# Tools, framework and library used

## **Front end:**

IDE: WebStorm 2021.2.1

This is an IDE for development of web applications, it comes with inbuilt git management functions and has many tools to assist easy debugging which is why I chose it. It can also automatically detect libraries that have not been installed and prompt you to one-click install them. However other IDEs like VSCode should work fine.

Libraries/Packages/External Tools:

* JavaScript language, ReactJS framework
* Node.js Version 14.17.6 (but the most recent of Node.js should do fine)
* Obtain a google API key so you can integrate google services (Maps, Places) to your front end app.

## **Back end:**

IDE: PyCharm 2021.2.2

This is an IDE for development of python applications. It has the same benefit as WebStorm. Likewise, you are free to use your preferred IDE. I chose PyCharm as it has the ability to automatically run the code within a virtual environment, thereby not affecting my local device libraries too heavily.

Testing API: Postman Version 9.15.3

This application is used to test out the APIs that are developed in the back end. By doing so, we don't need to have the front end running to test the API, thereby decoupling the development procedure.

Sending Emails: SendGrid API

When users register accounts in Carparkly, we need to send confirmation emails to users so they can verify their accounts. This is done with the help of SendGrid. It provides a way to programmatically create and send emails to users. The guide to do so can be found in the reference links below and it does not cost any money to use it.

Libraries/Packages:

* Python 3.9, Flask Web Framework
* Other required packages in “requirement.txt” of the backend source code

## 

## **Web Scraper:**

IDE: PyCharm 2021.2.2

Libraries/Packages:

* Python 3.9, Selenium Web driver
* ChromeDriver Version 100.0.4896.60: https://chromedriver.chromium.org/downloads

## **Cloud service provider:**

### AWS

I have chosen to implement AWS for this project with regards to data storage and deployment. If you wish to do the same, you would first need to set up your own personal AWS account. It involves providing payment details, however, you are only charged if you utilise services that cost money (For the scale of this project, most of the services I used in AWS were under the free tier).

### AWS DynamoDB:

This is a NoSQL database provisioned and managed by AWS. I used this option for storing and managing data for the web application. The details of the columns within the database are found in the appendix below. You can set up DynamoDB on your local machine or on the via your AWS cloud. I recommend setting it up locally for testing purposes first then replicating that to the cloud. The guide to set DynamoDB locally is found in the reference links below.

### AWS Secrets Manager:

This service was used to store all sensitive API keys. Because you will need to obtain API keys from Google and URA. You may store and retrieve them using this service.

### AWS Lambda:

This service provides the ability to run code serverlessly. Meaning I do not need to think about what type of computing resource I should allocate to run the code. This service allows me to provide the code, and I can schedule it to run at certain intervals.

This service is used to run the code to update the car park info and availability. The code files are provided in the zip file.

One thing to take note of for AWS Lambda, particularly for the web scraping code. There is a need to add “layers” to your AWS Lambda function for the web scraper (there is a link provided at the bottom of this document to understand how layers help). Essentially, layers allow you to install Python libraries that your Lambda function can use.

### AWS ElasticBeanstalk:

This service was used to deploy the Carparkly Backend, providing the APIs for the front end to call. It takes care of the provisioning of resources. And you can configure the application as well (environment variables, number of instances etc). Refer to the link at the bottom for an example guide on deployment to ElasticBeanstalk.

### AWS Amplify:

This service was used to deploy the Carparkly Frontend; the process is straightforward. An example guide on how to deploy is listed in the links below. The good thing about AWS Amplify is that it can sync with your github repository, meaning when you commit a new version of your front end app to github, AWS Amplify can automatically detect that new commit, then build and deploy a new version of your app onto the web.

### AWS VPC:

One important aspect of deployment is managing the network that your apps reside in. You would have to create a virtual private cloud (VPC) and configure subnets to allow for internet connectivity. Guides on how to set up a VPC are listed in the links below. However, further details on how to configure a VPC depends on what configuration you require.

