BikeBuddy Writing 3

Team Foobarbaz: Claes Boillot, Adham Popal, Ethan Cohen, Matthew Gouvin

Product Description

User Stories

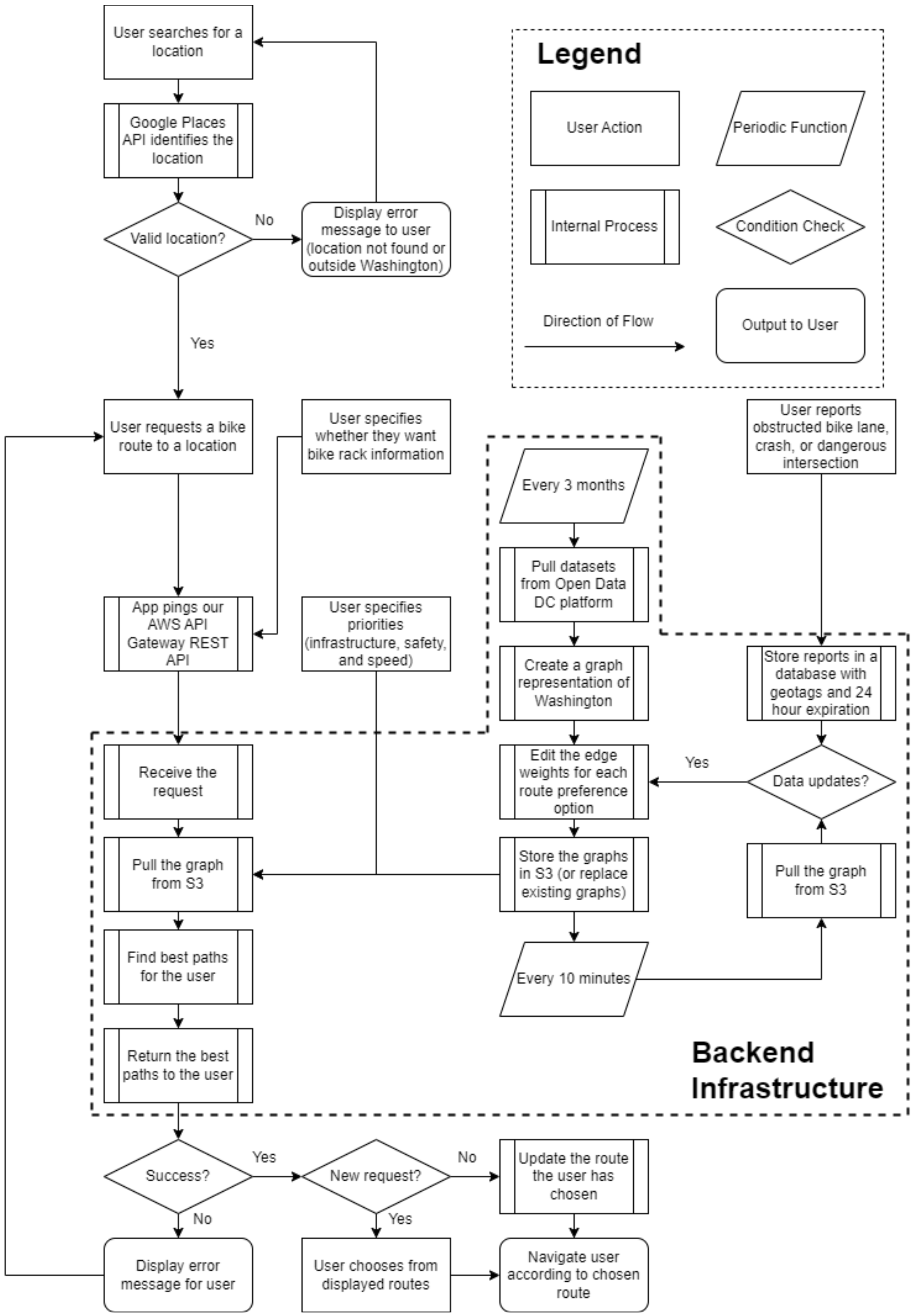
Cyclist / App User	• **As a** user of BikeBuddy, **I would like to** request a route from my current location to a destination **so that** I can cycle to my destination safely while also following an optimized cycling route. • **As a** user of BikeBuddy, **I would like to** search for nearby locations in a search input field **so that** I can find nearby locations that I want to bike to. • **As a** user of BikeBuddy, **I would like to** request a cycling route with input priorities (existing infrastructure, road safety, and speed) **so that** I can get a route that meets my needs. • **As a** user of BikeBuddy, **I would like to** make reports of a blocked bike lane, a cyclist crash, high traffic areas, and a difficult turning spot **so that** I can get a more optimized cycling route in the future.
Engineer	• **As an** engineer, **I would like to** create a specialized graph for biking in Washington D.C. and periodically update and store it.

Engineer (cont.)	• **As an** engineer, **I would like to** ensure that the backend logic of path finding is fault-tolerant and accurate **so that** our app will always be able to function for our users. • **As an** engineer, **I would like to** be able to quantify user preferences for the graph's edge manipulation and the path-finding of route requests. • **As an** engineer, **I would like to** use cloud resources **so that** we can save costs and only use infrastructure when we need it. • **As an** engineer, **I would like to** create a REST API for the application **so that** users can request a route and receive an optimized cycling route that is generated from our route-finding algorithm. • **As an** engineer, **I would like to** create a cron job that runs periodically **so that** we can update the data that is stored in the graph representation of DC. • **As an** engineer, **I would like to** retrieve data from the OpenData DC public API **so that** we can update data that accurately reflects the streets and bike lanes in DC. • **As an** engineer, **I would like to** properly ingest applicable data and clean it properly for graph processing. • **As an** engineer, **I would like to** store user-reported events in a non-relational database **so that** we can use that data as inputs to our path-finding algorithm.

