

Table of contents

1.1 Project Architecture	2
2.1 User Interface	3
3.1 Build instructions	6
3.1.1 Database and connection string configuration	6
3.1.2 Cors and server addresses configuration.....	9
3.1.3 API keys	10
3.1.4 Node.js/npm are required.....	11

1.1 Project Architecture

The backend is developed with .NET that consists of two parts: web api mvc and data layer. See Figure 1. On the other hand, frontend is another project developed on ReactJS. See figure 2.

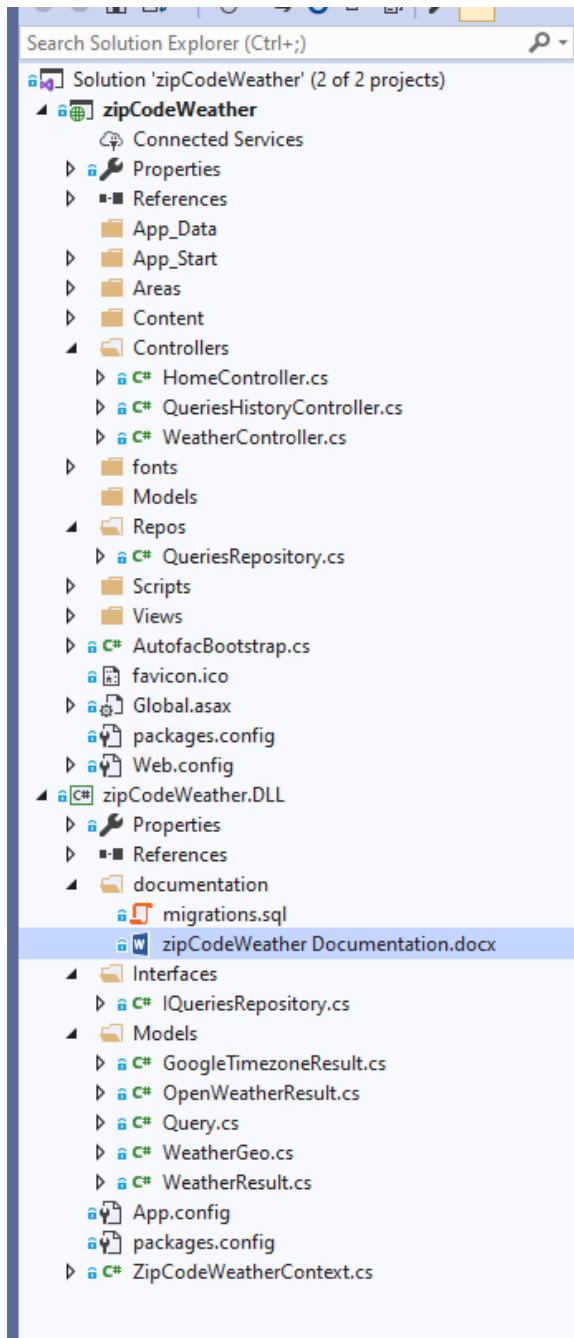


Figure 1. Solution explorer view of the project

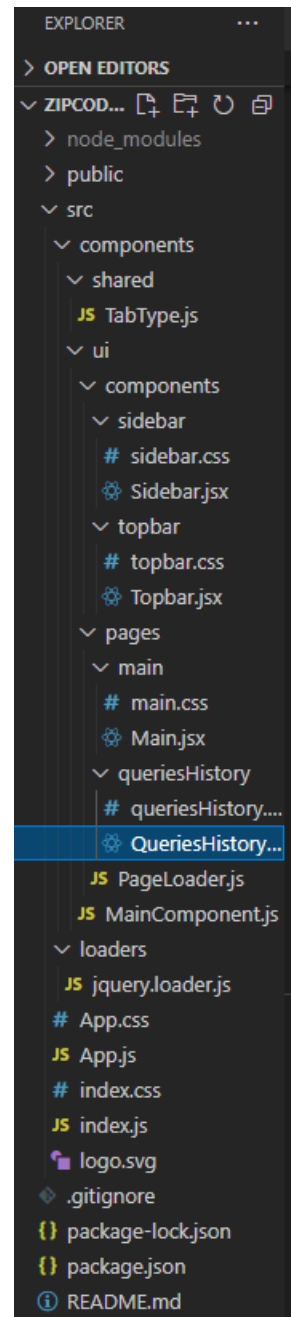


Figure 2. Frontend structure

2.1 User Interface

Let us take a look at the user interface that we see as soon as we launch the application. It should be noted that there are only two routes (pages) where we can get familiar with all requested queries or get information corresponding to the entered postal code such as the time zone, city and its current temperature. See figure 3 and 7.