You need to configure your VPC such that the AWS Lambda functions and AWS ElasticBeanstalk app can communicate over the internet (For AWS Amplify it auto manages

internet connectivity so there was no special effort needed).

### AWS Route53:

At this point in time, my back end was running on HTTP while my front end was running on HTTPS (AWS Amplify auto provisions SSL certificate to enable HTTPS but AWS ElasticBeanstalk does not). Therefore to ensure better security I had to upgrade the communication protocol of my back end. To do this, I had to purchase a domain name through AWS, this allowed me to obtain a SSL certificate and for my back end to use HTTPS protocol. Example guide is found below.

## 

## **Car park data sources:**

### HDB:

Car park info: <https://data.gov.sg/dataset/hdb-carpark-information>

Car park availability: <https://data.gov.sg/dataset/carpark-availability>

### URA:

Car park info: <https://www.ura.gov.sg/maps/api/#car-park-list-and-rates>

Car park availability: <https://www.ura.gov.sg/maps/api/#car-park-available-lots>

### LTA:

Car park info: <https://onemotoring.lta.gov.sg/content/onemotoring/home/owning/ongoing-car-costs/parking/parking_rates.1.html>

APIs provided by LTA (not used for this project but you can consider exploring:

<https://datamall.lta.gov.sg/content/datamall/en/dynamic-data.html>

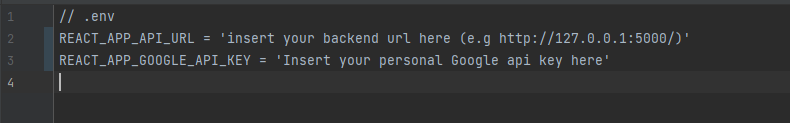
# 

# Setup instructions

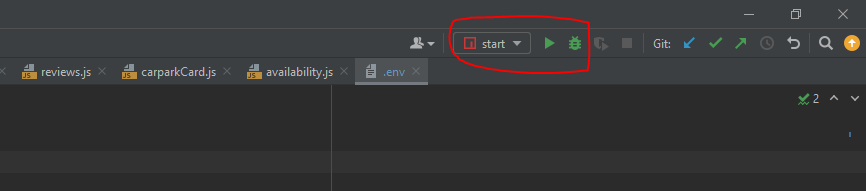
The following instructions describe how to set up and run the relevant code projects locally on your device.

## **Front end:**

1. Import the source code. It is a ReactJS project. The entry point of the project resides in the file “index.js”
2. You will need to create a new file in the root directory of the folder, this file must be called “.env”. It will store your application configuration such as your google api key and the back end api url of your flask application. Refer to the screenshot below for what to put in the “.env” file.



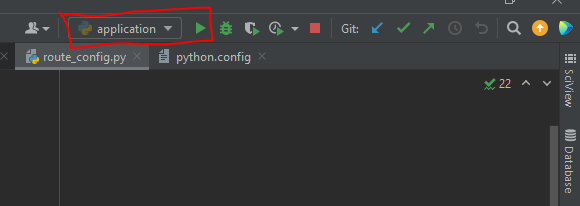
1. To run the project, navigate to the root folder of the project in a command terminal and execute the command “npm run start” (You must have Node.js installed)
2. Alternatively, if you are running the code in WebStorm IDE, you can configure the run configuration in the top right of the IDE. As shown in the screenshot below.



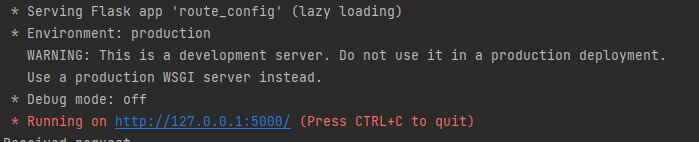
1. After running the code, give it a while to load on your local browser, this takes about 1 minute or so.
2. Access your browser with the following url “<http://localhost:3000/>” and you should be able to see the website loaded.
3. One benefit of React is that it supports “hot reloading”, meaning that you can edit the code WHILE the app is running, and the change will be reflected immediately upon any edits to your code. This is very helpful in saving time when you need to make many minor changes and want to avoid restarting the application.