Engineer (cont.)	• **As an** engineer, **I would like to** leverage the Google Places API **so that** users can search for nearby locations. • **As an** engineer, **I would like to** combine OpenDataDC nodes and OSMnx maps **so that** we can create an accurate map of bike lanes in DC. • **As an** engineer, **I would like to** use an AWS Lambda function **so that** we can reconstruct the graph every month. • **As an** engineer, **I would like to** use a DynamoDB database **so that** we can integrate user reports into our map. • **As an** engineer, **I would like to** post our map S3 bucket **so that** we can store and access our map cheaply and easily. • **As an** engineer, **I would like to** use AWS Lambda for routing requests **so that** we can quickly evaluate shortests paths between two nodes.
UI/UX Designer	• **As a** UI/UX designer, **I would like for** my app to be as easy to navigate as possible **so that** users do have any confusion about the functions of the app. • **As a** UI/UX designer, **I would like for** my app to provide an optimal user experience with visuals and elements that enhance the appearance and features of my app.

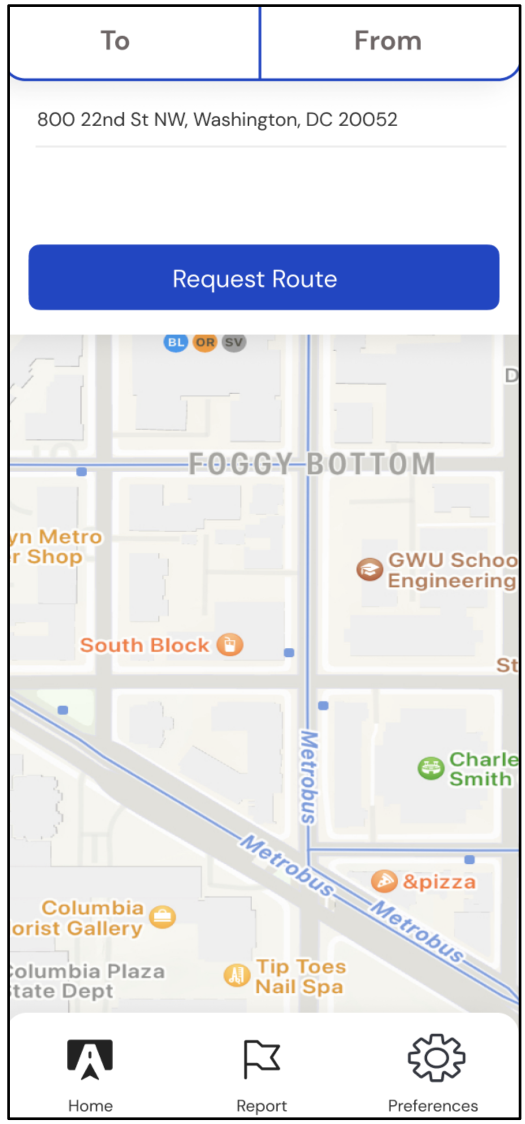
Flow Diagrams

The following diagram represents the key steps the user takes when interacting with our application.


User searches for a location
Google Places API identifies the location
Valid location?
No
Display error message to user (location not found or outside Washington)
Yes
Legend
User Action
Periodic Function
Internal Process
Condition Check
Direction of Flow
Output to User
User requests a bike route to a location
User specifies whether they want bike rack information
User reports obstructed bike lane, crash, or dangerous intersection
Every 3 months
Pull datasets from Open Data DC platform
App pings our AWS API Gateway REST API
User specifies priorities (infrastructure, safety, and speed)
Create a graph representation of Washington
Store reports in a database with geotags and 24 hour expiration
Receive the request
Edit the edge weights for each route preference option
Yes
Data updates?
Pull the graph from S3
Store the graphs in S3 (or replace existing graphs)
Pull the graph from S3
Find best paths for the user
Every 10 minutes
Return the best paths to the user
Backend Infrastructure
Success?
Yes
New request?
No
Update the route the user has chosen
No
Yes
Display error message for user
User chooses from displayed routes
Navigate user according to chosen route

UI Mockups

The following pictures are UI mockups of our application. They have been updated to be consistent with the current design on the application.



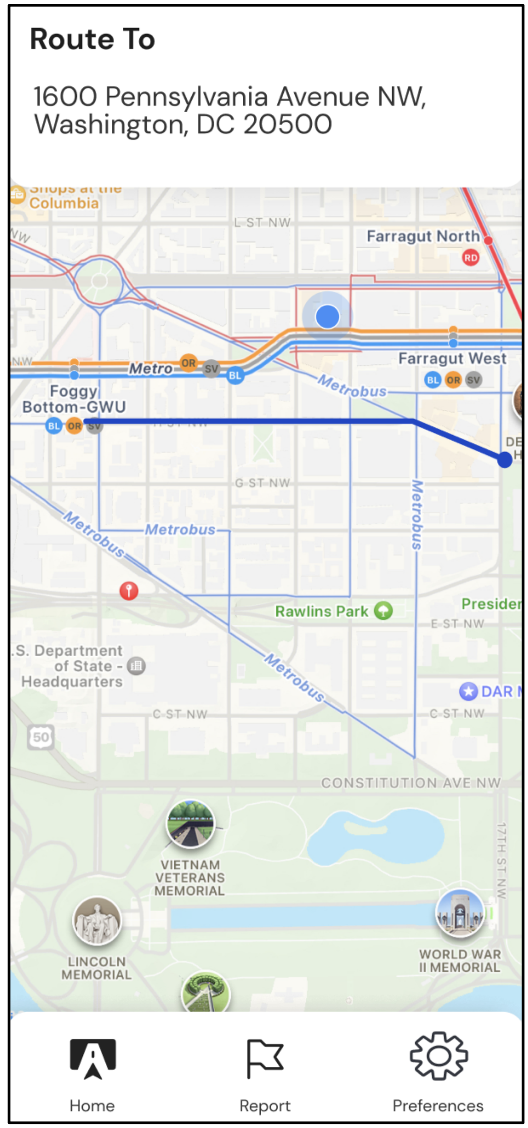
When a user opens the application they will be presented with a home screen that displays a map with their current location. The **Home** tab will be selected by default.

