Table of Contents

Introduction **4**

Section 1: Getting Started **5**

Getting Started With Vue.js **6**

Step #1: Install Node.js & NPM 6

Step #2: Install Vue Cli 8

Step #3: Create A Project 8

Obtain Google Maps API Key **13**

Step #1: Create an API Key 13

Step #2: Setup Billing 16

Step #3: Protect Your API Key 17

Step #4: Enable Libraries 20

Section 2: Detect User's Current Location **21**

What You'll Be Building By The End Of This Section **22**

Create a Vue Component and Route **23**

Step #1: Component vs Page Based 24

Step #2: Vue Route 26

Build User Location Form Using Semantic-UI **29**

Step #1: Include Semantic UI CSS Framework 29

Step #2: Design Form Layout 30

Step #3: Error Message 33

Step #4: Input Field 34

Step #5: Go Button 36

HTML5 Geolocation API **38**

Step #1: What is HTML5 Geolocation API 38

Step #2: Attach A Click Event 39

Step #3: Get User Location Using getCurrentPosition() 41

Step #4: Show Error Message 43

Make An HTTP Request To Google's Geocoding API **45**

Step #1: Google's Geocoding API 45

Step #2: HTTP Request Using Axios 47

Step #3: Show Address On The Input Field 51

Step #4: Error Checking 53

Handling Client and Server Errors **55**
Step #1: Set the Error Message 56
Step #2: Add a Spinner / Loader 58

Enable Autocomplete API **63**
Step #1: What is Autocomplete? 63
Step #2: Instantiate Autocomplete 64
Step #3: Restrict Suggested Addresses to a Specific Region 66
Step #4: Change Drop-down List Style 69

Show User's Current Location On The Google Map **73**
Step #1: Show the User's location Using Locator Button 74
Step #2: Show the Map Element Full Width 75
Step #3: Show Google Maps On The View 77
Step #4: Place Marker On The Map Using Locator 79
Step #5: Place Marker On The Map Using Autocomplete 80

Section 3: Build a Closeby App Using Places API **83**

What You'll Be Building By The End Of This Section **84**

What is Nearby Search Library? **86**
Step #1: What is Nearby Search Request? 86
Step #2: Required Request Parameters 86
Step #3: Output Response Object 88

Create A CloseBuy Component **89**
Step #1: Create a CloseBuy Component and Route 89
Step #2: Add Grid Layout 91
Step #3: Build User Input Form 93

Make An HTTP Request To Nearby Search **106**
Step #1: Compose Nearby Search Request URL 106
Step #2: Why CORS Error Occur? 109
Step #3: What is CORS ? 110
Step #4: Fix CORS Error 112

Get Nearby Places Data When Using Autocomplete **114**
Step #1: Define a variable called autocomplete and assign an autocomplete object to it. 114
Step #2: Attach a place_changed event to the autocomplete object. 114
Step #3: Invoke getPlace() 115
Step #4: Reset the Address 115

Display Places Data on the View **117**

Show Places Data on the Maps View **123**
Step #1: Instantiate Google Maps map object. 123
Step #2: Add Markers 127

Show Tooltip (Callout) on the Marker **130**
Step #1: Attach Click Event To Markers 130
Step #2: Instantiate the Info Window 131

Show More Information About A Place **135**
Step #1: Place Details Request 135
Step #2: Make A HTTP Request To Place Details 137

Auto Select Markers When List Item is Clicked **142**

About the Author **147**

Introduction

Welcome to the Vue.js & Google Maps API for Beginners course. You'll be learning how to build a few location based apps from start to finish with easy to follow STEP by STEP instructions.

This course is designed for anyone who has some experience with HTML and JavaScript seeking to design and build location based apps.

By the end of this course, you will be fluent with coding in **Vue.js**, the popular front end framework and utilizing the **Google Maps Platform** to build your own location-based apps.

The ideal student for this course is an entry level Javascript Developer who wants to expand their current skills or VueJs developers curious to learn how the Google Maps Platform works.

****

Section 1: Getting Started

****

Getting Started With Vue.js

Vue.js is a front-end framework and is component based, similar to Angular and React, but it is very popular amongst developers because it is very simple to use.

Let's see how to install and get started with Vue.js in three easy steps:

1. Install Node.js & NPM
2. Install Vue Cli
3. Create a Project

Step #1: Install Node.js & NPM

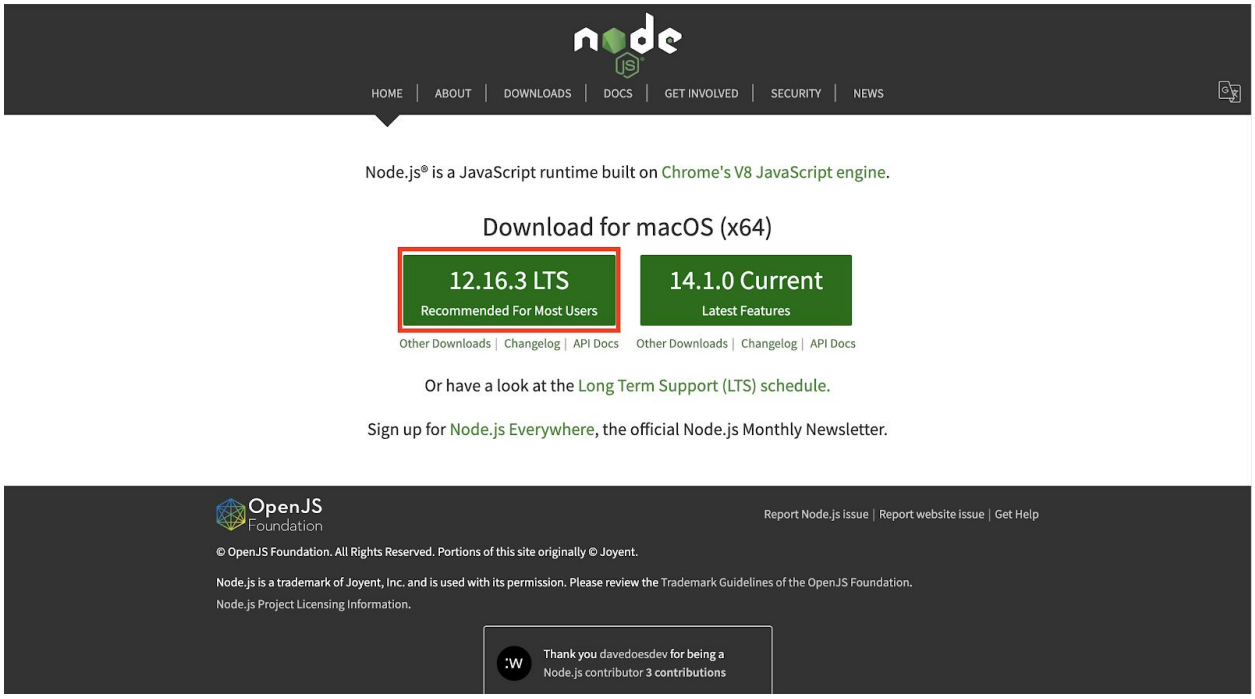
To get started, we need to install Node.js & NPM (Node Package Manager) to your computer. You can check to see if they are already installed by going to the Terminal window or command prompt and running the following commands:

```
node -v
```

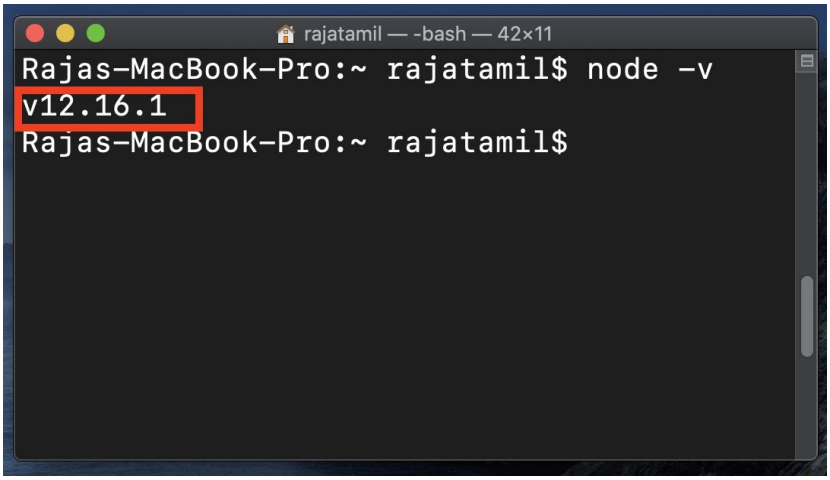
and

```
npm -v
```

If they're not installed, head over to https://nodejs.org/en/ and choose the LTS version to start downloading and installing it on your computer.



At the time of writing this book, I have installed node **v12.16.1** and npm version **6.13.4**. You may have a higher version by the time you're reading this book.



Step #2: Install Vue Cli

There are multiple ways to install a Vue project on your computer. Let's see how to do it using Vue Cli.

Vue CLI is a NPM package that allows us to create a Vue project quickly. It will ask a series of questions about how to configure our project such as do you need a router included, how you want to build the Vue App, and so on.

Let's install it **globally**, so that we can create a Vue project anywhere on our computer.

Open up the terminal window and run:

```
npm install vue-cli -g
```

The **-g** is added at the end to install it globally.

It will take a few seconds to complete the installation process.

Step #3: Create A Project

Once it's done, the next step is to create a new Vue project. To create it on your desktop, go to the desktop and run the following command:

```
cd ~/Desktop
```

Then, run the webpack command:

```
vue init webpack
```

At this stage you may be asked to install the **cli-init package**. If so, go ahead and install it by running the following command:

```
npm install -g @vue/cli-init
```

Once it is done, run the **vue init webpack** command again which will start with asking a series of questions.

- **Project name** (yourprojectname) – The name must be URL friendly (no space)
- **Project description** (A Vue.js project)
- **Author** (SoftAuthor)
- **Vue build** (Use arrow keys) **› Runtime + Compiler: recommended for most users**
- **Install vue-router?** (Y/N) **›** Y (We are using vue-router in this course)
- **Use ESLint to lint your code?** (Y/N) **›** N
- **Set up unit tests** (Y/N) **›** N
- **Setup e2e tests with Nightwatch?** (Y/N) **›** N

Once the questions are answered, it will then ask you how you want to install the dependencies; choose **npm install**.

It will take a few seconds to complete the installation process and it will show you two commands when it is done:

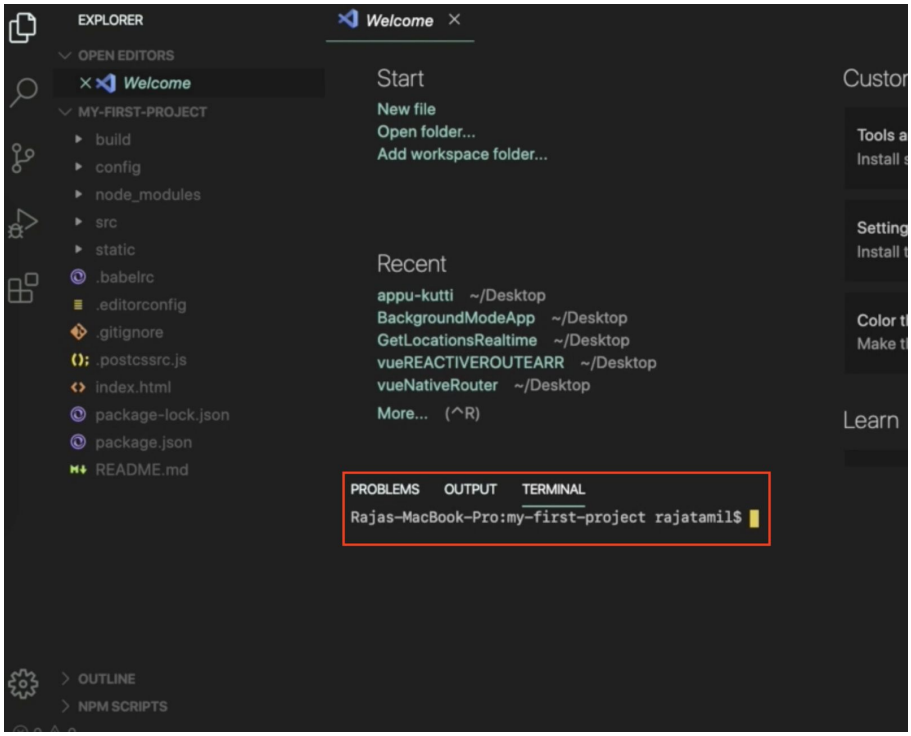
```
cd yourprojectname
npm run dev
```

The first command is to go into the project directory. The second command is to run the Vue project which will start the server and run the Vue project on it.

Rather than running it from the command line, I am going to use Visual Studio Code and it has an inbuilt Terminal.

To open up the project on the Visual Studio Code editor, drag the project folder and drop it into the Visual Studio Code Icon in the dock or go to File -> Open -> choose your project.

Then, go to the Terminal menu from the Visual Studio menu bar and choose **New Window**.

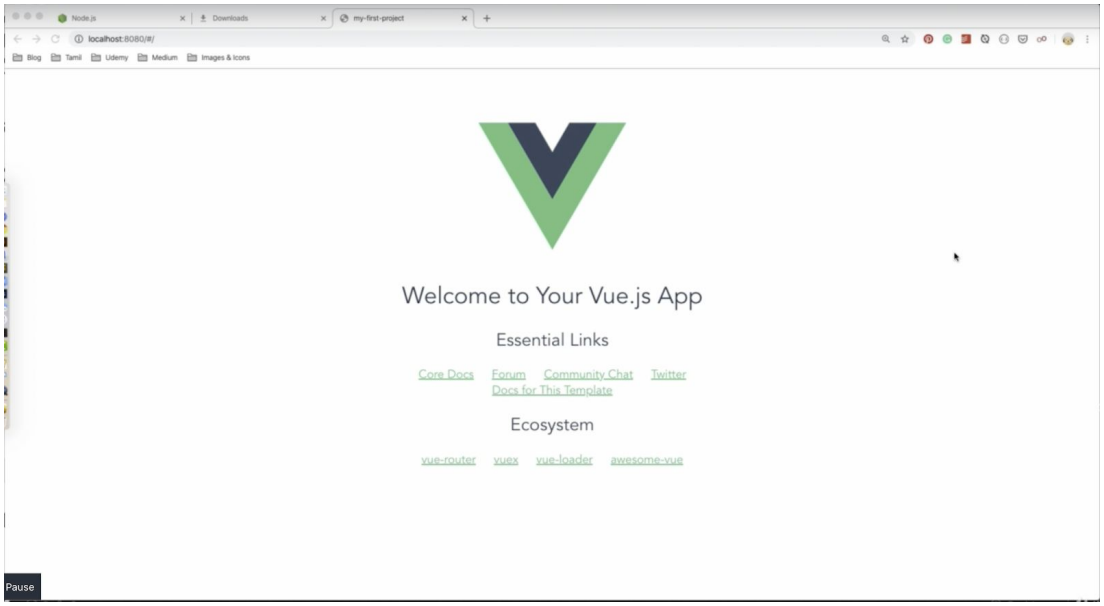


As you can see, it's already inside the project. Run the app using the following command:

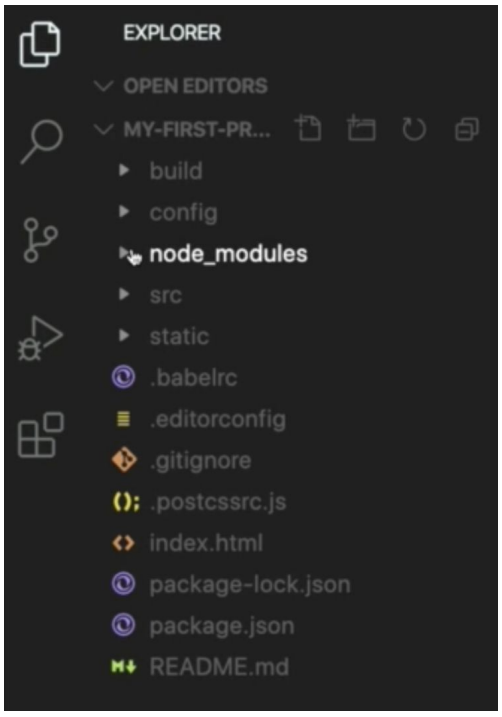
```
rpm run dev
```

This will start the server and give you the localhost URL. Copy the URL and paste it on the browser.

Excellent! Now we have our first Vue App up and running.



Let’s explore the project folder structure.



- **node_modules**: This folder contains the installed npm packages.
- **src**: This folder contains the source files for your project. Most of the work will be done here.

- **src/assets**: This folder contains the project's assets such as logo.png and any other file that will be imported into the components.
- **src/components**: This folder contains the Vue components that are not the main views or pages. When we repeat a set of code in multiple places, it would be a separate component created inside this components folder.
- **src/router**: This is where you create a route and attach it to page based components.

We'll learn more details about the file structure and how to organize it later in this course.

Obtain Google Maps API Key

In order to use any of the Google Maps API services, we need to obtain an API Key from the Google Cloud Console Website.

By the end of this chapter, you'll be able to:

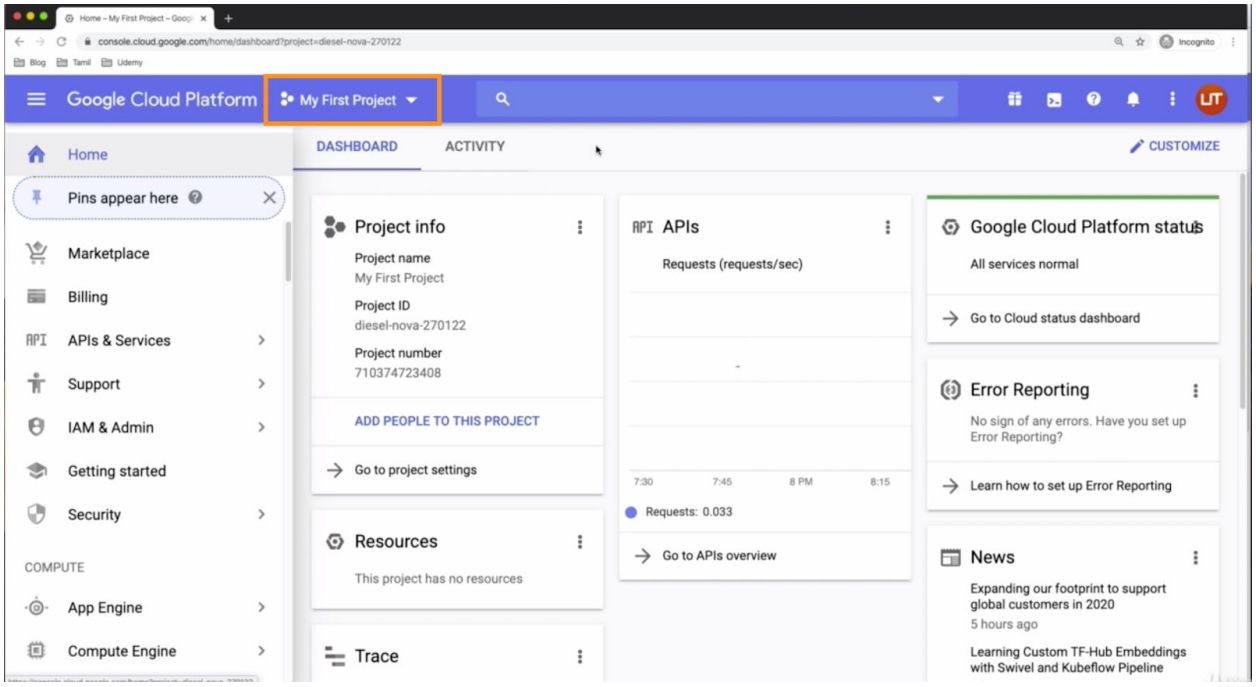
1. Create an API Key.
2. Enable Billing (APIs do not work without it).
3. Protect the API key (as it can be exposed to users).
4. Enable different Google Maps Libraries.

Step #1: Create an API Key

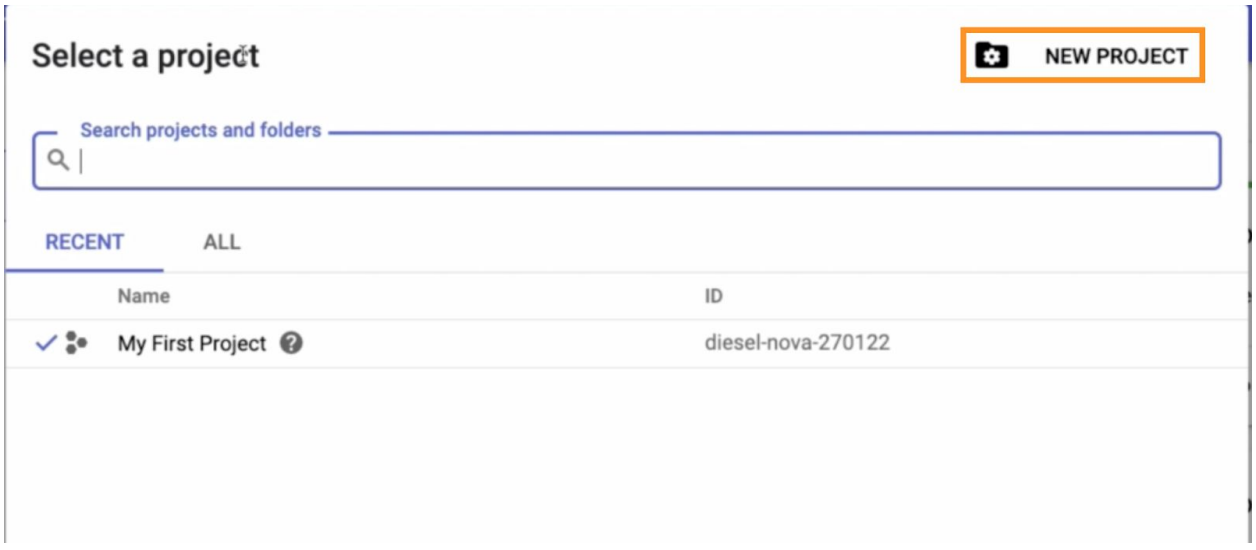
Open the Google Cloud Console Website on the browser.

Then, it will ask you to login with your Gmail Account. If you are already logged in, it will take you straight to the Google Cloud Platform Dashboard.

In order to create an API Key we need to create a project. To create a new project, go to the top and choose **Select a Project** in the drop down menu.



It will open up a project window where you can see any existing projects. At the top right, create a new project by clicking the **New Project** button.

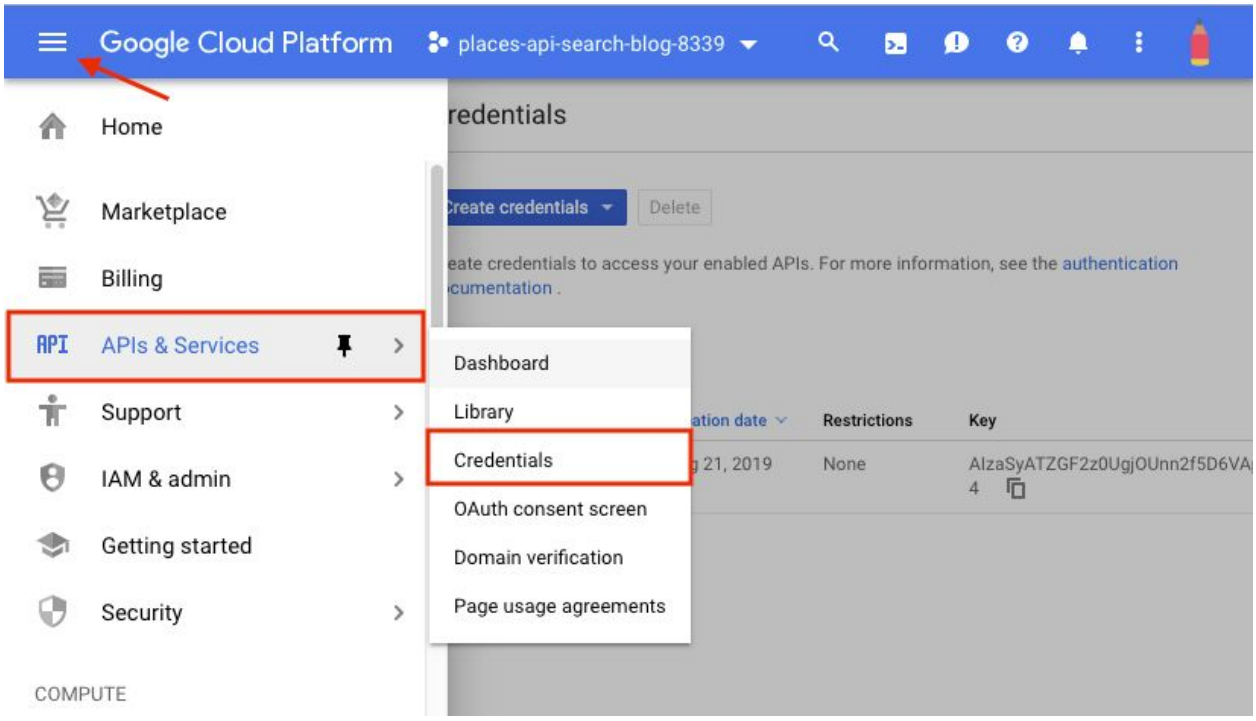


Then, give it a project name, and have the location be **No Organization** for now. Hit **Create**. It will take a few seconds for the project to be set up.

Once it's done, you can go to that project by clicking the link from the notification icon at the top right or you can go to the **Select Project** drop down menu at the top and choose the project that you want to create an API Key for.

Once the project is selected, it will take you to the project dashboard.

Go to the top left bar button, then choose the **Api & Services** option from the sidebar and choose **Credentials**. Click **Create Credentials** and choose **API Key** and then **Done**.

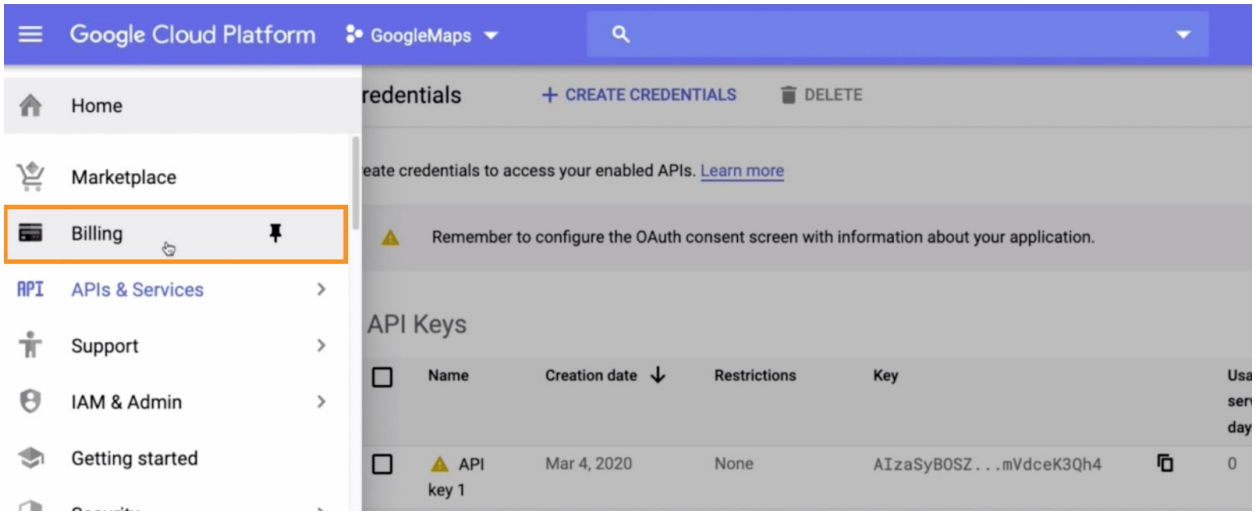


This is the API Key that you will need throughout this course.

