

# Angular Demo Walkthrough

This is a step-by-step guide for building the example Angular application [Mercury](#).

## Table of Contents

---

- [Technology Used](#)
- [Prerequisites](#)
- [Project Setup](#)
- [Building Our First Component](#)
- [Passing Data Between Components](#)
- [Calling AWS Webservice](#)
- [Promises, Promises, and more Promises](#)
- [Display the Data](#)
- [Our Second Component – Main Content](#)
- [Our Third Component – Angular Material Table](#)
- [Our Fourth and Fifth Components - Large Icon Cards](#)
- [Last Component - the "Graph" cards](#)

## Technology Used

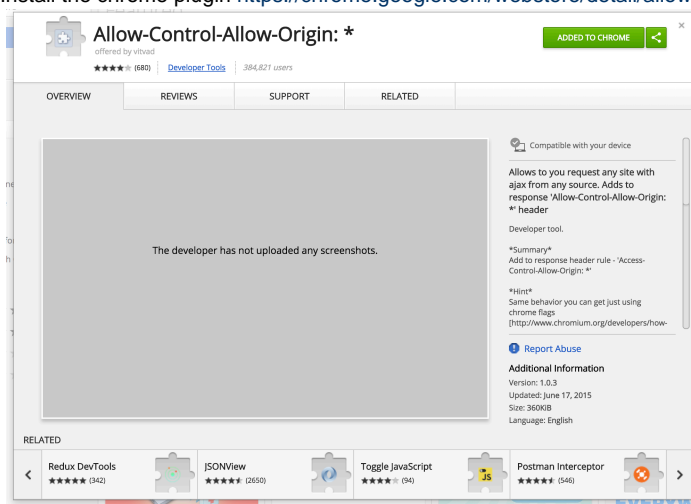
---

- [Angular](#)
- [Angular CLI](#)
- [Angular Material](#)
- [nodejs](#)
- [AWS](#)
- [SASS](#)
- [Git](#)
- [Typescript](#)

## Prerequisites

---

- See [CSX Computer Setup for Angular](#)
- [Chrome browser](#)
  - Install the chrome plugin <https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfgkhkpbjhdihlkkiljbi>



## Project Setup

---

We will start by getting the initial code for the project. This code is currently hosted on [GitHub](#) and [Bitbucket](#):

Github link:

<https://github.com/csx-technology/csx-ng-gps>

Bitbucket link:

<https://git.csx.com/users/t5693/repos/csx-ng-gps/browse>

To start, clone the repo, cd into folder, grab all the tags, and checkout the tag shown below:

```
git clone {{REPO NAME }}
```

```
cd csx-ng-gps/
```

```
git fetch --tags
```

```
git checkout -b my-branch v1.0
```

You should now see a folder "NgGps". cd into this folder.

We will now install all the external dependencies needed by our project using npm. Interested in npm? Learn more about it [here](#).

```
npm install
```

Once the dependencies are finished downloading, we will serve our Angular project using the angular-cli. The command to do this is:

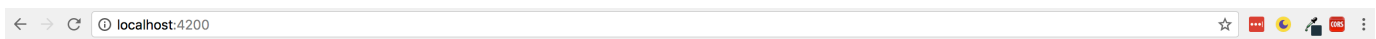
```
ng serve
```

You should see something similar to below in the console:

```
[Wed Feb 21 11:42:49] anthonymiscialo : ~/Documents/CSX/Hackathon/Hackathon-2018/walkthrough/ng-loco-gps/NgLocoGps
ng serve
** NG Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
Date: 2018-02-21T16:43:18.294Z
Hash: 71d4add4c554658f2175
Time: 6000ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 20.6 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 566 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 34.6 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.44 MB [initial] [rendered]

webpack: Compiled successfully.
```

This serves the application on <http://localhost:4200/>. For the purpose of this demo, we will be using the Chrome browser, so open up the link in Chrome. You should be seeing the below welcome page:



Welcome to app!



Here are some links to help you start:

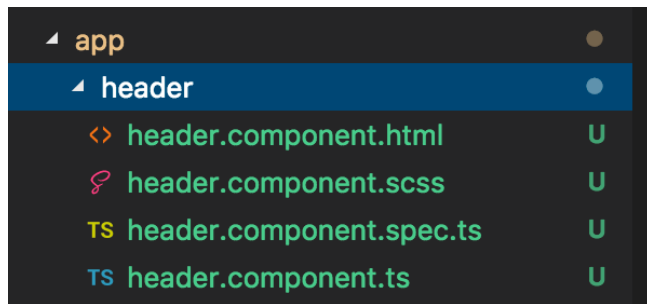
- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

## Building Our First Component

We will now use the angular-cli to build our first angular component, the header. The angular-cli provides us with an easy to use approach to building a component in our app. To generate a new component, we will run the command:

```
ng g c header
```

This is shorthand for "ng generate component header". This basically says "angular-cli, generate a new component, called header, and provide any setup needed so i can use this component". You should now see the header component in your project, shown below:



The files generated include:

- header.component.html
  - This is the file that holds all the html specifically for this component
- header.component.scss
  - The Sass file that will hold our component specific styles
- header.component.spec.ts
  - Used for automated testing of the component
- header.component.ts
  - Our typescript file that holds the logic of our component.

To use a component in Angular, we need to tell Angular that the component exists and will be used. This is done in the app.module.ts. Luckily, when we generate the component using the angular-cli, it does this for us!

If you open up the app.module.ts, you will see the "HeaderComponent" is both imported, and declared under the declarations. The app.module.ts is shown below:

```

import { MaterialModule } from '../material/material.module';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserAnimationsModule } from
'@angular/platform-browser/animations';

import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    MaterialModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

If you open up header.component.ts, you will see that our typescript class is decorated with the "@Component" decorator, telling Angular that this is an Angular component. You will also notice that the component has a "selector" called "app-header". This is the selector we will use in our code to generate and show this component.

```

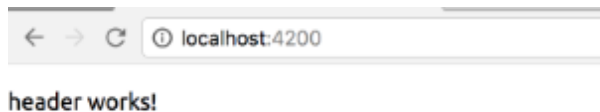
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnInit {

```

To show our app header, we will simply add it to our app.component.html page. Open that file, delete all the code out of it, and add the line shown below:

```
<app-header></app-header>
```

You should now see "header works!" on your page.



If you open up header.component.html, you will see that this is where the code is coming from. Another interesting thing you can do is inspect the console in Chrome and see the component that was generated.



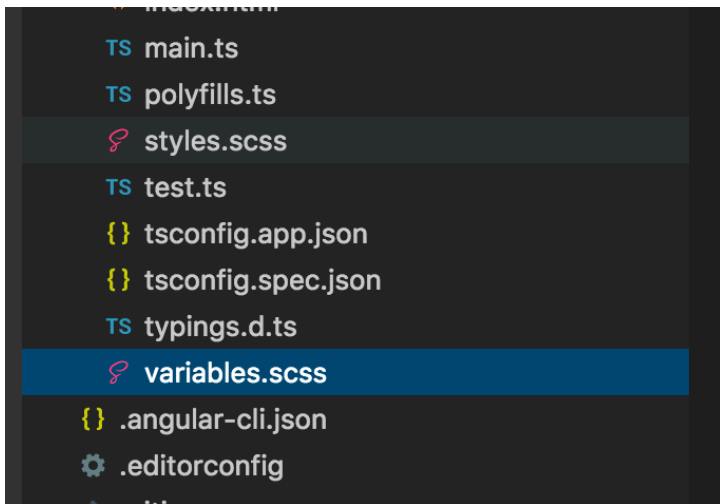
Here we can see our app-root (generated by our app.component.ts) and our app-header (generated by our header component).

Let's now write the actual code for our header. Delete the code generated from the angular-cli in the app-header and the basic code structure shown below:

```
<div class="header">
  <div class="header-top">
    <div class="logo-container">
    </div>
    <div class="header-left">
      <div class="title">Mercury</div>
    </div>
  </div>
  <div class="header-bottom">
    <div class="user-input-container light-input">
      <div class="user-input">
      </div>
      <div class="user-input">
      </div>
      <div class="user-input">
      </div>
    </div>
  </div>
</div>
```

This is the basic html structure of our header. We have an overall "header" container, as well as a "header-top" for the the app name, and the "header-bottom" for the actual user input.

Let's work on getting the "header-top" section working. Let's start by working on some styles. For this, we are going to make our fist use of Sass by using a Sass variable. Open the variables.scss file shown below:



This is a file we are going to use to hold all of our Sass variables, so we can make use of the @import tag, another cool Sass feature that imports one Sass file into another. In variables.scss, let's add the below variables:

```
$lite-purple: #3f51b5;  
$purple: #283593;  
$pink: #e91e63;  
$grey-title: #57595b;
```

The '\$' tells Sass that this value will be used as a variable. We can think of this like a variable you may have used in Java/JavaScript.

Let's now make use of the @import feature and import variables.scss into header.component.scss. Open header.component.scss and add the below code:

```
@import "~variables.scss";
```

We now have use of our colors that will be using throughout our application. Adding the below CSS with the variable \$lite-purple should give us a header with a purple background of #3f51b5.

```
.header-top {  
  background-color: $lite-purple;  
  height: 66px;  
}
```

Mercury

If you inspect the header in the Chrome developer console, you can see that Sass actually converts our \$lite-purple variable to #3f51b5.

```

}

.header-top[_ngcontent-c1] {
  background-color: #3f51b5;
  height: 66px;
}

```

We can add our logo as well. Images for this application are located in `/src/assets/images`. The image we will be using is `logo.svg`. We can use an `svg` image the same as a `png` or `jpeg` using an `html <img>` tag. Add this image to the page between the "logo-container" div using the below code:

```

<div class="logo-container">
  
</div>

```

Let's also add some styles to make our header look the same as the mockup presented above. Try to style it yourself first and then look at the code below to see one possibility:

▼ See CSS...

