Team Phase 1:

**Data we are drawing from:**

The database that we used consists of 508 JSON objects. These objects had 7 key value pairs that we used generate the results that you see on our project. These value pairs are name, length, height, speed, type, park, and continent.

**How we chose to display the information:**

We used 1 bar chart and 9 pie charts to display our information. Our bar chart illustrates the total number of roller coasters on each continent. Our pie charts are broken into 3 length related charts, 3 height related charts, and 3 speed related charts. The first chart in each category shows how many rollercoaster from our database fall into the given category ranges. The second chart shows the amount of steel coasters vs wooden coasters within its given category. The final chart in each category shows the top 200 coasters in the divided into their respective continent of origin.

**Looking at our raw data:**

If you were curious about the data that we are using to construct these charts, you can click on our table button to display the information. The table has the option to sort our data by length, height, or speed, and display it in ascending or descending numerical order.

**Angular Overview:**

-Angular is a platform and framework for building client applications in html and typescript. Angular applications are written using typescript (similar to javascript).

-Angular is developed by google.

-Angular has 3 versions (Angular js, Angular 2, Angular 4, Angular 5 on the way?)

-Websites built with angular (The Guardian, Paypal, Netflix, Vevo, . . . )

**Angular Basic Terminology:**

-The basic building blocks of angular are ngModules which provide a compilation context for your components.

-Angular components define the views (what the user sees on the screen).

-Templates are a combination of HTML and Angular markup that modify the HTML displayed in the view.

-Angular directives are programing logic that can be put into the HTML markup.

-Property Binding is a feature that allows you to pass data from the component class to an element in the component’s template.

-Event Binding is making your app listen for certain user actions and then updating the app accordingly.

-Information is supplied to the components via services. Services perform operations that are not related to the view.

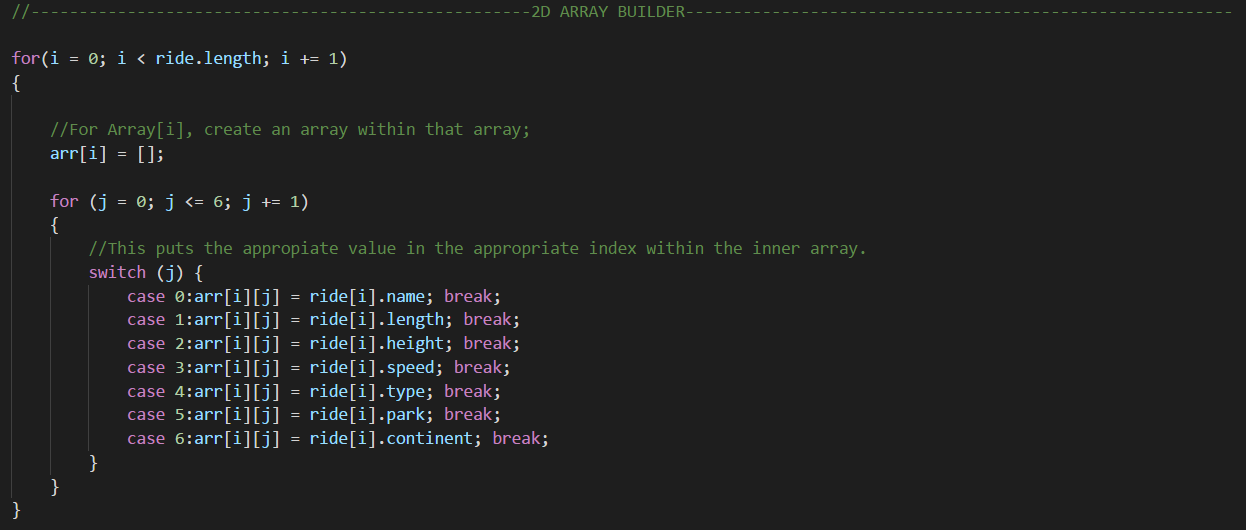
**Insertion Sorting Algorithm:**

The algorithm that our group was assigned is the insertion sort. This algorithm sorts the data presented one item at a time. It is not considered an efficient algorithm when working with large data sets, but is works great when working with smaller amounts of data.