## **Backend:**

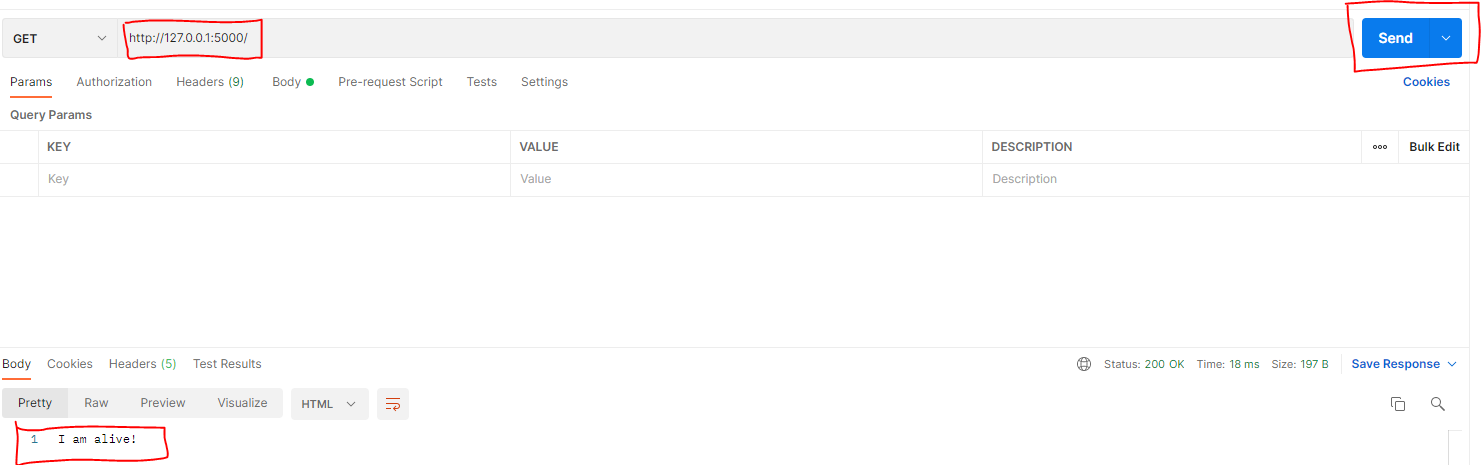
1. Import the source code. It is a Python project. The entry point of the project resides in the file “application.py”.
2. The required libraries to install reside in “requirements.txt” for reference.
3. To run the project, navigate to the root folder of the project in a command terminal and execute the command “python application.py” (Or “python3 application.py” depending on your version of python)
4. Alternatively, if you are running the code in PyCharm IDE, you can configure the run configuration in the top right of the IDE. As shown in the screenshot below.



1. After running the code, in the terminal, it should show what URL the backend api is configured to be on your local device. As per the screenshot below.



1. For example, based on the screenshot above, the backend is running on “<http://127.0.0.1:5000/>”. This can vary based on your device.
2. Now that you have the URL, you can test it by using Postman. Refer to the screenshot below.

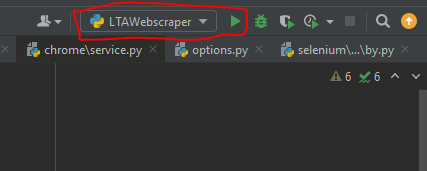


1. This Python application supports a form of “hot reloading” as well. Any changes you make to the back end code, it will automatically detect, and rerun the application to reflect the new changes you made.

## **Web scraper:**

Note: The web scraper project is also run as a Python project.