From the home page users will have the ability to request routes from a starting location and a destination location.

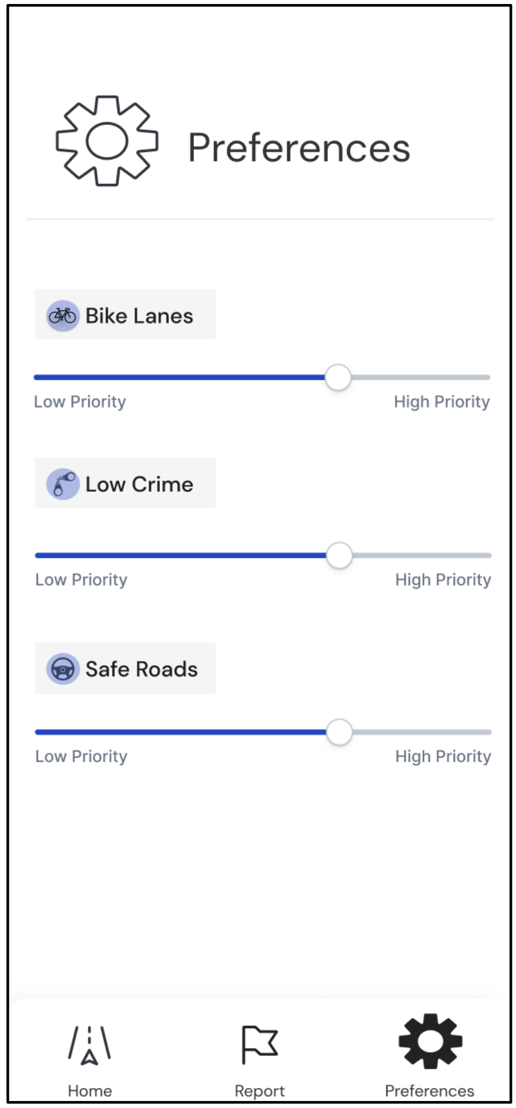
When users type in their locations, a dropdown list of places will be rendered underneath the search bar. These places will be suggested by making requests and receiving responses from the Google Places API. Users will select the place they want to choose at their location. Once they input both a starting and destination location, they can click the "Request Route" button to request a route.

There are also tabs at the bottom of the application's interface to navigate between the three menus: **Home**, **Report**, and **Preferences**.

When a user makes a request for a route the route that is outputted by our algorithm will be rendered on the map. The actual route is displayed on the map interface.

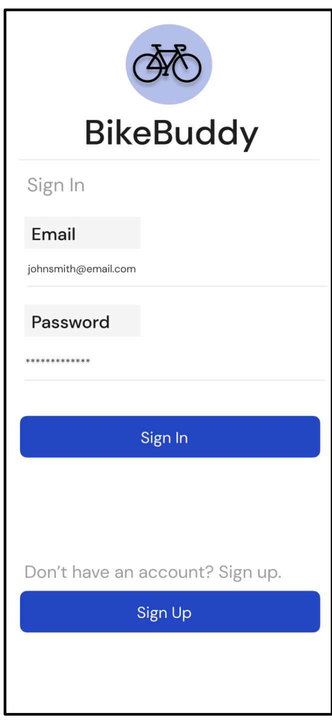


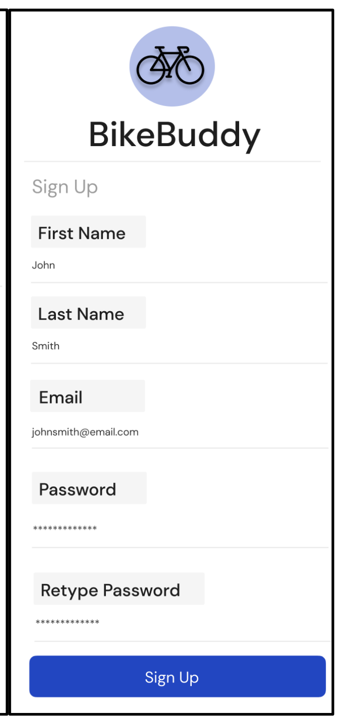
When users click on the **Preferences** tab, a component like the one on the right will render. Users will have the ability to rank the priority of three preferences. These preferences will be used as inputs to the route-finding algorithm.



If time permits, the preferences menu may be transformed into a profile menu for more detailed user customizations.