```

.header-top {
  background-color: $lite-purple;
  display: flex;
  align-items: center;
  height: 66px;
}

.logo-container {
  width: 5%;
  display: flex;
  justify-content: center;
  align-items: center;
}

.logo-container img {
  width: 50px;
  height: 50px;
}

.title {
  color: #fff;
  font-size: 24px;
}

```

Your header should look similar to the below screen grab:



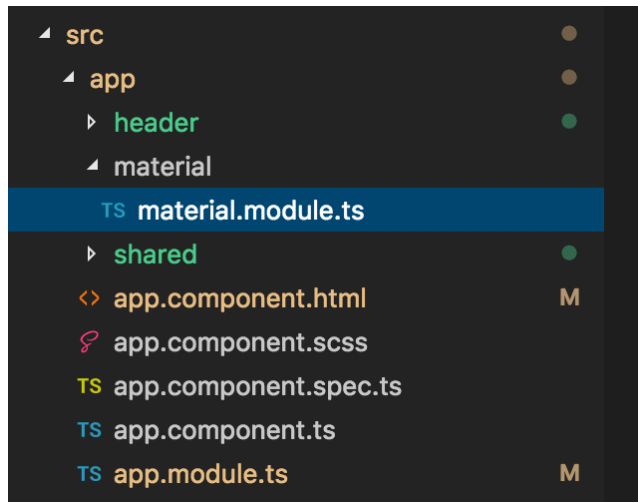
Let's now work on the user input portion of the header.

For our user input controls, we will be making use of an awesome library maintained by Google called [Angular Material](#). Clicking on the link will show you the docs and all the cool components that are available.

For the user input, we will use the inputs provided by Angular Material. The code for adding these to the bottom header is shown below:

```
...  
<div class="user-input-container light-input">  
  <div class="user-input">  
    <mat-form-field>  
      <input matInput placeholder="Train ID">  
    </mat-form-field>  
  </div>  
  <div class="user-input">  
    <mat-form-field>  
      <input matInput placeholder="Subdivision">  
    </mat-form-field>  
  </div>  
  <div class="user-input">  
  </div>  
</div>  
...
```

If you save, you should see an ERROR! This is because we haven't told Angular that we are using a "mat-form-field" component. We can add this to our "Material" module, which is already a dependency in the project. Open material.module.ts:



In this file, we will add our new component dependencies to both the import and export sections, as well as importing them at the top. This code is shown below:

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { MatFormFieldModule, MatInputModule } from '@angular/material';

@NgModule({
  imports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule
  ],
  exports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
  ],
  declarations: []
})
export class MaterialModule { }

```

This module is already added as a dependency in the app.module.ts so all of our Angular Material components we add here can be used anywhere in our project. This separate module is one of the great things about Angular, MODULARITY. We can add/remove modules easily, and only include what we are actually using in our project.

We will need to add some more styles to our page to see the inputs, as they are styled white right now (these will be placed in header.component.scss):

```

.header-bottom {
  background-color: $purple;
}

.user-input-container {
  display: flex;
  align-items: center;
  justify-content: space-between;
  width: 600px;
}

```

Here you can see we once again make use of Sass variables and Flexbox.

Let's add our button to the page now as well. We will use the mat-button Angular Material component. Start by adding it to material.module.ts, and then add it to the page:

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { MatFormFieldModule, MatInputModule, MatButtonModule } from
 '@angular/material';

@NgModule({
  imports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule
  ],
  exports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule
  ],
  declarations: []
})
export class MaterialModule { }

```

```

<div class="user-input-container light-input">
  <div class="user-input">
    <mat-form-field>
      <input matInput placeholder="Train ID">
    </mat-form-field>
  </div>
  <div class="user-input">
    <mat-form-field>
      <input matInput placeholder="Subdivision">
    </mat-form-field>
  </div>
  <div class="user-input">
    <button mat-raised-button color="primary">Search</button>
  </div>
</div>

```

We are also going to add a app wide style that will be used by multiple pages called "inner-wrapper" that will help keep our content in the center of the page. We will add this new style to styles.scss:

```

.inner-wrapper {
  padding: 0 5%;
}

```

We can add it to our "header-bottom" class like shown below and our content will now be aligned nicer:

```
...  
<div class="header-bottom inner-wrapper">  
...
```

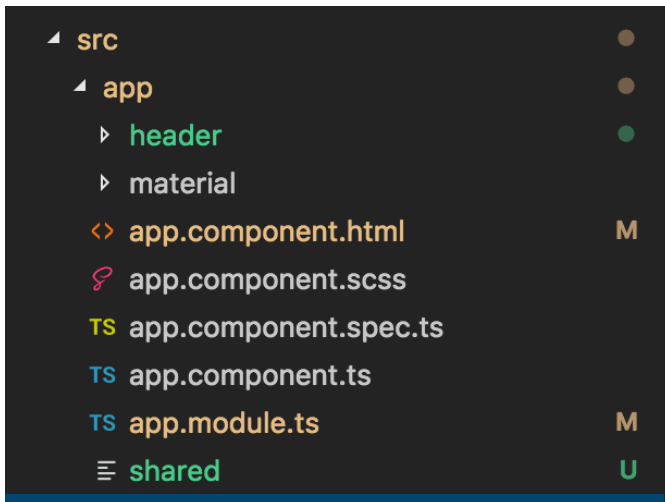
Your header should now look similar to below:



We can now move onto hooking up our UI with the nuts and bolts of our application.

We will start by opening up header.component.ts and adding our properties for train id and subdivision. You can see that this is a .ts file, which stands for Typescript. Typescript is a super set of Javascript that provides OPTIONAL static typing (you'll see this throughout the demo application), classes, interfaces, and much more. When we build our application, our Typescript files are transpiled into plain vanilla Javascript.

For our user input, lets create a model object that will hold the train id and subdivision. Start by creating a folder under /src/app called "shared" where all of our model objects will live. This folder is shown below:



In this shared folder, create a file called headerUserInput.model.ts. The format for a Typescript class is shown below:

```
export class HeaderUserInput {  
  constructor() {}  
}
```

In this file, lets add the two fields we need (train id and subdivision) and statically type them as strings. Interested in why static typing is useful in development? Read this article [here](#).

```
export class HeaderUserInput {  
    public trainId: string;  
    public subdivision: string;  
    constructor() {}  
}
```

Our header.component.ts will now make use of this new model object. We can use it similar to how we would model objects in Java. Open header.component.ts and add the code below:

```
export class HeaderComponent implements OnInit {  
  
    userInput = new HeaderUserInput();  
  
    constructor() { }  
  
    ngOnInit() {  
    }  
  
}
```

Don't forget to import HeaderUserInput!

```
import { HeaderUserInput } from '../shared/headerUserInput.model';
```

To hook up the UI to our header component typescript file, we will use 2-way databating in Angular. This simple means if a value changes on the UI, it will AUTOMATICALLY change in the Typescript, and if a value is changed in the Typescript, it will AUTOMATICALLY udpate the UI. This is one of the amazing and truly usefull features of Angular. We can have dynamic web applications with minimal effort from you, the developer. To implement 2-way databinding, we will make use of ngModel attribute of the input element.

```

...
    <div class="user-input">
      <mat-form-field>
        <input matInput placeholder="Train ID"
[(ngModel)]="userInput.trainId">
      </mat-form-field>
    </div>
    <div class="user-input">
      <mat-form-field>
        <input matInput placeholder="Subdivision"
[(ngModel)]="userInput.subdivision">
      </mat-form-field>
    </div>
...

```

The "[ ( ) ]" syntax in the above example tells Angular we want to do 2-way databinding. More info on ways to bind in Angular can be found [here](#). The ngModel is a special property of the input component and can be used by a few other html components as well.


To show that this is working, add the below code to the ngOnInit() method in header.component.ts. This method will fire after the creation of the component, similar to the @postConstruct decorator in Java.

```

...
ngOnInit() {
  this.userInput.trainId = 'Q21602';
  this.userInput.subdivision = 'C3';
}
...

```

You should now see these values on the UI"


**Mercury**

Train ID	Subdivision	
Q21602	C3	<button>Search</button>

I would leave these hardcoded values in the ngOnInit() for now, as it makes testing the application a little easier (you won't have to keep entering those values before hitting search).

Let's now wire up the Search button. For this, we will use [event binding](#). In our header.component.ts file, lets add a search() method and log the user input to the console:

```

...
search() {
  console.log(this.userInput);
}
...

```

The code for the UI is:

```
...  
<button mat-raised-button color="primary"  
(click)="search()">Search</button>  
...
```

This basically says when the "click" event is triggered, call search(). If you test this, you should see the entered values in the console.

We've now completed the first part of our demo!

## Passing Data between components

---

To get all the code from above up to this point, you can check out the next tag v1.1:

```
git checkout -b my-branch-v1 v1.1
```

We will now work on passing data between between the header and our controller, the app component (app.component.ts).

Open header.component.ts. We are going to add an [EventEmitter](#) to that will push our data from the header component to the app component. To use an event emitter, we are going to add a new property to our header component called searchDataset that will emit our user info when the user clicks search.

```
...  
  @Output()  
  searchDataset = new EventEmitter<HeaderUserInput>();  
...
```

Notice we decorate our searchDataset with the @Output decorator, and have it be set to a new EventEmitter that is paramitorized with our HeaderUserInput model.

Don't forget to add the @Output and EventEmitter as imports (hint: they are apart of @angular/core)!