How it works:

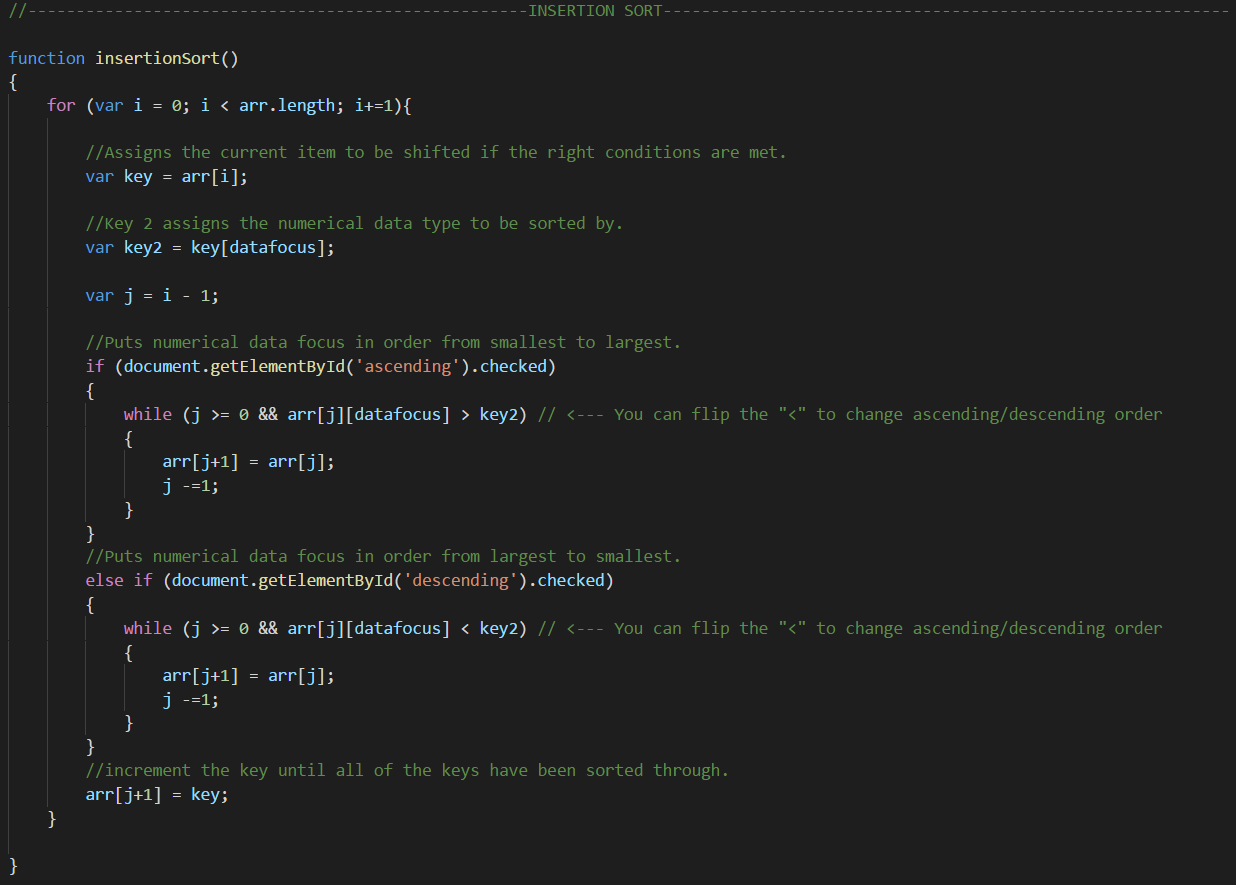
Step #1:

First, we put all of the info from our JSON file into a 2D array. The code below would create 508 lists containing our 7 value pairs. The reason that we used a 2D array is because when we sort the items and switch the placement of 1 item, we can switch the placements of all the corresponding items by referencing (i) instead of (j). Keeping the correct value pairs within parallel 1 dimensional arrays would have been less efficient and more confusing.



