐	☐	☐	☐	
Did you collaborate with a partner?	Worked with partner			Worked alone		Do you need to clarify? No
	X					
Did you put in earnest effort and provide an articulate summary of your experience?	Excellent	Pretty good	About average	Could be improved	Not this time	What supports this? Our process is well documented with adequate pictures that show every step of the project.
	X	☐	☐	☐	☐	

Was the assignment complete, with minimal errors, correct output, and good style?	**Excellent**	**Pretty good**	**About average**	**Could be improved**	**Not this time**	**What supports this?** **The writeup is sufficient enough for what the teaching team expects and our code is neatly organized.**
	X		☐	☐	☐	
How much EXTRA effort did you put into the assignment?	**A lot of extra effort**		**Some extra effort**		**Not this time**	**What supports this?** **Understanding how to use an API was a bit of a headache and took up most of my effort.**
			X		☐	
Reflection in section above.						