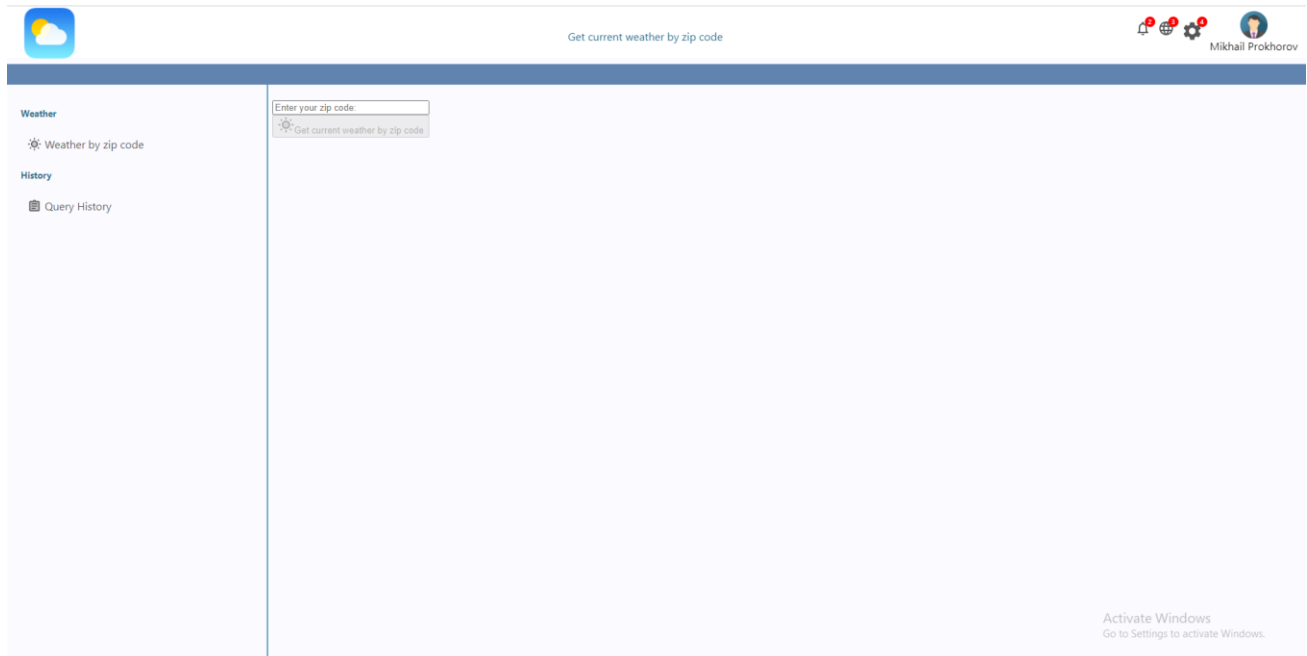


Figure 3. Initial user interface

To begin with, we can click the button to get the information about weather of the city that related to the entered postal code only after the input field is changed. If the postal code is not in supported format, the error will be displayed to a user (See figure 4).