```
...  
import { Component, OnInit, Output, EventEmitter } from '@angular/core';  
...
```

We now want to emitt this event when the user clicks search, and pass our user info with it. To accomplish this, we will do the following in our search method:

```
...
  search() {
    this.searchDataset.emit(this.userInput);
  }
...
```

Our header component is emitting an event, but our app component isn't listening for it. To accomplish this let's create a method in `app.component.ts` that will trigger when an event is emitted from our header component:

```
...
export class AppComponent {
  title = 'app';

  onSearchDataset(userInput: HeaderUserInput) {
    console.log(userInput);
  }
}
...
```

Don't forget to import `HeaderUserInput` into the `app.component.ts`!

To trigger this event, we will tell our app component to listen for a "searchDataset" event emitted from the header component, and when this event occurs, to call "onSearchDataset()"

```
<app-header (searchDataset)="onSearchDataset($event)"></app-header>
```

The "\$event" is convention in Angular and is our "userInput" we passed in from the header component. So the line above is saying "when the searchDataset event is triggered, call onSearchDataset in `app.component.ts` and pass in \$event (the user input in this case)"

If you click the search button, you should see the user input displayed in the Chrome console.

## Calling AWS Web Service

---

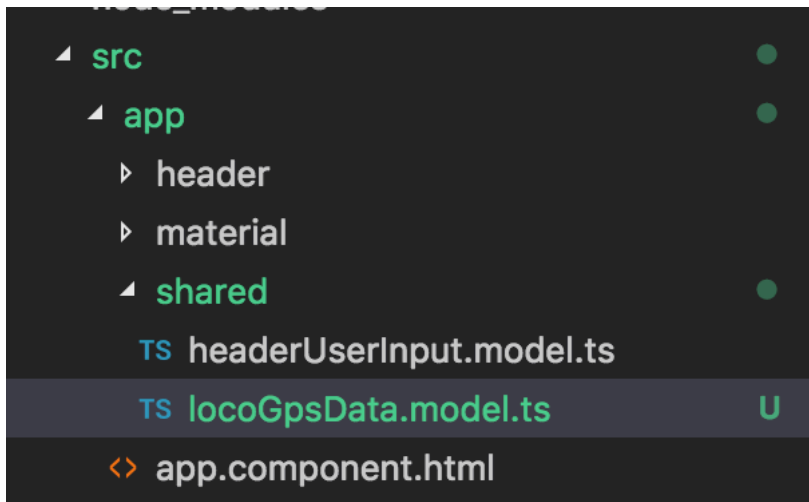
To get all the code from above up to this point, you can check out the next tag `v1.2`:

```
git checkout -b my-branch-v1-2 v1.2
```

Let's now work on creating a service and injecting it into our app component. This service will take in a train id and subdivision as parameters, and return a list of loco gps pings.

To begin, let's create a new model object called `locoGpsData` that will be a representation of the data returned by the service. The class will look as follows:

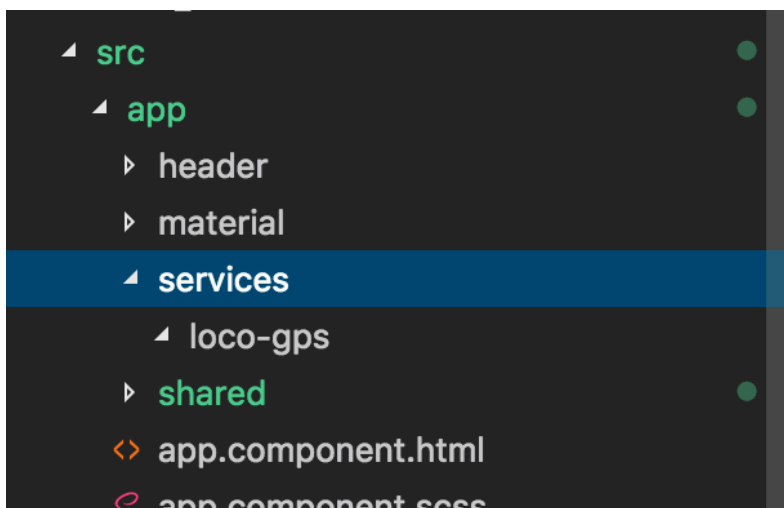




```
export class LocoGpsData {
  constructor(
    public device_i,
    public latitude_m,
    public loco_i,
    public loco_speed_m,
    public longitude_m,
    public report_est_d,
    public signal_quality_c,
    public signal_strength_m,
    public train_i,
    public trans_subdiv_c) {}
}
```

We are now going to create our first service class. Before we do, create a service folder in /src/app.

Inside this folder, create another folder just for our locoGpsService called loco-gps:



Navigate to this folder on the command line (/src/app/services/loco-gps) and we can use the angular-cli to generate a service here for us:

```
ng g s loco-gps
```

This will generate 2 files, the service and the automated testing for the service. If you open loco-gps.service.ts, you'll notice it is decorated with the @Injectable decorator. This tells Angular that this class can be injected into another class with [dependency injection](#) (more on this later).

```
import { Injectable } from '@angular/core';

@Injectable()
export class LocoGpsService {

  constructor() { }

}
```

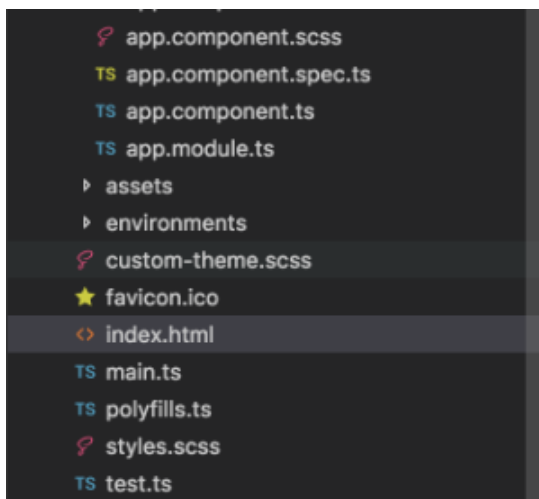
In order to use our AWS service, we have to take a few setup steps:

1. Add the script tags to <head> in index.html:

```

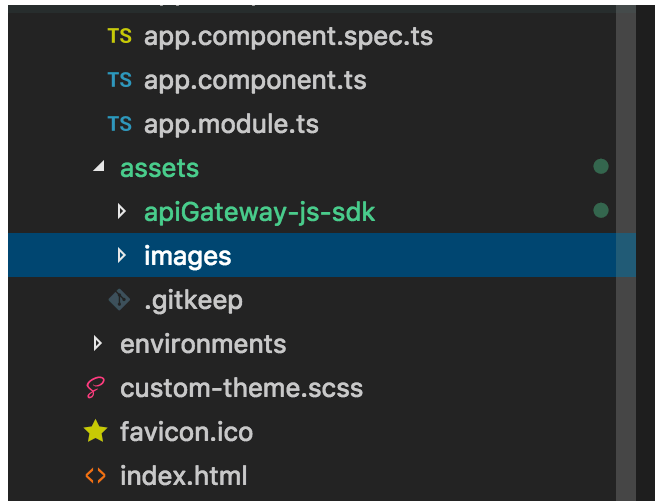
...
<head>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/axios/dist/axios.standalone.js"></s
cript>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/CryptoJS/rollups/hmac-sha256.js"></
script>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/CryptoJS/rollups/sha256.js"></scrip
t>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/CryptoJS/components/hmac.js"></scri
pt>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/CryptoJS/components/enc-base64.js">
</script>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/url-template/url-template.js"></scr
ipt>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/apiGatewayCore/sigV4Client.js"></sc
ript>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/apiGatewayCore/apiGatewayClient.js"
></script>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/apiGatewayCore/simpleHttpClient.js"
></script>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/lib/apiGatewayCore/utils.js"></script>
  <script type="text/javascript"
src="assets/apiGateway-js-sdk/apigClient.js"></script>
...

```

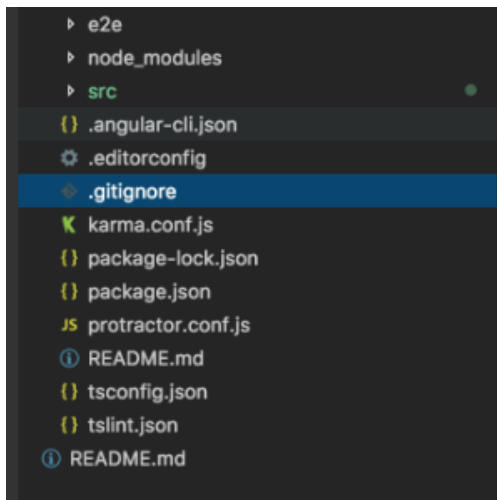


2. Download the "apiGateway-js-sdk" from S3 and put it into /src/assets
  - a. Sign in to <https://bit.ly/csx-aws>

- b. Download the apiGateway-js-sdk.zip from <https://s3.console.aws.amazon.com/s3/buckets/csx-hackathon-2-service-ohio/?region=us-east-1&tab=overview>
- c. Extract the files and then put into project
  - i. \*You may have to stop the server, run an npm install, and an ng serve to get rid of any errors



- 3. Add a new folder in /src/service called "secret". This will hold our aws specific access and secret key (they will be provided the day of the hackathon). Files stored here SHOULD NOT BE UPLOADED TO ANY REPOSITORY! How can we accomplish this? By adding a new rule to our .gitignore:



```
# See http://help.github.com/ignore-files/ for more about ignoring
files.