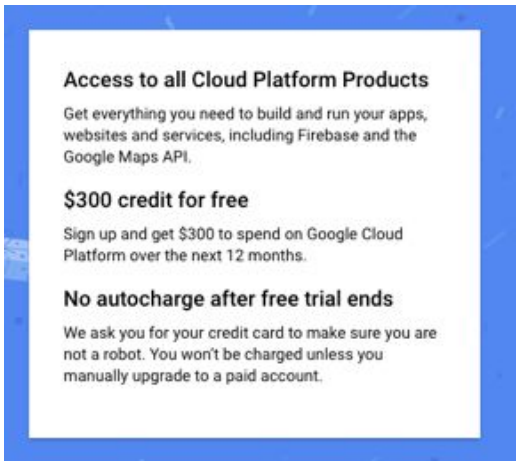
Step #2: Setup Billing

Now that we have an API Key, we need to enable billing otherwise our requests to the Google Maps API services won't work.

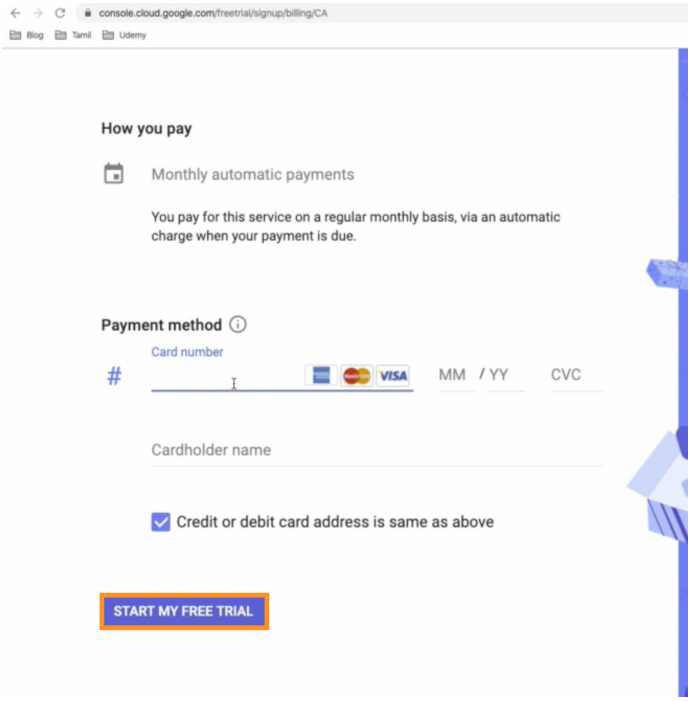
Go to the sidebar and choose **Manage Billing Account** and then choose **Add Billing Account**.



Select your country and then select the **Terms and Conditions**. At the time of writing, google gives a $300 credit for free, which will be more than enough to complete all the projects in this course without paying for the service.

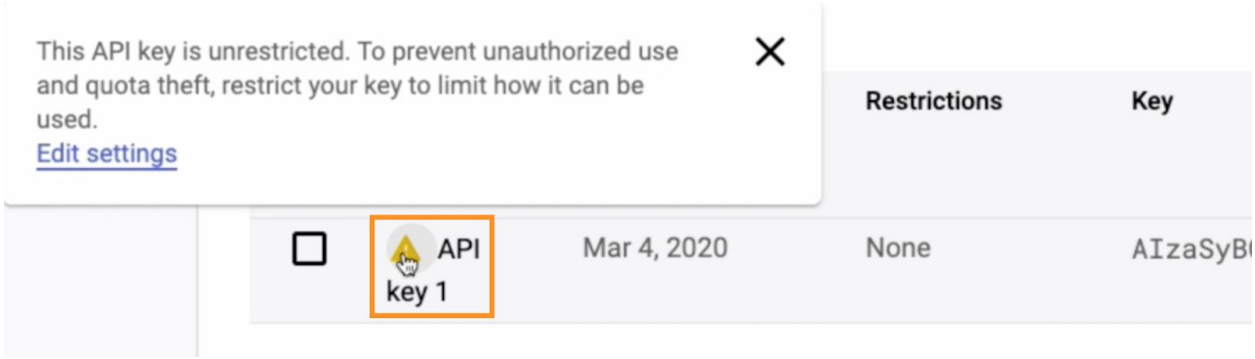


Click **Next**, then choose the **Account Type**, **Address**, **Add** then enter your credit card information and choose **Start My Free Trial**.



Step #3: Protect Your API Key

The warning icon before the API Key name indicating that the Key is NOT safe.



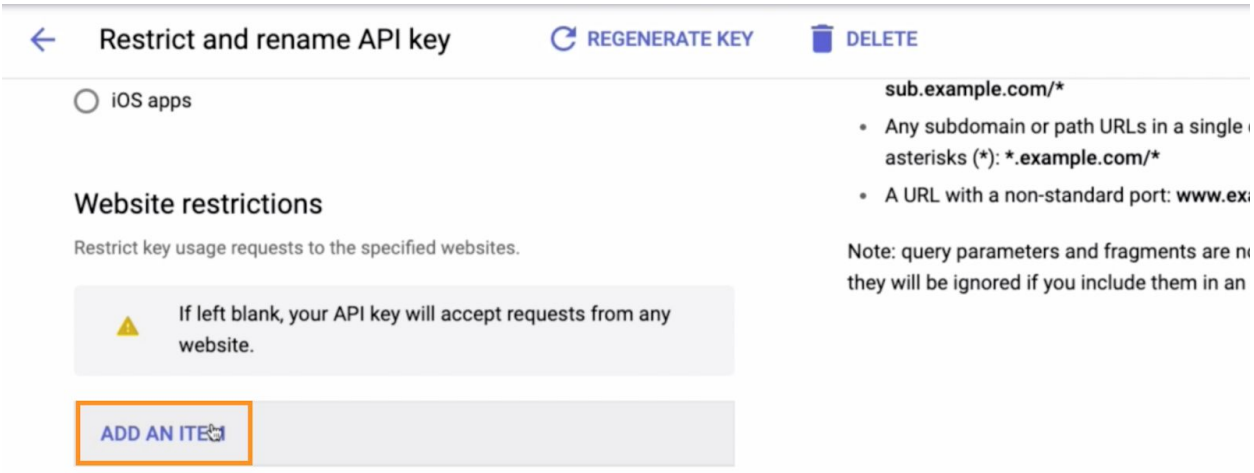
The billing is based on the number of HTTP requests that we are making to the Google Maps API.

So, we need to protect the API Key so no one can make HTTP requests using our account in other words quota theft.

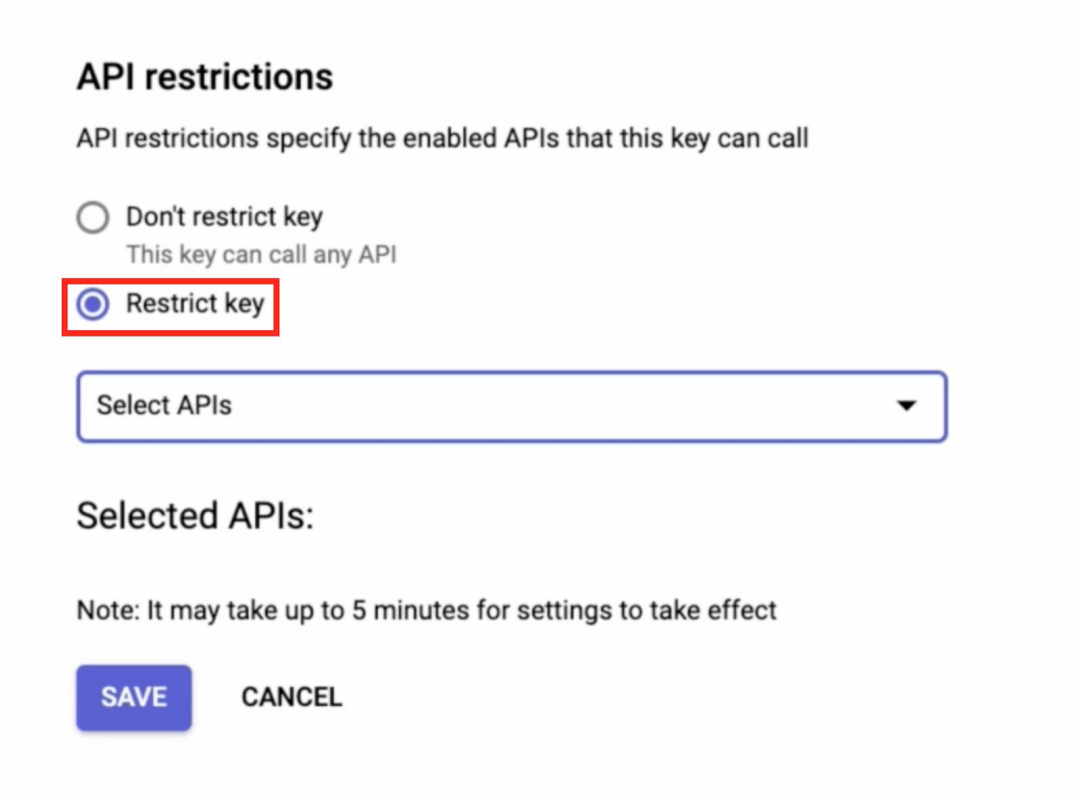
We will be using this API Key inside the vue components in some projects and it will be exposed to the client. This means anyone can open up the source code on the browser and find the API Key and be able to make HTTP requests with our API Key.

Luckily, we can protect our API Key by allowing HTTP requests that are ONLY coming from a specific domain in this case your domain.

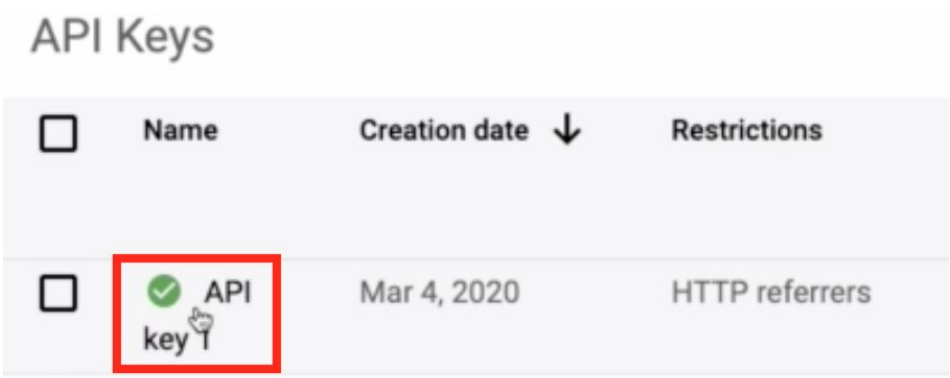
To do that, go to the **Credentials Page** and click on the name of the API Key. Then, inside the **Application Restriction** section, choose **HTTP Referrers** and go down to **Website Restrictions** and click on **Add Item** to add the domain. In my case, softauthor.com and choose **Done** and hit **Save**.



We can also restrict the API Key by giving permission to a specific Maps Library.



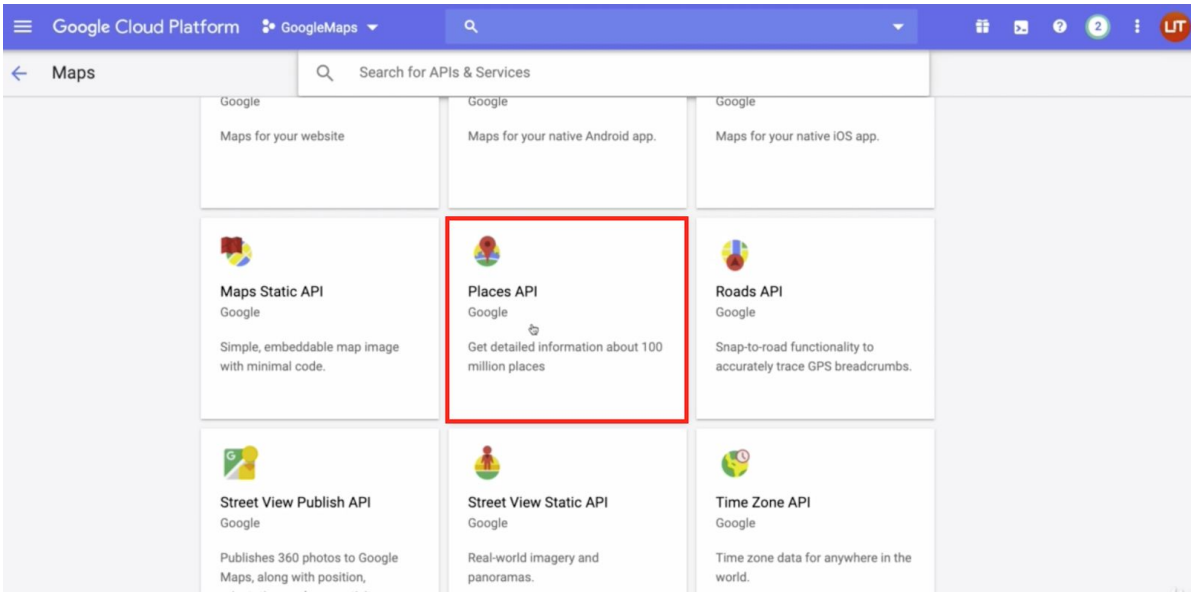
Once you've added to the domain name, you can now see a green check circle before the API Key name, indicating that the API Key is safe.



Step #4: Enable Libraries

We will be using different Google Maps Libraries throughout this course, such as Geocoding API, Places API, and so on. By default, the Libraries are not enabled by default and we will need to enable them as we need them.

To enable the Place API Library for example, go to the **Library** from the left bar and you will see all the available Google Maps API related Libraries. Choose **Places API** and click the **Enable** button at the top.



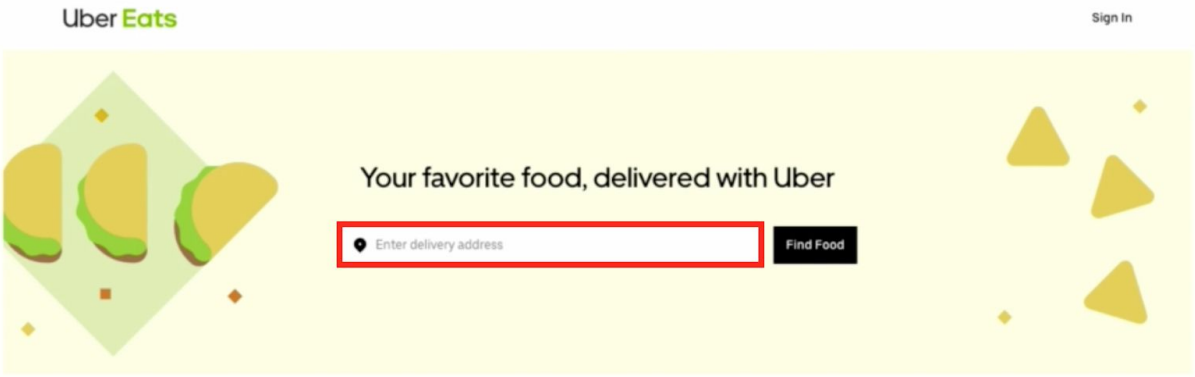
****

Section 2: Detect User's Current Location

****

What You'll Be Building By The End Of This Section

When you think about any food delivery app, the first thing it will ask for is the user's current location; where the food needs to be delivered.



You will be able to get the user's current location when a user gives permission to a locator.

If the permission is denied or the locator gives a wrong address for some reason, users will still be able to enter their addresses manually by typing it in.

By the end of this section, you'll be able to:

- Create a brand new Vue Component and the route for it
- Get User's Location Using HTML5 Geolocation API
- Convert Geographic coordinates to Address Using Geocoding API
- Get Address as user types Using Autocomplete API
- Handle Client & Server Side Error errors
- Change Autocomplete Drop Down List Styling

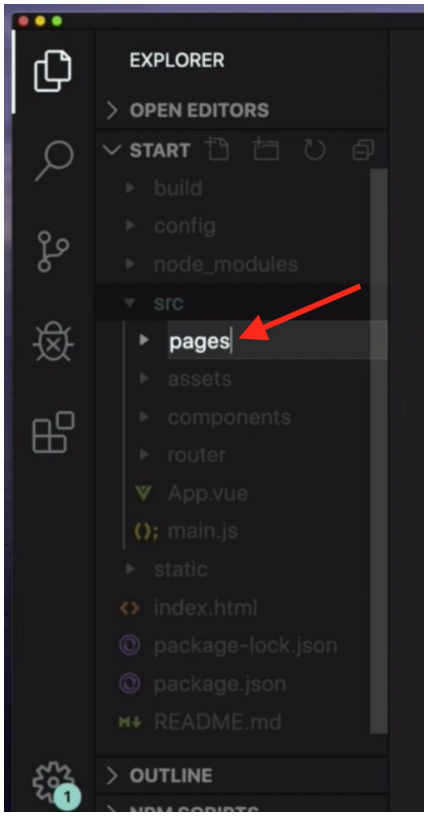
Create a Vue Component and Route

By the end of this chapter, you'll be able to:

1. Know the difference between Component vs Page Based
2. Attach a Vue Route

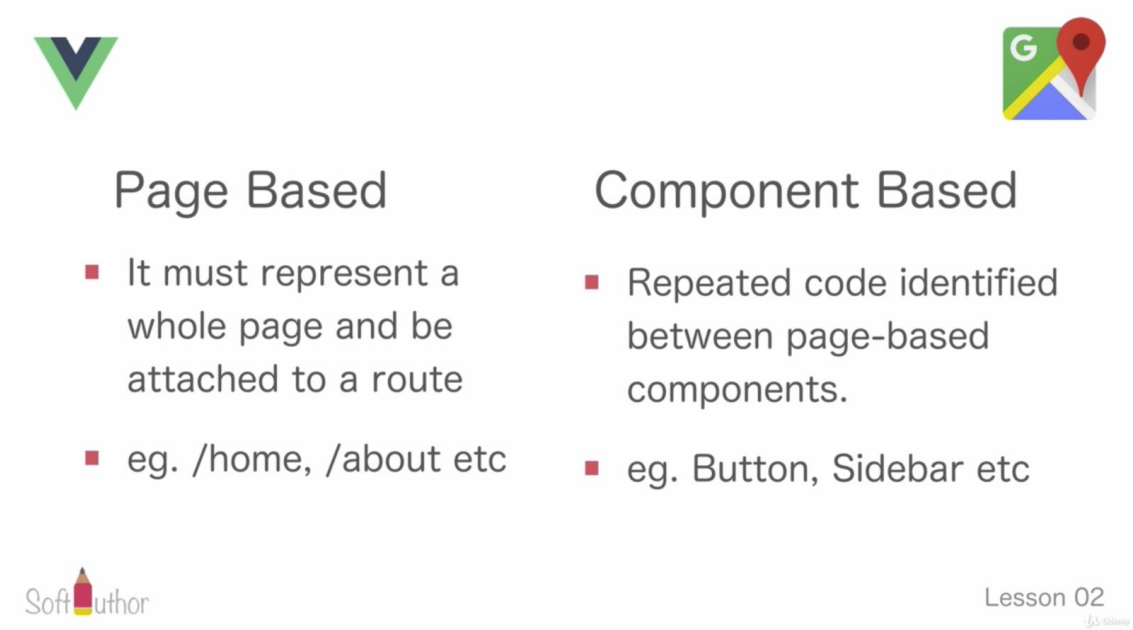
If you look at the Project Folder Structure, there is a Source Folder that has a few folders and files in it. The **Components Folder** is the place normally where Vue Component files are created.

Create a new folder called **Pages** under the **Source Directory** and create a User Location Component there. Right click the **Source Folder** and choose **New Folder** and call it **Pages**.



Step #1: Component vs Page Based

Here is the quick difference about when to create a new component inside the pages folder and the components folder.

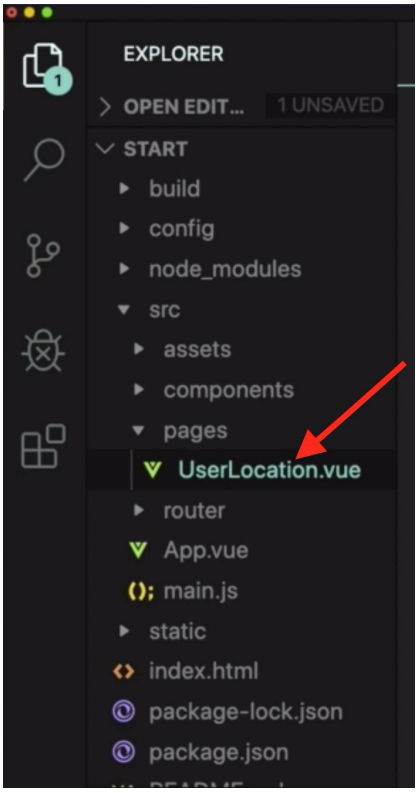


When your component is going to take a whole page on the browser and be attached to a route such as /login or /home, create a component inside the pages folder.

Whenever you have a piece of code that is repeated in more than one page based component or repeated multiple times inside a single page based component. Then, create it inside the Components Folder.

For example: navigation bar, buttons and input fields.

We are going to create a page based component. Select the **Page Folder** and right click on it to create a new file called **userLocation.vue**. I use Title Case, which is the common best practice when naming a Vue Component.



Inside the **userLocation.vue** component, create a `<template>` element, which is where all the HTML code lives inside a single div container.

Define `<h1>` tags inside with some text so that we can make sure we're in the right component when viewing on the browser.

```
<template>
 <h1>User Location</h1>
</template>
```

Now we have a simple page based Vue Component ready.

Step #2: Vue Route

A route needs to be set to this component like /home or /about, so that when a user enters any specific route on the browser, the attached component will be visible on the browser.

To attach the **home** route (/) to the **userLocation.vue** component, we need to do two steps.

Head over to the **router** folder and inside the **index.js** file:

1. Import the component at the top:

```
import UserLocation from '@/pages/UserLocation'
```

2. Add a Javascript object which should have at least two properties inside the **routes** array:

```
export default new Router({
  routes: [{
    path: '/',
    component: UserLocation
  }]
})
```

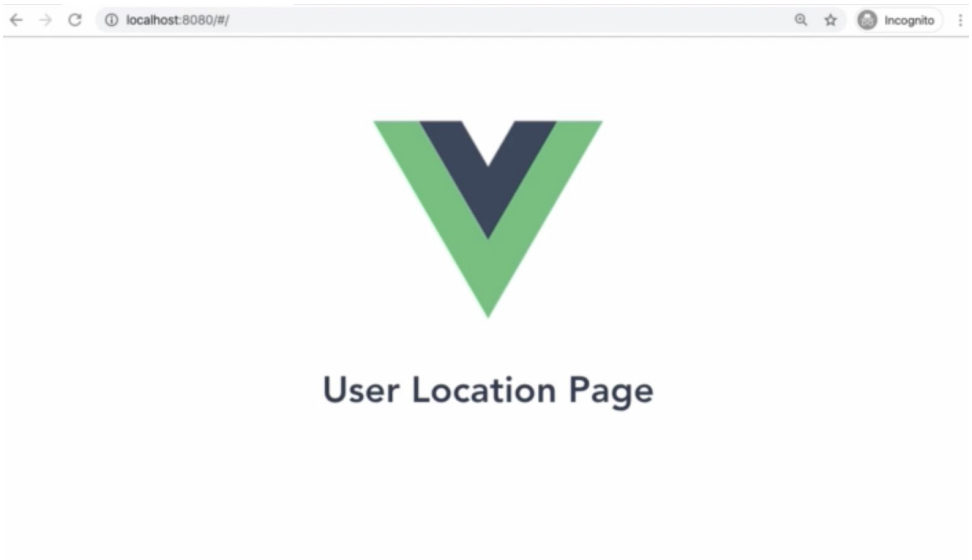
The first property will be the **path**, all lowercase with a colon. The value will be in between the quotes, which is the default route. This is the route you will go to first when running the application.

The second property is the **component**. The value of this property is the actual component **name** that you want to attach to, in this case **UserLocation**

component. This name must match the **name** that is given when importing the Vue component file.

Let's run it by opening up the terminal window from Visual Studio and going to the project using CD space, your project folder name, then executing the NPM run dev command.

```
cd yourprojectname
npm run dev
```

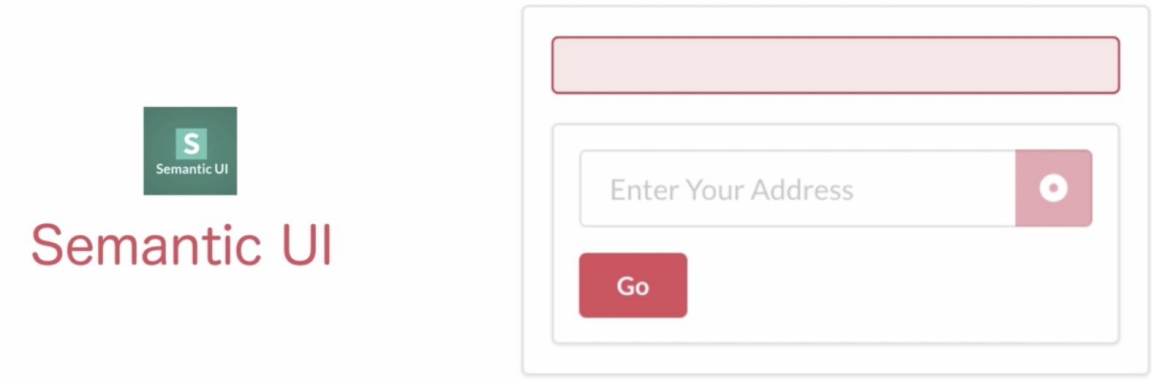


To remove the Vue logo, go to **app.vue** file and get rid of the image tag which is above the router vue directory.

Also clear the style and hit save. Then go over to the browser and the logo is gone.

At this stage, we have successfully created a user location component and assigned it to the home route (/).

In the next chapter, let's create this beautiful UI design inside a userLocation.vue file using Semantic UI.



Build User Location Form Using Semantic-UI

I'll be using Semantic UI CSS Framework throughout this course to design UI elements faster.

By the end of this chapter, you'll be able to:

1. Use Semantic UI CSS Framework
2. Design the Form Layout
3. Include Three Visible Elements (Error Message, Input Field, Go Button)

Step #1: Include Semantic UI CSS Framework

To use the Semantic UI CSS framework, we need to link it to our project.

I'll be using the CDN (Content Delivery Network) format of the CSS framework which is hosted on the Semantic UI Website.

```
<!DOCTYPE html>
<html>
<head>
  <title>User Location Detector</title>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/semantic-ui@2.4.2/dist/sem
antic.min.css">
</head>
<body>
  <div id="app"></div>
</body>
```

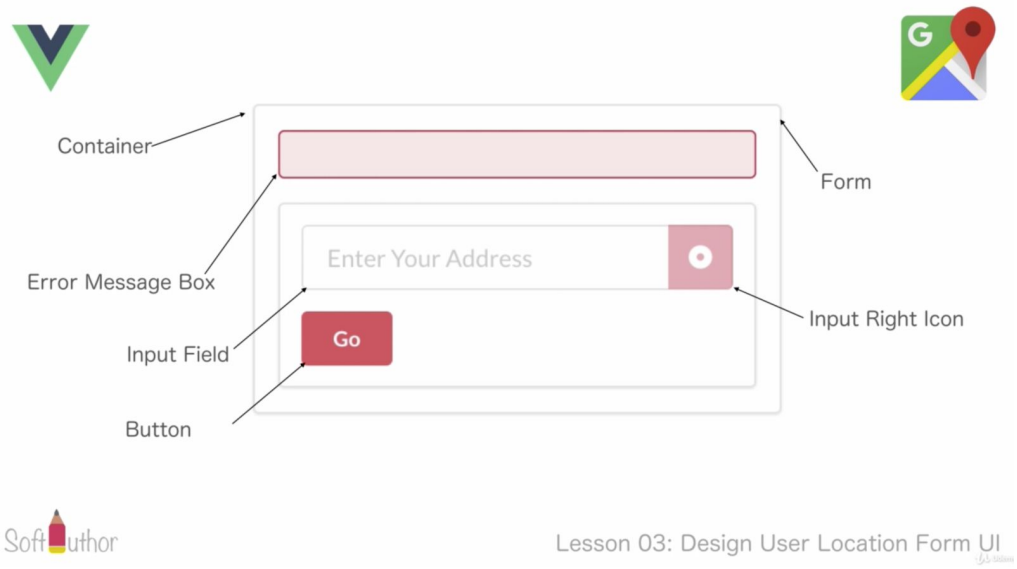
```
</html>
```

Copy the above `<link>` tag and head over to our project. Open the **index.html** file and paste it in between the head tags and hit save.