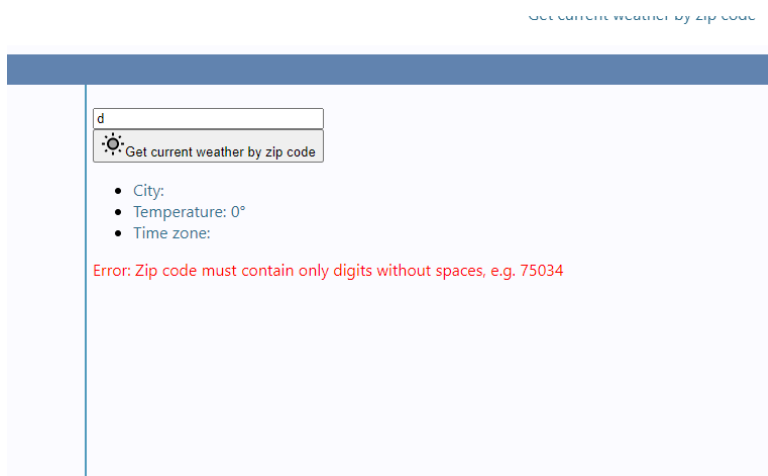
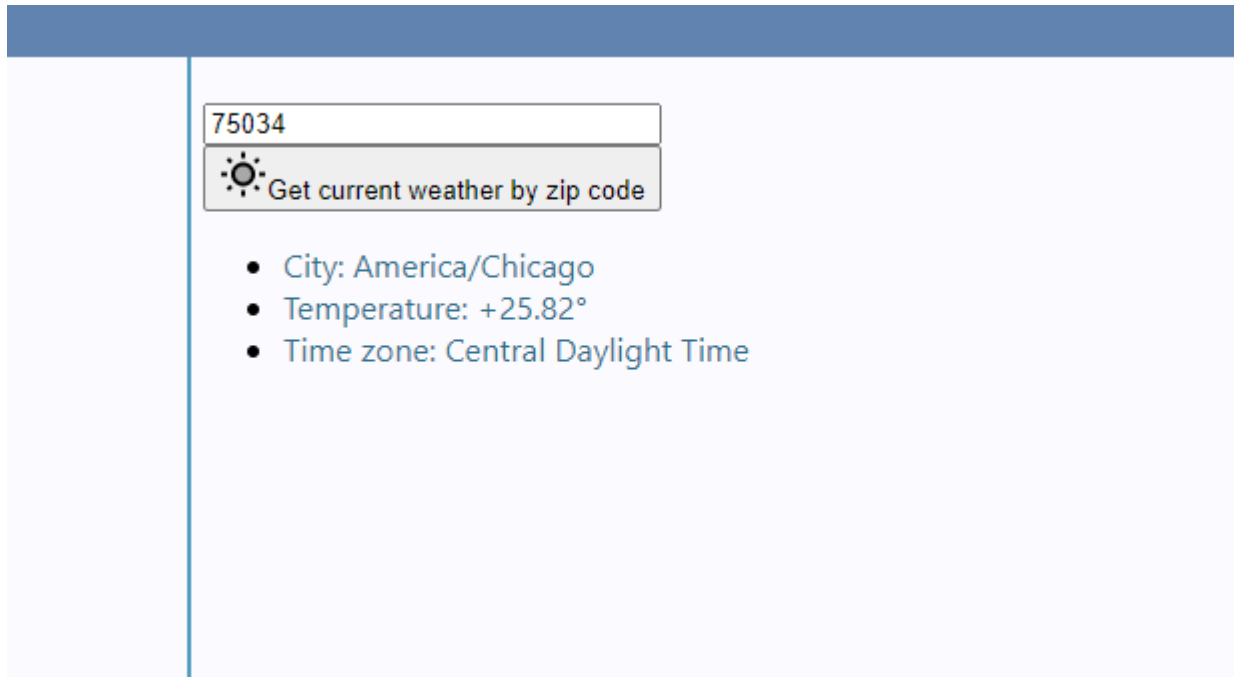


Figure 4. Displayed error of not supported format with valid format example

Otherwise, if the system was able to find weather information based on the postal code, we will see the city, its current temperature and time zone information. See figure 5.

