Geocoding in Java with the Google Maps API

By Airen Surzyn // September 14, 2020

Table of Contents [hide]

Location data is an integral part of a variety of modern software applications. Fortunately for developers, the Google Maps API along with many others grants us access to a rich collection of geographical data and mapping tools.

In this tutorial, we will set up a simple Java Spring Boot application that calls the Google Maps API and returns Geocoded data.

If you are not familiar with APIs I would recommend a brief of review API Terms and Concepts before getting started.

For this tutorial, you will need the following:

- Around 15 or 20 minutes
- Maven 3+
- JDK 8 or higher
- Text Editor or IDE

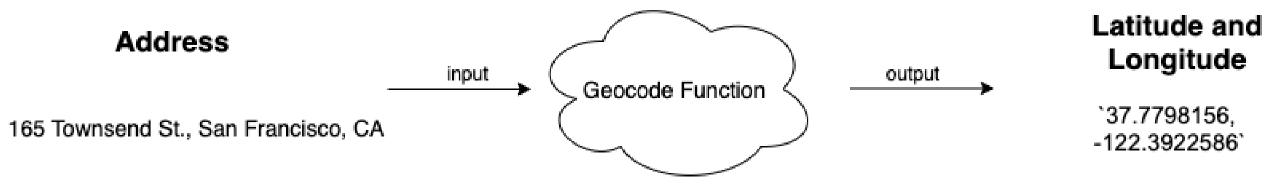
- An internet connection

Connect to the Google Maps API

Geocoding API

Geocoding is the process of converting a human-readable address or location name into a latitude-longitude coordinate. Reverse geocoding is the opposite process.

For example:



The Google Maps Geocode API has two endpoints:

1. GET Geocoding: takes a human-readable address, returns location data

   GET Reverse Geocoding: takes latitude and longitude, returns an address

The Geocoding API returns other useful information, but we can look into that a little later. For now, let’s start building!