Step #2: Design Form Layout

Go to the **userLocation.vue** component and change the container element from `<h1>` to `<section>` element, which is where we will write all of our HTML code.

Let's see the design.



The first step is to create a container element and center it on the browser.

To do that, add a class attribute to the section tag with a few semantic UI classes which are **ui two column grid**.

```
<section class="ui two column grid">
```

```
</section>
```

Yes, we are making a two column grid wrapper here, and we will discover why in just a moment.

Then, declare two divs inside it with a class name **column**. Give different colors and class names for each DIV so that we can see them on the browser.

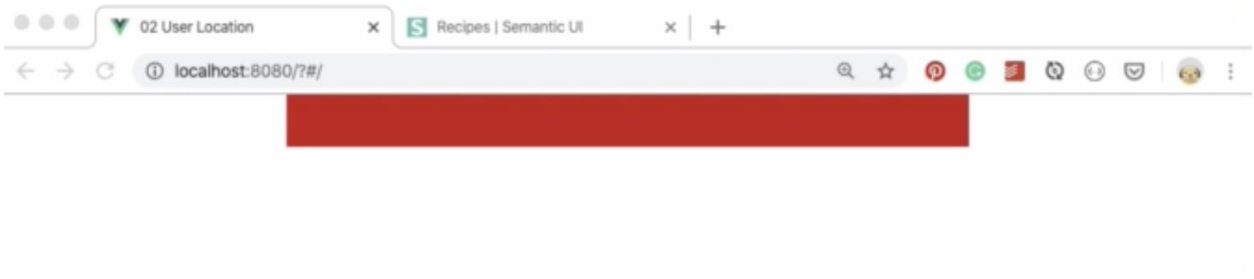
```
<section class="ui two column grid">
  <div class="column red"></div>
  <div class="column blue"></div>
</section>
```

And the view will look like this.

Now, what we want is one column that is centered. So, get rid of one of the columns and add a **centered** class to the section element.

```
<section class="ui two column centered grid">
    <div class="column red"></div>
</section>
```

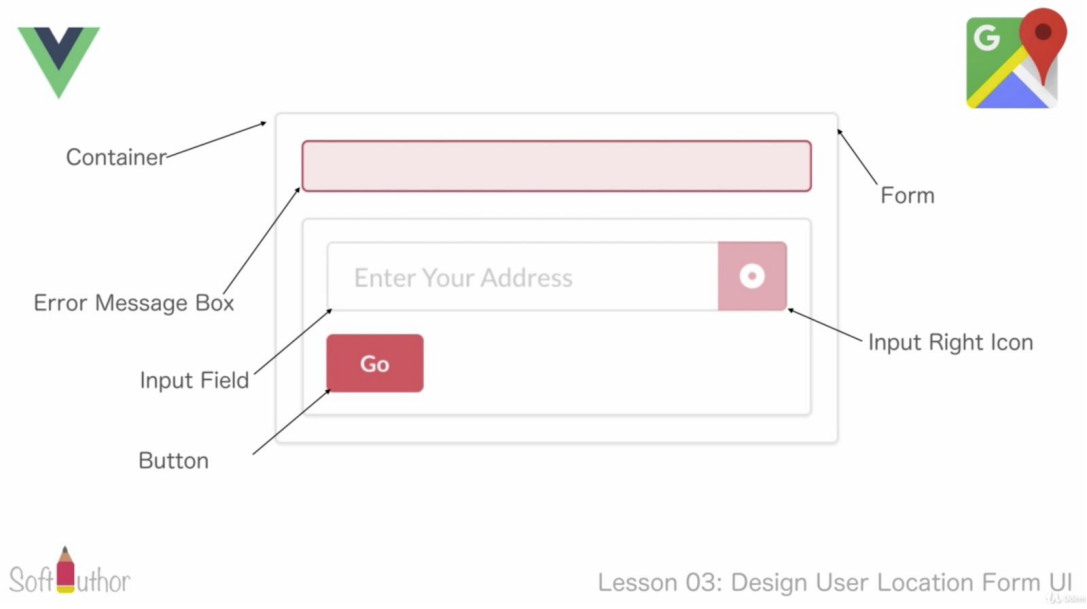
That will give a nice centered box.



Then, define a form tag inside the column with some semantic ui css classes:

The **ui segment** classes create a nice thin outline border and others are straight forward.

```
<section class="ui two column centered grid">
    <div class="column red">
            <form class="ui segment large form">
            </form>
   </div>
</section>
```



Inside, there will be three visible elements which are:

- the error message box at the top
- the actual input field with the locator button on the right
- the go button

Let's take a look at each one by one.

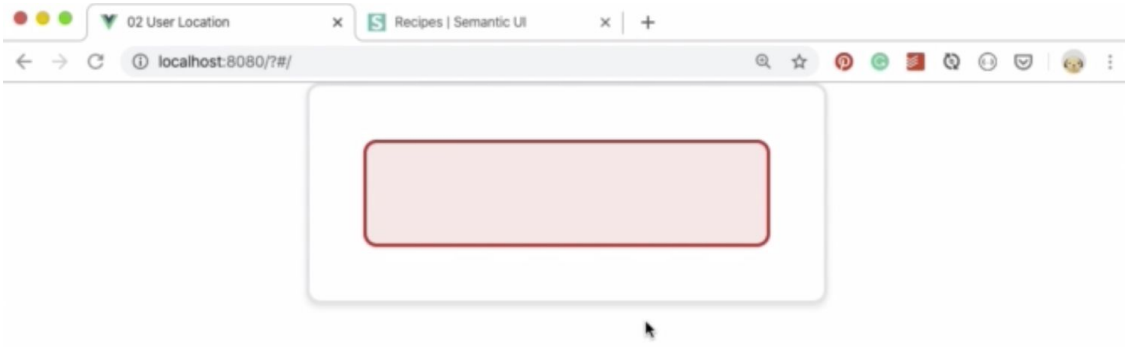
Step #3: Error Message

Inside the `<form>` element, create a `<div>` with the class names: **ui message red**.
The **ui** class is a common class name that you use all the UI elements that are visible on the browser. The **message** class puts the rounded corner border to the div. **Red** is the background color as well as the border color.

This will be toggled later based on if there is an error or not.

```
<form class="ui segment large form">
    <div class="ui message red"></div>
</form>
```

And the view will look like this.



Step #4: Input Field

Let's create a wrapper for the input field after the error message element but inside the `<form>` element.

Starting with a `<div>` element with two semantic UI CSS classes: **ui** and **segment**.

As I mentioned before, the **segment** class is very similar to the **message** class in that it adds a rounded corner border to the element.

```
<div class="ui segment">
   <div class="field">
      <div class="ui right icon input large">
         <input
            type="text"
            placeholder="Enter your address"
            />
         <i class="dot circle link icon"></i>
      </div>
   </div>
</div>
```

Then, create another `<div>` with the class name **field** which is an input group container.

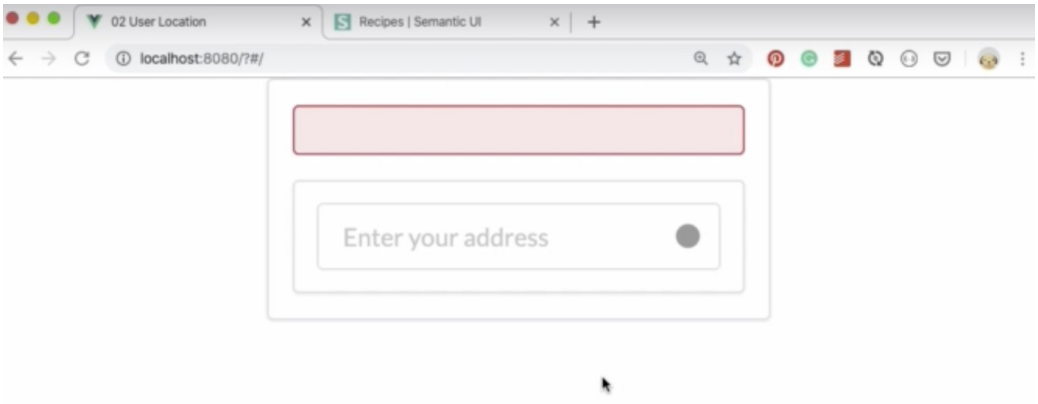
Define another `<div>` with a few semantic class names: **ui, right, icon, input, large** which is where we are going to add the input field and icon.

The **right icon** class will put the icon on the right side of the input field.

The `<input>` tag type is set to **text** and the placeholder attribute is set to **enter your address**.

Semantic UI ships with a lot of icons and you can explore them here. To get the dotted circle one, define an `<i>` element with the classes **dot circle link icon**.

At this stage, the view will look like this:



Step #5: Go Button

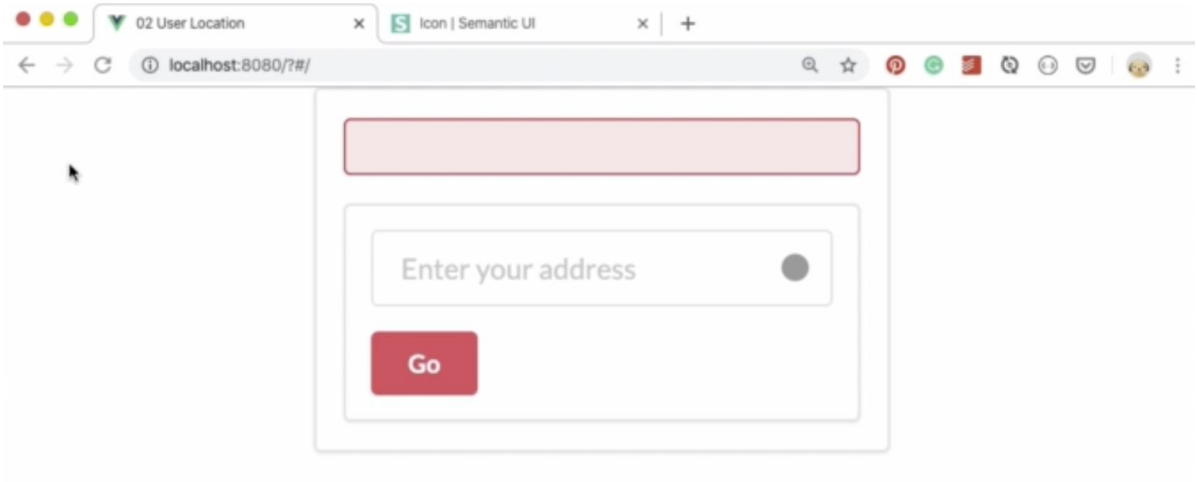
Declare a `<button>` element after the `<div>` that has input class:

```
<button class="ui button pink">Go</button>
```

Here is the full HTML code for the User Input Form:

```
<template>
  <section class="ui two column centered grid">
    <div class="column">
      <form class="ui segment large form">
        <div class="ui message red"></div>
        <div class="ui segment">
          <div class="field">
            <div class="ui right icon input large">
              <input
                type="text"
                placeholder="Enter your address"
              />
              <i class="dot circle link icon"></i>
            </div>
          </div>
          <button class="ui button pink">Go</button>
        </div>
      </form>
    </div>
  </section>
</template>
```

And the final result will look like this.



Now you know how easy it is to build this beautiful form with a few semantic UI classes.

In the next chapter, we will get the user location when a user presses the locator button, which is an icon located on the right side of the input field.

HTML5 Geolocation API

In this Chapter, we will be talking about HTML5 Geolocation API and how to get geographic coordinates (latitude and longitude) from it.

By the end of this chapter, you'll be able to:

1. Understand what HTML5 Geolocation API is and verify browser support.
2. Attach a Click Event.
3. Get Latitude and Longitude using the getCurrentPosition() Method.
4. Show an Error Message when the user denies permission

Step #1: What is HTML5 Geolocation API

HTML5 Geolocation is a browser based API which basically allows us to get a user's location in the form of geographic coordinates which are latitude and longitude values.

Here is the chart that shows which browser and it's version that supports this Geolocation API.

API					
Geolocation	5.0 - 49.0 (http) 50.0 (https)	9.0	3.5	5.0	16.0

One thing worth pointing out here is that Google Chrome version 50 or above only enables this API when using an HTTPS connection. However, it works fine on a local host which is exactly where we will be testing this.

Step #2: Attach A Click Event

Let's attach a click event to the locator button to get the user location. Add a click event handler using **@click** with a callback function called **locatorButtonPressed** to the locator icon, which is an i element.

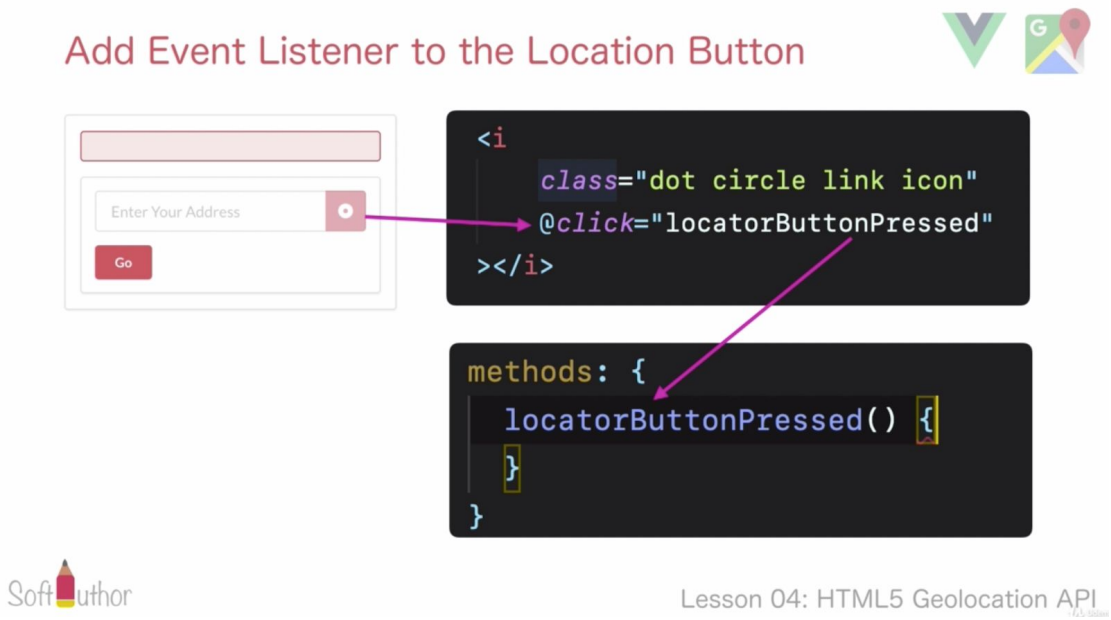
```
<i class="dot circle link icon"
@click="locatorButtonPressed"></i>
```

After the ending `</template>` tag, create a script element which is where all of the javascript code would go.

```
<script>
export default {
}
</script>
```

In Vue, we create all the event handler callback functions inside the methods object, which is the standard best practice.

```
<script>
export default {
  methods: {
        locatorButtonPressed() {
        }
  }
}
</script>
```



The first thing we want to do inside the function is to check if the Geolocation API is supported by the user's browser or not.

Inside the **locatorButtonPressed** callback function, check to see if the Geolocation is supported by the using **navigator.geolocation** object.

```
export default {
  methods: {
    locatorButtonPressed() {
      if (navigator.geolocation) {
        console.log("Your browser supports Geolocation API");
      } else {
        console.log("Your browser does not support
Geolocation API");
      }
    }
  }
};
```

If the browser supports the Geolocation API, we can get the geographical coordinates using one of it's methods, called **getCurrentPosition** upon the user's permission.

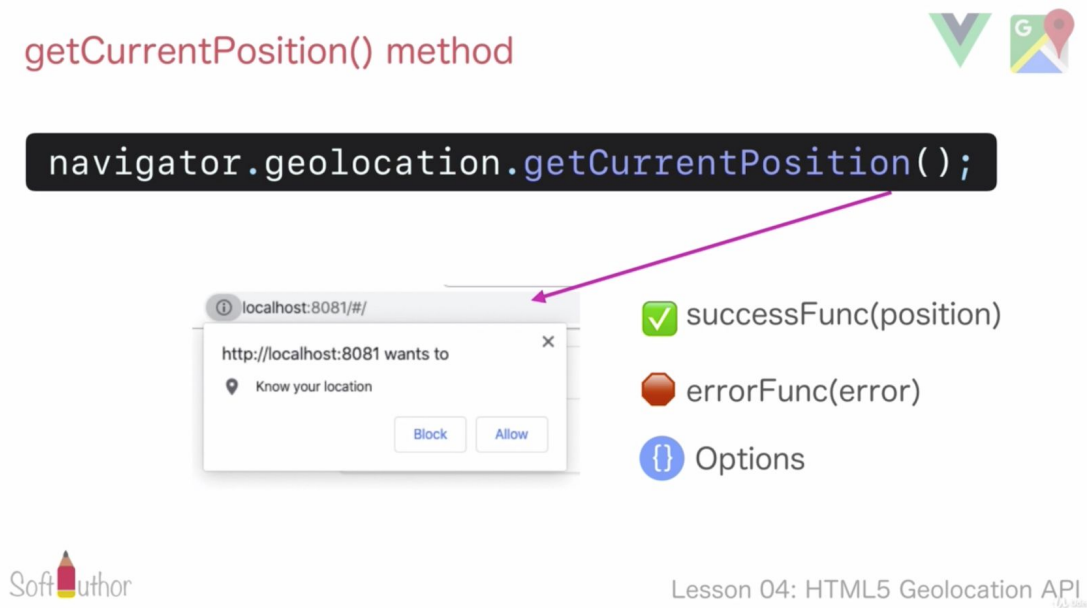
Otherwise, users will still be able to type their address using the Autocomplete API, which will be covered later in this section.

Step #3: Get User Location Using getCurrentPosition()

The getCurrentPosition() method takes three arguments:

- The success function - successFunc(position)
- The error function - errorFunc(error)
- An options object

When the getCurrentPosition is called, a browser notification window will appear close to the URL bar on the browser and state, in my case, "the local host wants to know your location."



If the **Allow** button is pressed the **getCurrentPosition** method will return the coordinate object to the success function specified in the parameter called **position**.

On the other hand, when the **Block** button is pressed, the error function will be invoked and will return the object specified in the **error** parameter.

Inside the if block, invoke the **getCurrentPosition** method and semicolon.

In between the parenthesis, declare the first argument, which is the success error function, with a parameter position in which we can get latitude and longitude coordinates. As you can see, I have console logged them in the code.

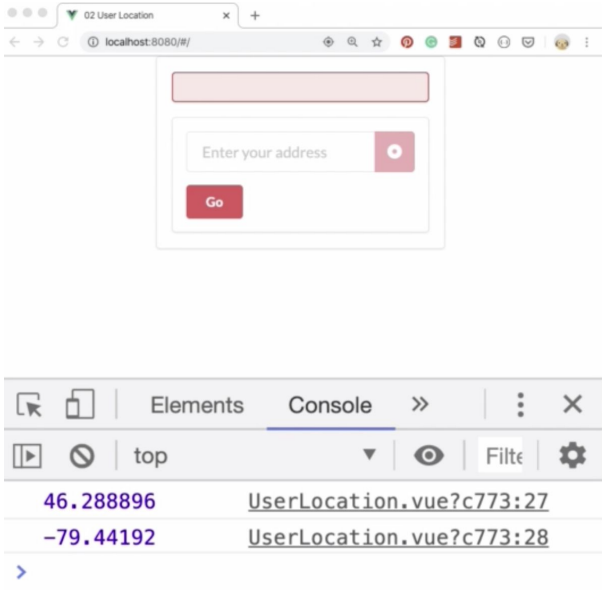
```
if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
          position => {
           console.log(position.coords.latitude);
           console.log(position.coords.longitude);
          },
          error => {
             console.log(error.message);
          }
        );
} else {
        console.log("Your browser does not support
geolocation API ");
}
```

Step #4: Show Error Message

Declare the second argument, which is the error arrow function, and **console.log** opening and closing parentheses **error.message**.

Save the file and switch back to the browser. Open the developer console by right-clicking the browser and select **Inspect**.

Click the locator button. The notification window pops up, choose **Allow**, and you can see the latitude and longitude coordinates are printed in the console.

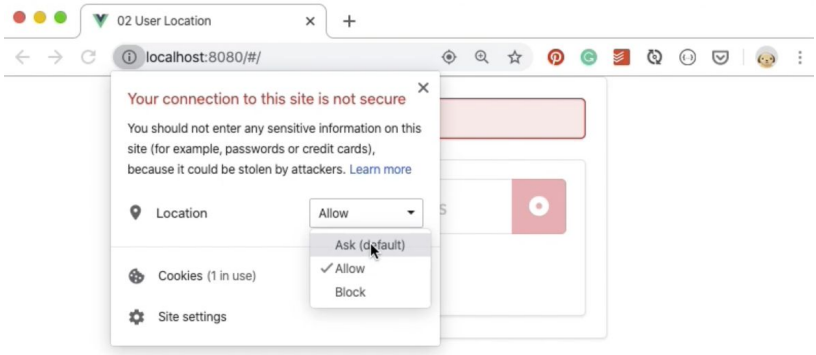


Let's see if the error message works as expected by blocking the location. Click the locator button again, and this time the notification does not show up.

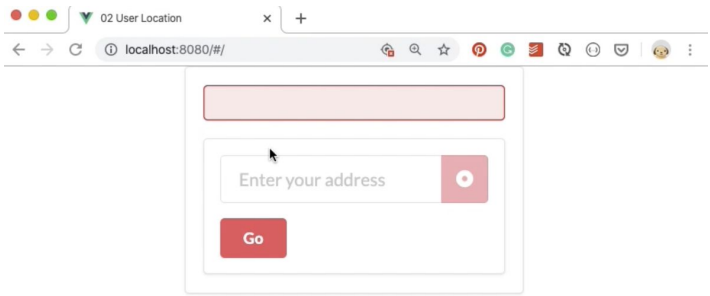
Why?

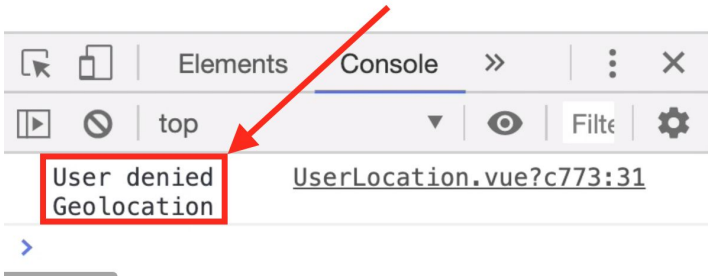
This is because we already gave permission to the browser to get our location.

To reset it in chrome, click on the i icon, which is before the address bar and change the location option from allowed to ask and reload the page.



Lets try it again. This time choose the block option and you will see the error message appear in the developer console.





The next step is to convert these coordinates to an actual human readable mailing address using Google's Geocoding API, which is exactly what we will be covering in the next chapter.

Make An HTTP Request To Google's Geocoding API

In this chapter, I will show you how to get an actual address using geocoding API by making an HTTP request using Ajax.

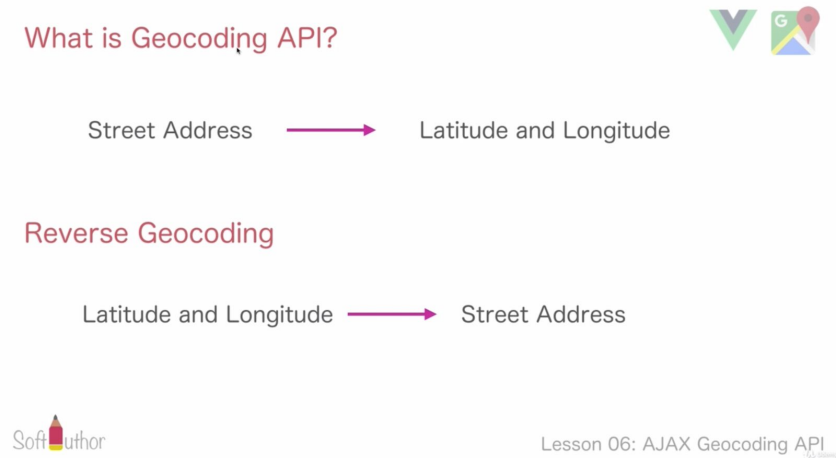
By the end of this chapter, you'll be able to:

1. Understand Geocoding API
2. Make an HTTP Request Using Axios
3. Show Address On The Input Field
4. Check Error Messages

Step #1: Google's Geocoding API

Geocoding is the process of converting street addresses to geographic coordinates, which are latitude and longitude.

However, what we want is the opposite as we already have the coordinates.



So, what we need is Reverse Geocoding, which is converting geographic coordinates to an actual human readable street address.

Using Google's Geocoding API service we can easily do that.

Here is the HTTP URL:
https://maps.googleapis.com/map/api/geocode/json

Here are the required route & query parameters:

Required Route Parameter

name	**Description**
json or xml	It determines what format the return output data should be.

Required Query Parameters

name	**Description**
latlng	The latitude and longitude values separated by a comma. *Example: 42.55454, -75.645354*
key	It's an API Key (I showed you how to get it in the earlier chapter).

Step #2: HTTP Request Using Axios

I am going to be using **axios** which is an HTTP client to make an ajax request.

This is going to be a three step process:

1. Install axios.
2. Import to the user location component.
3. Make a request using the **get** method on the axios object.

1. Open up the terminal from the visual studio code editor: view -> terminal and run:

```
npm install --save axios
```

It will take a few seconds to install it to our project.

2. Then, import the axios module inside the **userLocation.vue** at the top inside the script tags.

```
import axios from 'axios'
```

Now we have axios ready to use. Let's declare a function called **getAddressFrom** with two parameters, **lat** and **long** inside the **methods** object. This is where I'm going to make an HTTP request just in a moment.

```
getAddressFrom(lat, long) {
}
```

Then, call **getAddressFrom()** function inside the **getCurrentPosition()** success callback function where we get geographic coordinates and pass the latitude and longitude as arguments to that function.

```
navigator.geolocation.getCurrentPosition(
  position => {
    this.getAddressFrom(
      position.coords.latitude,
      position.coords.longitude
    );
  },
)
```

In Vue, we use **this** keyword to call a function that is declared inside a methods object.

In the **getAddressFrom** function, invoke the **get** method on the **axios** object and pass the geocoding HTTP endpoint URL.

As you can see, I am concatenating **lat** and **long** values to the URL separated by a comma and adding the API Key as a second query parameter using **&** sign.