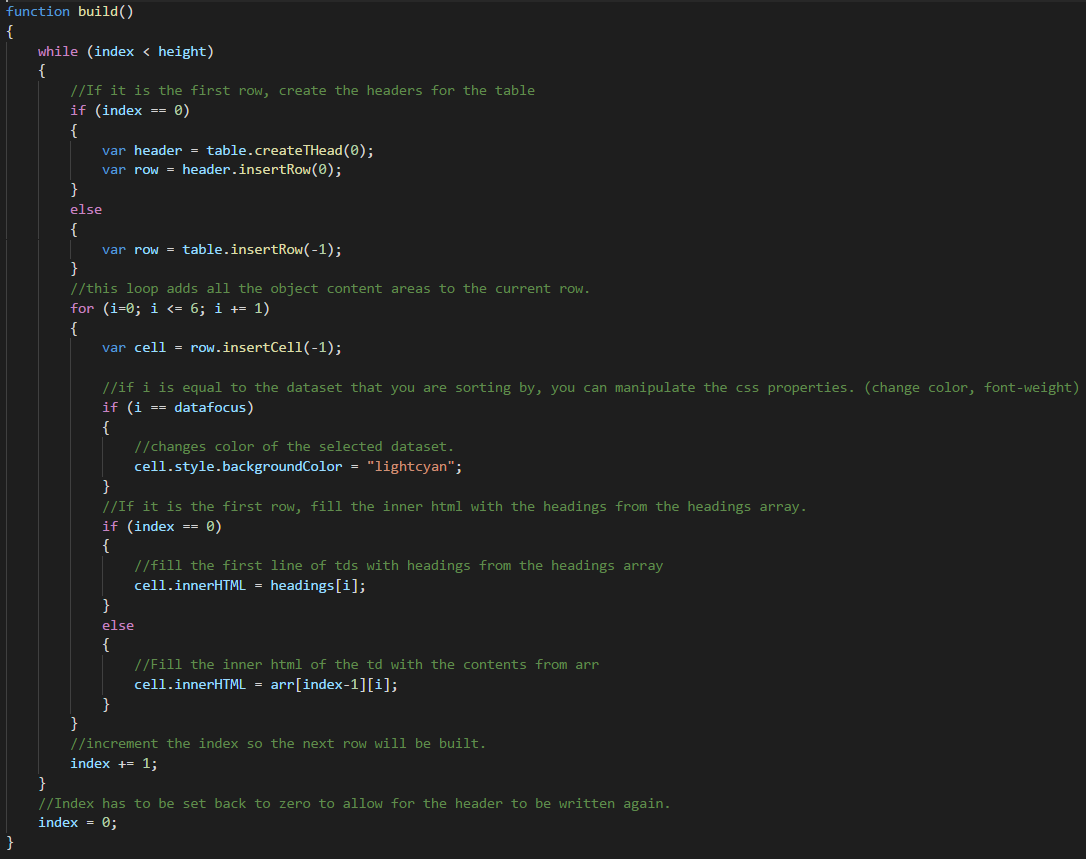
Step #2:

We take the arr[] array and use the insertion sort to manipulate the placements of arr[]’s values. The coaster being evaluated is assigned to key, whereas the value pair being sorted by, is assigned to key2.



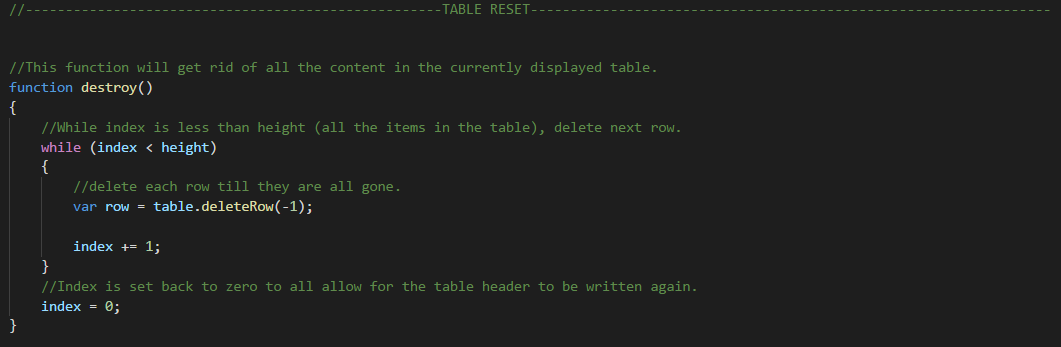
**Table in Phase 1:**

The table that we have in our team phase 1 version is not coded into the html page. It is procedurally generated from our build function. The while loop checks to see if our index variable is less than the number of items in our JSON file. If the while condition is true, a table row is generated. For the last generated row, 7 cells will be created (1 for name, height, length, speed, type, park, and continent). Once the cell is created, the correct value from our arr[] array is placed in the cell. These two loops happen until we have 508 rows with 7 cells each.



**Getting Rid Of The Table Or Sorting By a Different Category:**

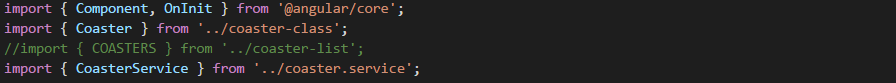
If you want to sort by a different value or get rid of the table, we opted to just destroy the entire thing and rebuild it from scratch if necessary. Trying to target individual cells and assign them different values seemed like it would be a lot of extra code, and destroying it seemed like it would be quicker and more efficient to us. The code below just loops 508 time and for every time deletes the corresponding row until the table is gone.