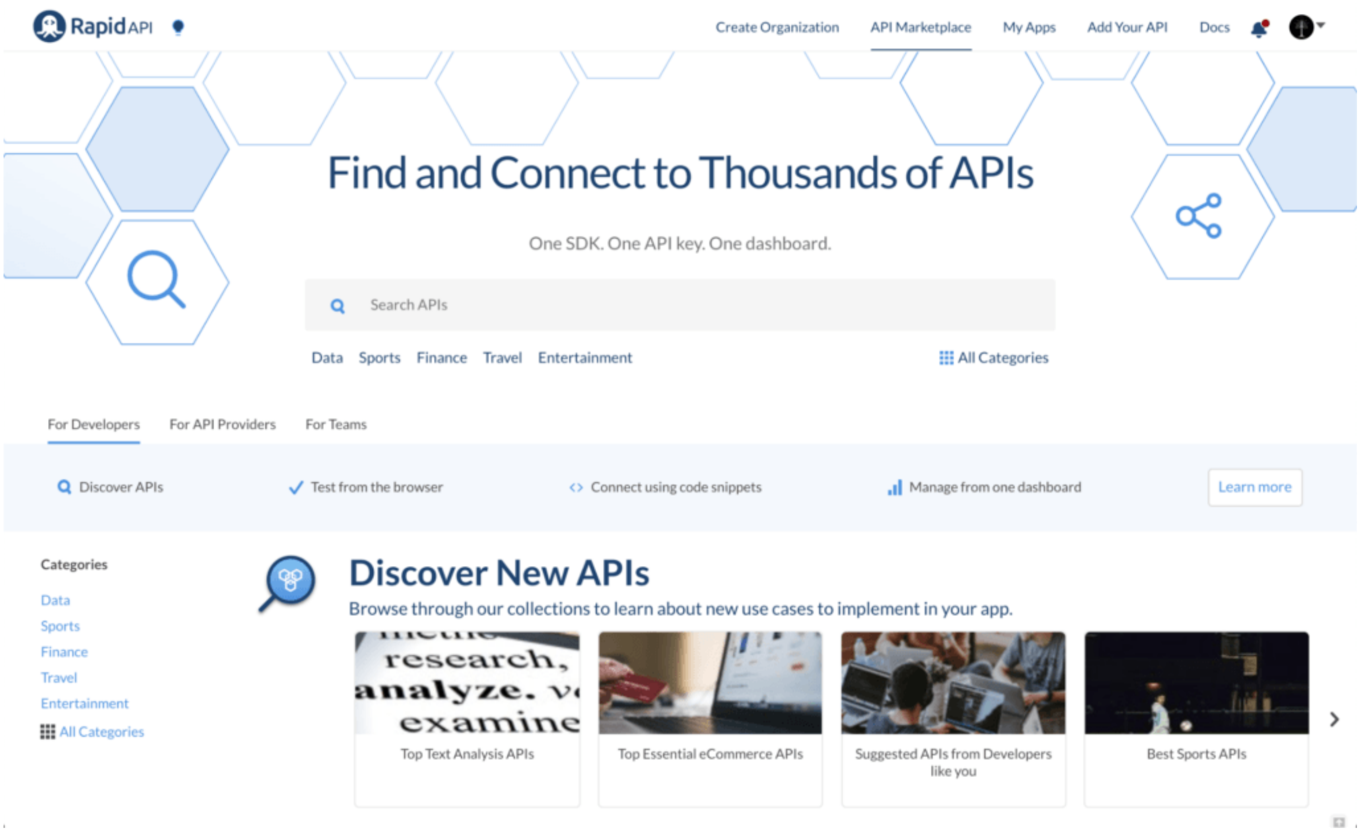
Connect to the Google Maps API

Create a Google Maps API Java Project

Getting a Google Maps API Key

API Keys are essential for accessing third-party APIs, we will need one to make calls to the Google Maps API.

To get an API Key for the Google Maps API, create a RapidAPI account here.


Rapid API

Create Organization | API Marketplace | My Apps | Add Your API | Docs

Find and Connect to Thousands of APIs

One SDK. One API key. One dashboard.

Search APIs

Data Sports Finance Travel Entertainment | All Categories

For Developers | For API Providers | For Teams

Discover APIs | Test from the browser | Connect using code snippets | Manage from one dashboard | Learn more