1. Import the source code. It is a Python project. The entry point of the project resides in the file “LTAWebscraper.py”.
2. Remember to download the ChromeDriver and take note of the directory it was downloaded to.
3. Change the path to the ChromeDriver in line 17 of “LTAWebscraper.py” with the path you obtained in step 2.
4. To run the project, navigate to the root folder of the project in a command terminal and execute the command “python LTAWebscraper.py” (Or “python3 application.py” depending on your version of python)
5. Alternatively, if you are running the code in PyCharm IDE, you can configure the run configuration in the top right of the IDE. As shown in the screenshot below.



1. There are two options for running the web scraper, “headless” or “non-headless”. You can change this option by commenting/uncommenting out line 20 of “LTAWebscraper.py”.
2. “Headless” means that your device will not open a new Chrome browser to let you see the steps taken by the web scraping code. “Non-headless” will open a Chrome browser and this will let you visually observe how your code navigates through the different web pages, allowing for debugging. The reason why some use cases prefer “headless” mode is when you deploy the application, there is no need to view the web pages anymore.

## **To deploy applications on AWS:**

The details to deploy the relevant services on AWS can be found in the “Cloud Service Provider” section under “Tools, framework and library used”.

DO NOTE THAT DEPLOYMENT ON AWS WILL INCUR SOME FORM OF MONETARY COST, PLEASE READ UP ON THE COSTS INVOLVED SO YOU DO NOT INCUR UNNECESSARY FEES.

One tip I have for deployment is try out deploying each service separately, then monitoring the cost incurred daily. This is because there are some services (e.g provision of NAT gateways) that will incur cost hourly as long as it is deployed, even if you are not using it.

By understanding the deployment process for each service better and will not confuse you as you have multiple services to manage. You can then avoid unintended costs.

# Writeup on code

## **Overview of the contents of each folder within the source code**

## **Carparkly Front End:**

File Name: “index.js”

This file is the entry point of the web application. It loads the route for each component. And will call the corresponding component depending on the route specified in the URL.

Folder: “Components”

This folder contains multiple sub folders. Each sub folder contains the code to render a specific page. Some subfolders have further levels of subfolders, this is to ensure that components are broken up into reusable and manageable code.

The following are the main subfolders within the “Components” folder:

* “navigation” - This is the first page that the user sees when the web application first loads, it contains the code for the rendering Google Maps and displaying the list of car parks based on location input
* “carParkDetails” - This folder contains the code to render the car park details page, items like the Google Street View and the review section for the car park can be found here
* “costCalculator” - This folder contains the code to render the cost calculator page. It takes in the car park ID as input and renders the page accordingly.
* “register” - This folder contains the code to render the registration page for user accounts.
* “shared” - This folder contains reusable components such as buttons and titles which are called across different components.

## **Carparkly Back End:**

File Name: “application.py”

This file is the entry point of the back end application. At the top of this file, it imports from another file called “route\_config.py”, and in that file is where the API routes are defined.

File Name: “route\_config.py”

All API routes are defined with the start “@application.route”. The type of http request that is acceptable for that route is also defined. If no request method is specified, it defaults to a GET request.

So from this file you can observe how each API route is mapped to a specific function. Therefore, when the appropriate request is made to a particular route, the back end will call the function associated and return to the caller.

Please refer to the appendix below to see the functions of each API route.

File Name: “requirements.txt”

This file contains all the libraries that were used in this Python app. It is needed for when deploying the app to AWS ElasticBeanstalk. Refresh it when necessary.

Folders:

* “api” - This folder contains the python functions that will be directly called by the API routes. Functions such as identifying the list of nearby car parks when a location is provided.
* “utils” - This folder contains the utility functions used throughout the application. Functions such as packaging the return response, calculating distance between two latitude and longitude values, calculating the cost of stay at a car park etc.
  + “static” - This folder exists within the utils folder, it contains the static data such as the list of public holidays for the year and the list of HDB car parks that are defined as “central” by HDB. These static data are required to calculate the cost of stay at a car park. You would have to manually change them accordingly if there are any updates.
  + I have included the python files from the **Lambda** functions in this folder as well for local testing purposes. “URA\_lambda\_functions.py” and “HDB\_lambda\_functions.py”
* “templates” - This folder contains the html templates for the email verification function. So when users click “verify email” then one of the html templates will; be rendered on the browser for the user to see if they are successful or not.
* “.ebextensions” - This folder contains the config file for when deploying to AWS ElasticBeanstalk. It does not affect any core functionality.