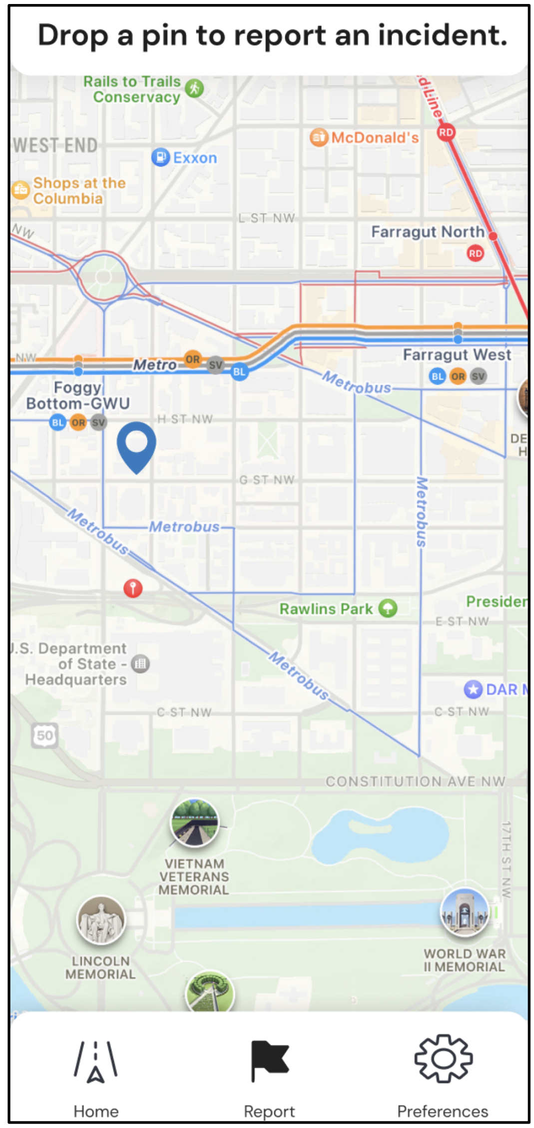
Here is a sign in and sign up menu that would be displayed when a user downloads and opens the app for the first time.





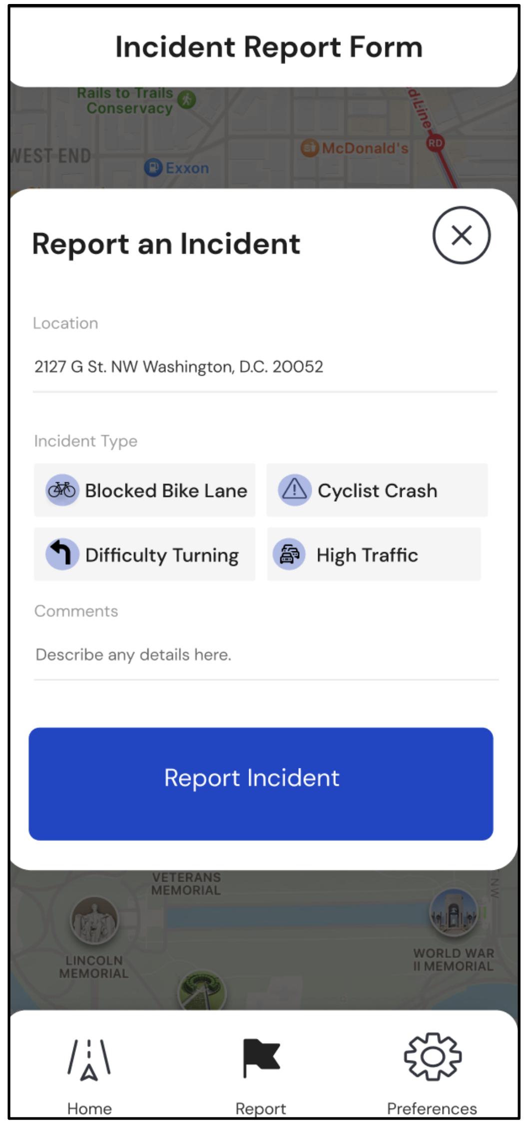
When users click on the **Report** tab they will see a menu that asks them to report an incident. Users will be able to pin a location on a map and report an incident at that marked location.

After they place the pin, a report form pop-up menu will appear.



Users will be able to verify the location, select an incident type from the options that are provided, and they can write additional comments about the incident.

When they would like to report the incident, they can click the “Report Incident” button and their incident report data will be sent to our application’s application programming interface.



REST API Endpoints

There are two resources for our application's REST API: routes and reports.

Specifically, when users request a route from the mobile applications, the application will make a **GET** HTTP request to the **/route** resource with the following query parameters (the * indicates that the parameter is required):

Parameter	**Type**
Source longitude*	Integer
Source latitude*	Integer
Destination longitude*	Integer
Destination latitude*	Integer
Bike Lanes	Boolean
Safe Roads	Boolean
Low Traffic	Boolean
Low Crime	Boolean

This **GET /route** endpoint will return a route (or possibly 2-3 routes) to the application. The format of this response will be an array of coordinate-like JSON objects with the following attributes:

Attribute	**Type**
Longitude	Integer
Latitude	Integer
Bike Lane	Boolean

The array of coordinate-like JSON objects will be able to be displayed as a line (representing a route) on the UI application's map.

When a user reports an incident while cycling, the application will make a **POST** HTTP request to the **/report** with the following body attributes (the * indicates that the attribute is required):

Attribute	**Type**
Longitude*	Integer
Latitude*	Integer
Incident Type*	String
Details	String

This endpoint has no response body.

On success these endpoints will return a 200 status code with their respective response bodies (if applicable). On failure these endpoints will return a 400 or 500 status code.

If we decide to create user profiles, we will have to add two new routes: one for **registration** and one for **authentication**.

When users sign up for an account and click Sign Up on the application, the application will make a **POST** HTTP request to the **/registration** resource with the following query parameters (the * indicates that the parameter is required):

Attribute	**Type**
First Name*	String
Last Name*	String
Email*	String - of type email
Password*	String

On success this endpoint will return a 200 status code and they will be automatically signed into the app. On failure these endpoints will return a 400 or 500 status code with an error message.