Categories: Data, Sports, Finance, Travel, Entertainment, All Categories

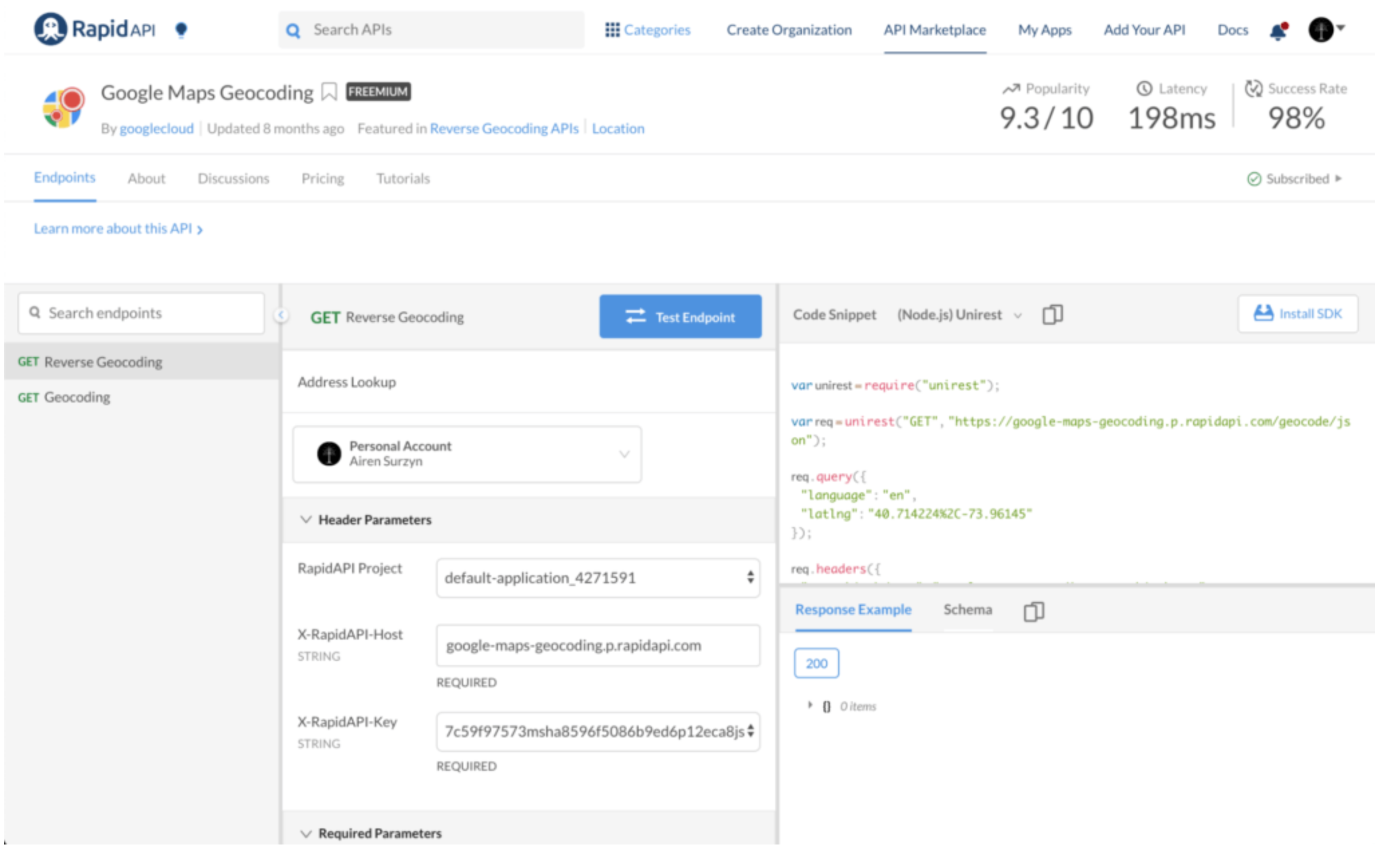
Discover New APIs

Browse through our collections to learn about new use cases to implement in your app.

Top Text Analysis APIs | Top Essential eCommerce APIs | Suggested APIs from Developers like you | Best Sports APIs

Subscribe to the Google Maps Geocode API

Once you have your account, simply subscribe to the Google Maps Geocode API by searching through your Rapid API Dashboard or accessing it here.


Rapid API | Search APIs | Categories | Create Organization | API Marketplace | My Apps | Add Your API | Docs

Google Maps Geocoding FREEMIUM

By googlecloud | Updated 8 months ago | Featured in Reverse Geocoding APIs | Location

Popularity 9.3 / 10 | Latency 198ms | Success Rate 98%

Endpoints | About | Discussions | Pricing | Tutorials | Subscribed

Learn more about this API

Search endpoints

GET Reverse Geocoding

GET Geocoding

GET Reverse Geocoding | Test Endpoint

Address Lookup

Personal Account
Airen Surzyn

Header Parameters

RapidAPI Project	default-application_4271591
X-RapidAPI-Host STRING	google-maps-geocoding.p.rapidapi.com REQUIRED
X-RapidAPI-Key STRING	7c59f97573msha8596f5086b9ed6p12eca8js REQUIRED