The value of the key is your actual Google API key and make sure to update with yours where it says `[YOUR_API_KEY].`

The **axios.get()** method will return a promise, so add the **then()** function which will have a return response object specified in the parameter called **response** if the request is fulfilled.

```
getAddressFrom(lat, long) {
      axios
        .get(

"https://maps.googleapis.com/maps/api/geocode/json?latlng=" +
            lat +
            "," +
            long +
            "&key=[YOUR_API_KEY]"
        )
        .then(response => {
          if (response.data.error_message) {
            console.log(response.data.error_message);
          } else {

console.log(response.data.results[0].formatted_address);
          }
        })
        .catch(error => {
          console.log(error.message);
        });
}
```

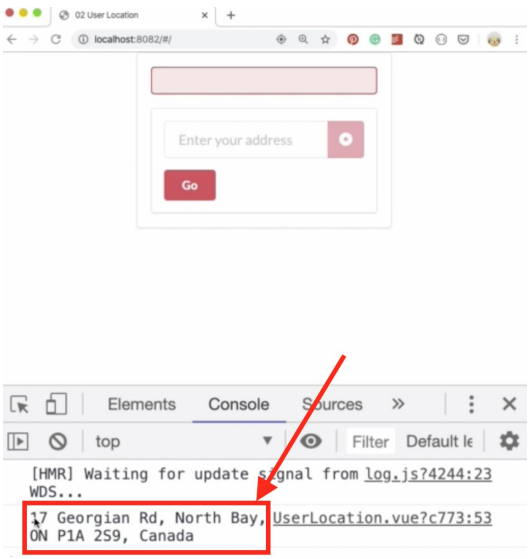
If not, the catch function will be executed and we can capture the error using **error.message**.

Sometimes, the response object itself will have an error message such as “API key is not valid,” “billing is not enabled” or “you need to enable the library,” etc and we can capture that as well using **response.data.error_message**.

If everything goes well, the actual address will be in the following path in the response object.

```
console.log(response.data.results[0].formatted_address);
```

Run the application and you'll be able to see the actual human readable address in the developer console.



🛑 Caution!

If you get a CORS error when making an HTTP request to Geocoding API like this:

Access to XMLHttpRequest at 'https://maps.googleapis.com/maps/api/geocode/json?latlng=7.9411343,98.3942565&key=xxxxx from origin 'http://localhost:8080' has been blocked by CORS policy: Cross-origin requests are only supported for protocol schemes: HTTP, data, chrome, chrome-extension, https.

Then:

✔️ Append “https://cors-anywhere.herokuapp.com/“ to the URL.

Example:

https://cors-anywhere.herokuapp.com/https://maps.googleapis.com/maps/api/geocode/json?latlng=7.9411343,98.3942565&key=[YOUR_API_KEY]

Also, please make sure to add your own **API KEY** at the end. We will go into detail about CORS error in the next chapter

Step #3: Show Address On The Input Field

Let’s show the address on the input field so that the user can see what address he/she selects.

To do that:

1. Define a property called **address** inside the data model at the top inside the export default object.

```
 export default {
  data() {
    return {
      address: "",
    };
  },
}
```

2. Bind **address** property to the input field using v-model directive.

```
<input
     type="text"
     placeholder="Enter your address"
     v-model="address"
/>
```

3. After that, all we have to do is set the actual address from the response object to the address property.

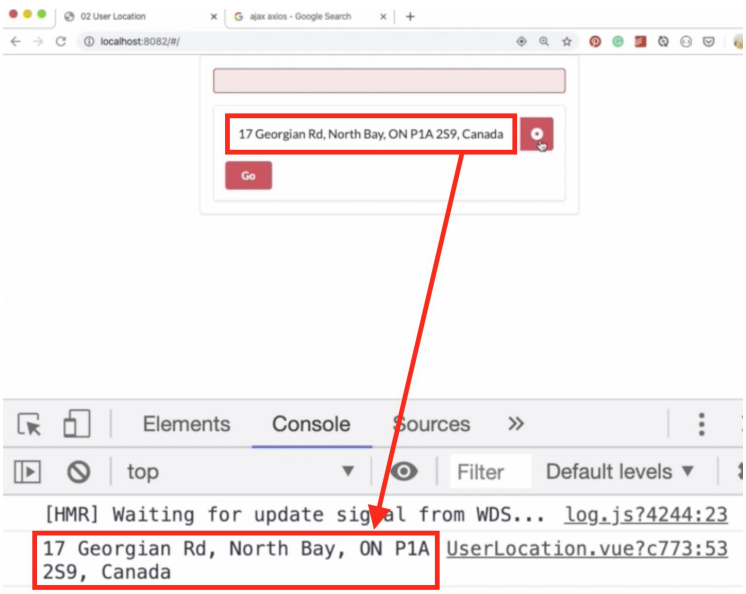
So instead of this:

```
console.log(response.data.results[0].formatted_address);
```

Replace with this:

```
 this.address = response.data.results[0].formatted_address;
```

Let's test it out.



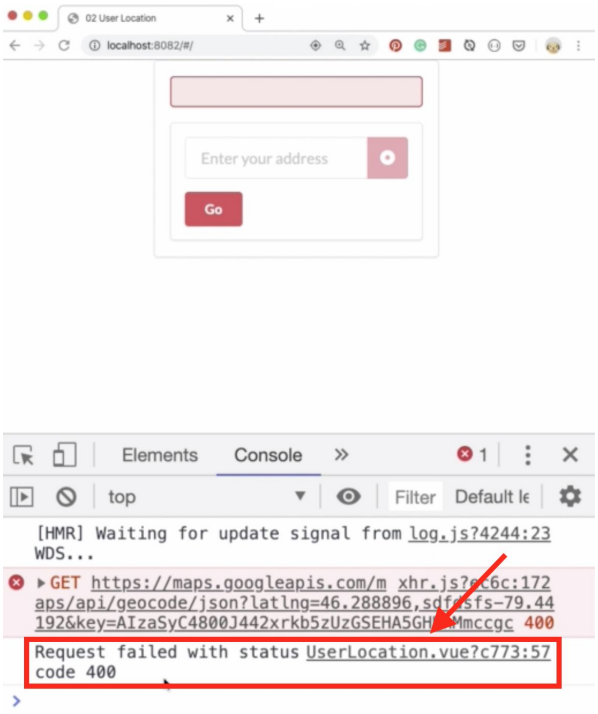
Perfect, but we are not done yet.

Step #4: Error Checking

Let's see if the errors are working as expected.

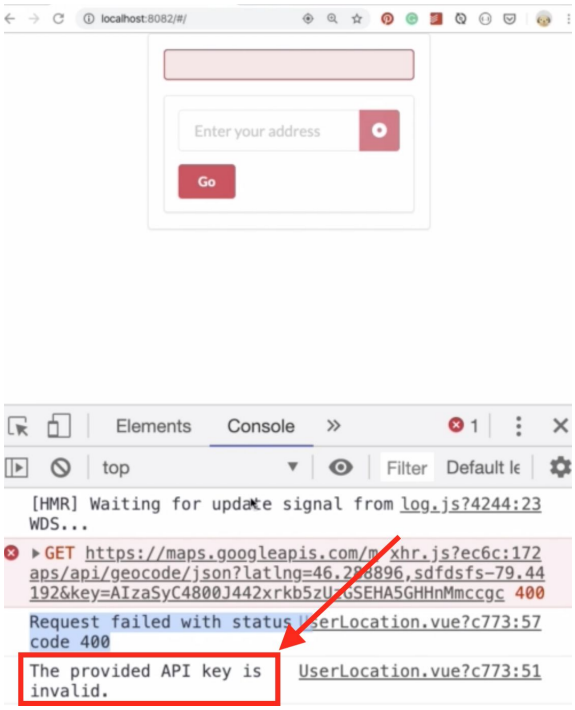
Make some changes to the HTTP endpoint, maybe add some text after the comma and save the file.

Switch back to the browser, and you can see the request failed with status code 400 in the developer console when trying to get the user location.



As you can see, the line number actually matches so we know it's coming from the catch function.

To check the error message in the response object, make the API key invalid by adding some text to it and try it.



This time it says a different error “the provider API key is invalid” which is coming from the **then** function inside the **if error** check.

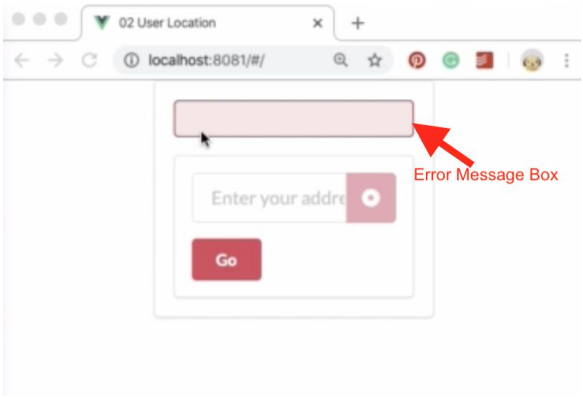
In the next chapter, I'll be showing you how to show the errors when it happens in the error message box at the top of the form.

Handling Client and Server Errors

By the end of this chapter, you'll be able to:

1. Set the Error Message
2. Add a Spinner / Loader

As you can see, the error message box shows regardless at the top.



Let's change it so that it is only visible if it's an actual error - otherwise hide it.

Declare a property called **error** inside the data model and set its initial value to an empty string.

```
data() {
   return {
     address: "",
     error: "",
   };
 },
```

When the error property contains an empty string, the error message element will be hidden, otherwise it will be visible with the error text.

So we can easily toggle the error message element by adding a vue directive called **v-show** to it.

```
<div class="ui message red" v-show="error">{{error}}</div>
```

The name **error** in the quotes is referring to the property name **error**. Then, bind the actual error message to the element using double curly braces in-between the tags.

Step #1: Set the Error Message

Set the error property to all the places where we can get a potential error message.

1. In the **locatorButtonPressed** function, inside the error arrow function, which is the second argument of **getCurrentLocation**, set the error message to the error property.

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(
    ...
    error => {
      this.error = error.message;
    }
  );
} else {
  this.error = error.message;
}
```

2. Do the same in the **else** block where we check if the Geolocation API is supported.

*You will need to use **this** keyword in front of any property which is declared inside the data function, similar to calling a function that was declared in the methods object that we saw in the earlier chapter.*

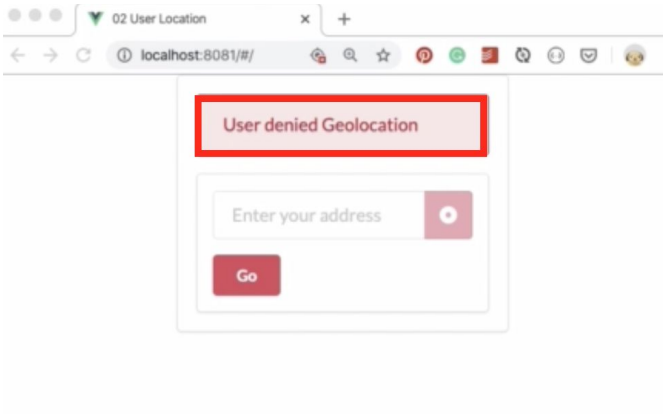
3. Let's set the error message inside the **getAddressFrom** function as well.

The first one is inside the **then** function **if** block and the other one is inside the **catch** block:

```
getAddressFrom(lat, long) {
  ...
  .then(response => {
      if (response.data.error_message) {
        this.error = response.data.error_message;
      } else {
        this.address =
response.data.results[0].formatted_address;
      }
    })
    .catch(error => {
      this.error = error.message;
    });
}
```

That's it. Let's try it out.

To get the error message, deny the permission, then the error message element will show up with the text.



That's awesome.

Step #2: Add a Spinner / Loader

Sometimes getting the user location will take a few seconds depending on various factors such as Internet connectivity, user computer speed, and so on.

It would be nice if we showed a spinner until the user gets an actual address or error message.

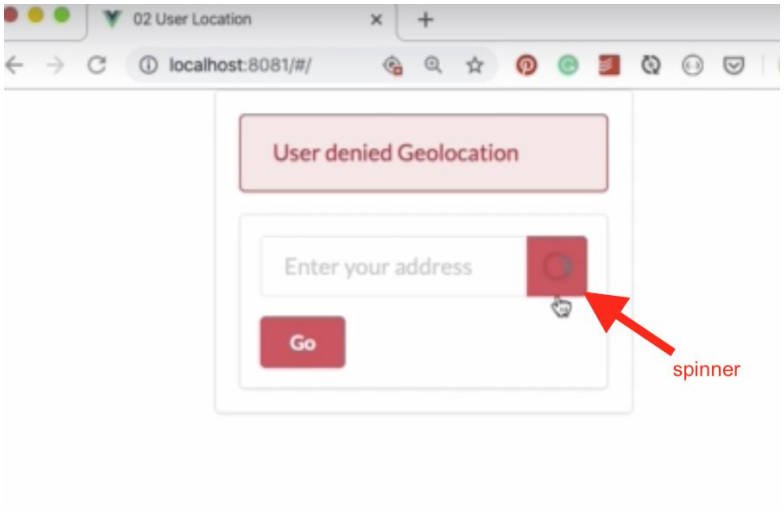
Luckily, we can add a spinner animation quickly on the locator icon on the right side of the input field using a semantic UI class called **loading**. :)

Let's take a look at how to implement that.

Inside the input container, add a loading class along with other classes.

```
<div class="ui right icon input large loading">
    <input
       type="text"
       placeholder="Enter your address"
       v-model="address"
     />
     ...
</div>
```

You can see the spinner inside the locator button like below.



Now, let's add the **loading** class dynamically whenever there is a delay before getting an address or error.

To do that:

1. Create a property called **spinner** inside the data model with an initial value of **false**.

```
data() {
    return {
      address: "",
      error: "",
      spinner: false
    };
},
```

Using this property, add or remove (toggle) the loading class dynamically from the input container.

I've set its initial value to **false** so that the spinner is hidden by default.

2. Let's add the loading class to the input container when the spinner property is true.

Before the end of the starting tag of the input container, add dynamic class using **:class**.

```
<div class="ui right icon input large"
:class="{loading:spinner}">
      ...
</div>
```

Inside the quotes and curly braces, the **loading** on the left side is referring to the class name that we want to add to the element when the **spinner** on the right, which is referring to the property name, is set to true.

3. Let's set the **spinner** to **true** at the top of the locaterButtonPressed function.

```
methods: {
    locatorButtonPressed() {
        this.spinner = true;
        ...
    }
}
```

This way as soon as the user presses the located button, the spinner becomes visible.

Let's hide the spinner as soon as we get an actual address or error in 4 places like the code below.

```
methods: {
    locatorButtonPressed() {
        ...
          error => {
            this.error = error.message
            this.spinner = false;
          }
        );
      } else {
        this.error = error.message;
        this.spinner = false;
      }
    },
    getAddressFrom(lat, long) {
      ...
        .then(response => {
          if (response.data.error_message) {
```

```
            this.error = response.data.error_message;
          } else {
            this.address =
response.data.results[0].formatted_address;
          }
          this.spinner = false;
        })
        .catch(error => {
          this.error = error.message;
          this.spinner = false;
        });
    }
  }
```

One thing worth pointing out here about the error is, whenever the user denies the permission to access the location, you get a message like “User Denied Geolocation.”

Sometimes it's recommended to use your own error message depending on the situation.

In this case, rather than saying “User Denied Geolocation,” say “locator is unable to find your address, please type your address manually,” which will direct the users on to what to do next.

```
~~this.error = error.message;~~
this.error = "Locator is unable to find your address. Please
type your address manually.";
```

However, we don't have the autocomplete functionality yet that will allow users to get their own addresses manually.

Enable Autocomplete API

By the end of this chapter, you'll be able to:

1. Know what Autocomplete is.
2. Instantiate Autocomplete
3. Restrict Suggested Addresses to a Specific Region.
4. Change Drop-down list style

Step #1: What is Autocomplete?

Let's show you how to useAutocomplete API to show suggested addresses in a dropdown list when users start typing their addresses in the input field.

This is very useful when a user denies sharing his or her location accidentally or when the HTML Geolocation API does not support their browser. We can easily get this functionality working using autocomplete.

Autocomplete is a part of Google's Places Library in the Google Map Javascript API.

The first thing we need to do is include Google Places Library inside the index.html file before the ending body tag.

And make sure that Maps JavaScript API and Places API libraries are enabled in the Google Cloud Console.

```
<script
src="https://maps.googleapis.com/maps/api/js?libraries=places
&key=[YOUR_API_KEY]"></script>
```

As you can see, the URL is generic and I've added two query parameters at the end.

name	description
libraries	The value of this will be the library name that we want to use, in this case **places**.
key	API key from the Google Cloud Console

Step #2: Instantiate Autocomplete

The next thing we want to do is instantiate the autocomplete object.

To do that, switch back to the user location component. After the data function, create another function called mounted:

```
mounted() {
},
```

This will be called whenever the DOM is ready, so it's a great place to do any DOM manipulation.

Let's instantiate the autocomplete object in there:

```
mounted() {
   new google.maps.places.Autocomplete()
```

```
},
```

It will take a few parameters.

The main one is the input element that we want to add the auto complete functionality into.

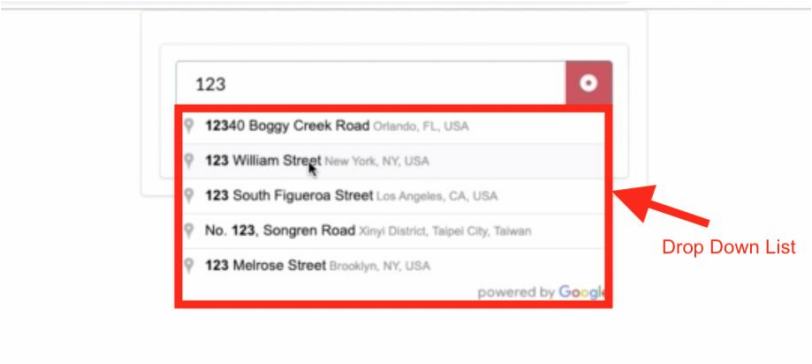
To do that, set a **id** attribute called **autocomplete** to the input field:

```
<input
     type="text"
     placeholder="Enter your address"
     v-model="address"
     id="autocomplete"
 />
```

Then inside the parenthesis, add the input DOM element as a first argument.

```
mounted() {
   new google.maps.places.Autocomplete(
      document.getElementById("autocomplete")
   )
},
```

That’s it! Let's save the file and run the app.

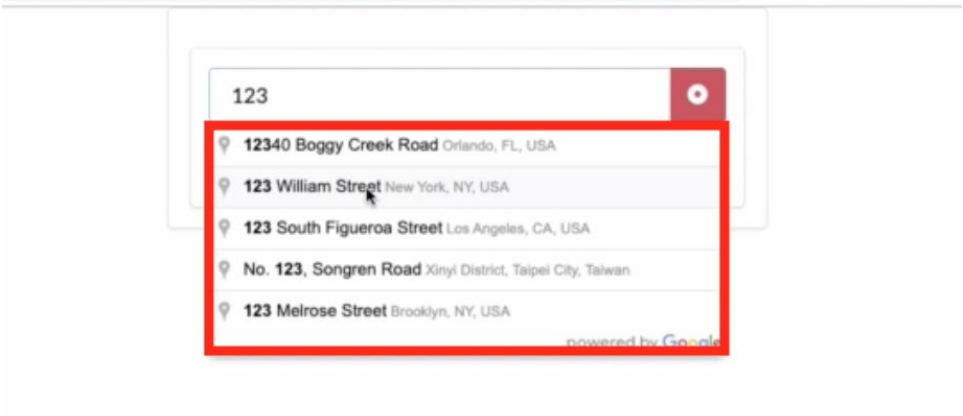


That's great!

Step #3: Restrict Suggested Addresses to a Specific Region

Let's see how we can show addresses from a specific region close to a user’s location so that it's easy to find his or her address with minimal typing.

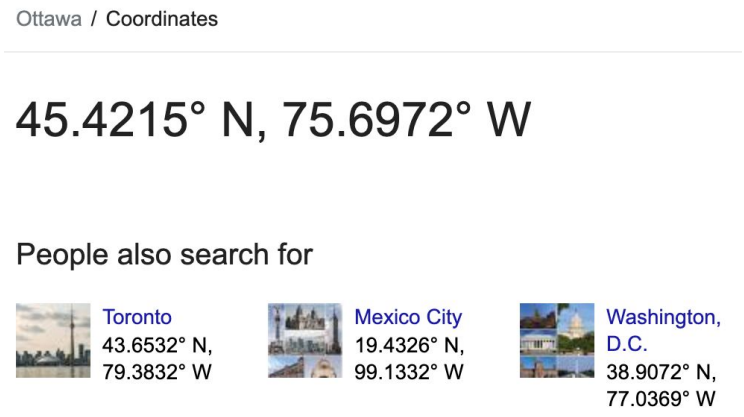
For example, I live in Canada and when I type 1 2 3 the suggested addresses are from the U.S. and Taiwan.



I can still get my address but it takes longer.

Let's say I want to show Ottawa street addresses first before showing suggestions from other cities or countries.

To do that, we need to have the geographic coordinates of the city, in this case, Ottawa. You can find that information by doing a simple search like Ottawa Ontario Geographic Coordinates... and there it is.



This is actually DD coordinates (decimal degree). What we want are simple standard coordinates.

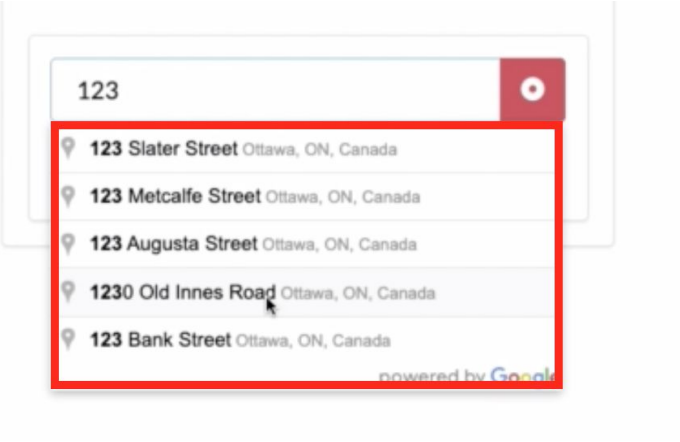
Click the first link, and the simple standard is the first option.

Now that we have the coordinates, to add them into the Autocomplete object we need to pass a Javascript object as a second argument.

After the comma, create a Javascript object and define a property called **bounds** inside it. Set its value to a **LatLng** object passing Ottawa coordinate values as arguments to it.

```
mounted() {
   new google.maps.places.Autocomplete(
      document.getElementById("autocomplete"),
      {
     bounds: new google.maps.LatLngBounds(45.4215296,
-75.6971931)
      }
   )
},
```

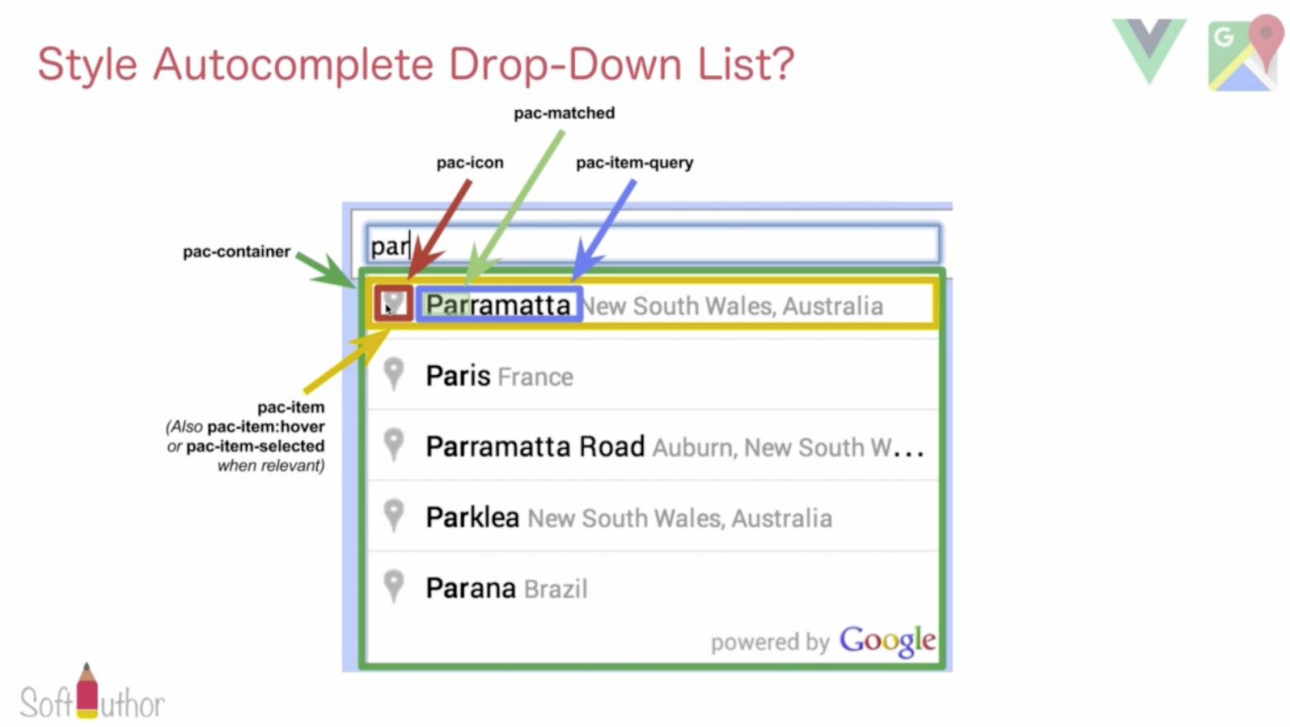
Let's switch back to the browser. As you can see, it shows all of the Ottawa addresses first followed by other matching addresses.



Step #4: Change Drop-down List Style

Let's change some of the style of the dropdown list to match our theme.

This image shows what CSS classes we need in order to change the list style.

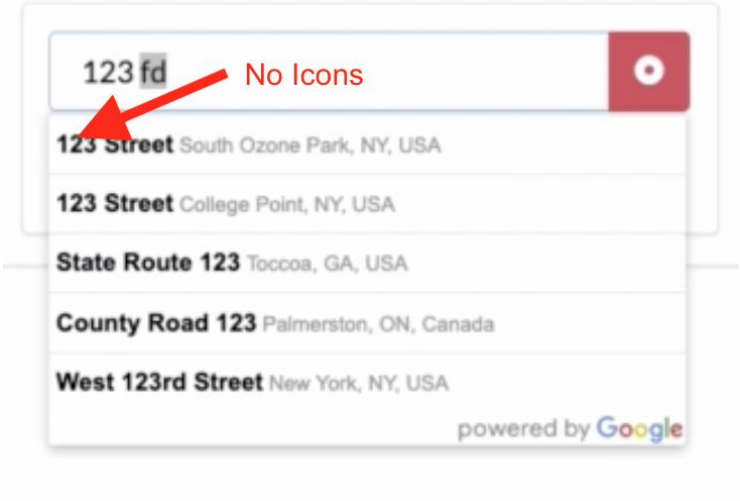


Create an opening and closing style tag after the end of the script tag in the userLocation.vue component.

First, get rid of the icon on the left side of each address which is the **.pac-icon** class.

Inside the class, add **display: none;** to hide that.