Note:  
For the back end to communicate with your database you have two choices.

1. Set up a local version of dynamodb (guide linked below)
2. Set up dynamodb in aws cloud (This is the “end” goal)

You may choose to set up a local version first to test out the functionality first. Then when your data is more or less ok, you can proceed to create the dynamodb tables via the AWS cloud.

In order to change whether the backend is communicating with the local version or the cloud version of dynamodb, please change the config accordingly in the “utils/utils.py” file.



The function in the screenshot above shows the config for the backend to communicate with the dynamodb tables in the cloud.

## 

## **Carparkly Web Scraper:**

File Name: “LTAWebscraper.py”

This file is the entry point of the web scraping application. It uses the Selenium Python library and requires a “chromedriver.exe” file. I have included it in the source code bundle for the application.

File Name: “scraper\_functions.py”

All utility functions for the web scraper, such as converting the html elements to a list, and using the GeoPy library to find the lat long of an address.

File Name: “DB\_update.py”

Contains the function to upload the data retrieved from web scraping to the DynamoDB table. You may configure accordingly to upload to local DynamoDB or cloud DynamoDB, similar to the back end.

## **AWS Lambda Functions:**

In the Lambda functions folder, it contains the files that are deployed to AWS Lambda for usage. However, should you need to test the functionality locally, you can find the local versions of the functions within the “Carparkly Backend” and the “Carparkly Web Scraper” source code folder.

File Name: “HDB\_Carpark\_Availability”

This file is the lambda function which updates the HDB car park available lots from the API.

File Name: “URA\_Carpark\_Availability”

This file is the lambda function which updates the URA car park available lots from the API.

File Name: “HDB\_Carpark\_Information”

This file is the lambda function which uploads the HDB car park information from the API.

File Name: “URA\_Carpark\_Information”

This file is the lambda function which uploads the URA car park information from the API.

File Name: “LTA\_Carpark\_Webscraper”

This file is the lambda function which uploads the URA car park information from the API.

Note:  
I have included 3 zip files inside a folder called “Layers for web scraper”. It contains the geopy layer, chromedriver layer, and selenium layer. These 3 layers are provided to the web scraper AWS lambda function. The purpose of the layers is to “pre-install” the Python libraries and chromedriver executable so that the web scraper code can have access to it at run time.

# Potentials areas for future development

Here are some areas for consideration that can be developed for CarParkly:

1. Optimizing the web app for mobile usage (currently only desktop experience)
2. Adding in more edge case handling for both front and back end functions (e.g more HTTP error response codes, different inputs etc)
3. Machine learning to predict occupancy of car parks
4. Adding regex search functionality to search specific car parks by name
5. Adding in LTA car park availability data (need to find a reliable source)
6. Integrate google maps navigation into the web app itself (currently it opens a new tab for navigation)
7. Developing a system to inform users of the optimal length of stay in a carpark (e.g, if I stay till 4:05pm instead of 4pm I get charged for the whole hour. So maybe the app can recommend that I leave at 4pm)
8. Integration with parking.sg app so users can pay for parking
9. Provide number of handicap/Electric vehicle charging points
10. Integration with Google assistant/Amazon Echo

# Resource/Literature links

## **Links to tutorials and resources used in this project.**

## **Front end:**

Learning how to code with React: <https://reactjs.org/tutorial/tutorial.html>

This link provides a great tutorial on how to start on React projects and even teaches the main concepts of React. I highly recommend viewing it if you are starting out to learn React.