Required Parameters

Code Snippet (Node.js) Unirest | Install SDK

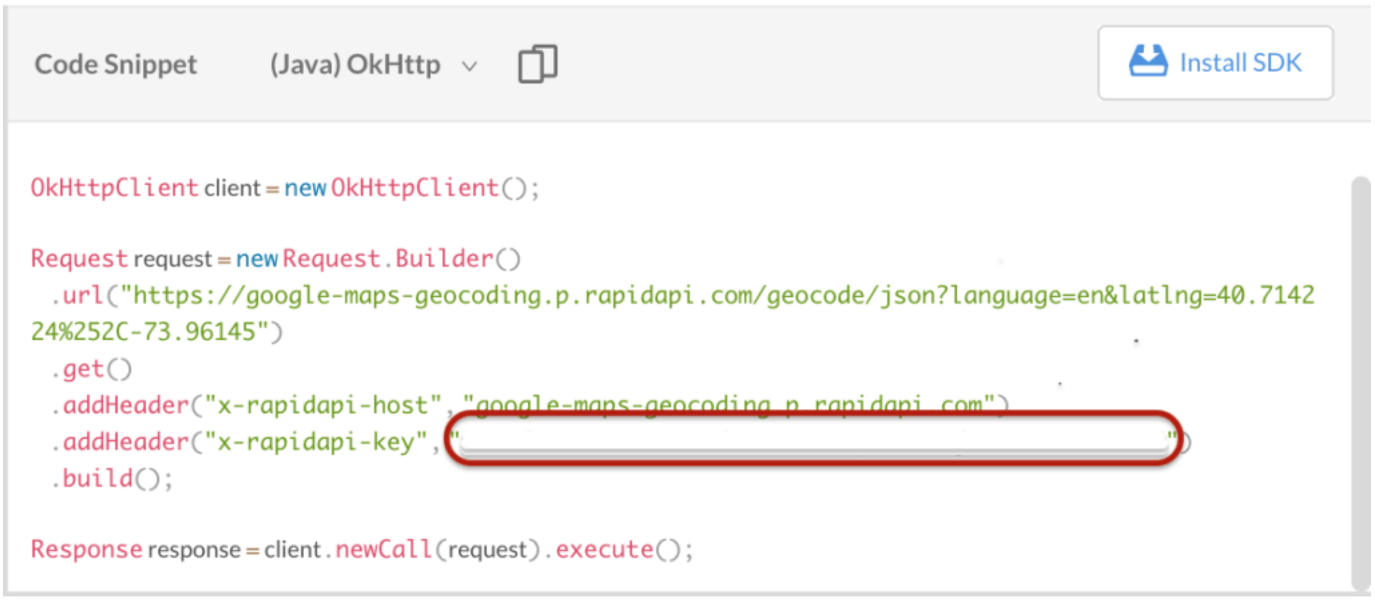
```
var unirest = require("unirest");

var req = unirest("GET", "https://google-maps-geocoding.p.rapidapi.com/geocode/js
on");

req.query({
 "language": "en",
 "latlng": "40.714224%2C-73.96145"
});

req.headers({
```

Response Example | Schema

200

[] 0 items

Your API Key is visible in the code snippet on the right side or at your dashboard.



```java
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()
  .url("https://google-maps-geocoding.p.rapidapi.com/geocode/json?language=en&latlng=40.7142
24%252C-73.96145")
  .get()
  .addHeader("x-rapidapi-host", "google-maps-geocoding.p.rapidapi.com")
  .addHeader("x-rapidapi-key", "")
  .build();

Response response = client.newCall(request).execute();
```