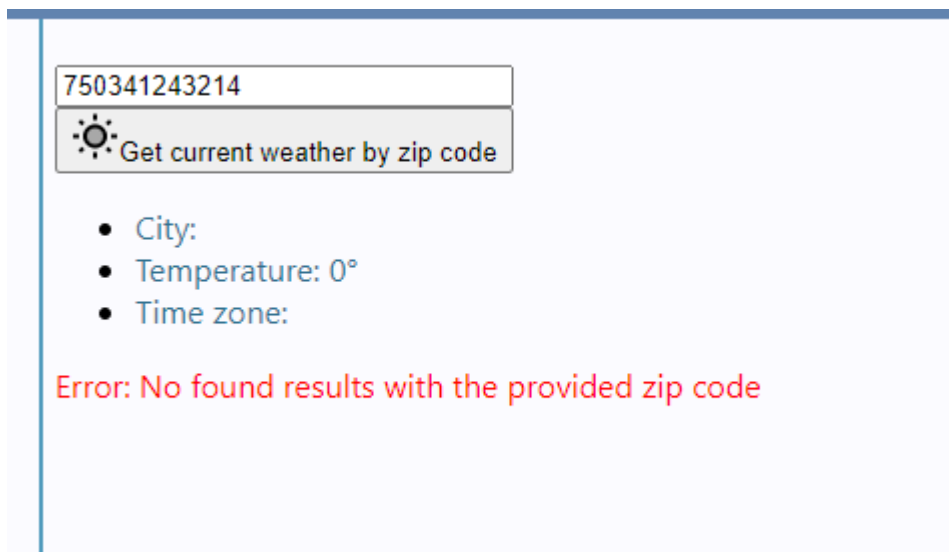


The screenshot shows a web interface with a blue header bar. Below it, there is a light blue sidebar on the left. The main content area is white. At the top of the main area, there is a text input field containing '75034'. Below the input field is a button with a sun icon and the text 'Get current weather by zip code'. Below the button, there is a list of three items:

- City: America/Chicago
- Temperature: +25.82°
- Time zone: Central Daylight Time

Figure 5. Weather information result

To add to this, if the zip code is in correct format but nonexistent or no weather results were found, the corresponding message will be shown. See figure 6.



The screenshot shows a web interface with a blue header bar. Below it, there is a light blue sidebar on the left. The main content area is white. At the top of the main area, there is a text input field containing '750341243214'. Below the input field is a button with a sun icon and the text 'Get current weather by zip code'. Below the button, there is a list of three items:

- City:
- Temperature: 0°
- Time zone:

Below the list, there is a red error message:

Error: No found results with the provided zip code

Figure 6. No found results message

Regarding the second page, we could analyze all query requests by clicking on “Query History” link. See figure 7.



Figure 7. Query History page

3.1 Build instructions

3.1.1 Database and connection string configuration

As regards to build instructions, first of all, we need to create a database and query table. There is an SQL file included in the backend project that is located at `\zipCodeWeather\zipCodeWeather.DLL\documentation\migrations.sql`. All we need to do is to copy and paste script text from line 3 to 21 to SQL Server Management Studio and execute it. See figure 8.



Figure 8. migrations.sql file

Secondly, we can execute the script from line 25 to 36 to get the connection string that we need to include in the web.config file. See figure 9, 10, 11 and 12.