When a user already has an account, they will sign into their account. When they fill in their email and password, they will Sign In on the application. The application will make a **POST** HTTP request to the **/auth** resource with the following query parameters (the * indicates that the parameter is required):

Attribute	**Type**
Email*	String - of type email

Password*	String

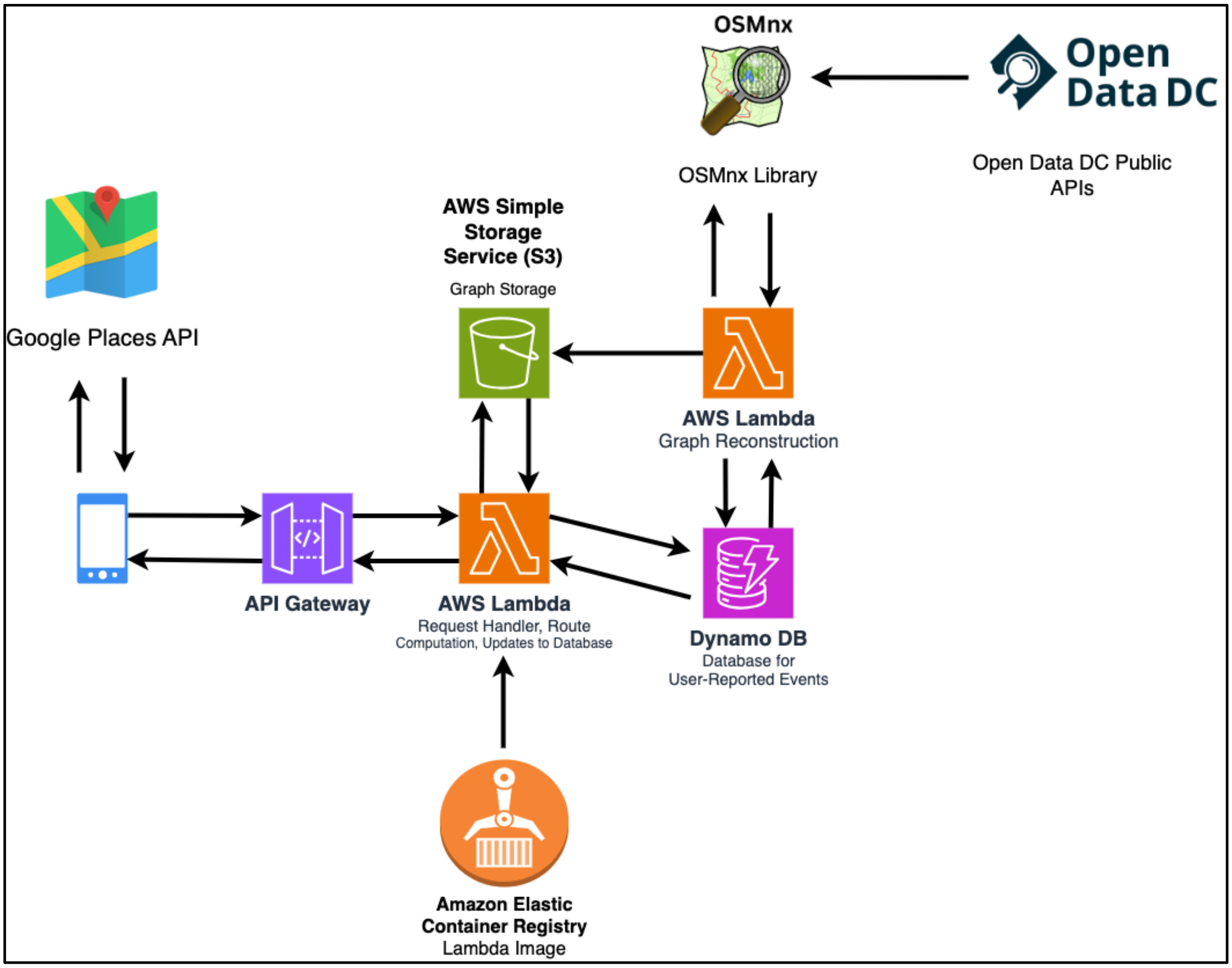
On success this endpoint will return a 200 status code and the users will be signed into the app. On failure these endpoints will return a 400 or 500 status code with an error message.

Technical Specifications

Architecture Diagram

In this section, we will give an overview of the application's architecture as well as providing detailed descriptions of how the different components of our application interact. .

Here is a high-level design of our architecture that is hosted using the Amazon Web Services (AWS) cloud and deployed using AWS's infrastructure as code software called AWS Cloud Development Kit (CDK).



Here is a description of the individual components of our application:

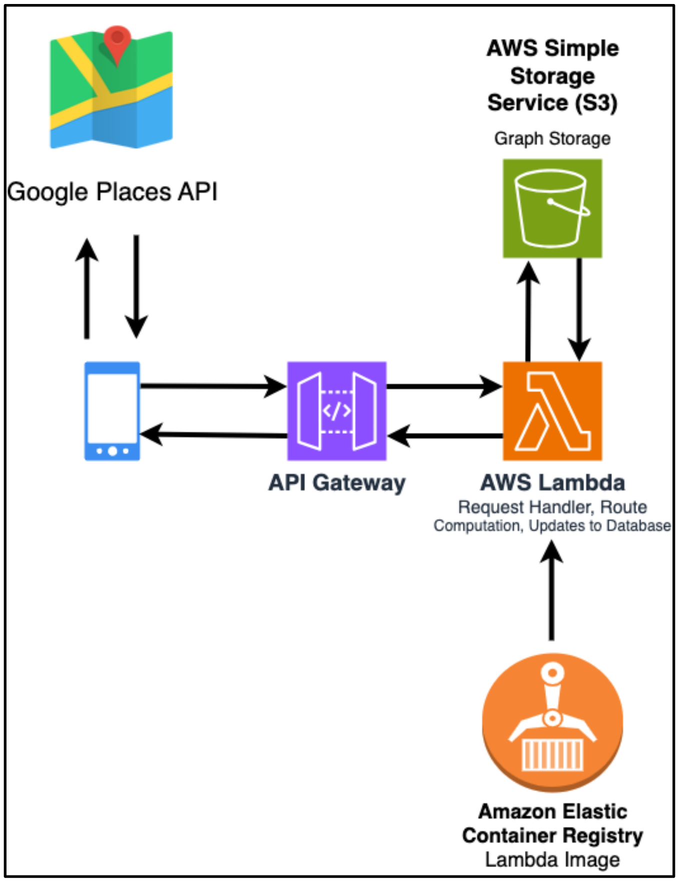
- <u>**Mobile Application**</u>: Our React Native mobile application will allow users to look for nearby locations, request cycling routes from a starting and destination location, report events about bike-lane obstacles and road conditions, and record their user preferences. We can build this mobile application application, publish the built package, and users can download this package so that they can have the mobile application on their mobile device.