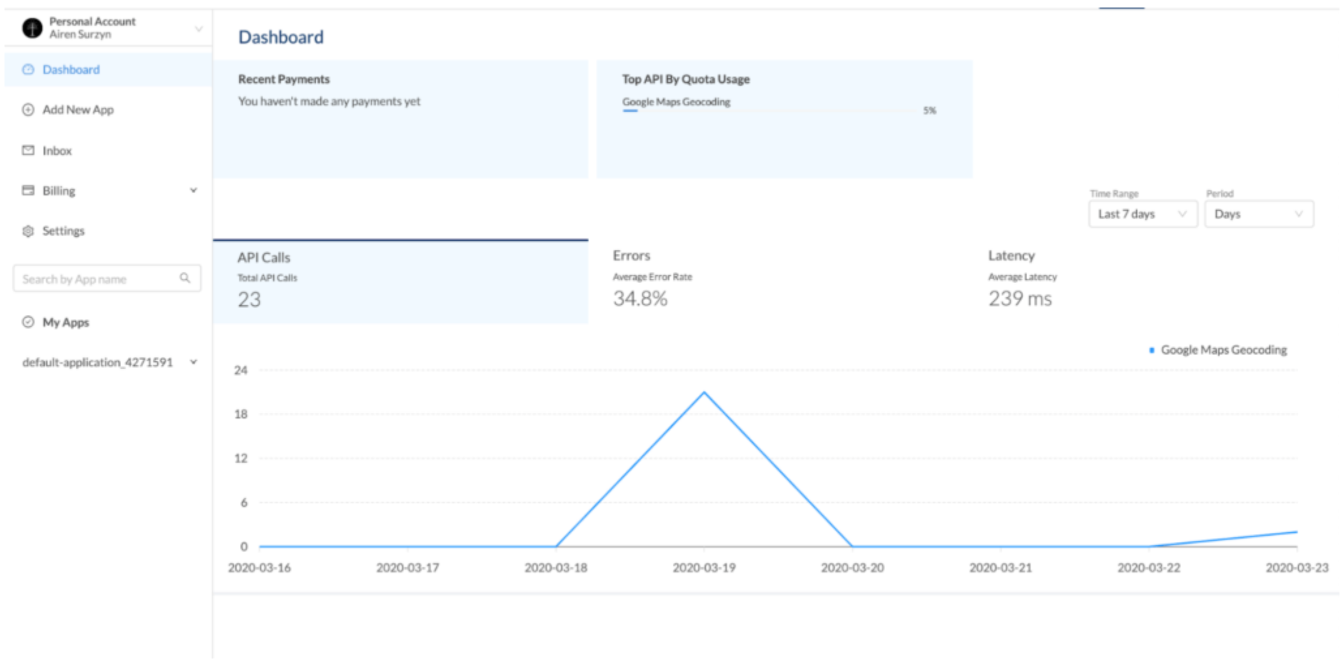
Google Maps API Pricing

When you subscribe, select the Basic plan which will ensure your first 500 calls to the Google Maps API are free. That will more than suffice for this tutorial.

However, if you are thinking about integrating the API into a public application, it would be worth considering the Pro or Ultra plan depending on your expected traffic.

For example, if you expect 5,000 requests/month upgrading to the Pro plan will cost you only $25 compared with $45 if you stay with the Basic plan.

If you are concerned about the number of requests your app will receive, the [RapidAPI Dashboard] will help you monitor the number of requests your application is handling.



Create Your Java Project

Now that we have our API Key, we will create our Java Spring Boot project.

To initialize a Java Project, we will use the Spring Initializr. Spring is a convenient Java framework for handling dependencies and project structure.

From your terminal, issue the command:

```
curl https://start.spring.io/starter.zip
-d language=java
-d dependencies=web
-d packageName=io.example.rest
-d name=spring-boot
-d type=maven-project
-o geocode-api.zip
```

This command queries the Spring Initializr endpoint which returns a full Maven project, saving you the extra configuration steps.

Unzip the file with this command:

```
unzip geocode-api.zip -d geocode-api
```

Add our Dependencies

We will need two additional dependencies for our application, an HTTP client and a JSON parser.

Navigate to your pom.xml file and add the following maven dependencies for the OkHTTP client and Jackson Json parser.

```
<dependency>
<groupId>com.squareup.okhttp3</groupId>
<artifactId>okhttp</artifactId>
<version>4.4.0</version>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.9.9</version>
</dependency>
```

Make sure to import these pom.xml changes into your maven build before moving onto the next step.

Setting Up Java Classes

Next, we will create a Controller that will handle requests to our application and redirect them to the Google Maps API.

The Spring Initializr has already created an Application.class which contains your main() method. At the same directory level in your project, create a new class called GeocodeController.java.

```
package io.example.rest;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.ResponseBody;
import org.springframework.web.bind.annotation.*;
import java.io.IOException;
import java.net.URLEncoder;
@RestController
public class GeocodeController {
@RequestMapping(path = "/geocode", method = RequestMethod.GET )
public String getGeocode(@RequestParam String address) throws IOException {
OkHttpClient client = new OkHttpClient();
String encodedAddress = URLEncoder.encode(address, "UTF-8");
Request request = new Request.Builder()
.url("https://google-maps-geocoding.p.rapidapi.com/geocode/json?language=en&address=" +
encodedAddress)
.get()
.addHeader("x-rapidapi-host", "google-maps-geocoding.p.rapidapi.com")
.addHeader("x-rapidapi-key", {your-api-key-here}/* Use your API Key here */)
.build();
ResponseBody responseBody = client.newCall(request).execute().body();
return responseBody.string();
}
}
```

**Note, be sure to substitute your own API Key into your code.

The @RestController annotation at the top of the class is a Spring annotation that denotes the class as a Spring Component.

With the @RequestMapping annotation, we define a URL path for our method and the acceptable REST action (in this case GET).

In our method declaration, we see another annotation, @RequestParam which defines the address variable that our endpoint will accept.