/src/app/services/secret/*

# compiled output
...
```

- 4. In the "secret" folder, create another model clas called "apiKey.model.ts" and add the below code to it

```
export class ApiKey {
  public accessKey = 'XXXXXXXXXXXXXXXXX';
  public secretKey = 'XXXXXXXXXXXXXXXXX';
  public region = 'us-east-2';
}
```

5. Replace the 'XXXXXXXXXXXXXXXXX' with the provided access/secret keys that are located [here](#)
6. In loco-gps.service.ts, add a const to our apiClientFactory, an instance of our apiKey, and an apiClient used to access the service (or just copy and paste the code below, this part is a little weird 😊):

```
import { ApiKey } from '../secret/apiKey.model';
import { Injectable } from '@angular/core';

declare const apigClientFactory;

@Injectable()
export class LocoGpsService {

  apiKey = new ApiKey();

  apigClient = apigClientFactory.newClient({
    accessKey: this.apiKey.accessKey,
    secretKey: this.apiKey.secretKey,
    region: this.apiKey.region
  });

  constructor() { }

}
```

That takes care of the AWS specific things needed for this app. Let's now implement the service!

Let's create a method in our service called getLocoGpsData() that takes in a trainId: string and subdivision: string. Right now the service will not return anything, but later on we will have it return a Promise<>.

```
...
getLocoGpsData(trainId: string, subdivision: string) {
}
...
```

Let's introduce a little modularity in our code as well because we are going to have getLocoGpsData call AWS, but if we wanted we could have it call a Mock Service for testing as well (this is a little outside the scope of our tutorial, but something to be aware of!). Create a method called invokeAws that takes in a trainId: string and subdivision: string, and prints out to the console the values.

```

...
getLocoGpsData(trainId: string, subdivision: string) {
  this.invokeAWS(trainId, subdivision);
}

invokeAWS(trainId: string, subdivision) {
  console.log('In service! ' + trainId + ', ' + subdivision);
}
...

```

Now that we have the skeleton structure of a service, let's inject it and see if we can print to the console.

Open up app.component.ts. Add a constructor and have the constructor take in a private variable, our LocoGpsService

```

...
constructor(
  private locoGpsService: LocoGpsService
) {}
...

```

Don't forget to import the service in the component as well! This unique syntax is how dependency injection is handled in Angular. Angular is clever enough to provide an instance of this service to us. The syntax is also unique in that values that are passed in as private in the constructor are ALSO available to use throughout the component. Its essentially the same as declaring the following (YOU DON'T HAVE TO ADD THE BELOW TO YOUR CODE, IT IS JUST AN EXAMPLE):

```

...
private locoGpsService;
constructor( private locoGpsService: LocoGpsService) {
  this.locoGpsService = locoGpsService;
}
...

```

Javascript is really awesome like that sometimes 😊!

We are now going to call our service in our previously created onSearchDataset() method (located in app.component.ts) :

```

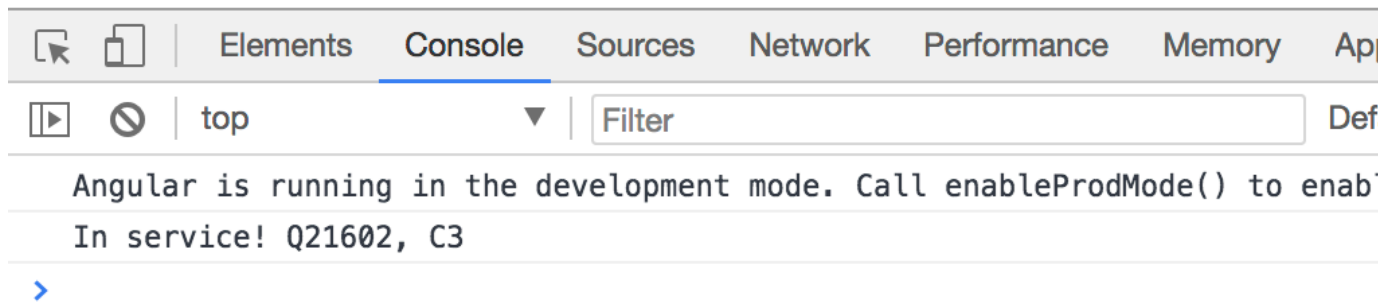
...
onSearchDataset(userInput: HeaderUserInput) {
  this.locoGpsService.getLocoGpsData(userInput.trainId,
  userInput.subdivision);
}
...

```

The last thing we need to do to get our service to work is to add it as a provider in the app.module.ts. This tells Angular that it needs to "provide" this service when needed. We simply add it in the providers [ ], and don't forget to import it up top!

```
...
@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    MaterialModule
  ],
  providers: [LocoGpsService],
  bootstrap: [AppComponent]
})
...
```

If you run the code, you should see the following output in the console:



Congrats! We're starting to make some progress here

## Promises, Promises, and more Promises!

To get all the code from above up to this point, you can check out the next tag v1.3 (if you jumped ahead, you must follow step 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v-1-3 v1.3
```

Now that we have our service correctly injected into our app component, let's get some data.

In `loco-gps.service.ts` remove the print statement and add the below line of code:

```

...
  invokeAWS(trainId: string, subdivision) {
    const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};
  }
...

```

This is the body of our request. We are now going to see our first use of [Promises](#)! Promises are a great feature available to us in Angular (and actually in ES6) for implementing asynchronous communication. In suuuuuper simple terms a promise says "if you make this request, i promise i will give you ONE response" (see this great eli5 explanation [here](#)).

There are multiple formats of promises and we will see a few in our application. Our first format is below:

```

...
  invokeAWS(trainId: string, subdivision) {
    const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};
    const promise = this.apigClient.rootPost('', body, '')
      .then()
      .catch();
  }
...

```

This says "apiClient, post this body of data, THEN do something, or CATCH an error". This apiClient.rootPost actually returns a promise that is then handled by the then clause.

In the then clause, lets see what is being returned. To do this, we can do the below:

```

...
  invokeAWS(trainId: string, subdivision) {
    const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};
    const promise = this.apigClient.rootPost('', body, '')
      .then(
        res => console.log(res)
      )
      .catch(
        res => console.log('ERROR' + res)
      );
  }
...

```

This syntax makes use of ES6 [arrow functions](#). An arrow function is a cool new ES6 feature that provides lexically scoped \$this variables (no more \$that = \$this!) as well as a cool new shorthand for functions. Using arrow functions, the below 2 calls are equivalent:

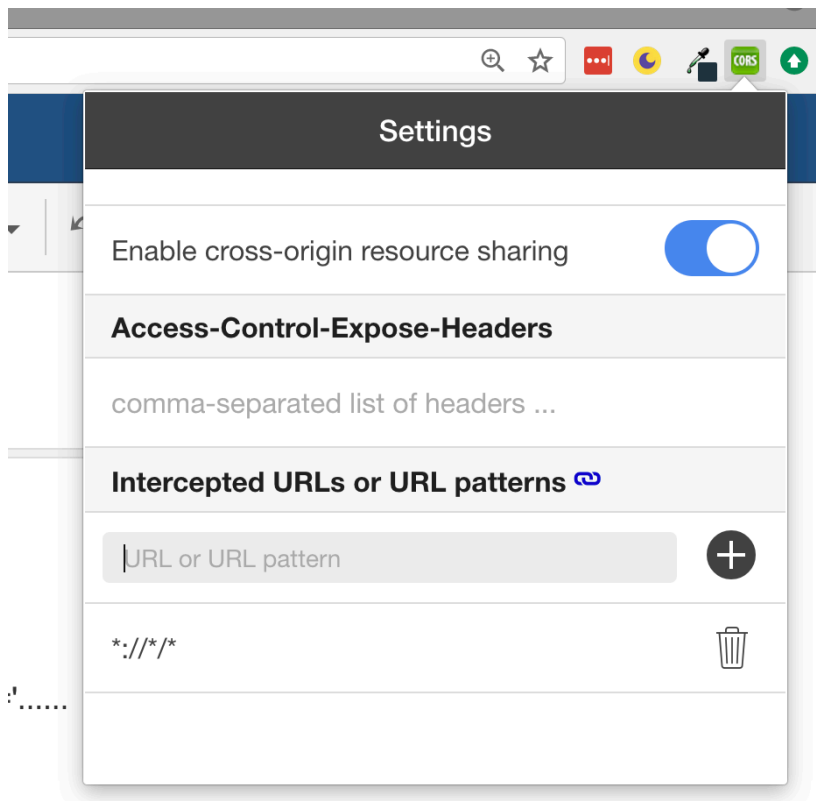


```
res => console.log(res);

function(res){
  return console.log(res);
}
```

So in our above then clause, we are saying taking in res, and print it to the console.

The last thing you need to do before the code will run is start the "CORS" plugin we installed earlier. Interested in what exactly CORS is and why we have to turn it off? Look [here](#). To enable our Cors plugin, click on it in the top right corner of Chrome, and click "enable cross-origin resource sharing". The "CORS" logo should now be green.



If you run the code, you should now get a printout in the console:

```
▼ {data: Array(163), status: 200, statusText: "", headers: {...}, config: {...}} ⓘ  
  ► config: {method: "POST", headers: {...}, timeout: 0, transformRequest: Array(1), tran  
  ▼ data: Array(163)  
    ► [0 ... 99]  
    ► [100 ... 162]  
      length: 163  
    ► __proto__: Array(0)  
  ► headers: {content-type: "application/json"}  
    status: 200  
    statusText: ""  
  ► __proto__: Object
```

What were really interested is in 2 things. We got a status of 200 (YESSSSS GREAT SUCCESS WE ARE AWESOME) and some data was returned (see the "data: Array(163)" portion).