```
<style>
.pac-icon {
  display: none;
}
</style>
```



Let's increase the padding for each item. The target CSS class for that would be pac-item. Give padding of 10 pixels to all sides.

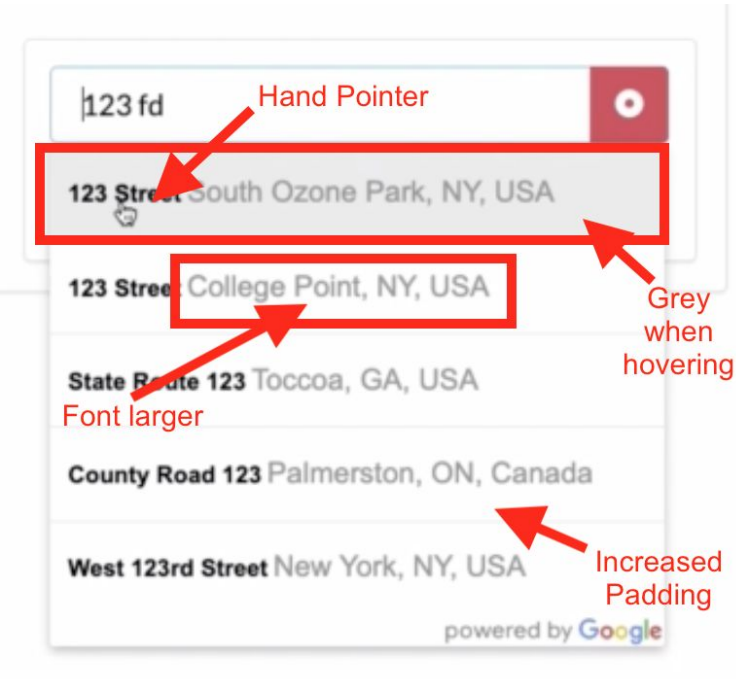
Then, increase the font size of the text to 16 pixels and let's change the cursor to a pointer (which is the hand symbol).

Finally, give a hover state for the pac item class with a grey background color.

```
.pac-item {
  padding: 10px;
  font-size: 16px;
  cursor: pointer;
}

.pac-item:hover {
  background-color: #ececec;
}
```

Let's look at the changes we made.



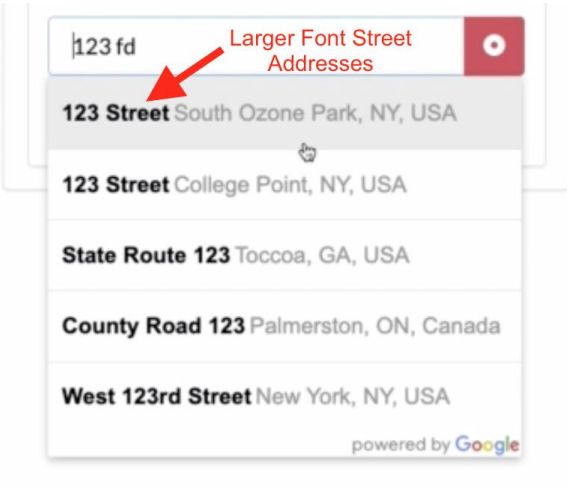
That looks better.

As you can see, the city, province and the country has a larger font size as expected, but the actual street name did not change.

To change the street name font size we need to use another CSS class named pac-item-query.

```
.pac-item-query {
  font-size: 16px;
}
```

Let's try it one more time and it looks great!

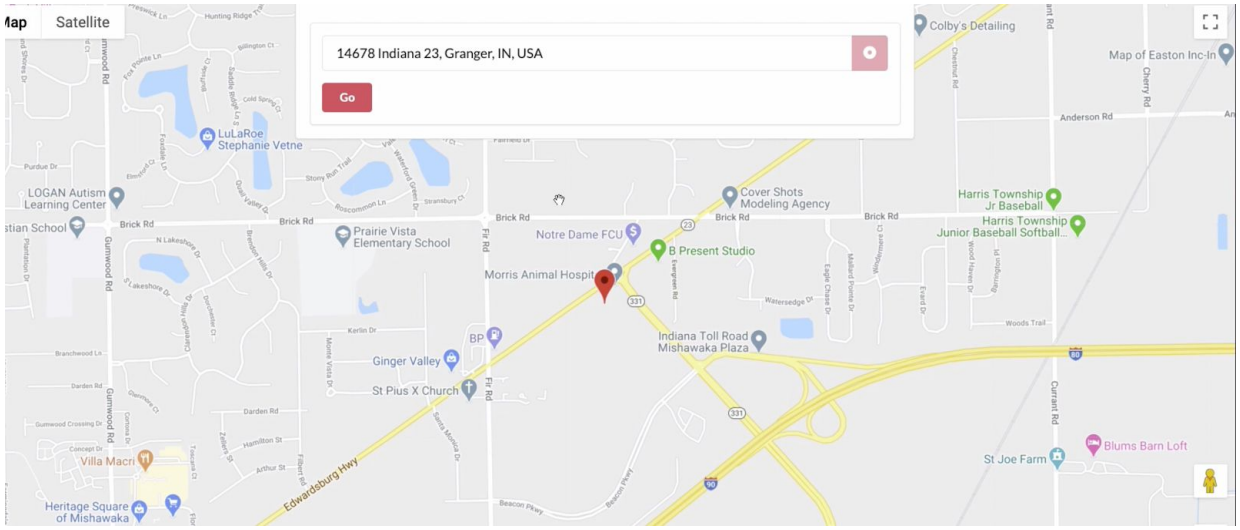


Show User's Current Location On The Google Map

In this chapter, we will be learning how to show the user's current location on the Map; whether the user gets a location by pressing the locator button or by selecting a specific address from the Autocomplete drop down list.

By the end of this chapter, you'll be able to:

1. Show the User's Location Using Locator Button
2. Show the Map Element Full Width
3. Show Google Maps On The View
4. Place Marker On The Map Using Locator
5. Place Marker On The Map Using Autocomplete



Step #1: Show the User's location Using Locator Button

As you can see in the previous image, the map is in full screen mode behind the form.

First, add an HTML Element that will be used to display google maps.

Go to the template at the top and after the end of the section tag, define another element called section with an id attribute and set its value to map.

```
<template>
    <section class="ui two column centered grid">
      ...
    </section>
    <section id="map"></section>
</template>
```

As soon as we add this element, the code editor gives an error when we mouse over it:

The template root requires exactly one element.

The reason for this error is because, in Vue, there should be only one element, which is a parent, inside the template tags. All other HTML elements should be inside that parent element.

To fix that, wrap both section elements with a div element:

```
<template>
  <div>
    <section class="ui two column centered grid">
      ...
    </section>
    <section id="map"></section>
  </div>
</template>
```

And the error is now gone!

Step #2: Show the Map Element Full Width

Let's add some CSS properties to the HTML map element to make it the full width and move it behind the form.

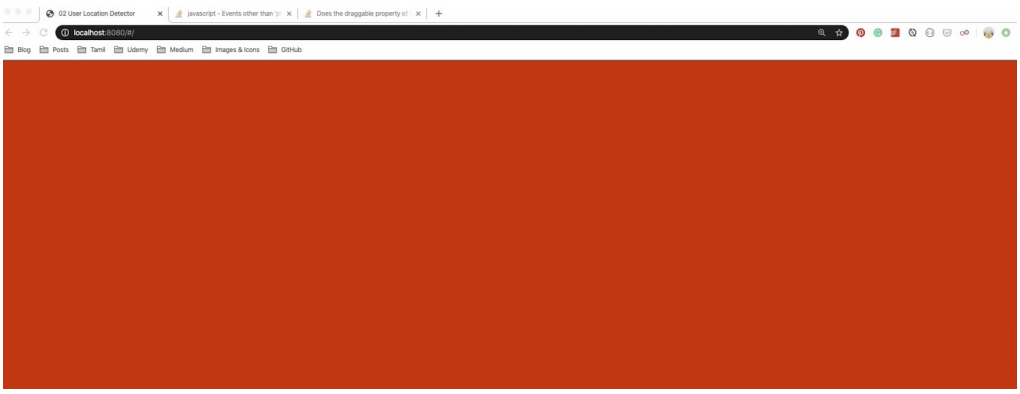
To do that, scroll down to the style element and define a new CSS rule called **#map** to target the map element:

```
#map {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  background: red;
}
```

Inside it, set the **position** property to **absolute** and set the **top**, **left**, **right** and **bottom** properties to 0, which will make sure the element covers the entire screen.

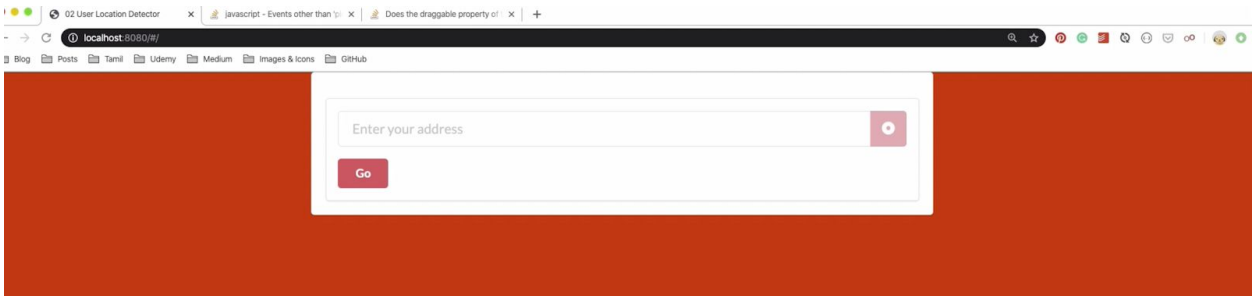
Let's also change the background colour to red to see if the HTML element is positioned correctly.

Nice! It does take the full width, but it now hides the form.



To bring the form to the front, set z-index value to -1.

```
#map {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  background: red,
  z-index: -1;
}
```



Nice!

Step #3: Show Google Maps On The View

Go inside the **locator ButtonPressed()** function, after the **getAddressFrom** function, and call this function:

```
this.showUserLocationOnTheMap()
```

We will declare it in a moment. First, pass the latitude and longitude values as arguments to it.

Then, go to the methods object and declare it with latitude and longitude parameters:

```
showUserLocationOnTheMap(latitude, longitude) {}
```

This is where we will create a map object.

Inside the function, define the map object and assign it to the variable called map.

This will take two arguments. The first one is an HTML element where the map is going to be added to.

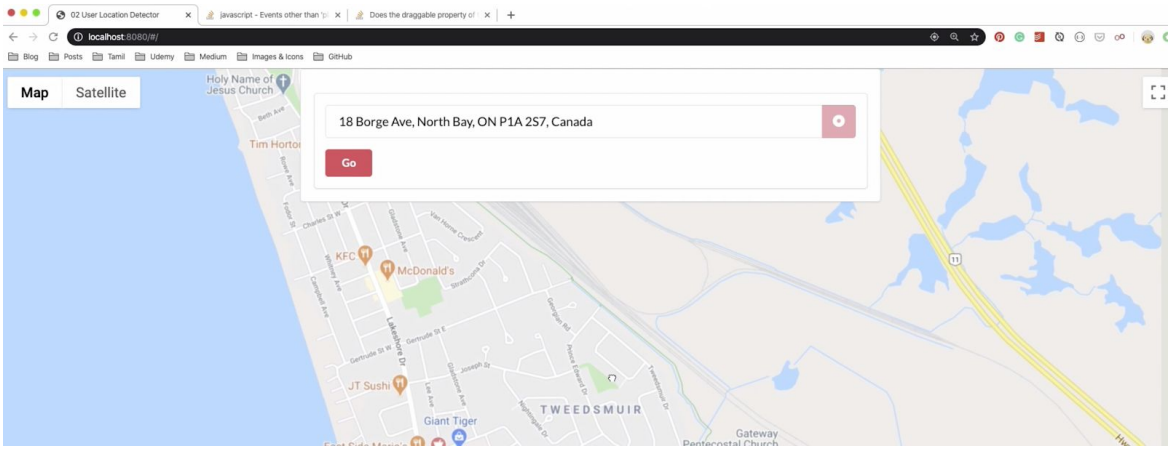
```
let map = new
google.maps.Map(document.getElementById("map"));
```

The second argument is a Javascript object in which we will add three properties to it.

```
let map = new google.maps.Map(document.getElementById("map"),
{
        zoom: 15,
        center: new google.maps.LatLng(latitude, longitude),
        mapTypeId: google.maps.MapTypeId.ROADMAP
});
```

name	**value**
zoom	15 This will determine how much the map will be zoomed in. The higher the number, the closer the place will be.
center	`new google.maps.LatLng(latitude, longitude)` It will take the user's coordinates as an argument of LatLng Object which will make sure that the map is centered based on the user's location.
mapTypeId	`google.maps.MapTypeId.ROADMAP` Type of map.

Let's try it!



And the map is working great!

Step #4: Place Marker On The Map Using Locator

The map is centered based on the user's current location, however, there is no marker indicating where the user's actual current location is.

Let's add the marker to it.

After the map object, define the marker object inside the **showUserLocationOnTheMap()** function.

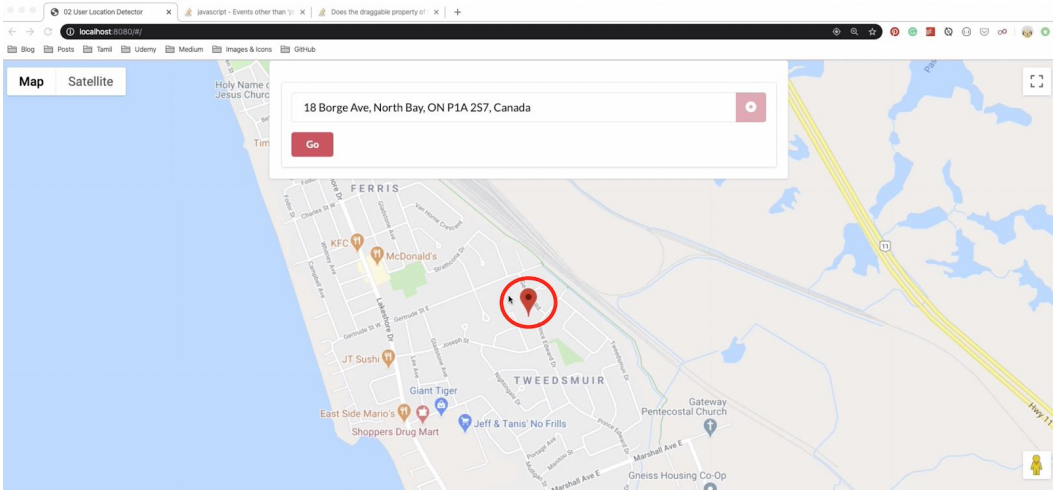
```
new google.maps.Marker({
    position: new google.maps.LatLng(latitude, longitude),
    map: map
});
```

This takes a Javascript object as an argument that has two properties in it.

The first parameter is the **Position** property which tells the marker where to be placed on the map. The value of it will be a LatLng object, passing the current user's coordinates as an argument.

The second parameter is the **Map** property, which tells the marker which map it should be placed into. Pass the map object that we declared above.

When we try it now using the locator button, the marker will be placed exactly where the user's current location is like in the image below.



Step #5: Place Marker On The Map Using Autocomplete

Next, let's show a location on the map when getting a user's address from the autocomplete drop down list.

First, attach an event listener called **place_changed** to the autocomplete object. This event will be triggered when a user selects a specific address from the autocomplete drop down list.

Go inside the **mounted()** function and give a name to the autocomplete object.

```
let autocomplete = new google.maps.places.Autocomplete(
      document.getElementById("map"),
      {
        bounds: new google.maps.LatLngBounds(
          new google.maps.LatLng(45.4215296, -75.6971931)
        )
      }
);
```

Then, attach a **place_changed** event listener to it.

```
autocomplete.addListener();
```

This method will take two parameters. The first parameter is the event name in quotes, "**place_changed**".

And, the second parameter is a callback arrow function.

```
autocomplete.addListener("place_changed", () => {
});
```

To get the coordinates of the selected place, invoke the **getPlace()** method on the autocomplete object and assign it to the variable called **place**.

```
autocomplete.addListener("place_changed", () => {
      let place = autocomplete.getPlace();
      console.log(place)
});
```

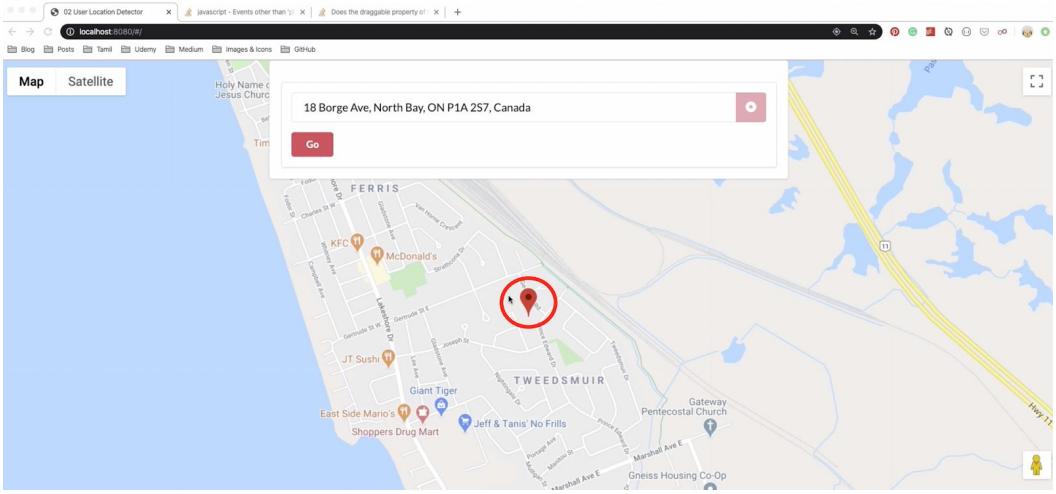
At this stage, all we need from the **place** object is the **latitude** and **longitude** values.

Then, we can simply call the **showUserLocationOnTheMap()** function and pass the latitude and longitude values to it.

```
autocomplete.addListener("place_changed", () => {
      let place = autocomplete.getPlace();
      this.showUserLocationOnTheMap(
        place.geometry.location.lat(),
        place.geometry.location.lng()
      );
});
```

And the rest will work automagically.

Let’s check it out by choosing an address from the autocomplete drop down list and the map will show the location like a charm!



****

Section 3: Build a Closeby App Using Places API

****

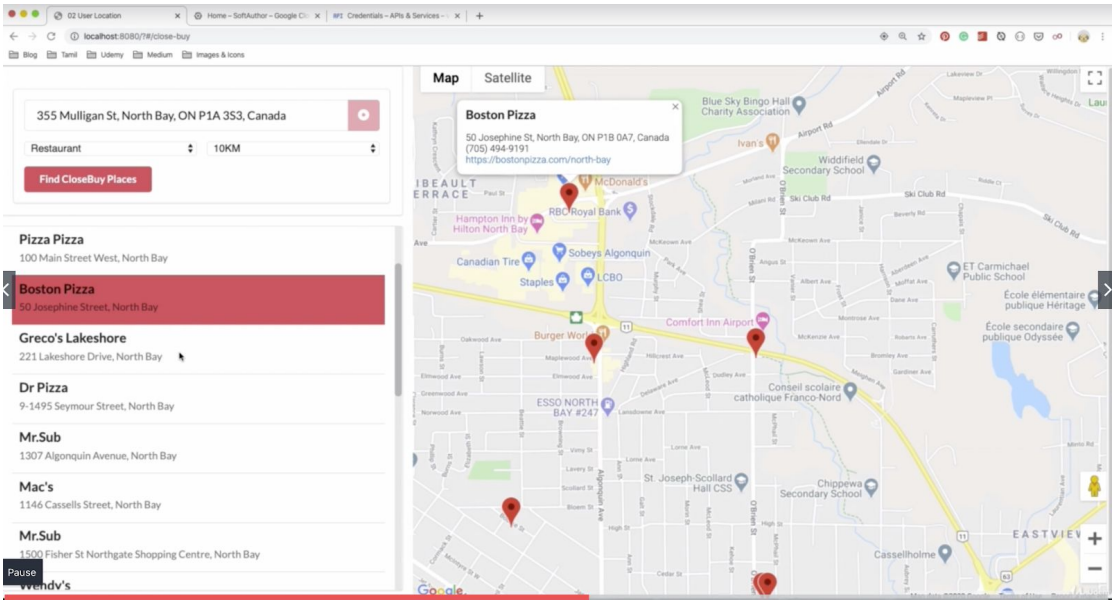
What You'll Be Building By The End Of This Section

We're going to be taking the knowledge that you have learned much further in this section, by building a more complex app called **CloseBuy** using Places API.

By the end of this section, you'll be able to:

- Build a two column grid (Semantic UI)
- Get Data from Nearby Search Request
- Fix CORS Error
- Show Places on the Google Maps
- Attach Event to a Marker and Show InfoWindow()
-

Here is the screenshot of the final app.



The CloseBuy app will get nearby restaurants based on the user's current location, type and the radius.

Then, users will be able to see a maximum of 20 restaurants in a list format on the left side, as well as a map on the right side with 20 markers on it representing each restaurant from the sidebar.

It also allows you to get more information about a restaurant by clicking the marker on the map as well as clicking the list item.

Sounds fun? Let’s get started!

What is Nearby Search Library?

By the end of this chapter, you'll be able to understand:

1. What Nearby Search Request is.
2. What the required Request Parameters are that we need to make an HTTP call.
3. What the output response object will look like.

Step #1: What is Nearby Search Request?

Nearby Search Request is a part of the Google Places Library in the Maps JavaScript API. It allows us to get different places based on:

- **location**: could be either the user's current location or any other location that you want to get nearby places from.
- **Type**: could be restaurants, bars, etc
- **radius**: determines how far you want to get the places from.

Let's take a look at the parameters that we need in order to get the places from a nearby search request.

Step #2: Required Request Parameters

The Nearby Search Request URL is pretty straight forward:

```
https://maps.googleapis.com/maps/api/place/nearbysearch/
```

It has a few required **route** and **query** parameters.

Required Route Parameter

name	description
Json or xml	This will determine what format the output response object will be. *(json recommended)*

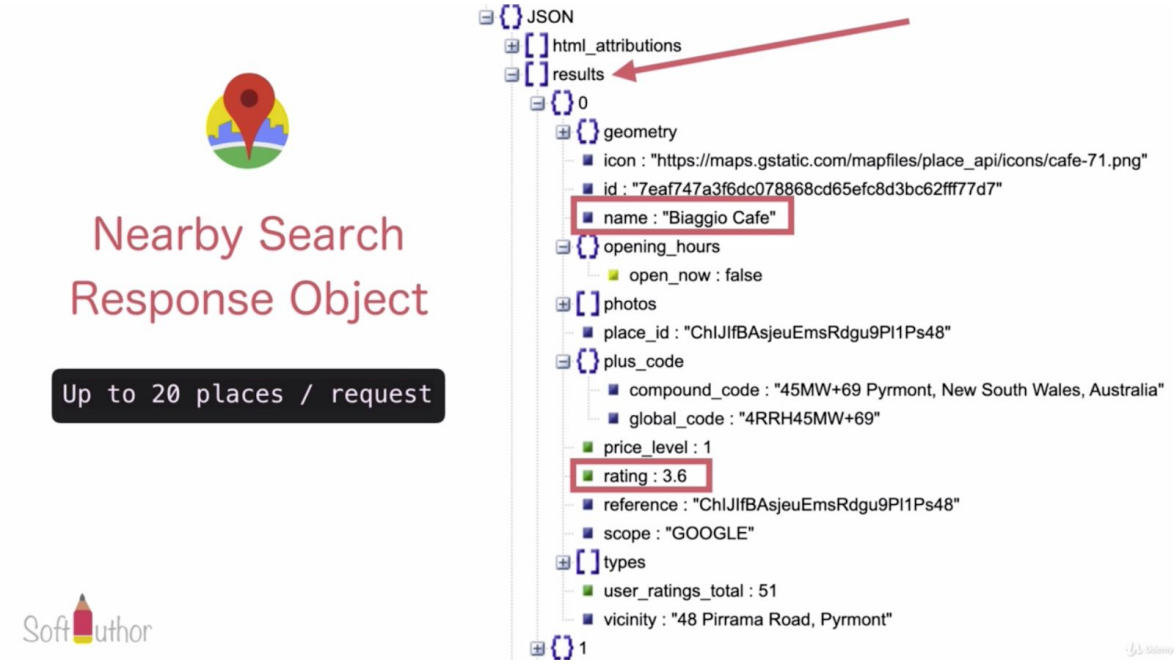
Required Query Parameters

There are four required Query Parameters. Let's take a look at them one by one.

name	description
key	The API key that we got from the Google Cloud Console.
location	This will be a user's current location in the form of latitude and longitude as a single value separated by a comma.
type	The value will be one from this list such as a **restaurant**, **school** and so on.
radius	It's a numeric value and it must be in meters. If your search range is **5KM**, the value will be **5000**.

Step #3: Output Response Object

Once the request is complete and everything goes well, we'll get an output in JSON format like in the screenshot below.



The results array will have a list of places and you can see each place has some information like name and rating etc.

The output results array will only have up to 20 places. If you want to see the next 20 places you will need to make a separate request.

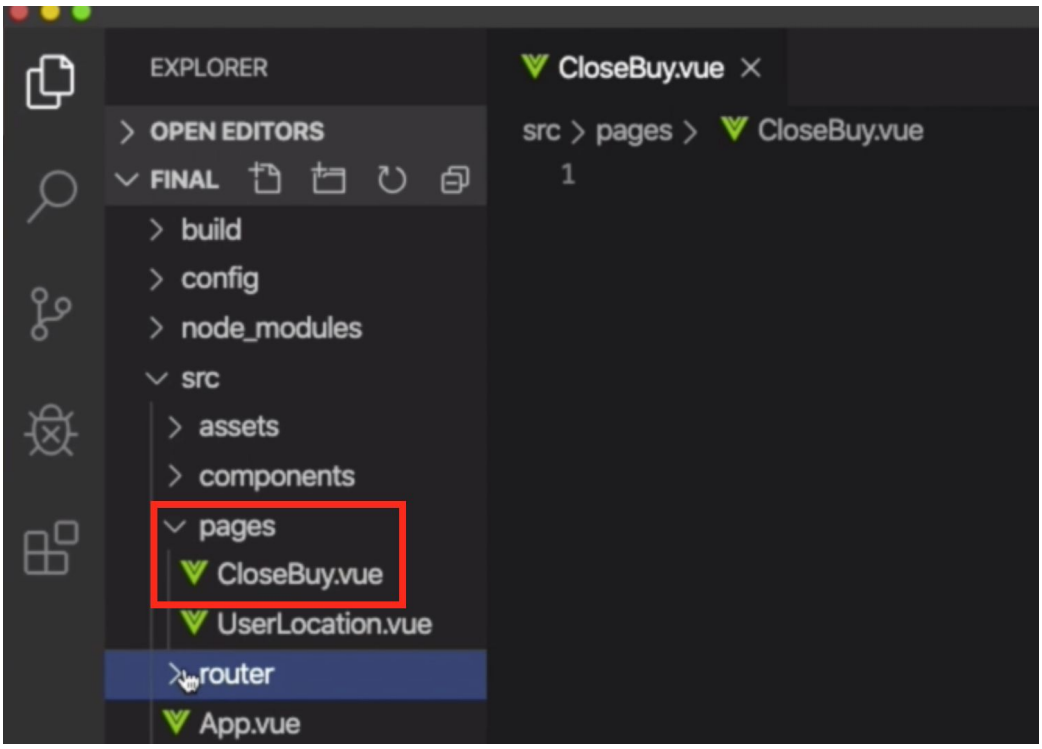
Create A CloseBuy Component

In this chapter, we are going to cover how to:

1. Create A “CloseBuy” App Component and Route for it.
2. Define a grid layout to match the final app.
3. Build a user input form.

Step #1: Create a CloseBuy Component and Route

Open up the project and create a component called closeBuy.vue inside the pages folder.



It is a page based component, so we need to create a route for it.

Go to the **index.js** file inside the router folder and import the CloseBuy component at the top.