Figure 9. Script to get the database connection string for web.config file

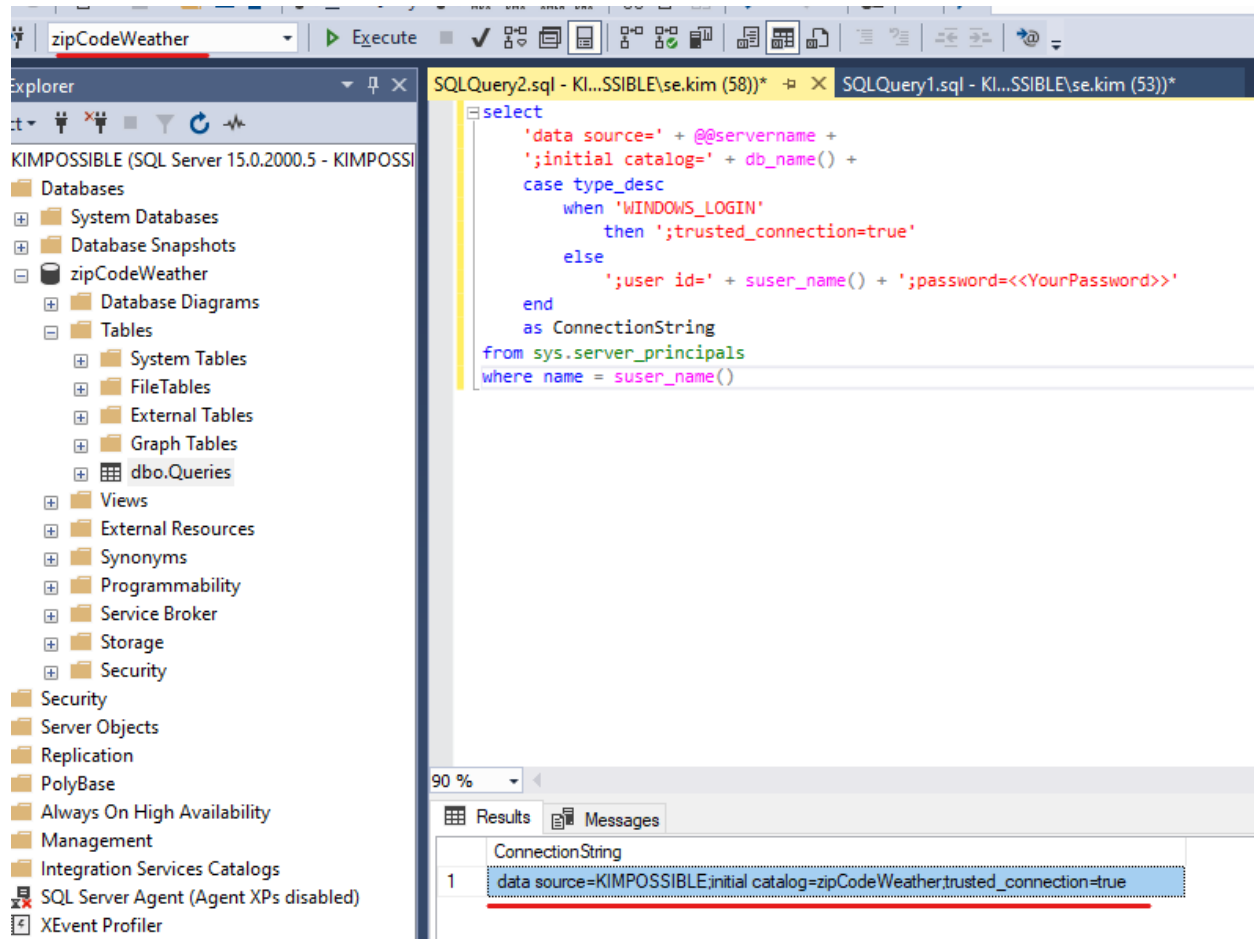


Figure 10. The connection string result after executing the script

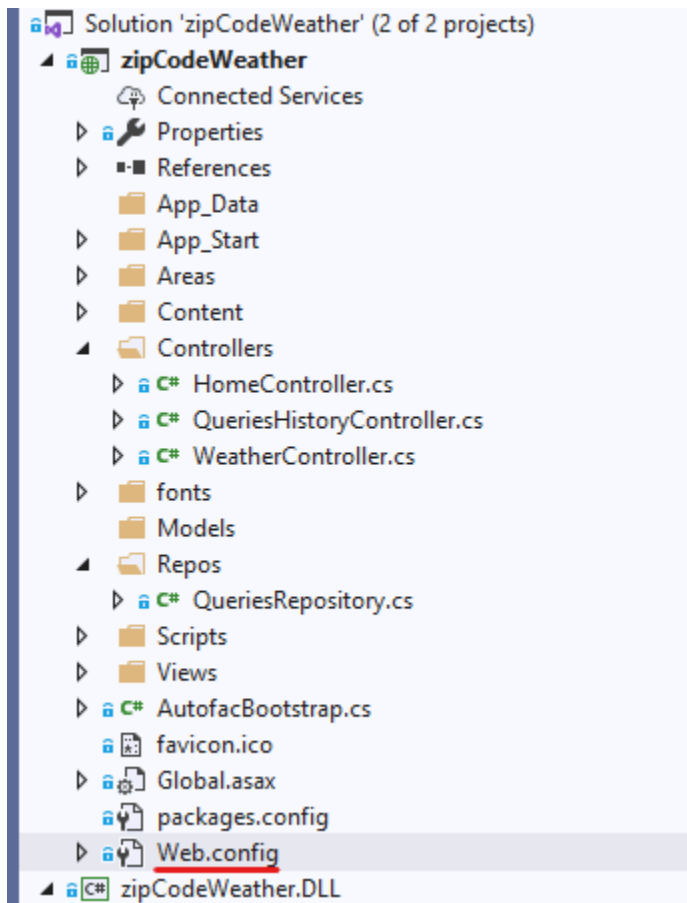


Figure 11. Web.config file that needs to be edited

```
-->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561" />
  </configSections>
  <connectionStrings>
    <add name="ZipCodeWeatherContext" connectionString="data source=KIMPOSSIBLE;initial catalog=zipCodeWeather;trusted_connection=true" providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
</configuration>
```

Figure 12. The connection string record in web.config file that should be added

3.1.2 Cors and server addresses configuration

Since the frontend and backend are two different object and they need to communicate with each other, we need to configure cors policies. Basically, all we need to do is to ensure that server addresses are referring to each other on both ends. Firstly, we need to check so that the server address on which the frontend runs is the same as in the “origins” field in QueriesHistory and Weather controllers. See figure 13.