To get his data, we will simply access it as a property of res, as show below:

```
...  
  invokeAWS(trainId: string, subdivision) {  
    const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};  
    const promise = this.apigClient.rootPost('', body, '')  
      .then(  
        res => res.data  
      )  
      .catch(  
        res => console.log('ERROR' + res)  
      );  
  }  
...
```

Now one of the REALLY cool thing about promises, is you can [chain them](#). Basically the then clause of a promise will always return a promise, so we can break our code into FUNCTIONAL peices of code ([functional programming is really cool!](#)). Let's add another then clause that takes our data and maps it to the locoGpsData model we created earlier. We can use a cool maping function available to use in Javascript:

```

...
invokeAWS(trainId: string, subdivision) {
  const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};
  const promise = this.apigClient.rootPost('', body, '')
    .then(
      res => res.data
    ).then(
      data =>
        data.map(item => {
          return new LocoGpsData(
            item.device_i,
            item.latitude_m,
            item.loco_i,
            item.loco_speed_m,
            item.longitude_m,
            item.report_est_d,
            item.signal_quality_c,
            item.signal_strength_m,
            item.train_i,
            item.trans_subdiv_c
          );
        })
    )
    .catch(
      res => console.log('ERROR' + res)
    );
}
...

```

This code basically says "take the data from the previous then clause (its a promise!) and map each item into a new LocoGpsData object".

The last thing we want to do is "chain" this promise back up to the calling functions, so they know an answer has returned. How do we do that? Simply return the promise:

```

...
invokeAWS(trainId: string, subdivision): Promise<any> {
  const body = { 'train_i': trainId, 'trans_subdiv_c': subdivision};
  const promise = this.apigClient.rootPost('', body, '')
    .then(
      res => res.data
    ).then(
      data =>
        data.map(item => {
          return new LocoGpsData(
            item.device_i,
            item.latitude_m,
            item.loco_i,
            item.loco_speed_m,
            item.longitude_m,
            item.report_est_d,
            item.signal_quality_c,
            item.signal_strength_m,
            item.train_i,
            item.trans_subdiv_c
          );
        })
    )
    .catch(
      res => console.log('ERROR' + res)
    );
  return promise;
}
...

```

Notice we added a return statement that returns the promise ("return promise") AND added to the function signature the ": Promise<any>". This is how we do return statements in Typescript.

We also need to return the promise in the function that called invokeAws().

```

...
getLocoGpsData(trainId: string, subdivision: string): Promise<any> {
  return this.invokeAWS(trainId, subdivision);
}

invokeAWS(trainId: string, subdivision): Promise<any> {
  ...
}
...

```

Our loco-gps service is now complete, but we still have a little bit of work to do in app our app component so we can actually see the data.

Open app.component.ts. In the onSearchDataset() method, can you figure out how to log the results returned to the console? Hint: our service returns a promise.....

▼ [See Code...](#)

```

...
onSearchDataset(userInput: HeaderUserInput) {
  this.locoGpsService.getLocoGpsData(userInput.trainId,
userInput.subdivision)
    .then(results => console.log(results));
}
...

```

You should now be seeing something like the below output in the console:

Angular is running in the development mode. Call enableProdMode() to enable production mode.

```

▼ (163) [LocoGpsData, LocoGpsData, LocoGpsData, LocoGpsData, LocoGpsData]
  ► [0 ... 99]
  ► [100 ... 162]
    length: 163
  ► __proto__: Array(0)

```

## Display The Data

To get all the code from above up to this point, you can check out the next tag v1.4 (if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-4 v1.4
```

Now that we have our data in being returned in our app component, let's display it on the UI. To achieve this, we are going to create our going to be creating our second Angular component (main-content) and second service (main-content service).

But before we do that, let's create a model that our main conten will use for data. Under our shared folder, create a new model called mainContent.model.ts:

```

import { LocoGpsData } from './locoGpsData.model';
export class MainContent {
  public locoGpsData: LocoGpsData[];
  constructor() {}
}

```

For now our main content model will only have one property, called locoGpsData, which will be an array of LocoGpsData objects.

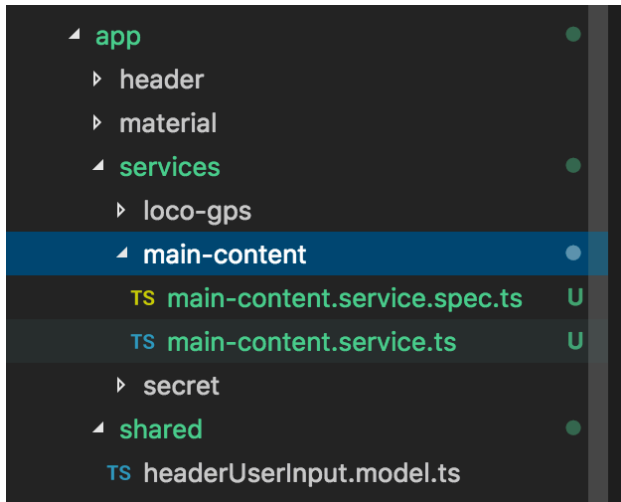
We will now create another service which will take a LocoGpsData array and store it in a MainContent model (obviously this is trivial and we will be doing muucchhhh more in this service later 😊)

To create the new service, navigate to /src/app/services, create a new folder called "main-content". Navigate into main-content and use the

angular-cli to generate a new main-content service:

```
ng g s main-content
```

This will produce the below folder structure:



We will be using this service with dependency injection again, so we must add it to our providers list in app.module.ts, as we did with the loco gps service:

```

import { MainContentService } from
'./services/main-content/main-content.service';
import { LocoGpsService } from './services/loco-gps/loco-gps.service';
import { MaterialModule } from './material/material.module';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserAnimationsModule } from
'@angular/platform-browser/animations';

import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    MaterialModule
  ],
  providers: [
    LocoGpsService,
    MainContentService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

In the newly created MainContentService class, create a new method that takes in an array of LocoGpsData and returns a MAIN CONTENT promise

```

...
buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {

}

...

```

Notice we paramatarized our promise, now saying we will not return any type of promis, but a MainContent object. Statically typing our data is a GREAT way to reduce errors, and one of the many improvements of Typescript.

In our buildMainContent method, we are now going to see a different format for creating promises:

```

...
    buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
        return new Promise((resolve, reject) => {

            });
    }
...

```

This has a similar look and feel to the older way we used to develop javascript applications using callbacks. The reject/ resolve params are essentially saying "if on success, call resolve, if on failure, call reject".

For now, we will only worry about the resolve (hmmmm might be interesting to see someone implement error handling in this application....wink wink). All we want to do for now is create a new MainContent object, add our locoGpsData array to it, and return it as a promise:

```

...
    buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
        return new Promise((resolve, reject) => {
            const mainContent = new MainContent();
            mainContent.locoGpsData = locoGpsData;
            resolve(mainContent);
        });
    }
...

```

The "resolve(mainContent)" basically says "take this mainContent object, wrap it in a promise, and return it".

We now have a service that returns us a MainContent object, let's make use of this in our app component. In the app.component.ts, inject the mainContentService in using dependency (the same way we did for the locoGpsService):

```

...
    constructor(
        private locoGpsService: LocoGpsService,
        private mainContentService: MainContentService
    ) {}
...

```

Don't forget to import it!

Now what we want to do is when our locoGpsService returns the locoGpsData array, we want to pass it to our MainContentService service, which will wrap it up nicely for us in a MainContent model, and return for us to use later.

To accomplish this, let's go back to promise chaining!

In onSearchDataset(), instead of printing the result, let's pass it to our mainContentService



```

...
onSearchDataset(userInput: HeaderUserInput) {
  this.locoGpsService.getLocoGpsData(userInput.trainId,
userInput.subdivision)
    .then(results => this.mainContentService.buildMainContent(results));
}
...

```

buildMainContent() returns a Promise<MainContent>....soooooo let's chain to that and save the result

Create a new property in app.component.ts called mainContentData, with the type MainContent:

```

...
onSearchDataset(userInput: HeaderUserInput) {
  this.locoGpsService.getLocoGpsData(userInput.trainId,
userInput.subdivision)
    .then(results => this.mainContentService.buildMainContent(results));
}
...

```

Don't forget to import MainContent!

```

...
mainContentData: MainContent;

...

```

To save the result to main content, we can make use of our cool arrow function we learned earlier

```

...
onSearchDataset(userInput: HeaderUserInput) {
  this.locoGpsService.getLocoGpsData(userInput.trainId,
userInput.subdivision)
    .then(results => this.mainContentService.buildMainContent(results))
    .then(mainContent => this.mainContentData = mainContent);
}
...

```

This basically says grab the Promise<MainContent> returned from buildMainContent(), and save it to mainContentData

Now if we want to see this data on the UI, we can easily add it. We are going to make use of a few Angular specific constructs, [interpolation](#) and [pi ping](#). String interpolation is very similar to what we do in jsf today.

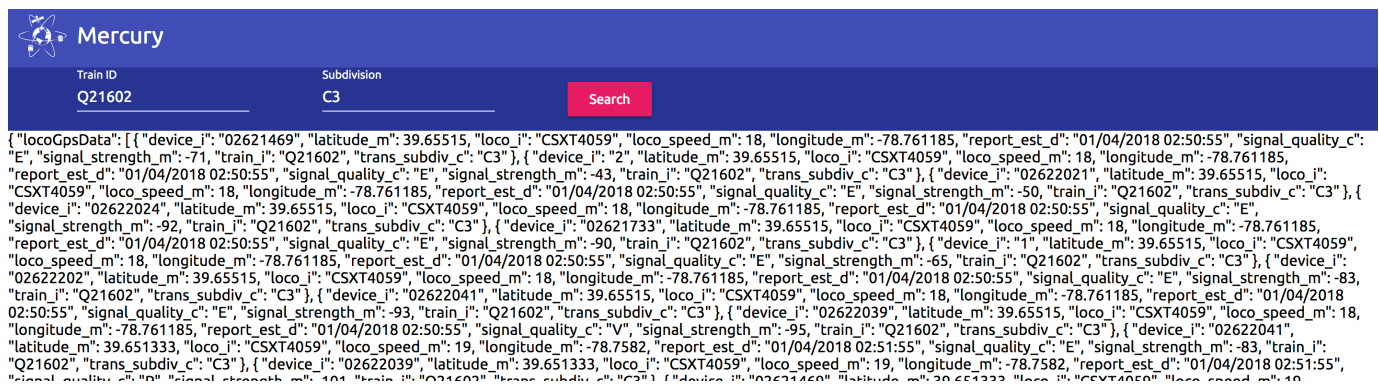
If I have a "value = 'test';" and want to display that on the UI in Angular, I can simply use {{ value }}. (This is verrrrrry similar to jsf el expressions). Piping is a way to easily 'translate' our values in the UI. So I essentially 'push' a value through a pipe, and it comes out the other side in a different format.