- **<u>Google Places API</u>**: When a user is trying to find their source and destination locations, we will make GET requests to the Google Places API. This will allow our application to return nearby locations that match the searched input text.
- **<u>API Gateway REST API</u>**: Our API Gateway REST API will redirect all HTTP requests to a Lambda function. We will refer to this Lambda function as our **App Handler**. This REST API will also direct all responses from the App Handler Lambda function back to the client who made the request. The REST API will be created using Lambda proxy integration; this means that the App Handler Lambda function will handle the HTTP logic that the REST API traditionally handles. .
- **<u>App Handler Lambda Function</u>:** This AWS Lambda function will parse the request that is made by the client and parse the query parameters that might be passed through the request. If a user is requesting a route, this Lambda function will load in the graph from the S3 bucket and perform the shortest-path route-finding algorithm on this graph, given the input parameters. If a user makes a request that needs to update the database — such as reporting an incident, then this Lambda function will handle the logic for updating the DynamoDB table that stores this information.
- **<u>S3 Bucket for Graph Storage</u>**: We have an AWS Simple Storage Service (S3) bucket that stores the graph representation of DC streets and bike lanes in GraphML format. This graph will be updated periodically using an AWS Lambda function that handles graph reconstruction. When a user requests a route, the GraphML file that is stored in this S3 bucket will be read by the Lambda function and the route-finding algorithm will be performed using this graph.

- **<u>DynamoDB Tables</u>**: We will use the DynamoDB tables to report user-report incidents and events that impact cyclists in DC. This data will be used to update the graph that is recreated during the Graph Reconstruction functionality. If we have enough time to implement user profiles, then we will use another DynamoDB table to store user profile information.
- **<u>Graph Reconstruction Lambda Function</u>**: This AWS Lambda function will periodically make automatic updates to the GraphML file that is stored in the S3 bucket. We will refer to this Lambda function as the **Graph Reconstruction** Lambda function. This function will perform graph reconstruction by pulling in data from the Open Street Maps library, the Open Data DC API as well as user-reported data in our AWS DynamoDB database.
- **<u>Amazon Elastic Container Registry (ECR)</u>**: We will use the Amazon Elastic Container Registry to upload an image of a container environment that will be used to create our Lambda function. Our App Handler Lambda function will need to be created using Amazon ECR because we need to ensure that the Lambda function has all of the dependencies that it needs to function properly and perform the route-finding functionality.

We will now give three scenarios that illustrate how components of our infrastructure interact.

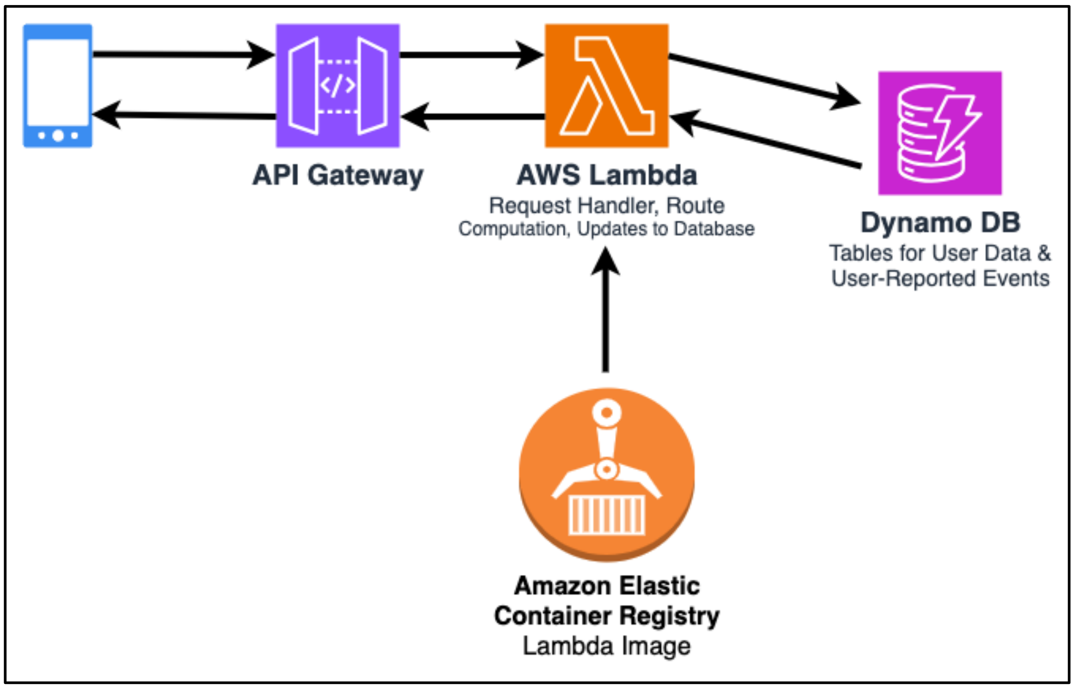
<u>Scenario #1</u>: Requesting a Bike Route



The first scenario describes the operations that are performed when a user requests a route from a starting and ending location. The user will search for the location in the search input field. As a user types in this input field for the starting location, the application will make GET requests to the Google Places API, which returns a list of relevant locations. The user will see a dropdown list of the relevant locations and will click the location that they desire. The user will do the same for the destination location. Once both inputs are selected, the user will click the "Request Route" button which will send an HTTP GET request to our REST API's **/route** endpoint with query parameters that include the location inputs and user preferences that are configured in the Preferences tab of the mobile application.