**Table in Phase 2:**

In Team Phase 2 we created a component to create and manipulate our table. Creating the table this time around was much easier because of some of Angular’s features.

**Here is the list of our imports necessary for this component:**

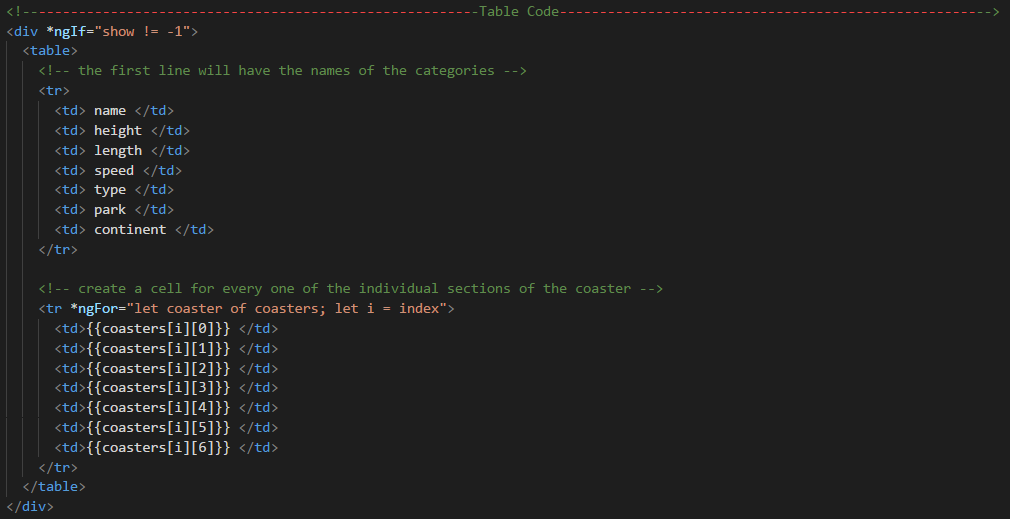


**Here is out component controller’s code:**



In the constructor we declare a private instance of our CoasterService and then in the getCoasters function we call methods from that service and assign them to our coasters property.

**Our HTML template for the table component looks like this:**

­

We set up the table element and define the first row and first 6 cells in our template. From here, we create every other row and cell in the table with Angular’s ngFor directive. ngFor allows us to loop through all of the JSON objects in our database and create a row and 7 cells for each. We then bind the data from our CoasterService.getCoasters function to each cell using interpolation. Because all of the data presented in the table is bound to a specific row and cell, we didn’t need a destroy function this time around. If you sort by a different parameter, the correct value is automatically place in the right spot.

**Slider in Phase 1:**

In team phase 1 we created our sliders by creating a parent container and giving it the position absolute property and then placing a child container in it and giving it the position relative property. We then manipulate the position of the child container by adding or subtracting from the left property of the child in our slider.js file. If you add to the left value, the slider moves to the right. If you subtract from the left property, the slider move to the left. Were things got tricky was using 1 js file to manipulate 3 separate sliders. We had to use document.getElementByClassName a lot and things got confusing pretty quick.

**Slider in Phase 2:**

We created a slider component in phase 2. Creating the sliders using components allowed us to make things less complicated because each slider operates independently, whereas all 3 sliders in team phase 1 were operated by one js file. If you wanted to add an additional slider to our team phase 1, you would have had to make a lot of adjustments in the slider.js script to account for an additional element added to the class. In phase two you could add or take away sliders and things would continue to function as normal because the code manipulating each one exists independently.