To see this, open app.component.ts and add the below interpolated object, and pipe it through and json

```
...
<app-header (searchDataset)="onSearchDataset($event)"></app-header>
{{ mainContentData | json }}
...
```

The json pipe is super useful and used heavily in Angular development, put for now it just helps us see our data on the UI.

You should be seeing a giant data dump on the screen:



And we are now calling a MULTIPLE services and displaying them to the UI!

## Our Second Component - Main Content

To get all the code from above up to this point, you can check out the next tag v1.5 (if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-5 v1.5
```

We have our data displaying on the page, but it's not very uX friendly. Let's build starting building some components to display our data.

We will start with a parent component called "MainContent" that will house all of our display components.

Navigate to /src/app and generate this new component.

```
ng g c main-content
```

Open up main.component.ts. We are going to add a property to this class called mainContentData that will be a MainContent model object. Don't forget to import it!

```
...
mainContentData: MainContent;
...
```

We also want to tell the main component that this data will be coming in as an INPUT. To do this, we will decorate it with the `@Input` decorator.

```
...
@Input()
mainContentData: MainContent;
...
```

This decorator tells Angular that this value will be passed in as an input. To accomplish this, open `app.component.html`. In this component, add our `main-content` component (you can remove the `{{ mainContent | json }}` we added earlier).

```
...
<app-header (searchDataset)="onSearchDataset($event)"></app-header>
<app-main-content></app-main-content>
...
```

If you save, you should see the below:



main-content works!

We can now pass the data FROM our app component TO the main content component by tying `mainContentData` from `app.component.ts` to `mainContentData` from `main-content.component.ts`

```
...
<app-header (searchDataset)="onSearchDataset($event)"></app-header>
<app-main-content [mainContentData]="mainContentData"></app-main-content>
...
```

To verify our main content is receiving the `mainContentData`, we can use the same interpolation/pipe we used earlier in `main-content.component.html`.

```
<p>
  main-content works!
</p>

{{ mainContentData | json }}
```

If you search, you should be seeing the data below "main-content works!".

## Our Third Component - Angular Material Table

To get all the code from above up to this point, you can check out the next tag v1.6 (if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-6 v1.6
```

Our data is being passed to our main component correctly, lets work on displaying it a little better. We will now work on the table portion of the mockup shown at the beginning:

LOCO ID	TRAIN ID	DEVICE ID	LAT	LONG	SPEED	STRENGTH	QUALITY	SUBDIVISION	DATE
CSXT4059	Q21602	02621469	39.65515	-78.761185	18	-71	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	2	39.65515	-78.761185	18	-43	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622021	39.65515	-78.761185	18	-50	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622024	39.65515	-78.761185	18	-92	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02621733	39.65515	-78.761185	18	-90	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	1	39.65515	-78.761185	18	-65	E	C3	01/04/2018 02:50:55

To do this, lets create a component to house our table called "loco-gps-table". We will want this component to live inside of our main-content component, so navigate to /src/app/main-content/ and generate the loco-gps-table component:

```
ng g c loco-gps-table
```

Let's add this component to main-content.component.html (you can delete all the code currently in it):

```
<app-loco-gps-table></app-loco-gps-table>
```

The page should now just show "loco-gps-table" works. The data we need for this table is just the locoGpsData array, so let's pass that to our component. Open loco-gps-table.component.ts and add locoGpsData: LocoGpsData[] as a property and decorate it with the @input decorator:

```
...
@Input()
locoGpsData: LocoGpsData[];
...
```

We also need to pass the locoGpsData array to this component using property binding. In main-content.component.html, add the following:

```
<app-loco-gps-table
[locoGpsData]="mainContentData.locoGpsData"></app-loco-gps-table>
```

To get this to work, we also need to make a small change to app.component.ts. We need to create a new instance of MainContent so that we don't get a NullPointerException (or 'undefined') exception:

```
...
mainContentData = new MainContent();
...
```

Let's now get our data to display correctly in an [Angular Material](#) table. Open loco-gps-table.component.html and add a container that will hold our table:

```
<div class="loco-gps-table-container">

</div>
```

In this container, add the mat-card that our table will be inside:

```
<div class="loco-gps-table-container">
  <mat-card>

  </mat-card>
</div>
```

If you save, you should get an error...womp womp. You have to add the mat-card to the imports/exports of our Material module!

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { MatFormFieldModule, MatInputModule, MatButtonModule,
MatCardModule } from '@angular/material';

@NgModule({
  imports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule,
    MatCardModule
  ],
  exports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule,
    MatCardModule
  ],
  declarations: []
})
export class MaterialModule { }

```

Look up the syntax for an Angular Material table and attempt to add it to your component. The code is below for you to look after you've made an attempt:

▼ [See html...](#)

```

<div class="loco-gps-table-container">
  <mat-card>
    <mat-table [dataSource]="locoGpsData">
      <ng-container matColumnDef="locoId">
        <mat-header-cell *matHeaderCellDef>LOCO ID</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.loco_i}}
      </mat-cell>
    </ng-container>
    <ng-container matColumnDef="trainId">
      <mat-header-cell *matHeaderCellDef>TRAIN ID</mat-header-cell>
      <mat-cell *matCellDef="let element"> {{element.train_i}}
    </mat-cell>
    </ng-container>
    <ng-container matColumnDef="deviceId">
      <mat-header-cell *matHeaderCellDef>DEVICE ID</mat-header-cell>
      <mat-cell *matCellDef="let element"> {{element.device_i}}
    </mat-cell>
    </ng-container>
    <ng-container matColumnDef="lat">

```

```

        <mat-header-cell *matHeaderCellDef>LAT</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.latitude_m}}
</mat-cell>
    </ng-container>
    <ng-container matColumnDef="long">
        <mat-header-cell *matHeaderCellDef>LONG</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.longitude_m}}
</mat-cell>
    </ng-container>
    <ng-container matColumnDef="speed">
        <mat-header-cell *matHeaderCellDef>SPEED</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.loco_speed_m}}
</mat-cell>
    </ng-container>
    <ng-container matColumnDef="signalStrength">
        <mat-header-cell *matHeaderCellDef>STRENGTH</mat-header-cell>
        <mat-cell *matCellDef="let element">
{{element.signal_strength_m}} </mat-cell>
    </ng-container>
    <ng-container matColumnDef="signalQuality">
        <mat-header-cell *matHeaderCellDef>QUALITY</mat-header-cell>
        <mat-cell *matCellDef="let element">
{{element.signal_quality_c}} </mat-cell>
    </ng-container>
    <ng-container matColumnDef="subdivision">
        <mat-header-cell *matHeaderCellDef>SUBDIVISION</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.trans_subdiv_c}}
</mat-cell>
    </ng-container>
    <ng-container matColumnDef="reportDate">
        <mat-header-cell *matHeaderCellDef>DATE</mat-header-cell>
        <mat-cell *matCellDef="let element"> {{element.report_est_d}}
</mat-cell>
    </ng-container>
    <mat-header-row
*matHeaderRowDef="displayedColumns"></mat-header-row>
    <mat-row *matRowDef="let row; columns:
displayedColumns;"></mat-row>
</mat-table>
</mat-card>

```

```
</div>
```

▼ [See typescript...](#)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';


import { MatFormFieldModule, MatInputModule, MatButtonModule,
MatCardModule } from '@angular/material';

@NgModule({
  imports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule,
    MatCardModule
  ],
  exports: [
    CommonModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule,
    MatCardModule
  ],
  declarations: []
})
export class MaterialModule { }
```

Don't forget to add MatTableModule to our Material module!

Your table should look similar to below:




Mercury

Train ID

Q21602

Subdivision

C3

Search

LOCO ID	TRAIN ID	DEVICE ID	LAT	LONG	SPEED	STRENGTH	QUALITY	SUBDIVISION	DATE
CSXT4059	Q21602	02621469	39.65515	-78.761185	18	-71	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	2	39.65515	-78.761185	18	-43	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622021	39.65515	-78.761185	18	-50	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622024	39.65515	-78.761185	18	-92	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02621733	39.65515	-78.761185	18	-90	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	1	39.65515	-78.761185	18	-65	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622023	39.65515	-78.761185	18	-93	E	C3	01/04/2018 02:50:55

Now let's add some small style changes to make our table really look like the mockup. In `loco-gps-table.component.scss`, import our `variables.scss` and so we can use our app colors. We will change the top border and title to `$pink`.

```
@import "~variables.scss";

.mat-card {
  border-top: 2px solid $pink;
}

.title {
  color: $pink;
}
```

Open `main-content.component.html`. We will wrap our components in the `"inner-wrapper"` class we created earlier so that they stay centered on the page:

```
<div class="main-content-container inner-wrapper">
  <app-loco-gps-table
    [locoGpsData]="mainContentData.locoGpsData"></app-loco-gps-table>
</div>
```

Our page should be starting to look pretty good!