Figure 13. Cors configuration in Queries HistoryController

Secondly, in the same way, we need to ensure that our pages (Main.jsx and QueriesHistory.jsx) on the frontend send requests to the right backend server address and port. If it differs from the actual backend server address and port, we need to change it so that they were the same. See figure 14.

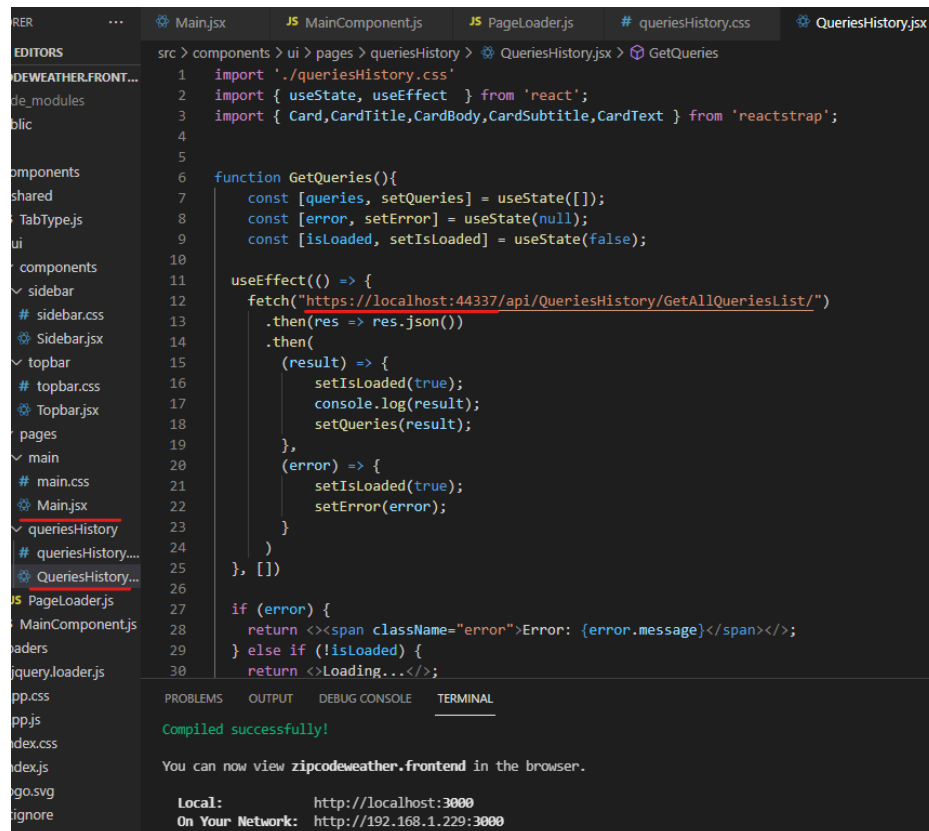


Figure 14. Main.jsx and QueriesHistory.jsx files' backend urls

3.1.3 API keys

Moreover, we need to have valid API keys for getting weather and timezone information from third party APIs. The current used keys are valid at the moment of writing the documentation. However, it would be great to pay attention to them in case of them not being valid by the time of users reading the document and using the system. There are two API keys variables `googleApiKey` and `openweatherApiKey` that are Google API and Open WeatherMap API keys, respectively. See figure 15. If they are not workable, they can be retrieved from <https://openweathermap.org/> and <https://console.cloud.google.com/>.

```
[EnableCors(origins: "http://localhost:3000", headers: "*", methods: "**")]
1 reference | KimSergey94, 14 hours ago | 1 author, 3 changes
public class WeatherController : ApiController
{
    private readonly IQueriesRepository _queriesRepository;
    0 references | KimSergey94, 14 hours ago | 1 author, 1 change
    public WeatherController(IQueriesRepository queriesRepository)
    {
        _queriesRepository = queriesRepository;
    }

    // GET: api/Weather/GetWeather
    [ResponseType(typeof(WeatherResult))]
    0 references | KimSergey94, 14 hours ago | 1 author, 3 changes
    public HttpResponseMessage GetWeather(string zipCode)
    {
        var weather = GetWeatherData(zipCode);
        _queriesRepository.SaveQuery(new Query() { Status = weather.Status, City = weather.City, ErrorMessage = weather.ErrorMessage, Tempera

        var res = Request.CreateResponse(HttpStatusCode.Created, weather);
        res.Content = new StringContent(new JavaScriptSerializer().Serialize(weather), Encoding.UTF8, "application/json");