Since the REST API is configured with Lambda proxy integration, the App Handler Lambda function will handle the GET request to the **/route** endpoint, load the graph from the S3 bucket, perform the route-finding algorithm, and return the output of the algorithm to the client application, which will render the route on the map that is in the application. Notice that the Lambda function is created using the container image that is stored in the Amazon Elastic Container registry.

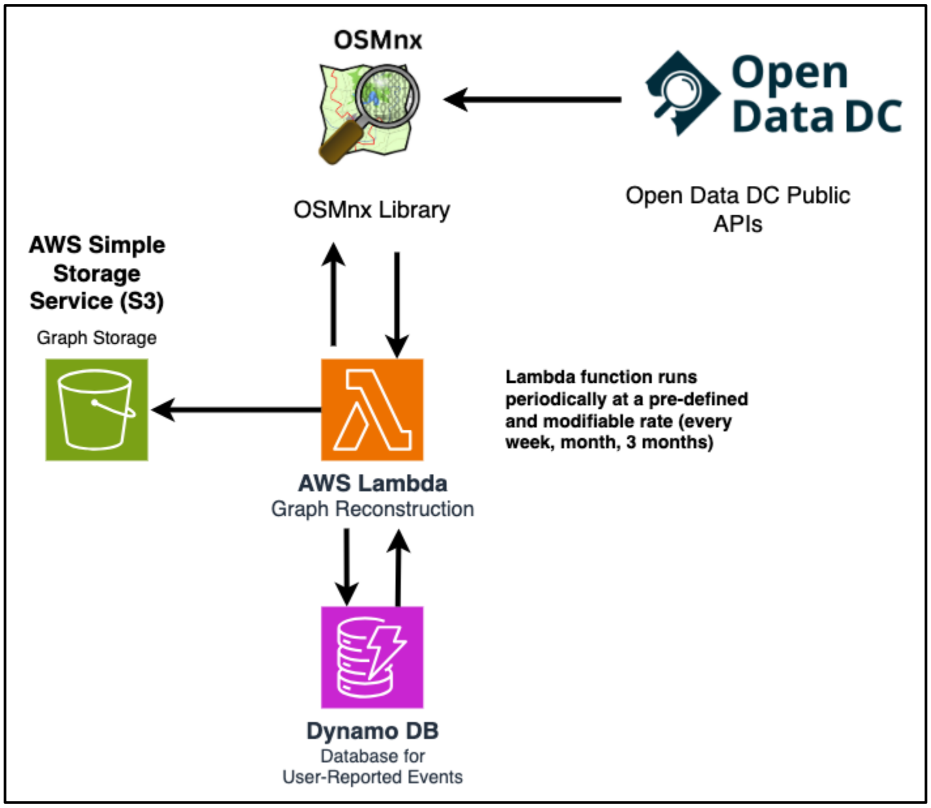
<u>Scenario #2</u>: Creating an Incident Report & Storing User Data



The second scenario describes the operations that are performed when a user makes an incident report. The user will fill out the fields in the incident report form on the mobile application. Then, when the user clicks the "Report Incident" button, the application will make an HTTP POST request to our REST API's **/report** endpoint with a body that includes all of the fields that were in the incident report form on the client application.

Since the REST API is configured with Lambda proxy integration, the App Handler Lambda function will handle the POST request to the **/report** endpoint and the Lambda function will update the DynamoDB table for user-report events accordingly. If we are able to implement the user profile functionality of the application, the operations that are performed when a user registers or logs in will be similar to the operations described here. These functional components will have different resource endpoints (**/registration** and **/authentication**) and the data will be stored in a separate DynamoDB table that stores user profile data.

<u>Scenario #3</u>: Graph Reconstruction



The third and final scenario describes the operations that are performed when the graph is reconstructed. On a periodic basis our Graph Reconstruction Lambda function will be invoked. The period on which the function is invoked can be altered and changed over time; for the purposes of testing, we will be using a three month period. The Graph Reconstruction Lambda will also be invoked when any update is made to the user-reported events table in DynamoDB. This will cause the graph that is stored in the S3 bucket to be updated with different weights that are assigned using the data that is stored in the user-reported events table, the OpenStreetMaps data, and the OpenDataDC data. Once the graph is updated, the routes that are returned from the route-finding algorithm may be different if the weight of the edges in the graph change dramatically.

External APIs and Frameworks

Google Places API	This API from the Google Maps platform will allow us to recommend search recommendations for the location input fields. When a user wants to input a location, they will type in the input text field. As the user types in this field, we will make GET HTTP requests to the Google Places API, which will return a list of nearby places that match the search input. The user will eventually select their desired location and the application will be able to retrieve the latitudinal and longitudinal coordinates for the location. These coordinates will be used in our application's API requests to the REST API that we created.
Open Data DC	This platform provides a wide range of datasets for public use. We pull seven datasets for our own use: bike lanes, bike trails, bike racks, prohibited riding areas, violent crime, car crashes, and Vision Zero (which is a catalog of miscellaneous factors that make a certain street, road, or intersection dangerous). The first four datasets go into creating new nodes for the graph so that we can route users through existing cycling infrastructure, as well as provide them a place to store their bikes when they aren't being ridden. The latter three datasets affect edge weights, as they all represent things that can make a road more dangerous. As all of this data is updated periodically, we also pull periodically to ensure the information we provide to the user is up to date.
OSMnx package	This Python package allows us to fetch a bi-directional graph from OpenStreetMaps that represents a network of city streets and intersections by city or coordinates.