Inside our method, we:

- Instantiate an OkHttpClient object
- Pass it our Google Maps API endpoint and encoded address
- Add our API key and host as headers
- Handle the Response

Right now, our controller takes an address at the endpoint http://localhost:8080/geocode , requests the Geocode from the Google Maps API and returns the json string as a response. Let's try it out.

Running our Application

If you are working in an IDE, you can run your application from your Application.class main() method.

Otherwise, from your terminal use the command:

```
./mvnw spring-boot:run
```

By default, your application will load on port 8080.

To test, we will use the address `164 Townsend St. San Francisco, CA`.

Once it has loaded, send the following request to your application through your command line:

```
curl
http://localhost:8080/geocode?address=164%20Townsend%20St.%2C%20San%20Francisco%2
C%20CA
```

*Note, I am passing in an encoded URL here to make it command-line friendly. You can encode an address string here.

API Response

In your terminal you should receive a JSON response that looks similar to the following:

```
{
"results" : [
{
"address_components" : [
{
"long_name" : "164",
"short_name" : "164",
"types" : [ "street_number" ]
},
{
"long_name" : "Townsend Street",
"short_name" : "Townsend St",
"types" : [ "route" ]
},
{
"long_name" : "South Beach",
"short_name" : "South Beach",
"types" : [ "neighborhood", "political" ]
},
{
"long_name" : "San Francisco",
"short_name" : "SF",
"types" : [ "locality", "political" ]
},
{
"long_name" : "San Francisco County",
"short_name" : "San Francisco County",
"types" : [ "administrative_area_level_2", "political" ]
},
{
"long_name" : "California",
"short_name" : "CA",
"types" : [ "administrative_area_level_1", "political" ]
},
{
"long_name" : "United States",
"short_name" : "US",
"types" : [ "country", "political" ]
},
{
"long_name" : "94107",
"short_name" : "94107",
"types" : [ "postal_code" ]
}
],
"formatted_address" : "164 Townsend St, San Francisco, CA 94107, USA",
"geometry" : {
"bounds" : {
"northeast" : {
"lat" : 37.7801547,
"lng" : -122.3919329
},
"southwest" : {
"lat" : 37.7795512,
"lng" : -122.3926987
}
},
"location" : {
"lat" : 37.7798156,
"lng" : -122.3922586
},
"location_type" : "ROOFTOP",
"viewport" : {
"northeast" : {
"lat" : 37.78120193029149,
"lng" : -122.3909668197085
},
"southwest" : {
"lat" : 37.7785039697085,
"lng" : -122.3936647802915
}
}
},
"place_id" : "ChIJy5Ij0Nd_j4ARewphNhlNJvE",
"types" : [ "premise" ]
}
],
"status" : "OK"
}
```

If you are seeing this, congrats you have successfully queried the Google Maps API!

As you can see, we get a lot more information than just a simple latitude and longitude. It returns other useful location information contained in the address components array such as neighborhood and county data.

Mapping the Result

Right now, our application returns a JSON string as a response.

Returning a JSON string is handy to visualize the result we are getting back, but it is bad Java practice to send a bare JSON string around our application. So, we have one more step to finish our Google Maps API implementation – we must map our result to a custom Java object.

Let’s define some classes to map our results. For the scope of this tutorial, we will focus on mapping only a few of the returned objects. We will create the following classes:

- GeocodeResult.class
- GeocodeObject.class
- AddressComponent.class
- GeocodeGeometry.class
- GeocodeLocation.class

Class names are up to your discretion, but variable names must either match corresponding JSON field names or use a custom mapping annotation as seen below.

GeocodeResult.class

```
package io.example.rest;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import java.util.List;
@JsonIgnoreProperties(ignoreUnknown = true)
public class GeocodeResult {
List<GeocodeObject> results;
String status;
public GeocodeResult() {
}
public String getStatus() {
return status;
}
public void setStatus(String status) {
this.status = status;
}
public List<GeocodeObject> getResults() {
return results;
}
public void setResults(List<GeocodeObject> results) {
this.results = results;
}
}
```