```
import CloseBuy from '@/pages/CloseBuy'
```

Next, declare a route object inside the routes array:

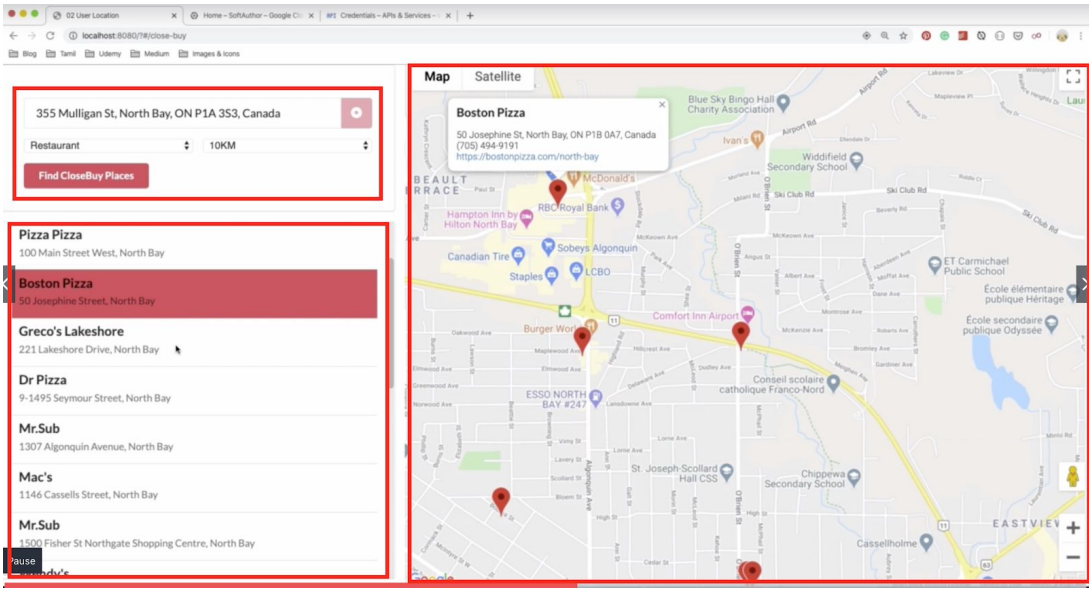
```
...
import CloseBuy from '@/pages/CloseBuy'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      component: UserLocation
    },
    {
      path:'/close-buy',
      component:CloseBuy
    }
  ]
})
```

Step #2: Add Grid Layout

As you can see in the final app screenshot, we need a grid layout with two columns in it.



The left column will have two divs, the top one will be the User Input Form and the bottom one will be a list of places. The right side column will be covered with the map.

Inside the template tags, define a grid using two classes: **ui** and **grid** and they are part of Semantic UI.

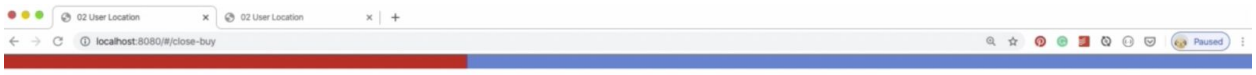
```
<template>
  <div class="ui grid">
    <div class="six wide column red"></div>
    <div class="ten wide column blue"></div>
  </div>
</template>
```

Inside the grid, create the first **div** with class names **six wide column** which is the left column and the second div is a ten wide column which is the right column.

The semantic grid is a 16 column layout to fill the screen, so the left div is six column wide and the right one is ten column wide.

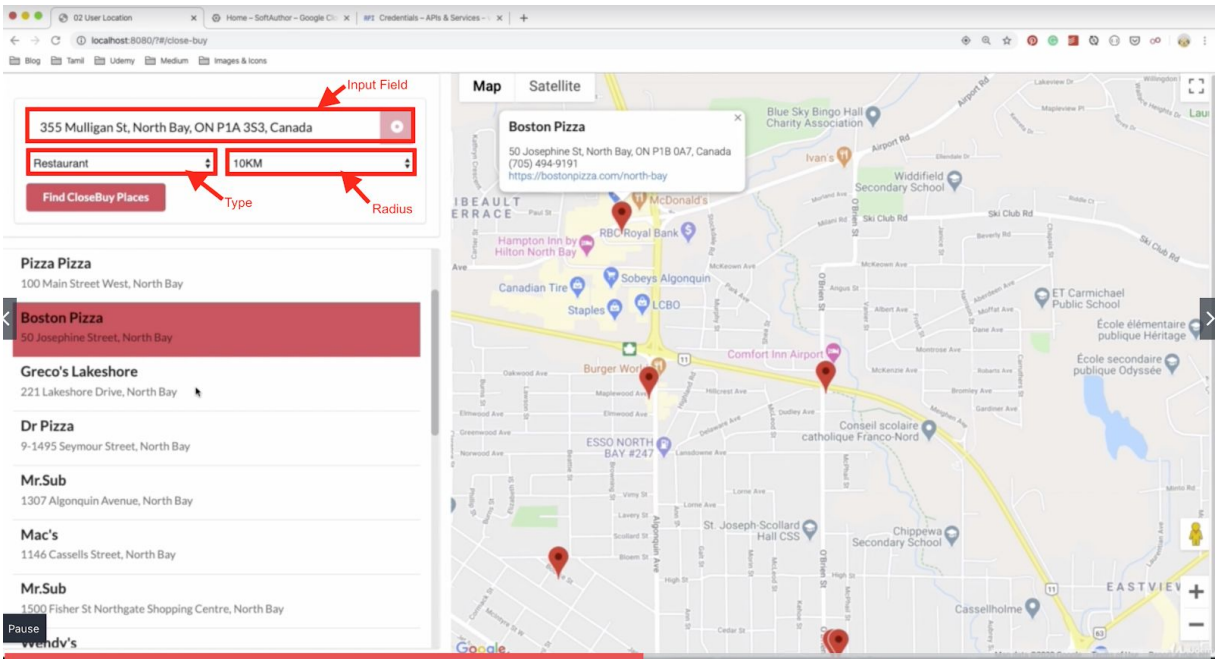
Add red class to the left column and blue class to the right column to see where they are on the browser.

You can see the columns are laid out as expected.



Step #3: Build User Input Form

As you can see, the user input form from the final app has three elements in it.



- The **input field** is where we can get the user's location.
- The place **type** can be a restaurant, bar etc, and it is a drop down menu.
- The **radius** on the right is also a drop down menu.

The first one is **user location** input and luckily we already covered how to get the User Location in the previous chapter.

Copy the script and style code from the User Location component and paste it into the CloseBuy component after the closing template tag.

If you're not sure about the code that I am talking about, you may need to complete the **Get User Location** chapter before continuing with this one.

The Script Code

```
<script>
import axios from "axios";
export default {
  data() {
    return {
      address: "",
      error: "",
      spinner: false
    };
  },

  mounted() {
    let autocomplete = new google.maps.places.Autocomplete(
      document.getElementById("map"),
      {
        bounds: new google.maps.LatLngBounds(
          new google.maps.LatLng(45.4215296, -75.6971931)
        )
      }
    );

    autocomplete.addListener("place_changed", () => {
      let place = autocomplete.getPlace();
      this.showUserLocationOnTheMap(
        place.geometry.location.lat(),
        place.geometry.location.lng()
      );
    });
  },

  methods: {
    locatorButtonPressed() {
```

```
      this.spinner = true;
      if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
          position => {
            this.getAddressFrom(
              position.coords.latitude,
              position.coords.longitude
            );
            this.showUserLocationOnTheMap(
              position.coords.latitude,
              position.coords.longitude
            );
          },
          error => {
            this.error =
              "Locator is unable to find your address. Please
type your address manually.";
            this.spinner = false;
          }
        );
      } else {
        this.error = error.message;
        this.spinner = false;
      }
    },
    getAddressFrom(lat, long) {
      axios
        .get(

"https://maps.googleapis.com/maps/api/geocode/json?latlng=" +
            lat +
            "," +
            long +
            "&key=[YOUR_API_KEY]"
```

```
        )
        .then(response => {
          if (response.data.error_message) {
            this.error = response.data.error_message;
            console.log(response.data.error_message);
          } else {
            this.address =
response.data.results[0].formatted_address;
          }
          this.spinner = false;
        })
        .catch(error => {
          this.error = error.message;
          this.spinner = false;
          console.log(error.message);
        });
    },
    showUserLocationOnTheMap(latitude, longitude) {
      let map = new
google.maps.Map(document.getElementById("map"), {
        zoom: 15,
        center: new google.maps.LatLng(latitude, longitude),
        mapTypeId: google.maps.MapTypeId.ROADMAP
      });

      new google.maps.Marker({
        position: new google.maps.LatLng(latitude,
longitude),
        map: map
      });
    }
  }
};
</script>
```

The Style Code

```
<style>
.ui.button,
.dot.circle.icon {
  background-color: #ff5a5f;
  color: white;
}

.pac-icon {
  display: none;
}

.pac-item {
  padding: 10px;
  font-size: 16px;
  cursor: pointer;
}

.pac-item:hover {
  background-color: #ececec;
}

.pac-item-query {
  font-size: 16px;
}

#map {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  background: red;
```

```
  z-index: -1;
}
</style>
```

Let's also grab the HTML code from the UserLocation component, but this time copy only the form element and everything underneath it and then paste it inside the left column.

```
<template>
    <div class="ui grid">
      <div class="six wide column red">
        <form class="ui segment large form">
          <div class="ui message red"
v-show="error">{{error}}</div>
          <div class="ui segment">
            <div class="field">
              <div class="ui right icon input large"
:class="{loading:spinner}">
                <input
                  type="text"
                  placeholder="Enter your address"
                  v-model="address"
                  ref="autocomplete"
                />
                <i class="dot circle link icon"
@click="locatorButtonPressed"></i>
              </div>
            </div>
            <button class="ui button">Go</button>
          </div>
        </form>
      </div>
```

```
      <div class="ten wide column blue"></div>
    </div>
  </template>
```

Switch back to the browser and you can see our User Input Form is nicely placed in the left column.

Perfect!

If you run the app, you may notice an error in the Google Chrome's developer console like below.

If you're getting an API error like this: Google Maps JavaScript API error: InvalidKeyMapError, make sure to replace your API Key inside the index.html file in the Google Places Library URL where it says **[YOUR_API_KEY]**.

Also, in the getAddressFrom() function, replace the **[YOUR_API_KEY]** placeholder text with your API Key.

Placing your API Key in those places will get rid of the error.

We will be using this API Key in a few places in the upcoming chapters, so I am going to create a property called apiKey in the data model object and set it's value to my actual API Key from the Google Cloud Console.

```
data() {
    return {
      ...,
      apiKey: [YOUR_API_KEY]
    };
},
```

Then, go down to the getAddressFrom() function, and simply delete the API Key placeholder text. After the end quote, add the **+** sign, then type **this.apiKey**.

```
getAddressFrom(lat, long) {
      axios
        .get(
"https://maps.googleapis.com/maps/api/geocode/json?latlng=" +
            lat +
            "," +
            long +
            "&key=" +
            this.apiKey
        )
        ...
}
```

Now, let's add HTML code for the type and radius drop down menus.

After the first div element with the class field, add another div element with field class. In there, create another div element with the classes **two** and

fields. These classes will make sure that the type and radius elements are laid out side by side.

```
<div class="field">
    <div class="two fields">
      <div class="field">
        <select>
          <option value="restaurant">Restaurant</option>
        </select>
      </div>

      <div class="field">
        <select>
          <option value="5">5 KM</option>
          <option value="10">10 KM</option>
          <option value="15">15 KM</option>
          <option value="20">20 KM</option>
        </select>
      </div>
    </div>
</div>
```

Then, define a div element for type with the class field. In there, create a select element with one option inside it. The option text will be **restaurant** and its value will also be restaurant. The value must be one of the type names from the Googles type list that you saw in the previous chapter.

In the second div with the field class for radius, define a select input element with four options: 5km, 10km, 15km and 20km.

Finally, change the button text to “Find CloseBuy Places” and attach a click event to it.

```
<button class="ui button"
@click="findCloseBuyButtonPressed">Find CloseBuy
Places</button>
```

Go ahead and declare this callback function inside the methods object.

```
findCloseBuyButtonPressed() {
}
```

This is where we will make an HTTP request to get nearby places. Let’s get all the required parameters from a user before we do that.

The first one is the location, as you know, it’s value must be in latitude and longitude.

To get those values, declare two properties inside the data() model.

```
 data() {
    return {
      ...
      apiKey: [YOUR_API_KEY],
      lat: 0,
      lng: 0,
    };
  },
```

Then, assign user location coordinate values to these properties inside the **getCurrentPosition()** method so that we can access them outside this function.

```
navigator.geolocation.getCurrentPosition(
          position => {
            this.lat = position.coords.latitude;
            this.lng = position.coords.longitude;
            this.getAddressFrom(
              position.coords.latitude,
              position.coords.longitude
            );
          },
          ...
)
```

Let's create two more properties inside the data() model as well.

```
data() {
    return {
      ...
      lat: 0,
      lng: 0,
      type: "",
      radius: "",
    };
  },
```

Then, bind these two properties to appropriate select tags using the v-model directive.

```
 <select v-model="type">
    <option value="restaurant">Restaurant</option>
  </select>

  <select v-model="radius">
    <option value="5">5 KM</option>
```

```
    <option value="10">10 KM</option>
    <option value="15">15 KM</option>
    <option value="20">20 KM</option>
  </select>
```

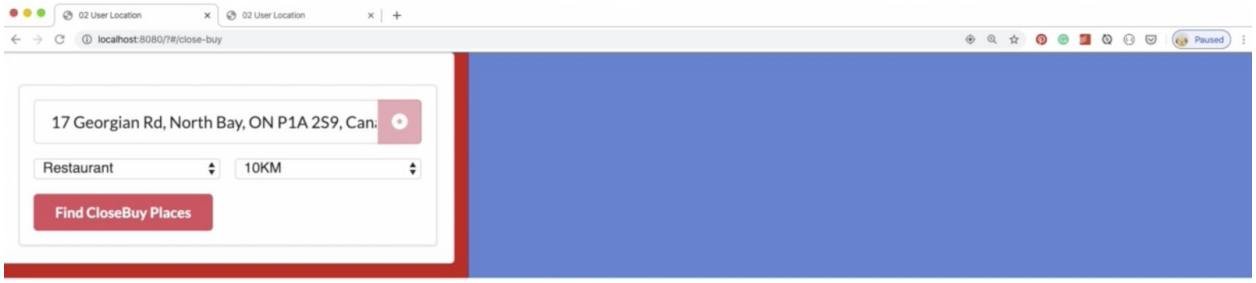
When a user selects an option from these drop down menus, type and radius properties will be set with selected values.

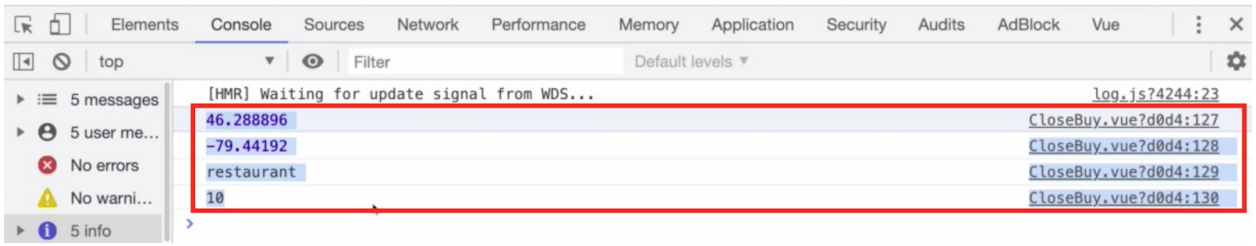
Now, console log all the four properties to make sure everything has correct values.

Inside the **findCloseBuyButtonPressed** function console.log():

```
findCloseBuyButtonPressed(){
   console.log(this.lat);
   console.log(this.lng);
   console.log(this.type);
   console.log(this.radius);
}
```

Let’s test it out!


02 User Location
02 User Location
localhost:8080/?#/close-buy
Paused
17 Georgian Rd, North Bay, ON P1A 2S9, Can
Restaurant
10KM
Find CloseBuy Places


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
AdBlock
Vue
top
Filter
Default levels
5 messages
5 user me...
No errors
No warni...
5 info
[HMR] Waiting for update signal from WDS...
log.js?4244:23
46.288896
CloseBuy.vue?d0d4:127
-79.44192
CloseBuy.vue?d0d4:128
restaurant
CloseBuy.vue?d0d4:129
10
CloseBuy.vue?d0d4:130

And it works!

Make An HTTP Request To Nearby Search

By the end of this chapter, you'll be able to:

1. Compose Nearby Search Request URL
2. Understand why CORS Error Occurs
3. Know what CORS is
4. Fix CORS Error

Step #1: Compose Nearby Search Request URL

Let's make an HTTP call to Nearby Search Request.

Go inside the **findClosebuybuttonPressed()** function and delete the console log messages that we created earlier..

Then, create a constant called URL and set it's value to the nearby search request base url inside the quotes like so:

```
 const URL =
"https://maps.googleapis.com/maps/api/place/nearbysearch";
```

Rather than using double quotes, change it to a backtick (`) which can be found at the top left side of the keyboard.

The reason we use the back tic is because we can do string interpolation. This means we can add any property in between the URL string using the **${propertyname}** format instead of using concatenation with the **+** operator.

```
 const URL =
`https://maps.googleapis.com/maps/api/place/nearbysearch`;
```

Now, let's add all the required parameters inside this URL string using string interpolation.

The first one is the required route param which is either json or xml.

At the end of the URL add: **/json**

As I mentioned earlier, this is the output format of the response object.

```
 const URL =
`https://maps.googleapis.com/maps/api/place/nearbysearch/json
`;
```

Then, add a question (?) mark to include query parameters.

The first one is location and it's value is latitude and longitude separated by a comma.

```
const URL =
`https://maps.googleapis.com/maps/api/place/nearbysearch/json
?location=${this.lat},{$this.lng}`;
```

Then, add type and radius query parameters using the and (&) sign to separate them.

```
const URL =
`https://maps.googleapis.com/maps/api/place/nearbysearch/json
?location=${this.lat},{$this.lng}&type=${this.type}&radius=${
this.radius * 1000}`;
```

As you can see, I have multiplied the radius value by 1000 to convert it from KM to meters.

Finally, add the API Key at the end.

```
const URL =
`https://maps.googleapis.com/maps/api/place/nearbysearch/json
?location=${this.lat},{$this.lng}&type=${this.type}&radius=${
this.radius * 1000}&key=${this.apiKey}`;
```

Once the URL string is composed, call the **get()** method on the **axios** object and pass the URL as an argument. This will return a promise.

The **then()** method will be called when the promise is resolved and the returned response object will be returned to the **response** parameter.

```
axios
      .get(URL)
      .then(response => {
        console.log(response);
      })
      .catch(error => {
        this.error = error.message;
      });
```

To catch any error, use the **catch** method which will be called when the promise is rejected. The error object will be returned to the error parameter.

Set the error message to the error property.

Switch back to the browser and get the user location by clicking the location button. Choose the place type and the radius and then hit the **Find CloseBuy Places** button.

There is an error in the browser console. 😕

It reads:

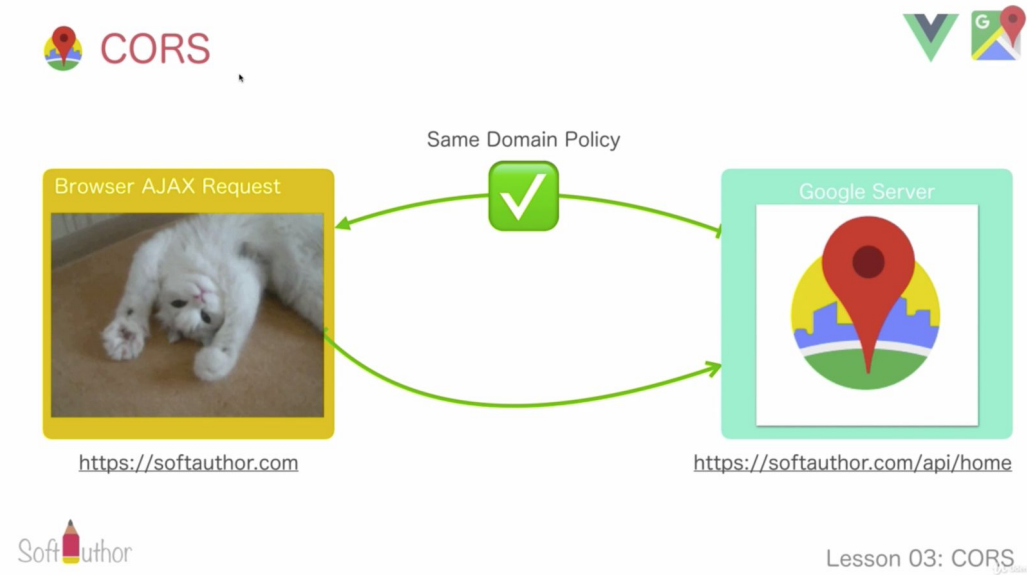
Access to XMLHttpRequest at 'https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=46.288896,-79.44192&type=restaurant&radius=10000&key=AIzaSyCM5Nz9ujxoPOL_R0aQAPxPlDHdurKYjxU' from origin 'http://localhost:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

This is called a CORS (Cross Origin Resource Sharing) error.

What this means is, the browser is blocking the response from the google server saying that the response object does not have an appropriate header to let the browser be okay with getting data from a different domain.

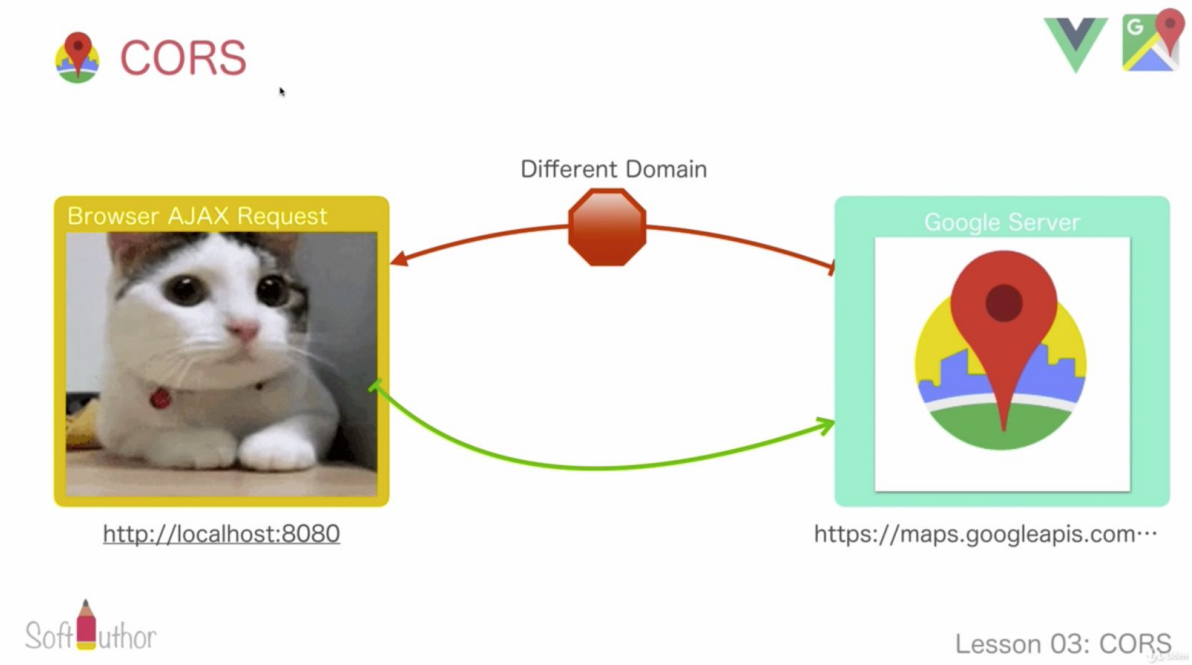
Step #2: Why CORS Error Occur?

By default, browsers will only allow communication between client and server (to share data) as long as they are from the same domain. This is called **Same Domain Policy** and the browser will not give any error for that.



As you can see, the client is softauthor.com and the server is also the softauthor.com domain, so sharing data between them will work just fine.

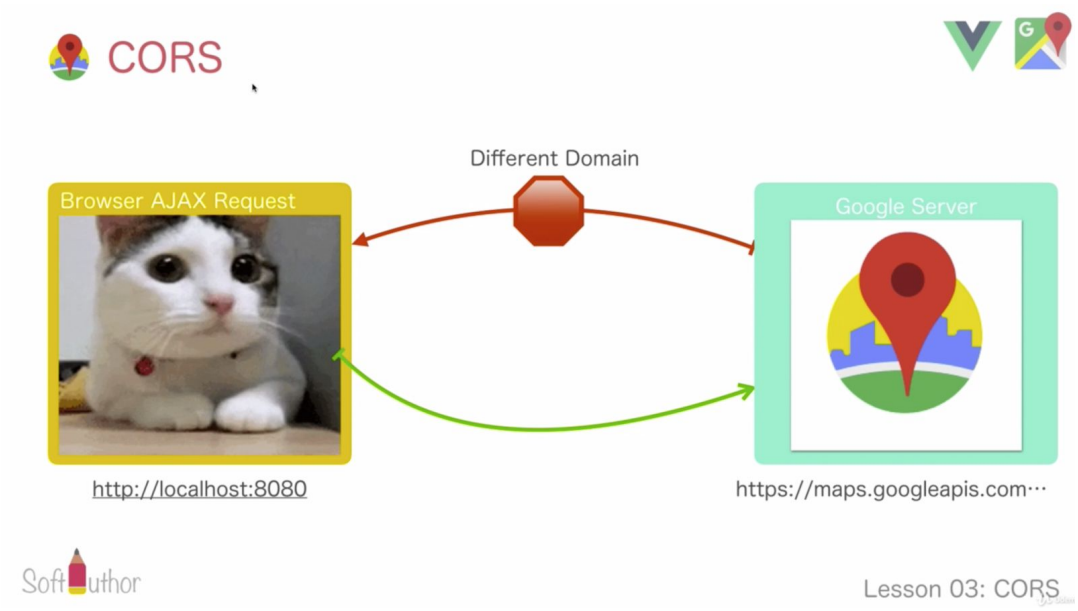
On the other hand, the browser won't allow getting a response from a different domain unless the response object has CORS enabled.



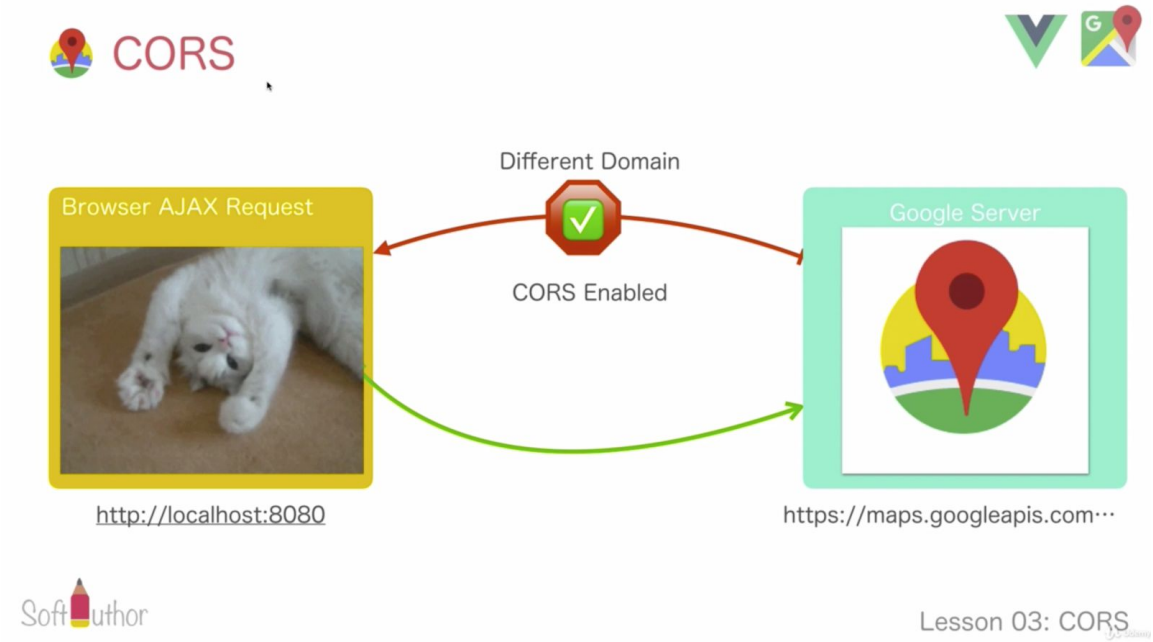
Step #3: What is CORS ?

CORS stands for **Cross Origin Resource Sharing** and it is a security mechanism that will safely allow any website to access resources from a different domain.

As you can see, the first client server communication will fail because CORS is not enabled on the server to let the browser know that it's okay to not block it.



If CORS is enabled everything will be fine.



Now we ask, how do we enable CORS?

Step #4: Fix CORS Error

To enable CORS, all we have to do is to add **“Access Control Allow Origin**” in the header and set it’s value to our domain in the response object on the server.

However, we do not have access to make changes to the google server and it does not have CORS enabled.

The alternative option is to make a request from our server rather than a client.

To get this working faster, we can use something called a **third party proxy server**.

To do this, get the proxy server URL link and add it in front of our nearby search request URL:

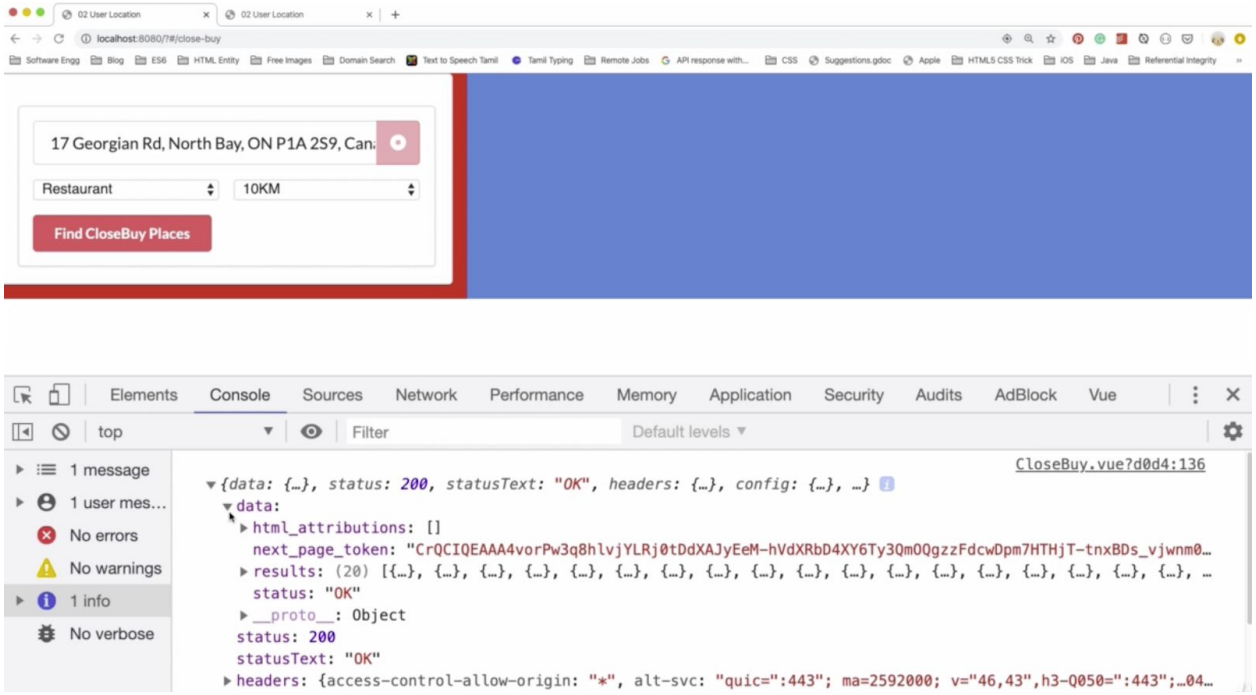
```
https://cors-anywhere.herokuapp.com/
```

This will actually make a request on the server and give back the results to the client. We do not have to do anything apart from adding the proxy server URL in front of our base URL.

Go to the URL link and add the proxy server link, save and then run the application.