How to persist a logged in user in react:

<https://www.freecodecamp.org/news/how-to-persist-a-logged-in-user-in-react/>

How to get an API key for google services (free):

<https://cloud.google.com/docs/authentication/api-keys>

How to integrate Google Maps and Google Places into React: <https://www.youtube.com/watch?v=WZcxJGmLbSo&t=0s>

Using Ant design to style your front end web application: <https://ant.design/docs/react/introduce>

The front end of this application uses the Ant design library to style the web pages. I chose this library because it has support for a wide variety of designs and objects. Whatsmore, their documentation is well updated.

## **Back end:**

Learning how to set up a Flask application in Python: <https://www.youtube.com/watch?v=MwZwr5Tvyxo>

This is part 1 of a multi part series. Depending on what you want to create/edit in the Flask app, you can watch different parts.

Learning how to send email verification for newly created user accounts: <https://realpython.com/handling-email-confirmation-in-flask/#flask-basic-registration>

How to use SendGrid API to send emails through your program:

<https://docs.sendgrid.com/for-developers/sending-email/api-getting-started>

Learning how to web scrape with Python and Selenium: <https://www.analyticsvidhya.com/blog/2020/08/web-scraping-selenium-with-python/>

How to set up dynamodb locally:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.DownloadingAndRunning.html>

## **Deployment:**

How to set up a VPC (generic use case):

<https://objectivefs.com/howto/how-to-setup-vpc-on-aws>

How to connect your lambda functions to the internet:

<https://aws.amazon.com/premiumsupport/knowledge-center/internet-access-lambda-function>

How to deploy a web scraper with selenium on AWS Lambda:

<https://www.youtube.com/watch?v=FcW-AXsirBE>

How to deploy Python Flask project to AWS Elastic Beanstalk:

<https://www.youtube.com/watch?v=4tDjVFbi31o>

How to connect your ElasticBeanstalk app to the internet:

<https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/vpc.html>

How to purchase a domain name for AWS:

<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/domain-register.html>

How to configure HTTPS for your AWS ElasticBeanstalk app:  
<https://www.youtube.com/watch?v=pBOcPxho_wg>

How to deploy a React app on AWS Amplify:

<https://www.youtube.com/watch?v=DHLZAzdT44Y>

# Appendix

## **Database Config**

| **Table Name** | **Attributes: Data Type** |
| --- | --- |
| **CarparkDetails** | CarParkID (Key): String  Name: String  Location: String  Agency: String  Rates: Map  TotalLots: Number |
| **Users** | Username (Key): String  Email: String  FirstName: String  LastName: String  Password: Hashed String  Confirmed: Boolean  ConfirmedOn: String |
| **Community\_Reviews** | CarParkID (Key): String  CommentID: String  Username: String  Review: String  Rating: Number  Votes: Number  Date: String  Voted\_Users: List(String) |
| **HDB\_Lots** | CarParkID (Key): String  LotType: String  TotalLots: Number  AvailableLots: Number  UpdateTime: String |
| **URA\_Lots** | CarParkID (Key): String  AvailableLots: Number  UpdateTime: String |

## 

## **API Routes**

| API Route name | Input | Output |
| --- | --- | --- |
| **/api/carparksByDistance** | Location latitude  Location longitude  Max distance radius to location | List of car parks, including availability and estimated cost of parking. List contains a maximum of 10 car parks.  Returned list is sorted according to criteria chosen |
| **/api/carparksByAvailability** |
| **/api/carparksByCost** |
| **/api/calculate** | Car park Id  Time in  Time out | Cost of parking at car park |
| **/api/rates** | Car park Id | Rates of the car park |
| **/api/getReviews** | Car park Id  Username | List of reviews for a car park. |
| **/api/addReview** | Car park Id  Username  Review  Rating | Success response |
| **/api/editVotes** | Car park Id  Comment Id  Username  Vote number | Success response |
| **/api/registerUser** | Username  Email  First name  Last name  Password | Success response |
| **/api/loginUser** | Username  Password | Success response |