        return res;
    }

    1 reference | KimSergey94, 20 hours ago | 1 author, 1 change
    private WeatherResult GetWeatherData(string zipCode)
    {
        long code = 0;
        if(!long.TryParse(zipCode, out code))
        {
            return new WeatherResult() { Status = "Error", ErrorMessage = "Zip code must contain only digits without spaces, e.g. 75034" };
        }
        try
        {
            var googleApiKey = "AIzaSyCwVwKTwK7KX9WNBx65CxU1m4v-Axf5_G0";
            var geocodeReq = WebRequest.Create(@"https://maps.googleapis.com/maps/api/geocode/json?key=" + googleApiKey + "&components=postal
            geocodeReq.ContentType = "application/json; charset=utf-8";
            var geocodeResponse = (HttpWebResponse)geocodeReq.GetResponse();

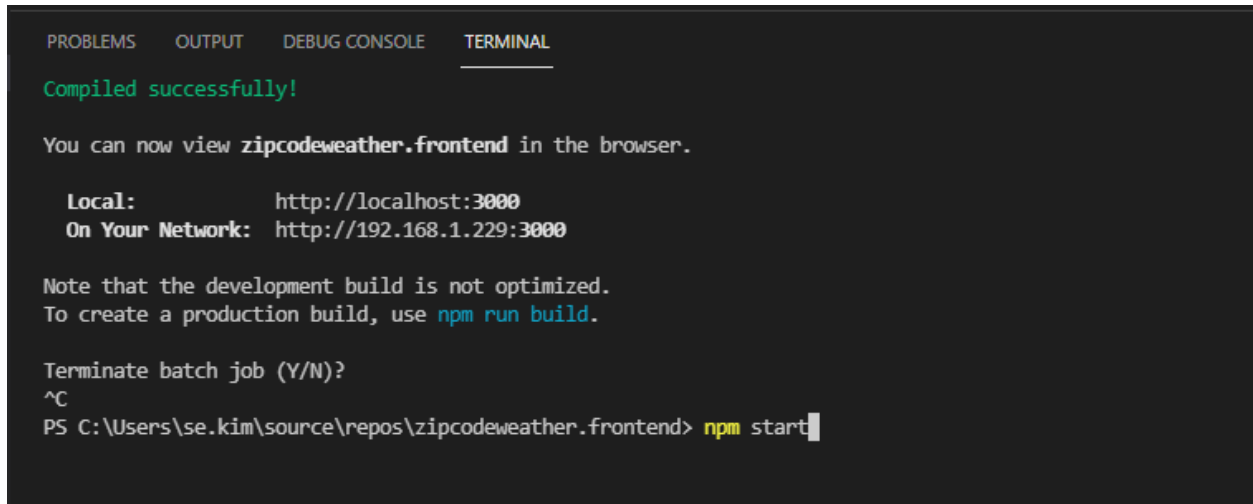
            if (geocodeResponse.StatusDescription == "OK")
            {
                var weatherGeo = GetWeatherLatitudeAndLongitudeFromGeocodeResponse(geocodeResponse);

                try
                {
                    var openweatherApiKey = "0053ea67e03320c6f7a322ed2b84ba0a";
                    var openweatherReq = WebRequest.Create("https://api.openweathermap.org/data/2.5/onecall?lat=" + weatherGeo.Latitude + "&l
                    var openweatherResponse = (HttpWebResponse)openweatherReq.GetResponse();
                    if (openweatherResponse.StatusDescription == "OK")
```

Figure 15. Google API and Open WeatherMap API keys

3.1.4 Node.js/npm are required

Finally, another thing that we should be aware of is that we need Node.js/npm to be installed and we are able to launch React application via Visual Studio Code with the “npm start” command from terminal. See figure 16.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Compiled successfully!

You can now view zipcodeweather.frontend in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.229:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

Terminate batch job (Y/N)?
^C
PS C:\Users\se.kim\source\repos\zipcodeweather.frontend> npm start
```

Figure 16. Visual Studio Code terminal window