```
const URL =
`https://cors-anywhere.herokuapp.com/https://maps.googleapis.
com/maps/api/place/nearbysearch/json?location=${this.lat},{$t
his.lng}&type=${this.type}&radius=${this.radius *
1000}&key=${this.apiKey}`;
```

Yay! We have a response back from the server with a list of places.



Get Nearby Places Data When Using Autocomplete

Previously, I showed you how to get nearby places when a user clicks the locator button - but not by using autocomplete.

In this chapter, we will get nearby places using Autocomplete in four steps:

1. Defining Autocomplete and Assigning Object to it
2. Attaching a place_changed Event
3. Invoking getPlace()
4. Reset the Address

Replacing the following code inside the **mounted()** will get nearby places data when a user types an address manually using autocomplete.

Step #1: Define a variable called **autocomplete** and assign an **autocomplete** object to it.

Step #2: Attach a **place_changed** event to the **autocomplete** object.

This will be triggered every time a user picks an address from the autocomplete drop-down list.

```
mounted() {
  const autocomplete = new google.maps.places.Autocomplete(
    this.$refs["autocomplete"],
    {
      bounds: new google.maps.LatLngBounds(
        new google.maps.LatLng(45.4215296, -75.6971931)
      )
```

```
    }
  );

  autocomplete.addListener("place_changed", () => {
    const place = autocomplete.getPlace();
    this.address = place.formatted_address;
    this.lat = place.geometry.location.lat();
    this.lng = place.geometry.location.lng();
  });
},
```

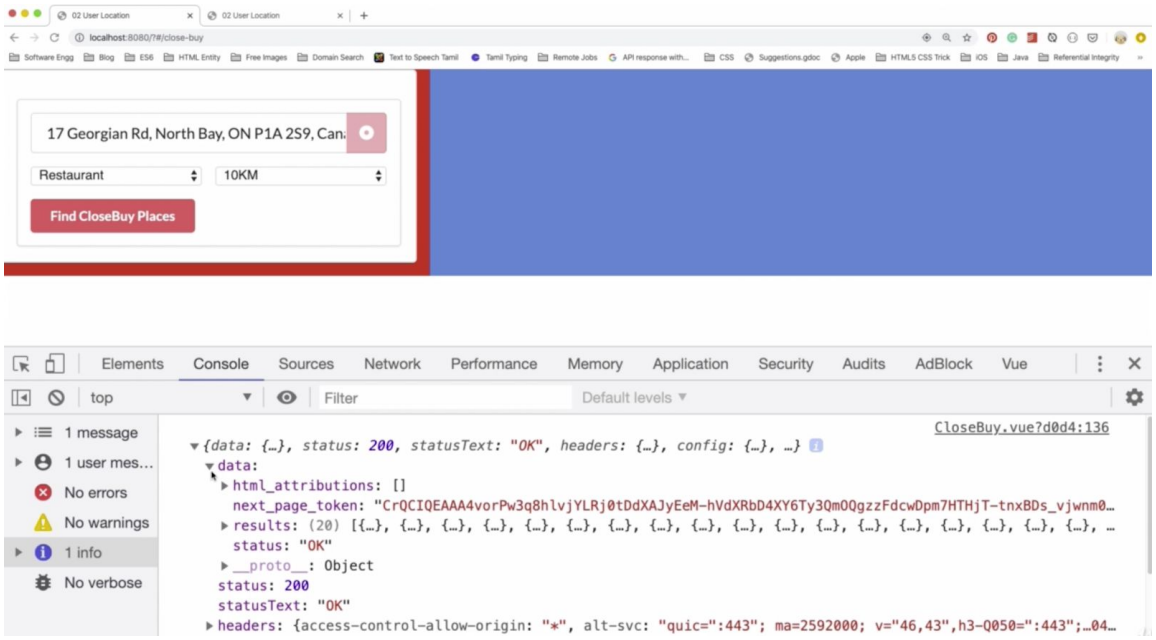
Step #3: Invoke getPlace()

Get the selected place object by invoking **getPlace()** on the autocomplete object inside the callback arrow function.

Step #4: Reset the Address

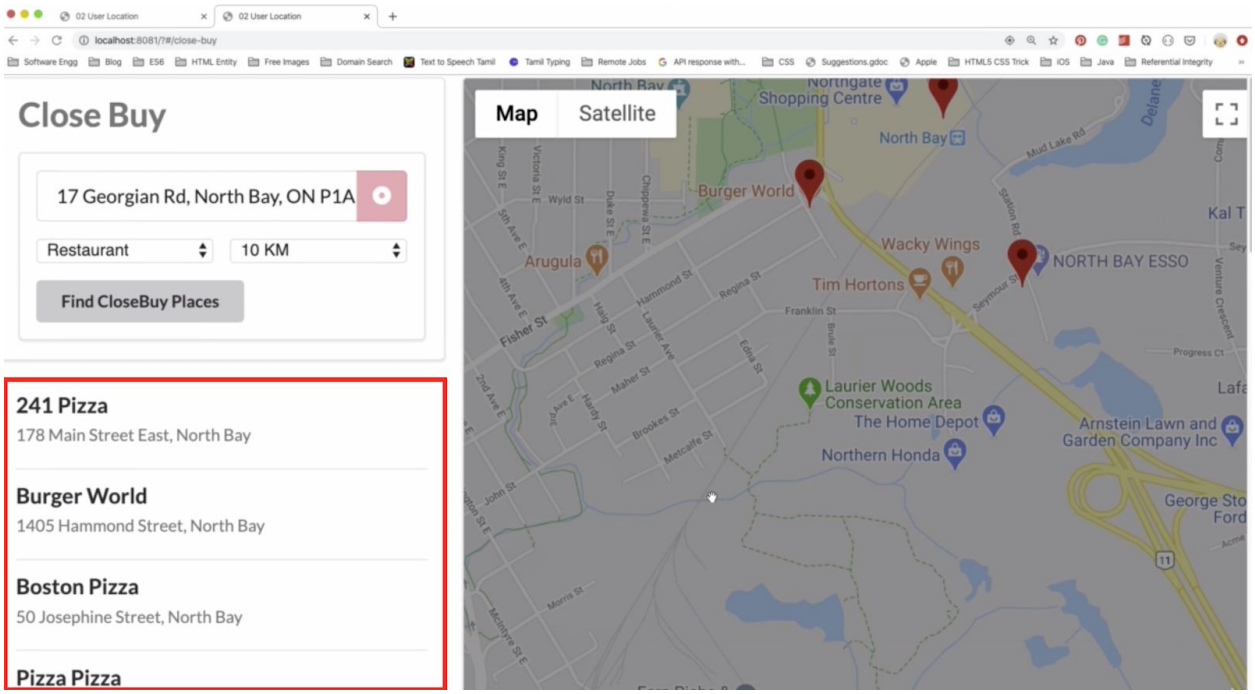
Finally, reset the address, lat, and lng properties with new values.

That should work!


02 User Location
localhost:8080/?#/close-buy
17 Georgian Rd, North Bay, ON P1A 2S9, Can
Restaurant
10KM
Find CloseBuy Places
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
AdBlock
Vue
top
Filter
Default levels
1 message
1 user mes...
No errors
No warnings
1 info
No verbose
CloseBuy.vue?d0d4:136
{data: {…}, status: 200, statusText: "OK", headers: {…}, config: {…}, …}
data:
html_attributions: []
next_page_token: "CrQCIQEAAA4vorPw3q8hlvjYLRj0tDdXAJyEeM-hVdXRbD4XY6Ty3QmOQgzzFdcwDpm7HTHjT-tnxBDs_vjwnm0…
results: (20) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, …
status: "OK"
__proto__: Object
status: 200
statusText: "OK"
headers: {access-control-allow-origin: "*", alt-svc: "quic=":443"; ma=2592000; v="46,43",h3-Q050=":443";…04…

Display Places Data on the View

Let’s learn how to show places data in a list format on the left column under the user input form.

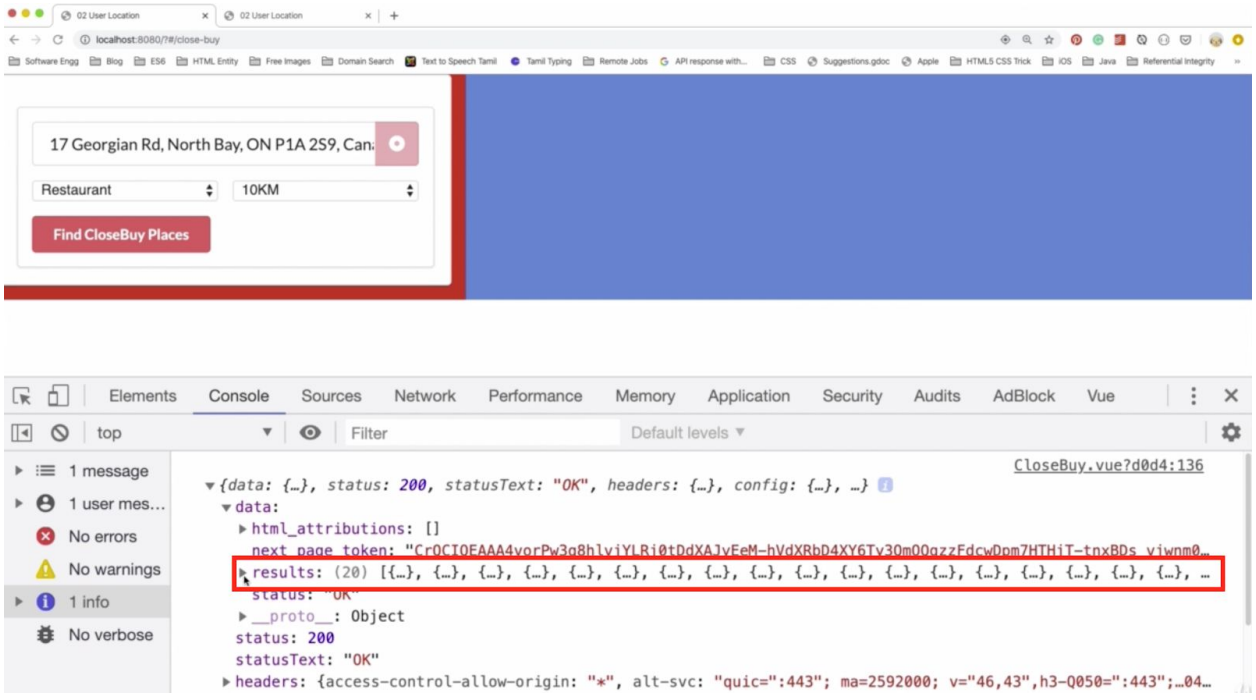


Create a property called **places** inside the data() model and set it’s initial value to an empty array.

This will hold all our places data that is coming from the Nearby Search Request.

```
data() {
    return {
     ...
      type: "",
      radius: "",
      places: [],
    };
  },
```

As you can see, the response object has more information than what we need. We can find places data inside the results array which is a property inside the data object.



Now, go to the **findCloseBuyButtonPressed()** function, inside the **then()** method, assign the results array to the places array.

```
axios
        .get(URL)
        .then(response => {
          this.places = response.data.results;
          console.log(response);
        })
        .catch(error => {
          this.error = error.message;
        });
```

Let's show the data on the view using the places array.

Create a single list item HTML markup and loop through the places array using **v-for** directive.

After the ending **</form>** tag, create a div element with two semantic ui classes — **ui segment** which will create a box with a nice thin rounded corner border.

```
<div class="ui grid">
    ...
    </form>
    <div class="ui segment">

    </div>
    <div class="ten wide column blue" id="map"></div>
</div>
```

Then, create another div inside it with a few classes — **ui divided items**, which will add borders between each list item.

Finally, define another div with a class name **item** which is where we're going to add a loop.

```
<div class="ui segment">
    <div class="ui divided items">
      <div class="item">
        <div class="content">
          <div class="header">name</div>
          <div class="meta">address</div>
        </div>
      </div>
    </div>
</div>
```

Inside that, define a div with a class called **content**. Then, create two divs: one for **name** with a header class and another one for **address** with the class name **meta**.

They are semantic ui classes to get the nice list item view.

Let's loop through the places array using **v-for** in the div element that has item class in it.

```
<div class="item" v-for="place in places">
   ...
</div>
```

The **place** variable, which is before the in keyword, will hold the place object on each iteration.

Then,the **in** keyword and **places,** which is the **array of places object** that we declared in the data() model below.

You may get an error inside the code editor at this stage that will say:

Elements in iteration expect to have 'v-bind:key'

Let's add the **:key** to the element to get the error to go away.

```
<div class="item" v-for="place in places" :key="place.id">
   ...
</div>
```

I could use **v-bind:key** or the shortcut is :key equals then double quotes.

The value of the key must be unique. We could use an index value but this time, let's use an **id** property which we can find inside each place object.

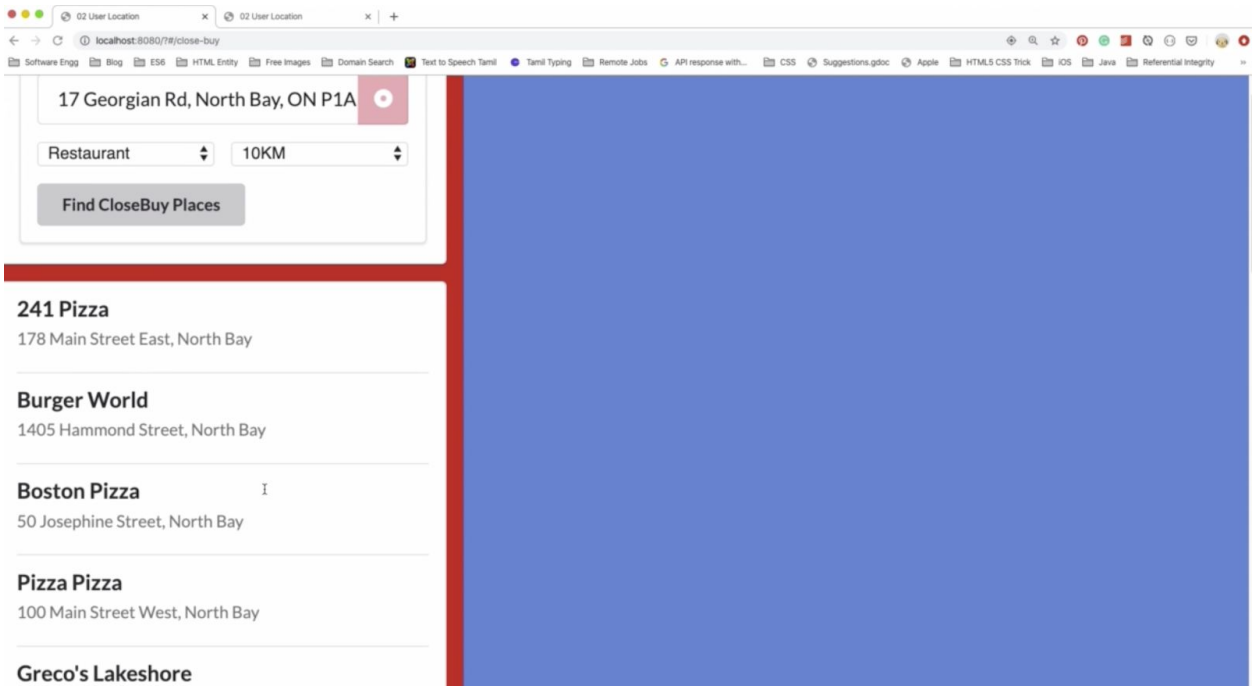
If we miss binding the key on the element where we loop through the array, it will give an error and it won't work.

The place object has a lot of properties. What we want is the name of the place which we can get from the **name** property and address of the place which is the **vicinity** property.

Let's render those values on the view using double curly braces.

```
<div class="ui divided items">
    <div class="item" v-for="place in places"
:key="place.id">
      <div class="content">
        <div class="header">{{place.name}}</div>
        <div class="meta">{{place.vicinity}}</div>
      </div>
    </div>
</div>
```

That's it! Let's run the application to see if it's working.



And it looks great!

Show Places Data on the Maps View

In this chapter, we are going to cover how to:

1. Instantiate Google Maps map object
2. Add Markers

Time to show all the places on the Google Maps using markers in the right column.

Step #1: Instantiate Google Maps map object.

Declare a function called **showPlacesOnMap()** inside the method object.

```
methods: {
    ...,
  showPlacesOnMap()  {
  }
}
```

This is where we will be creating a map object and adding markers to it.

We only want to show the map once we get data from the Places API, so right after setting the response object to the places array, call the **showPlacesOnMap()** function.

```
axios
        .get(URL)
        .then(response => {
          this.places = response.data.results;
          this.showPlacesOnMap();
        })
        .catch(error => {
          this.error = error.message;
        });
```

Let's create an instance of Google Maps map object inside the **showPlacesOnMap()** function.

Define a variable called map which will hold the map object. Then, pass a couple of parameters to it.

The first parameter will be the **DOM** element that we want to add the map into.

```
const map = new google.maps.Map();
```

Add a reference to the right column in the template. This time, add an attribute called ref="map" instead of id.

```
<div class="ten wide column" ref="map"></div>
```

The **ref** attribute is vue specific and it's more efficient than using the traditional **id** attribute.

Now that we have a reference to the right column, let's add the DOM element as a first parameter into the map constructor using **this.$refs**, which will be an object with all the **ref** attributes in this component.

What we want to choose is a **map** key from the object.

The second parameter is a Javascript object that has a few properties.

```
const map = new google.maps.Map(this.$refs["map"], {
  zoom: 15,
  center: new google.maps.LatLng(this.lat, this.lng),
  mapTypeId: google.maps.MapTypeId.ROADMAP
});
```

The first property is **zoom** level. Set it's value to 15 (I find that is a sweet spot but we can adjust the number - bumping it up makes the map view get closer to the places and vice versa).

The second property is **center**. This is where we center the map based on the location, which will be the user's current location.

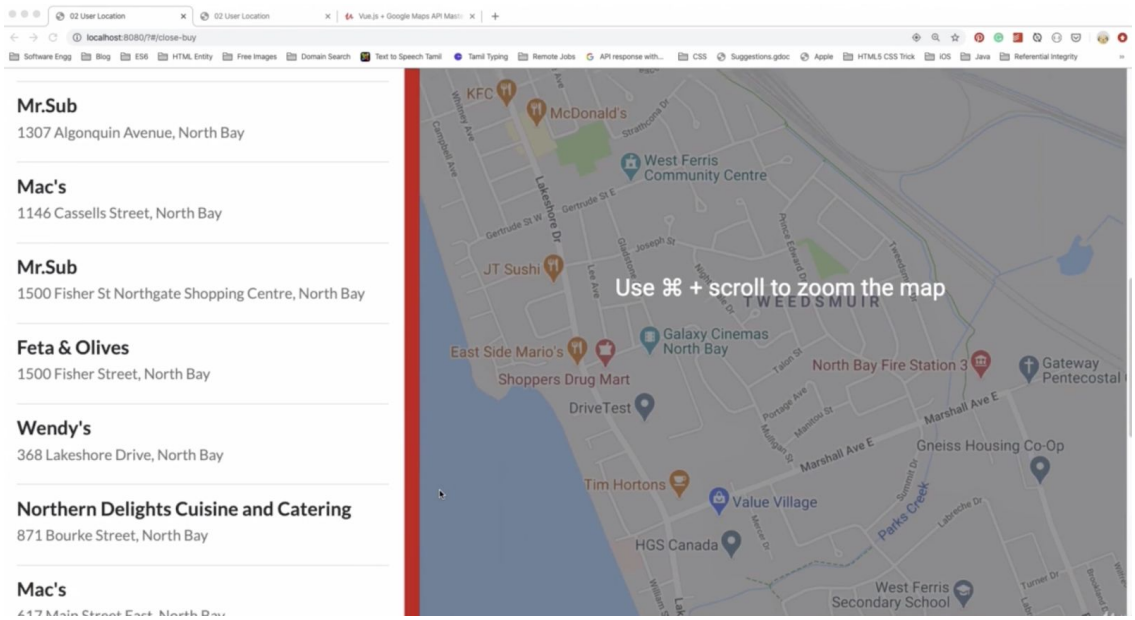
The value of the center property must be a LatLng Object.

Instantiate LatLng object and pass the users coordinates to it as an argument separated by a comma.

The third property is **mapTypeId** which will determine what type of map we want to use, in my case ROADMAP.

That's it.

Let's see if the map is visible as well as centered based on our current location.



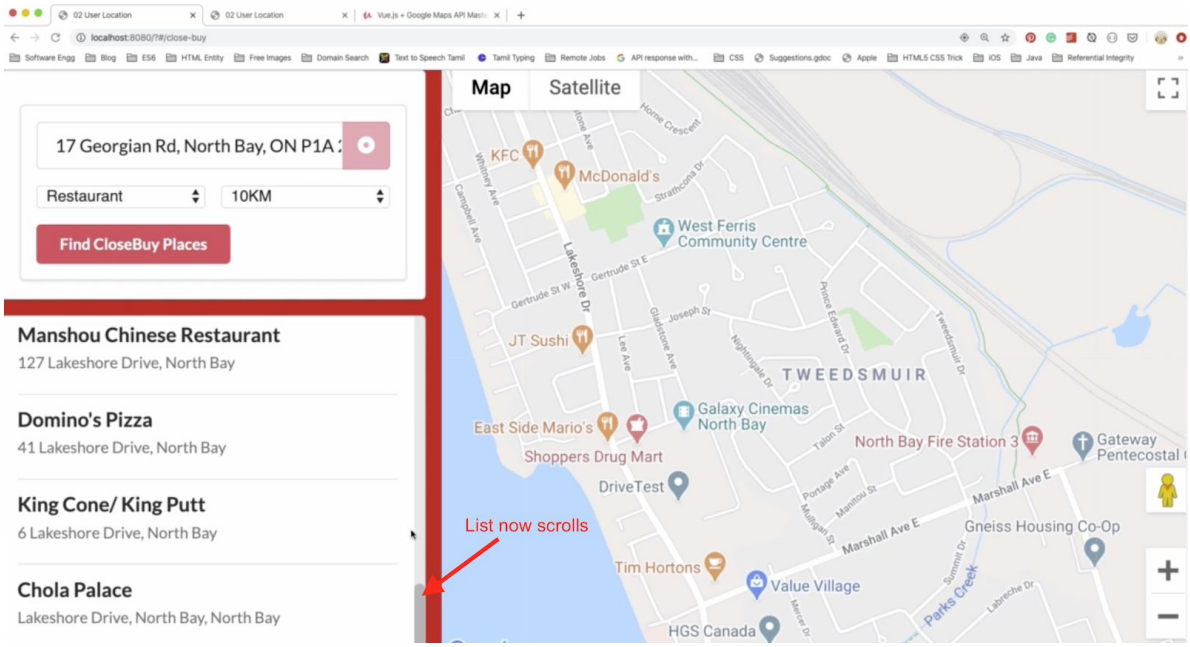
The map is working nice, but it is stretched vertically based on the places data in the left column.

So give a fixed height to the places list item container so that the left column as well as the right map column fit into the browser's screen view.

In the div with ui segment class, add a couple of inline styles which are max-height and overflow auto.

```
<div class="ui segment"
style="max-height:500px;overflow:auto;">
        <div class="ui divided items">
          <div
            class="item"
            v-for="place in places"
            :key="place.id"
          ...
 </div>
```

Let's try it again.



The list is scrollable, the map is not stretched vertically anymore and it's also centered nicely based on our current location.

Step #2: Add Markers

Let's add markers to the map using the **places** array. Loop through the **places** array after the map object inside **showPlacesOnMap()** function and get coordinates of a place on each iteration.