GeocodeObject.class

```
package io.example.rest;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import java.util.List;
@JsonIgnoreProperties(ignoreUnknown = true)
public class GeocodeObject {
@JsonProperty("place_id")
String placeId;
@JsonProperty("address_components")
List<AddressComponent> addressComponents;
@JsonProperty("formatted_address")
String formattedAddress;
GeocodeGeometry geometry;
public GeocodeObject() {
}
public List<AddressComponent> getAddressComponents() {
return addressComponents;
}
public void setAddressComponents(List<AddressComponent> addressComponents) {
this.addressComponents = addressComponents;
}
public String getPlaceId() {
return placeId;
}
public void setPlaceId(String placeId) {
this.placeId = placeId;
}
public String getFormattedAddress() {
return formattedAddress;
}
public void setFormattedAddress(String formattedAddress) {
this.formattedAddress = formattedAddress;
}
public GeocodeGeometry getGeometry() {
return geometry;
}
public void setGeometry(GeocodeGeometry geometry) {
this.geometry = geometry;
}
}
```

AddressComponent.class

```
package io.example.rest;
import com.fasterxml.jackson.annotation.JsonProperty;
import java.util.List;
public class AddressComponent {
@JsonProperty("long_name")
String longName;
@JsonProperty("short_name")
String shortName;
List<String> types;
public AddressComponent() {
}
public String getLongName() {
return longName;
}
public void setLongName(String longName) {
this.longName = longName;
}
public String getShortName() {
```

```java
return shortName;
}
public void setShortName(String shortName) {
this.shortName = shortName;
}
public List<String> getTypes() {
return types;
}
public void setTypes(List<String> types) {
this.types = types;
}
}
```

GeocodeGeometry.class

```java
package io.example.rest;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
@JsonIgnoreProperties(ignoreUnknown = true)
public class GeocodeGeometry {
@JsonProperty("location")
GeocodeLocation geocodeLocation;
public GeocodeGeometry() {
}
public GeocodeLocation getGeocodeLocation() {
return geocodeLocation;
}
public void setGeocodeLocation(GeocodeLocation geocodeLocation) {
this.geocodeLocation = geocodeLocation;
}
}
```

GeocodeLocation.class

```java
package io.example.rest;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
@JsonIgnoreProperties(ignoreUnknown = true)
public class GeocodeLocation {
@JsonProperty("lat")
private String latitude;
@JsonProperty("lng")
private String longitude;
public GeocodeLocation() {
}
public String getLatitude() {
return latitude;
}
public void setLatitude(String latitude) {
this.latitude = latitude;
}
public String getLongitude() {
return longitude;
}
public void setLongitude(String longitude) {
this.longitude = longitude;
}
}
```

Finally, we will add our mapper to the GeocodeController class we created earlier.

```
package io.example.rest;
import com.fasterxml.jackson.databind.ObjectMapper;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.ResponseBody;
import org.springframework.web.bind.annotation.*;
import java.io.IOException;
import java.net.URLEncoder;
@RestController
public class GeocodeController {
@RequestMapping(path = "/geocode", method = RequestMethod.GET )
public GeocodeResult getGeocode(@RequestParam String address) throws IOException {
OkHttpClient client = new OkHttpClient();
String encodedAddress = URLEncoder.encode(address, "UTF-8");
Request request = new Request.Builder()
.url("https://google-maps-geocoding.p.rapidapi.com/geocode/json?language=en&address=" +
encodedAddress)
.get()
.addHeader("x-rapidapi-host", "google-maps-geocoding.p.rapidapi.com")
.addHeader("x-rapidapi-key", {your-api-key-here}/* Use your API Key here */)
.build();
ResponseBody responseBody = client.newCall(request).execute().body();
ObjectMapper objectMapper = new ObjectMapper();
GeocodeResult result = objectMapper.readValue(responseBody.string(), GeocodeResult.class);
return result;
}
}
```

If you run your application again and send the same curl command :

```
curl http://localhost:8080/geocode?address=164%20Townsend%20St.%2C%20San%20Francisco%2C%20CA
```

You will now receive a proper Java object that is suitable to be sent around an application.

Connect to the Google Maps API

Conclusion

In this tutorial, we set up a simple Java Spring Boot web application that queries and packages Google Maps API data.

This is a great starting point for the backend service of an app that will use location data.

Next, try integrating the functionality we built here with another API like the Current Weather Conditions API to get location-based weather data or the Distance API to find the distance between any two lat-long geographical points.

Access to the Google Maps API opens up a lot of possibilities with other APIs and I hope this tutorial has given you the foundation to try many more.