**Here is the html template for our slider component:**



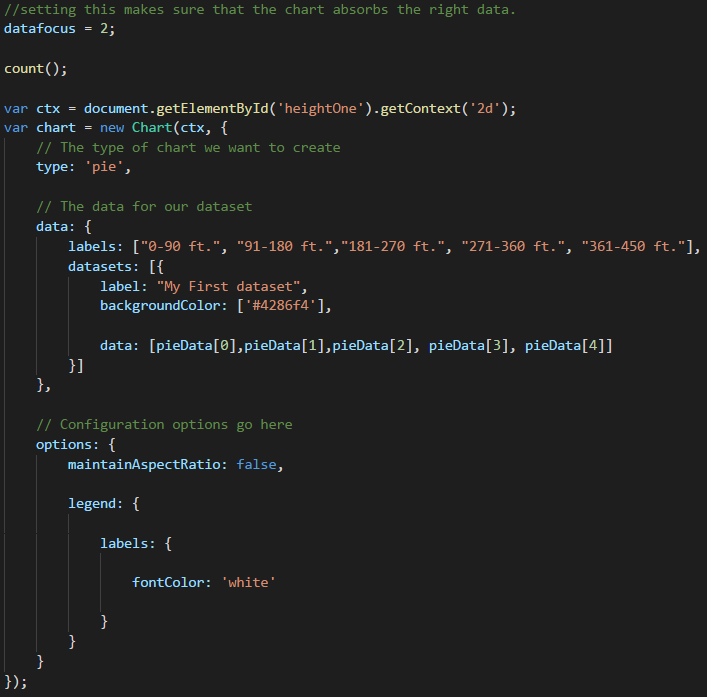
One of the aspects that made developing a slider in Angular easier than in vanilla js is property/attribute binding. In order to manipulate a specific element in vanilla js, you have use document.getblahblahblah. In Angular you can bind properties from components to elements which gets rid of the need to use document.get. In the example above we bound the style width css property to the variable screenwidth. Doing this made it so whenever the user resizes the window, the PieChartContainer class is resized to the appropriate width. We also bound the style left css property to the posX variable. This allowed us to control which chart within our slider was displayed. In our team phase 2 we did not use any document.getblahblahblah functions. All instances of that approach were redone using property binding.

The one last feature to take notice of within this component is the passing of information from the parent component to the child component. In this example the slider component is passing items from the chartNumber[] array to the child chart components. The chart component can then read this information and perform the appropriate task.

**Charts Team Phase 1:**

In team phase 1 we created the charts by just reusing the same js code over and over and over.

**The code for building the chart looked like this:**



Since we had so many charts we had this code copy and pasted about 12 times for all of our charts. The reason that we did this was because we needed a different ID for each canvas that we were drawing the chart on. This approach worked but was dumb at the same time because we had so much of the same code. It would have been nice to be able to re-use the same chart template for all of the charts.

**Charts Team Phase 2:**

In team phase 2 we created a chart component. This component allowed us to not have to re-use the same code over and over and over. The way that we approached this component was similar to the way that we approached the slider. Each chart is controlled independently rather than being controlled by 1 big js file.

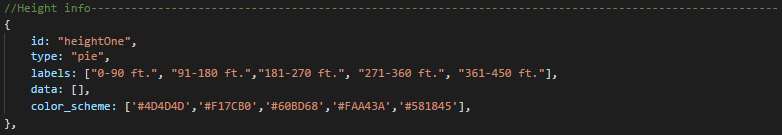
**Here is some the controller code for our chart component:**



This is the same basic charts js code template from team phase 1. The notable differences here are the assignment of new Chart at the top. Usually after new Chart you have to put an element id in the parentheses. In this case we are assigning the ids dynamically. This means that rather than hard coding the values we are retrieving them from the parent container. When we were passing the sliderComponent variable chartNumber[] to [childMessage] input, the chart component is reading that input and assigning it to this.elemid.

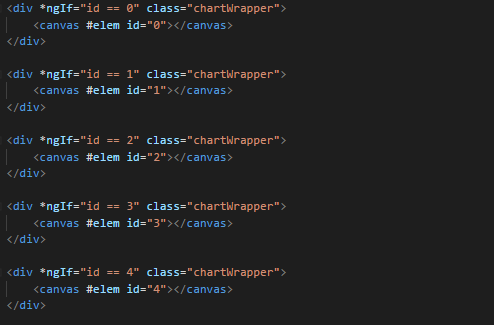
The info put into the chart template is supplied by our chart-info-list file. Our chart –info-list contains a list of chart-info objects. There is one chart-info object for each chart displayed in our application.

**Here is a look at an object from our chart-info-list:**



One are of difficulty with this component was getting the info to display the right way.Normally the components start with ngInit(){} but in this case we had to call ngAfterViewInit because the appropriate chart would load onto the canvas but it would not display the information provided. This was because the chart had not received the info before it loaded the view. Calling ngAfterViewInit allowed us to make sure that the component had all the info it needed before it loaded the view.

**Here is a look at the html template for the chart component:**



There is more instances of the ngIf directive here. The ngIf statements just make sure that the right chart is displayed.

CoasterService:

Services are an object that supply data to your various components. Rather than coding all of your logic and data retrieval directly into your components themselves, you can create a service that does all of this for them. By creating a service, you can then call that service to supply all the components with their necessary information. This allows for a more streamlined design because the components themselves are only controlling what is displayed on the screen rather than the data manipulation. If you had to get rid of a component, you would not need to find a new place to put your logic but with the service, you don’t have to worry about that. The service we created is the file that contains all of our sorting and data manipulation. The only thing that a component needs to access this information is to import the service and then call, this.coasterService.method name.