```
for (let i = 0; i < this.places.length; i++) {
    const lat = this.places[i].geometry.location.lat;
    const lng = this.places[i].geometry.location.lat;
}
```

Once we have the coordinates, create a marker with those coordinate values and add it to the map.

So, instantiate the marker object inside the loop and assign it to the marker variable.

```
const marker = new google.maps.Marker()
```

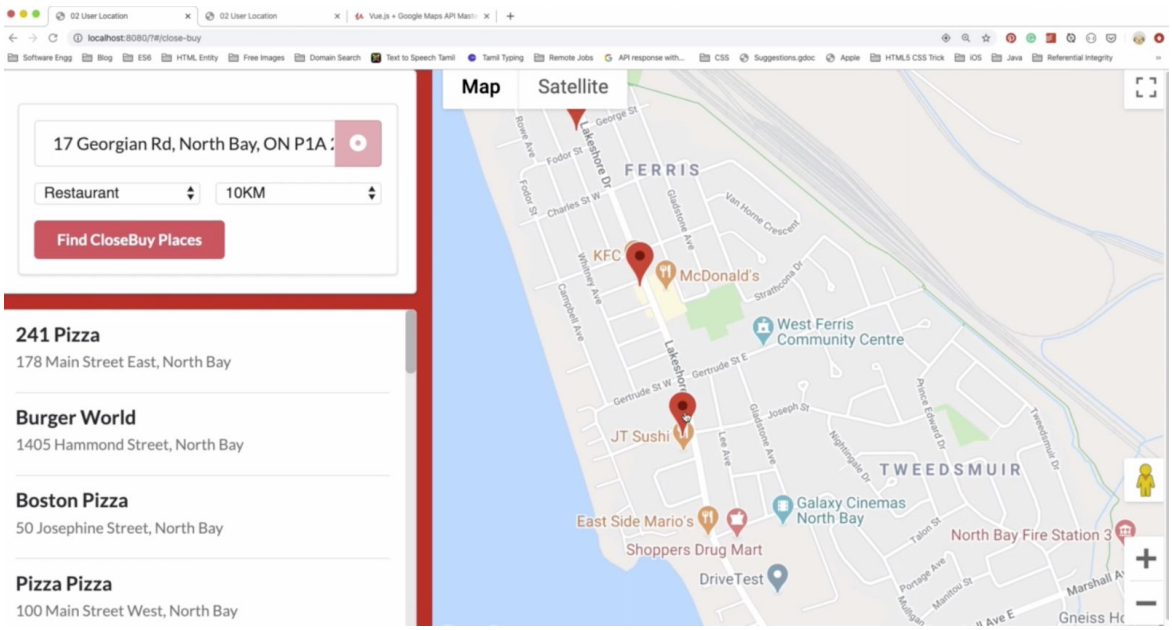
Then, add a Javascript object as a parameter to the marker object.

The first property in the parameter object is **position**, which is where we tell the marker where it needs to be positioned based on the coordinate values.

```
const marker = new google.maps.Marker({
    position: new google.maps.LatLng(lat, lng),
    map: map
});
```

The second property is **map** which will tell the marker which map it needs to be added onto.

Switch back the browser and we now have markers on the map!



We are now showing the places data in a list view and also on the map using markers.

It’s kind of confusing to see which place is at which marker. It would be nice to show a popup window *(InfoWindow)* about the place when we click on the marker.

Show Tooltip (Callout) on the Marker

Let's see how to display a popup window aka info window with some information about the selected place when a marker is clicked.

In this chapter, we are going to cover how to:

1. Attach a Click Event to Markers
2. Instantiate the Info Window

Step #1: Attach Click Event To Markers

Invoke the **addListener()** method on the google events object inside the **for** loop and after the marker object.

```
google.maps.event.addListener()
```

Then, pass three arguments to it.

```
 google.maps.event.addListener(marker, "click", function()
{})
```

The first argument is the object that we want to add a click event to, in this case, the **marker** object that I have declared above.

The second argument is the name of the event - **click** in quotes.

The final argument is the **anonymous callback arrow function** that will be fired when the event occurs.

Step #2: Instantiate the Info Window

Define an info window object above the for loop, which is the actual popup window that ships with Google Library.

```
const infowindow = new google.maps.InfoWindow();
```

Inside the marker's click event callback function, add some content to the **infoWindow** object using the **setContent()** method.

This method will take text or HTML content as an argument. Add some dummy HTML content for now using the back tic.

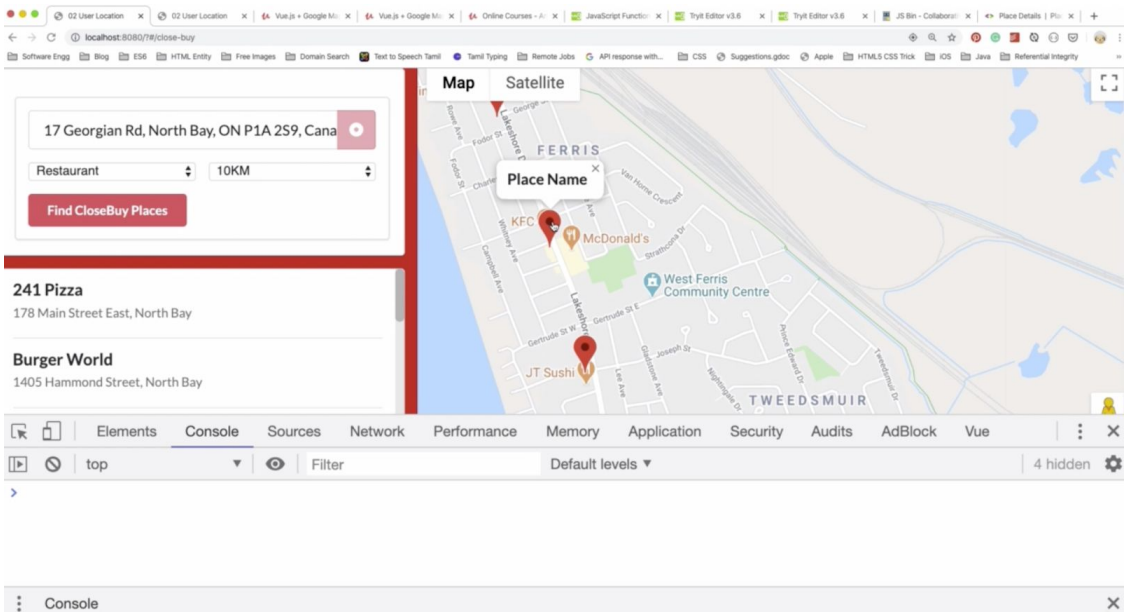
```
infoWindow.setContent( `<div class="ui header">place
name</div>`);
infoWindow.open(map, marker);
```

Finally, bring the info window object to the screen using the **open()** method.

This method takes two arguments that we want to add the info window into - **map** and **marker**.

Hop over to the browser.

Now, when we click on any of the markers, the infoWindow shows up with our dummy content.



Let's display the selected place name instead of dummy text, so replace the place name text to:

```
const infowindow = new google.maps.InfoWindow();

for (let i = 0; i < this.places.length; i++) {
   ...
  google.maps.event.addListener(marker, "click", function() {
    infoWindow.setContent(
      `<div class="ui header">${this.places[i].name}</div>`
    );
    infoWindow.open(map, marker);
  });
}
```

When running the app, you'll get an error something like this:

```
▸Uncaught TypeError: Cannot read property '4' of undefined
    at _.$f.eval (CloseBuy.vue?d0d4:61)
    at be.o (js?libraries=places&…Lh7mmValLvcw1ok:194)
    at Object._.N.trigger (js?libraries=places&…Lh7mmValLvcw1ok:191)
    at aU.<anonymous> (marker.js:31)
    at be.o (js?libraries=places&…Lh7mmValLvcw1ok:194)
    at Object._.N.trigger (js?libraries=places&…Lh7mmValLvcw1ok:191)
    at Object.onClick (marker.js:24)
    at Ko._.r.onClick (common.js:144)
    at HTMLAreaElement.ma._.Wo.Sb (common.js:63)
```

The issue is the scope of the keyword **this**.

When we use an anonymous function, the ***this*** keyword inside it is referring to the click event object NOT the export default object.

That means the event object does not have a property called **places** and it obviously throws an error.

To avoid the error, use an arrow function rather than the traditional anonymous function.

Why use an arrow function?

Arrow functions do not create a private scope to the **this** keyword, unlike the anonymous function.

When you use an arrow function, the **this** keyword is still referring to the original export default object.

Let's remove the **function** keyword and add the equals greater than symbol after the opening and closing parenthesis, which is also called a fat arrow.

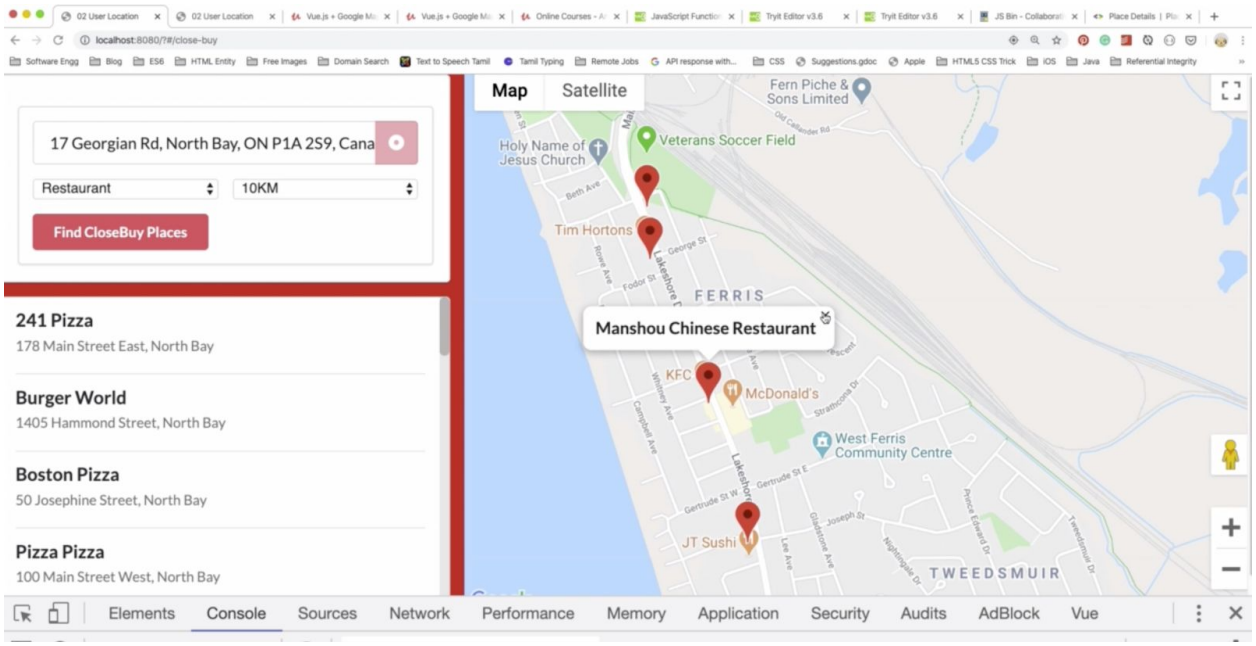
```
const infowindow = new google.maps.InfoWindow();

for (let i = 0; i < this.places.length; i++) {
   ...
  google.maps.event.addListener(marker, "click",() => {
    infoWindow.setContent(
      `<div class="ui header">${this.places[i].name}</div>`
```

```
    );
    infoWindow.open(map, marker);
  });
}
```

Let's try it one more time!

This time everything works as we intend it to. It has a close button on it and it also works.



Nearby Search Places data will have only limited information about a place such as it's name, address, etc. To get more information about a specific place, we need to make another HTTP call to Place Details Request which is also a part of Google Places API.

Show More Information About A Place

Place Details Request will provide more information about a place such as the website address, phone number, opening hours for a week or reviews and many more.

By the end of this chapter, you will:

1. Know about Place Details Request
2. Be able to make an HTTP Request to Place Details

Step #1: Place Details Request

Here is the Place Details URL. It has one required parameter that can be either json or xml.

```
https://maps.googleapis.com/maps/api/place/details/
```

Required Route Parameter

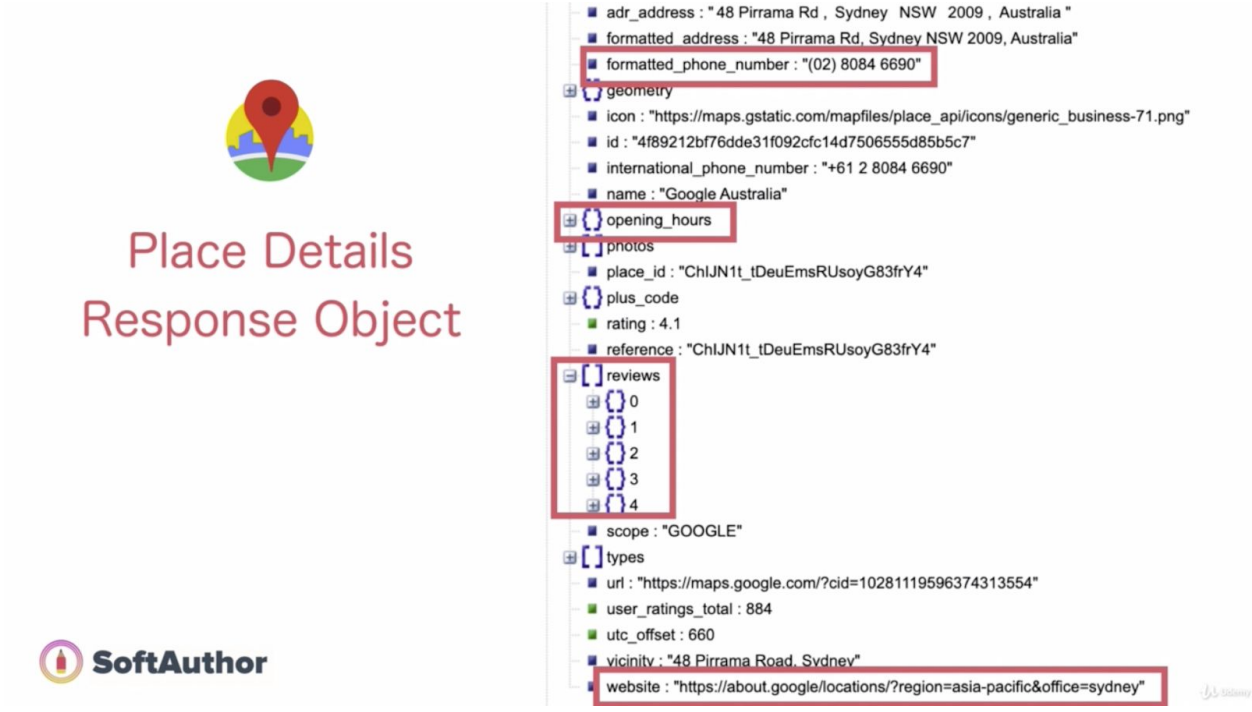
name	description
json or xml	This will determine what format the output response object will be. *(json recommended)*

Required Query Parameters

There are two required Query Parameters. Let's take a look at them one by one.

name	description
key	The API key that we got from the Google Cloud Console.
place_id	Unique ID of a place. We can find it in the place data from the Nearby Search Request response object.

Let's take a look at the output response object of the Place Details Request.



As you can see, it has more detailed information about the place such as phone number, opening hours, reviews website and so on.

Step #2: Make A HTTP Request To Place Details

Now that we have enough information, make an HTTP Call to the Place Details API.

Inside the **marker** click event listener callback function, get rid of the two lines of the info window.

```
  google.maps.event.addListener(marker, "click",() => {
    infoWindow.setContent(
      `<div class="ui header">${this.places[i].name}</div>`
    );
    infoWindow.open(map, marker);
  });
```

Define a const **URL** variable and add **Place Details Request HTTP URL** inside the back tics.

```
const URL =
`https://cors-anywhere.herokuapp.com/https://maps.googleapis.
com/maps/api/place/details`;
```

We are using back tics so that we can inject properties in-between the URL string using string interpolation.

Let's add the required route and query parameters to the URL string.

```
const URL =
`https://maps.googleapis.com/maps/api/place/details/json?key=
${this.apiKey}&place_id=${this.places[i].place_id}`;
```

Finally, append the proxy URL to it so that we avoid a CORS error.

```
const URL =
`https://cors-anywhere.herokuapp.com/https://maps.googleapis.
com/maps/api/place/details/json?key=${this.apiKey}&place_id=$
{this.places[i].place_id}`;
```

Now that we have the URL ready, let's make an HTTP call using axios.

This will return a promise, so use the **then()** method with a callback arrow function: **response => {}**

Add the catch block to catch any errors if the promise is rejected.

```
axios
.get(URL)
.then(response => {
  console.log(response.data.result);
})
.catch(error => {
  this.error = error.message;
});
```

Sometimes, this response object may have an error, so let's handle that next.

Go to the success callback function, check to see if there is an error message on the response object.

Then, add the else statement, which is where we get the actual response object.

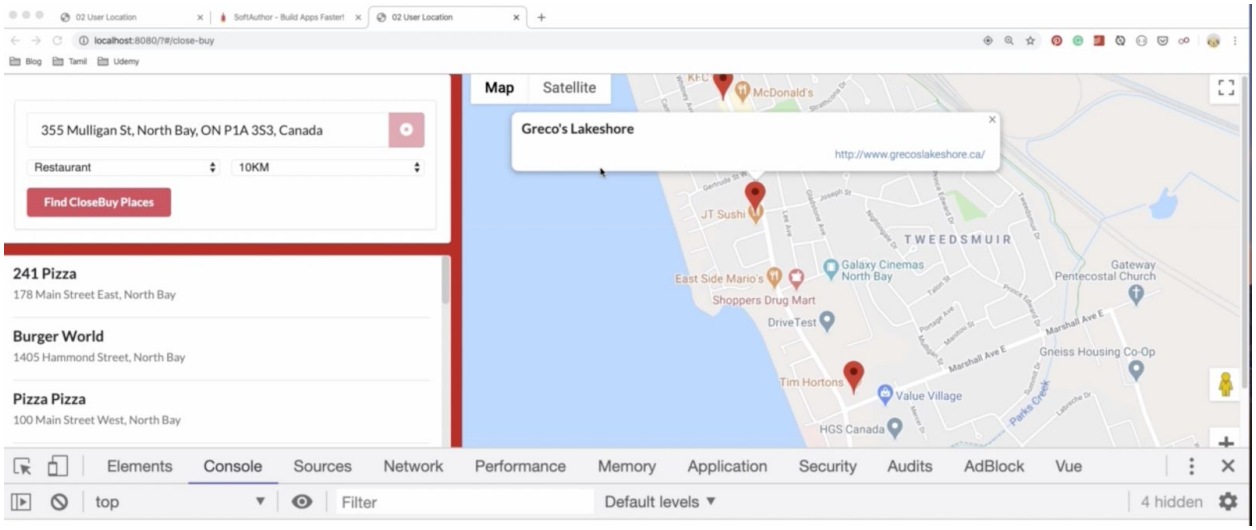
```
axios
.get(URL)
.then(response => {
  if (response.data.error_message) {
    this.error = response.data.error_message;
  } else {
    const place = response.data.result;
    console.log(place);
  }
})
.catch(error => {
  this.error = error.message;
});
```

To get the data, declare a variable called **place** and assign the response object to it. After that, set the place data; such as name, address, phone number, and/or website to the infowindow using it's **content** method and call the **open()** after.

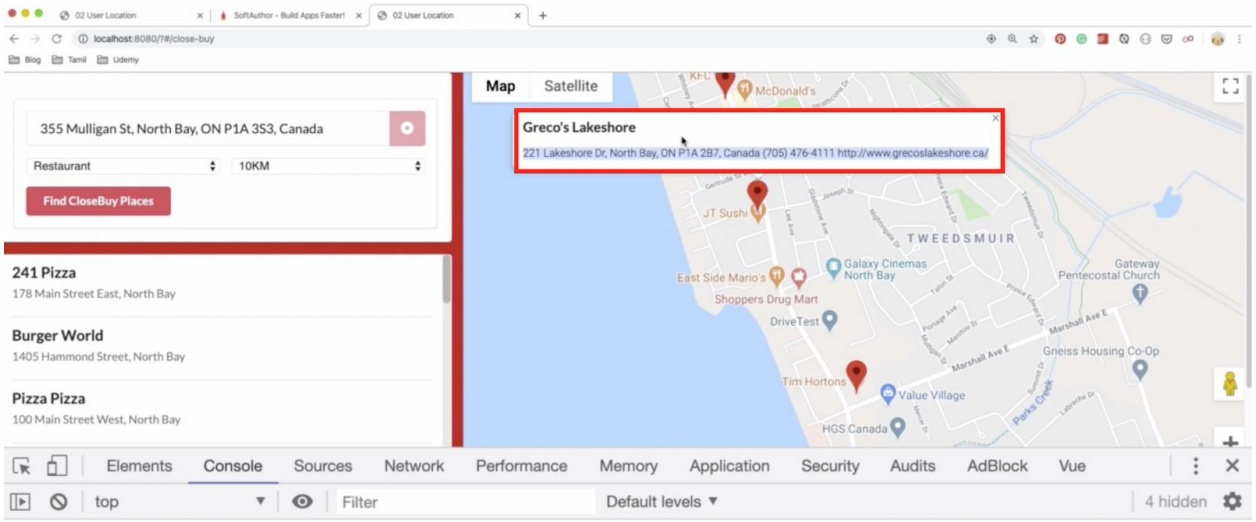
```
infoWindow.setContent(
  `<div class="ui header">${place.name}</div>
              ${place.formatted_address}
              ${place.formatted_phone_number}
<a href="${place.website}"
target="_blank">${place.website}</a>`
);
infoWindow.open(map, marker);
```

Now we have our own place object that has all the information we need, so we do not have to get data from the places array.

Let’s run the application. You may notice the title and website are there but not the address and phone number.



When we select all of the text in the info window, we can see the address and phone number, so their text color is set to white for some reason.

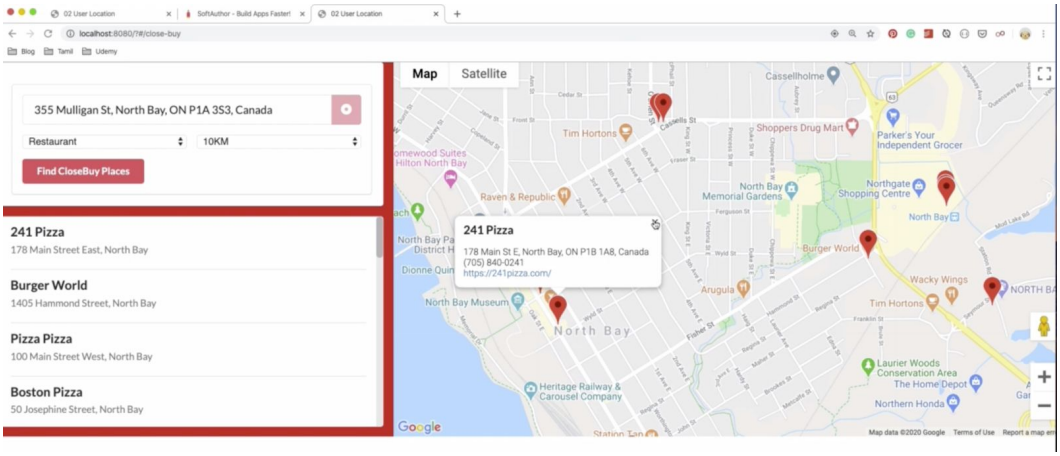


Actually, the class **blue** that is added to the map div container is causing this issue, so let's get rid of it.

```
<div class="ten wide column ~~blue~~" ref="map"></div>
```

Also, add a couple of break tags
 to the end of the address and phone number lines so that they are laid out one below the other.

```
infoWindow.setContent(
  `<div class="ui header">${place.name}</div>
              ${place.formatted_address} <br>
              ${place.formatted_phone_number}<br>          <a
href="${place.website}" target="_blank">${place.website}</a>`
);
infoWindow.open(map, marker);
```



Wouldn't it be cool to be able to select any place from the list on the left column and it will automatically highlight the appropriate marker in the map and open the info window showing more information about the place.

Users would easily be able to identify the place they are looking for without going through all of the markers.

Auto Select Markers When List Item is Clicked

First, attach a click event listener to the place item in the list. Go to the element where we loop through the places that have item class in it and create an event handler with a callback function called **showInfoWindow()**.

```
<div
        class="item"
        v-for="place in places"
        :key="place.id"
         @click="showInfoWindow()"
 >
    ...
</div>
```

This call back function will be triggered when any place item is clicked.

Use an index value in the v-for loop to determine which item is pressed as well as which infoWindow() needs to be visible.

To get the index value, add the second argument inside the v-for loop. After the place argument add comma index:

```
<div
        class="item"
        v-for="(place, index) in places"
        :key="place.id"
         @click="showInfoWindow(index)"
 >
    ...
```

```
</div>
```

When we add more than one argument, we need to wrap them with parentheses.

Now that we have access to the index variable, pass it as an argument to the **showInfoWindow()** function.

Declare the **infoWindow()** function inside the methods object.

```
showInfoWindow(index) {
   console.log(index)
}
```

This is where we will be getting the selected marker object and trigger the click event to it which will open up the **infoWindow()**.

To get the selected marker, we first need to add all the marker objects into an array.

Create a property called **markers** in the data model and set its value to an empty array.

```
data() {
    return {
      ...
      radius: "",
      places: [],
      markers: [],
    };
  },
```

Then, go to the **showPlacesOnMap()** function and inside the for loop, push all the markers objects to the markers array.

```
this.markers.push(marker);
```

Now we can easily get the selected marker object using the index value that is passed into the s**howInfoWindow()** function.

Go inside the **showInfoWindow()** function and trigger a click event method on the google event object.

```
new google.maps.event.trigger(this.markers[index], "click");
```

This takes two arguments. The first one is an actual **marker** object that we want to trigger the click event to. The second one is the **event name**, in my case **click** in quotes.

Finally, add an active class to the clicked place element in the left column.

Let’s create an active class inside the CSS block at the bottom:

```
<style>
...
.active {
  background: #ff5a5f !important;
}
</style>
```

Now we need to know which item is selected so that we can add the .**active** class to it.

Declare a new property called **activeIndex** and set its value to -1 so that no item is selected by default.

```
data() {
    return {
      ...
      places: [],
      markers: [],
      activeIndex: -1
    };
  },
```

Then, set the selected index value to the activeIndex property inside the **showInfoWindow()** function.

```
this.activeIndex = index
```

In the template, check if the **activeIndex** value is matched to any of the **index** values on each iteration.

```
 <div
    class="item"
    v-for="(place, index) in places"
    :key="place.id"
    @click="showInfoWindow(index)"
    :class="{'active' : activeIndex === index}"
  >
    <div class="content">
      <div class="header">{{place.name}}</div>
      <div class="meta">{{place.vicinity}}</div>
    </div>
  </div>
```

If the condition is true, then add the **.active** class to the appropriate item.

Finally, add an inline style:

```
style="padding:10px:
```

The active background color will have some padding around the text.

Run the app and it will work as expected!

About the Author

Raja Tamil is a self taught Web and iOS Developer with about five years experience working with various technologies.

Soon after, he transformed into a writer and founded SoftAuthor where he writes various technology based articles. SoftAuthor has over 15K visitors a month who are interested in learning new programming skills. It also has around 1000 Facebook followers.

Currently he is focussing on product and business development with his delivery service company, as well as converting his experience into online courses.