LOCO ID	TRAIN ID	DEVICE ID	LAT	LONG	SPEED	STRENGTH	QUALITY	SUBDIVISION	DATE
CSXT4059	Q21602	02621469	39.65515	-78.761185	18	-71	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	2	39.65515	-78.761185	18	-43	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622021	39.65515	-78.761185	18	-50	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02622024	39.65515	-78.761185	18	-92	E	C3	01/04/2018 02:50:55
CSXT4059	Q21602	02621733	39.65515	-78.761185	18	-90	E	C3	01/04/2018 02:50:55

## Our Fourth and Fifth Components - Large Icon Cards

To get all the code from above up to this point, you can check out the next tag v1.7 (if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-7 v1.7
```

We are now going to work on the large icon cards from the mock up



TRAINS  
1



LOCOS  
1



SUBDIVISIONS  
1



DEVICE  
14

We will start by generating TWO new components - large-icon-cards AND large-icon-card. large-icon-cards will be our parent container and large-icon-card will be our actual card. This will begin to show us how we can really start to modularize our Angular application.

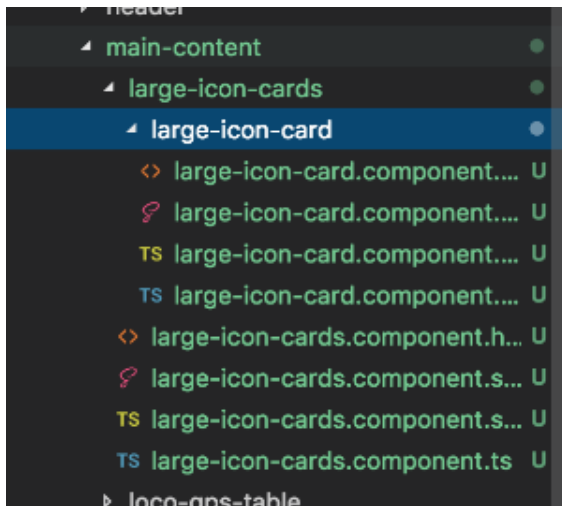
Start by generating large-icon-cards (make sure you are in /src/app/main-content)

```
ng g c large-icon-cards
```

Navigate into this newly created folder (/src/app/main-content/large-icon-cards/) and generate large-icon-card

```
ng g c large-icon-card
```

This will give you the below folder structure:



We also need to create a few new model objects for these components. In the "shared" directory, create a new file called `largeIconCard.model.ts`. This will hold the info for our large icon card and will look like:

```
export class LargeIconCard {
  public title: string;
  public value = 0;
  public uniqueValues: string[];
  public cardStyle;
  public cardImage: string;
}
```

We are also going to create a class that will hold some styles for us called `largeIconCardStyles.model.ts` which will hold an array of styles we can use in our large icon cards:

```
export class LargeIconCardStyles {
  public stylesList = [
    {
      'background': 'linear-gradient(#3F51B5, #fff)'
    },
    {
      'background': 'linear-gradient(#8B26AF, #fff)'
    },
    {
      'background': 'linear-gradient(#455A64, #fff)'
    },
    {
      'background': 'linear-gradient(#E91E63, #fff)'
    }
  ];
}
```

We also need to modify mainContent.model.ts to hold our large-card-icons (this will be an array):

```
import { LargeIconCard } from './largeIconCard.model';
import { LocoGpsData } from './locoGpsData.model';
export class MainContent {
  public locoGpsData: LocoGpsData[];
  public largeIconCardList: LargeIconCard[] = new Array();
  constructor() {}
}
```

This class would generally be a const or a properties file but we'll keep it as a model to simplify our demo a little bit. We also need a class to hold our icon values, called locoGpsIcons.model.ts:

```
export class LocoGpsIcons {
  public iconList = [
    'assets/images/train.svg',
    'assets/images/power.svg',
    'assets/images/flag.svg',
    'assets/images/device.svg'
  ];
}
```

We now have everything we need to build a large-icon-card. Let's build our model object in the service we created earlier, main-content.service.ts. For our large icon cards, we want the number of trains, number of locomotives, number of subdivisions, and the number of devices. There are a lot of libraries we can use to count unique items in a json array, but let's write our own Javascript that does this for us:

In buildMainContent, add the following:

```

buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
  return new Promise((resolve, reject) => {
    const mainContent = new MainContent();
    mainContent.locoGpsData = locoGpsData;

    const uniqueTrains = [];
    const uniqueLocos = [];
    const uniqueSubs = [];
    const uniqueDevice = [];

    for (let i = 0; i < locoGpsData.length; i++) {

      if (uniqueTrains.indexOf(locoGpsData[i].train_i) === -1) {
        uniqueTrains.push(locoGpsData[i].train_i);
      }
      if (uniqueLocos.indexOf(locoGpsData[i].loco_i) === -1) {
        uniqueLocos.push(locoGpsData[i].loco_i);
      }
      if (uniqueSubs.indexOf(locoGpsData[i].trans_subdiv_c) === -1) {
        uniqueSubs.push(locoGpsData[i].trans_subdiv_c);
      }
      if (uniqueDevice.indexOf(locoGpsData[i].device_i) === -1) {
        uniqueDevice.push(locoGpsData[i].device_i);
      }
    }

    resolve(mainContent);
  });
}

```

We will loop through the locoGpsData json array, and add any unique items to an array. With that data, we can build our large icon cards. Create a new method called buildLargeIconCard that takes in a title: string, uniquevalues: string[], cardStyle: any, and cardIcon: any. Here we will populate a card and return it:

```

buildLargeCard(title: string, uniqueValues: string[], cardStyle: any,
cardIcon: string): LargeIconCard {
  const largeIconCard = new LargeIconCard();
  largeIconCard.title = title;
  largeIconCard.value = uniqueValues.length;
  largeIconCard.uniqueValues = uniqueValues;
  largeIconCard.cardStyle = cardStyle;
  largeIconCard.cardImage = cardIcon;
  return largeIconCard;
}

```

We can grab references to our cardStyles and cardIcons model objects, and then populate a large-icon-card using buildLargeCard, and then add that to mainContent.largeIconList by doing the following:

```

const cardStyles = new LargeIconCardStyles();
const cardIcons = new LocoGpsIcons();

    mainContent.largeIconCardList.push(this.buildLargeCard('Trains',
uniqueTrains,
    cardStyles.stylesList[0], cardIcons.iconList[0]));
    mainContent.largeIconCardList.push(this.buildLargeCard('Locos',
uniqueLocos,
    cardStyles.stylesList[1], cardIcons.iconList[1]));

mainContent.largeIconCardList.push(this.buildLargeCard('Subdivisions',
uniqueSubs,
    cardStyles.stylesList[2], cardIcons.iconList[2]));
    mainContent.largeIconCardList.push(this.buildLargeCard('Device',
uniqueDevice,
    cardStyles.stylesList[3], cardIcons.iconList[3]));

```

(so the above code will be in buildMainContent)...

We now have the data for our large-icon-cards, let's add them to the UI. Open large-icon-cards.component.ts and add an input that is our largeIconCard array:

```

@Input()
largeIconCardList: LargeIconCard[];

```

Open large-icon-card.component.ts and add in the properties we need for the card (we could just pass in the entire card object, but you can see here that we can pass in MULTIPLE properties to a component as well):

```

@Input()
styles: {};

@Input()
title: string;

@Input()
value: number;

@Input()
icon: string;

```

We now need to pass the data alllllll the way down to our large-icon-card. Lets start by passing the list to the largeIconCardList property of large-icon-cards (so the below code goes in...main-content.component.html

```

<div class="main-content-container inner-wrapper">
  <app-large-icon-cards
    [largeIconCardList]="mainContentData.largeIconCardList"></app-large-icon-cards>
  <app-loco-gps-table
    [locoGpsData]="mainContentData.locoGpsData"></app-loco-gps-table>
</div>

```

We then need to pass the data from our large-icon-cards to our large-icon-card. To do this, we can pass each property individually (so this code goes in large-icon-cards.component.html):

```

<div class="icon-cards-container">
  <div class="icon-card" *ngFor="let iconCard of largeIconCardList">
    <app-large-icon-card [title]="iconCard.title" [value]="iconCard.value"
      [styles]="iconCard.cardStyle"
    [icon]="iconCard.cardImage"></app-large-icon-card>
  </div>
</div>

```

Notice we are using a new Angular directive here, the **"\*ngFor"**. This is essentially a for each loop that says, "for each iconCard in largeIconCardList, repeat the following code" (this is really similar to how a datalist in primefaces works). Notice we are setting each property individually. We also wrapped it in a container that we can use to style our components later.

We lastly need to display the data for the card in large-icon-card.component.html:

```

<mat-card>
  <div class="icon-container" [ngStyle]="styles">
    <img [src]="icon">
  </div>
  <div class="content">
    <div class="title">
      {{ title }}
    </div>
    <div class="value">
      {{ value }}
    </div>
  </div>
</mat-card>

```

Notice we are using interpolation (the "{{ }}" ). This provides us with 1-way data binding (from controller to view) and is really useful for displaying data. If you run the above code, it will work but probably look CRAZY. Let's add some styles. Open large-card-icon.scss and add try to style the component so it looks like the mockup. The code I wrote to do this is below:

▼ [See CSS...](#)

```
.mat-card {
  padding: 0;
  display: flex;
  height: 94px;
}

.icon-container {
  width: 22%;
  display: flex;
  justify-content: center;
  align-items: center;
}

.icon-container img {
  width: 50px;
  height: 50px;
}

.content {
  width: 78%;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}

.title {
  font-size: 20px;
  text-transform: uppercase;
}

.value {
  font-size: 36px;
}
```

Getting closer! But we still need to style the large-icon-cards.component.scss.

▼ [See CSS...](#)



```

.icon-cards-container {
  display: flex;
  margin: 28px 0;
}


.icon-card {
  flex-grow: 1;
  flex-basis: 0;
  margin: 0 12px;
}

.icon-cards-container .icon-card:first-child{
  margin-left: 0;
}

.icon-cards-container .icon-card:last-child{
  margin-right: 0;
}


```

You should see something similar to below. Starting to look really good!



**Mercury**

Train ID


Subdivision




**TRAINS**  
**1**



**LOCOS**  
**1**



**SUBDIVISIONS**  
**1**



**DEVICE**  
**14**

LOCO ID	TRAIN ID	DEVICE ID	LAT	LONG	SPEED	STRENGTH	QUALITY	SUBDIVISION	DATE
CSXT4059	Q21602	02621469	39.65515	-78.761185	18	-71	E	C3	01/04/2018 02:50:55

## Last Component - the "Graph" cards

To get all the code from above up to this point, you can check out the next tag v1.8 (if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-8 v1.8
```

The last thing we need to add is the graph-cards component. For this, we will need 2 new model objects `averageSpeedCard.model.ts` and

signalStrengthCard.model.ts. Both of these will live in the "shared" folder and are shown below:

averageSpeedCard.model.ts:

```
export class AverageSpeedCard {  
  public averageSpeed: number;  
  public speedList: number[];  
}
```

signalStrengthCard.model.ts:

```
export class SignalStrengthCard {  
  public low: number;  
  public high: number;  
  public strenghtList: number[];  
}
```

You'll notice we store some property values to display on the card, as well as a list of the values.