Google Algorithm

Goal: Identify Google's algorithm to use as a comparison and improvement guideline

Description: Google uses Djikstra's algorithm and the A* algorithm for path-finding. The distance, number of turns, and type of terrain determine the path-finding algorithm used which also may include more specialized algorithms than those stated above. Both of these use distance for edge weights as well as a combination of ongoing data such as traffic data to find the shortest paths. Image recognition algorithms are used to create the map and are detailed enough to identify roads, buildings, and landmarks. Machine learning is utilized to identify trends and patterns to further improve the weight modifications that the above path-finding algorithms will run on. These machine learning algorithms ingest data from user data, satellite imagery, and street view images. This is the primary component that helps Google have up-to-date information. Special cameras on vehicles catch 360 degree views of streets and are stitched together to also have an accurate and real representation of areas captured for the maps application. Lastly, Google uses geospatial data from large datasets to accurately show the visuals of the map and use it as consideration in its weight modification. Satellite imagery seems to be the overlying factor in having accurate and up-to-date information for the entire world, as compared to our focus on just the city of Washington D.C. This is where our app can shine by specializing just on DC and this specialization will be the competitive advantage against Google.

Graph Creation Algorithm

Goal: Create a graph that represents the city of Washington D.C. accurately and with a focus on biking specifically.

Description: The main graph that will be generated infrequently (e.g. every three months) is based on static infrastructure and the street layout. This graph is stored and used to service incoming requests as part of the graph modification algorithm detailed below. This graph is unlikely to change as bike lanes do not change quickly, thus it only needs to be rarely updated. First, the OSMnx package provides a base graph "G" of Washington D.C, that is coordinate-based and is simplified with nodes at intersections. "G" needs to be updated with bike-lane specific data that we pull from the OpenData DC API. Some preprocessing is needed to properly modify "G". A new "bike" node needs to be added for every element of each segment. Adding a node requires the deletion of a prior edge, and the addition of two edges upon its addition into the graph. To do this efficiently, the nearest_edges method of the package is too slow.

Two data structures are needed to identify the two nodes that create an edge in graph "G". The first data structure is a cKDTree which takes in a list of points (x and y coordinates) for its creation. The tree can then be queried with any inputted point and will return the nearest point for the inputted point in O(log(n)) time. The points inserted into this data structure are "midpoints" that are generated for each edge in "G" by interpolating a LineString object that represents two nodes for that edge. The second data structure is a list of dictionaries where each dictionary contains the data for each prior mentioned midpoint and the nodes that correspond to the original edge in "G". The returned result from the query of the tree is the index of the list that was used to create it. The indices of the second data structure line up with the indices of the plain list used to create the cKDTree. Thus, the tree can be queried to find the nearest point of a coordinate from our API call. This point is actually used to pull the edge from the dictionary of the second data structure in O(1) time. This is because the index returned by the cKDTree can instantly locate the applicable dictionary that has edge information needed.

Data can now be properly used by finding the nearest applicable nodes for each node to be added based on a coordinate read in. Data is cleaned and put into RoadNode objects that contain the latitude and longitude coordinates for a line segment that represents a bike lane from the API. RoadNode objects are organized to create a list of linked lists, where each list represents a bike lane segment with adjacent nodes linked to each other. Here, a graph named "G_Copy" will add all bike nodes. Each coordinate from each RoadNode object is queried with the cKDTree and looks up the two nodes for the edge it will affect. The node is added based on its coordinate; the edge for the original two nearest nodes is removed. The euclidean distances between the new node and the two nearest nodes that it interjects are calculated and the two edges are added with those distances. This process is repeated for every coordinate of every RoadNode object.

G_copy now has all bike nodes and nodes from the package itself. The final graph is obtained by overlaying certain nodes from G_copy onto the original graph "G"; this reduces the sheer amount of bike nodes that were obtained from the OpenDataDC dataset. The nodes that make up each edge from "G" are iterated upon and the shortest_path function from those nodes is called on G_copy. A list of nodes that start and end with OSMnx nodes and with internodes being bike lane nodes is reduced to a list of 3 or 4 nodes. These are then added to "G" with simple edge removals and additions upon addition of applicable internodes. Finally, a mathematical operation is used to modify the "length" attribute of these edges that are now confirmed to be bike lanes to bias bike lanes for user-requests.

Graph Modification Algorithm

Goal: Modify the edge weights of our stored graph from the algorithm above that is specific to the user's route request and preferences. This modification will influence the actual paths returned to a user.

Description: With a properly created graph, this graph now needs to be able to modify edge weights before running Djikstra's shortest path algorithm to actually service user requests. Unlike the Graph Creation algorithm which modifies certain weights on generally static data (bike-lanes), this algorithm is used to service specific route requests. Within their settings, users will be able to configure their preferences for route features; specifically, users will be able to choose how they want the app to prioritize certain qualities over others (i.e. preferring a route with fewer crashes or being comfortable with painted bike lanes). This dynamic information means different users can request routes with the same endpoints and get different routes. Each edge in the graph is modified according to this dynamic information such that there is a separate graph for every possible configuration of settings in the application. Those modifications could either be a positive change (making the weight smaller) or negative change based on the data and configuration. At the end, each graph is stored separately in S3, so that when the user makes a request, the app can download the file that matches their settings configuration. Finally, k_shortest_paths is performed on the graph corresponding to the user's configuration to return multiple paths that conform to the user's preferences.