Let's populate these models in our main-content.service.ts. For the average speed component, we need to calculate the average speed (duh!). This can be accomplished by adding up all the speeds and dividing by the total number of records (I removed some of our previous code so you can see what was added easier):

```
buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {  
  return new Promise((resolve, reject) => {  
  
    ...  
  
    // for avg speed card  
    let avgSpeed = 0;  
    const speedList = [];  
  
    for (let i = 0; i < locoGpsData.length; i++) {  
  
      ...  
  
      // for average speed  
      avgSpeed += locoGpsData[i].loco_speed_m;  
      speedList.push(locoGpsData[i].loco_speed_m);  
  
    }  
  
    ...  
  
    resolve(mainContent);  
  });  
}
```

Let's also add a new method that will create our averageSpeedCard for us:

```
buildAvgSpeedCard(avgSpeed: number, speedList: number[],
numAvgSpeedValues): AverageSpeedCard {
    avgSpeed /= speedList.length;
    const avgSpeedCard = new AverageSpeedCard();
    avgSpeedCard.averageSpeed = Math.round(avgSpeed);
    avgSpeedCard.speedList = speedList;
    return avgSpeedCard;
}
```

We also need to add the averageSpeedCard model and signalStrengthCard model as properties of our MainContentModel. Open mainContent.model.ts and add them:

```
import { SignalStrengthCard } from './signalStrengthCard.model';
import { AverageSpeedCard } from './averageSpeedCard.model';
import { LocoGpsData } from './locoGpsData.model';
import { LargeIconCard } from './largeIconCard.model';
export class MainContent {
    public avgSpeedCard = new AverageSpeedCard();
    public signalStrengthCard = new SignalStrengthCard();
    public locoGpsData: LocoGpsData[];
    public largeIconCardList: LargeIconCard[] = new Array();
    constructor() {}
}
```

Call the method to create the card:

```

buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
  return new Promise((resolve, reject) => {

    ...

    // for avg speed card
    let avgSpeed = 0;
    const speedList = [];

    for (let i = 0; i < locoGpsData.length; i++) {

      ...

      // for average speed
      avgSpeed += locoGpsData[i].loco_speed_m;
      speedList.push(locoGpsData[i].loco_speed_m);

    }

    // build average speed card
    mainContent.avgSpeedCard = this.buildAvgSpeedCard(avgSpeed, speedList,
    locoGpsData.length);

    ...

    resolve(mainContent);
  });
}

```

Now we can now do the same thing for signalStrengthCard model

```

buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
    return new Promise((resolve, reject) => {

        ...

        // for avg speed card
        let avgSpeed = 0;
        const speedList = [];

        // for signal strength
        let lowSignalStrength = Number.MAX_SAFE_INTEGER;
        let highSignalStrength = Number.MIN_SAFE_INTEGER;
        const signalStrengthList = [];
        for (let i = 0; i < locoGpsData.length; i++) {

            ...

            // for average speed
            avgSpeed += locoGpsData[i].loco_speed_m;
            speedList.push(locoGpsData[i].loco_speed_m);

            // for signal strength
            if (locoGpsData[i].signal_strength_m > highSignalStrength) {
                highSignalStrength = locoGpsData[i].signal_strength_m;
            } else if (locoGpsData[i].signal_strength_m < lowSignalStrength) {
                lowSignalStrength = locoGpsData[i].signal_strength_m;
            }

            signalStrengthList.push(locoGpsData[i].signal_strength_m);

        }

        // build average speed card
        mainContent.avgSpeedCard = this.buildAvgSpeedCard(avgSpeed, speedList,
        locoGpsData.length);

        ...

        resolve(mainContent);
    });
}

```

Let's add a new method called buildSignalStrengthCard that will build the model for us:

```
buildSignalStrengthCard(lowStrength: number, highStrength: number,  
signalStrengthList: number[]): SignalStrengthCard {  
  const signalStrengthCard = new SignalStrengthCard();  
  signalStrengthCard.low = lowStrength;  
  signalStrengthCard.high = highStrength;  
  signalStrengthCard.strenghtList = signalStrengthList;  
  return signalStrengthCard;  
}
```

Lastly call our newly created method and populate

```

buildMainContent(locoGpsData: LocoGpsData[]): Promise<MainContent> {
    return new Promise((resolve, reject) => {

        ...

        // for avg speed card
        let avgSpeed = 0;
        const speedList = [];

        // for signal strength
        let lowSignalStrength = Number.MAX_SAFE_INTEGER;
        let highSignalStrength = Number.MIN_SAFE_INTEGER;
        const signalStrengthList = [];
        for (let i = 0; i < locoGpsData.length; i++) {

            ...

            // for average speed
            avgSpeed += locoGpsData[i].loco_speed_m;
            speedList.push(locoGpsData[i].loco_speed_m);

            // for signal strength
            if (locoGpsData[i].signal_strength_m > highSignalStrength) {
                highSignalStrength = locoGpsData[i].signal_strength_m;
            } else if (locoGpsData[i].signal_strength_m < lowSignalStrength) {
                lowSignalStrength = locoGpsData[i].signal_strength_m;
            }

            signalStrengthList.push(locoGpsData[i].signal_strength_m);

        }

        // build average speed card
        mainContent.avgSpeedCard = this.buildAvgSpeedCard(avgSpeed, speedList,
locoGpsData.length);

        // signal strength card
        mainContent.signalStrengthCard =
this.buildSignalStrengthCard(lowSignalStrength,
            highSignalStrength, signalStrengthList);

        ...

        resolve(mainContent);
    });
}

```

Both of our cards should now be populated correctly, lets use them on the UI.

To do that, we need to generate a new component using the angular-cli. Make sure you are in /src/app/main-content and generate graph-cards:

```
ng g c graph-cards
```

Open graph-cards.component.ts and add our newly created models as properties of this component:

```
import { SignalStrengthCard } from
'../../shared/signalStrengthCard.model';
import { AverageSpeedCard } from '../../shared/averageSpeedCard.model';
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-graph-cards',
  templateUrl: './graph-cards.component.html',
  styleUrls: ['./graph-cards.component.scss']
})
export class GraphCardsComponent implements OnInit {

  @Input()
  avgSpeedCard: AverageSpeedCard;

  @Input()
  signalStrengthCard: SignalStrengthCard;

  constructor() { }

  ngOnInit() {
  }

}
```

We can now pass in our data as inputs, just like we've done in previous sections. Open main-content.component.html, add the graph-cards component, and pass in the values:

```
<div class="main-content-container inner-wrapper">
  <app-graph-cards [avgSpeedCard]="mainContentData.avgSpeedCard"
[signalStrengthCard]="mainContentData.signalStrengthCard"></app-graph-card
s>
  <app-large-icon-cards
[largeIconCardList]="mainContentData.largeIconCardList"></app-large-icon-c
ards>
  <app-loco-gps-table
[locoGpsData]="mainContentData.locoGpsData"></app-loco-gps-table>
</div>
```

We can now display our data in our card, shown below:



```

<div class="graph-cards-container">
  <mat-card class="avg-speed-card">
    <div class="title">Speed</div>
    <div class="value">{{ avgSpeedCard.averageSpeed }}</div>
  </mat-card>
  <mat-card class="signal-strength-card">
    <div class="title">Signal Strength</div>
    <div class="signal-value">
      <div class="low">{{ signalStrengthCard.low }} LOW</div>
      <div class="high">{{ signalStrengthCard.high }} HIGH</div>
    </div>
  </mat-card>
</div>

```

The last thing we need to do is style our newly created cards:

```

.icon-cards-container {
  display: flex;
  margin: 28px 0;
}

.icon-card {
  flex-grow: 1;
  flex-basis: 0;
  margin: 0 12px;
}

.icon-cards-container .icon-card:first-child{
  margin-left: 0;
}

.icon-cards-container .icon-card:last-child{
  margin-right: 0;
}

```



## Mercury

Train ID  
Q21602

Subdivision  
C3

Search



TRAINS  
1



LOCOS  
1



SUBDIVISIONS  
1



DEVICE  
14

LOCO ID	TRAIN ID	DEVICE ID	LAT	LONG	SPEED	STRENGTH	QUALITY	SUBDIVISION	DATE
CSXT4059	Q21602	02621469	39.65515	-78.761185	18	-71	E	C3	01/04/2018 02:50:55

AND WERE DONE!!!! WOO HOO CONGRATS!

(The final working code tag is below, if you jumped ahead, you must follow steps 3, 4 and 5 [here](#)):

```
git checkout -b my-branch-v1-9 v1.9
```