# Introduction

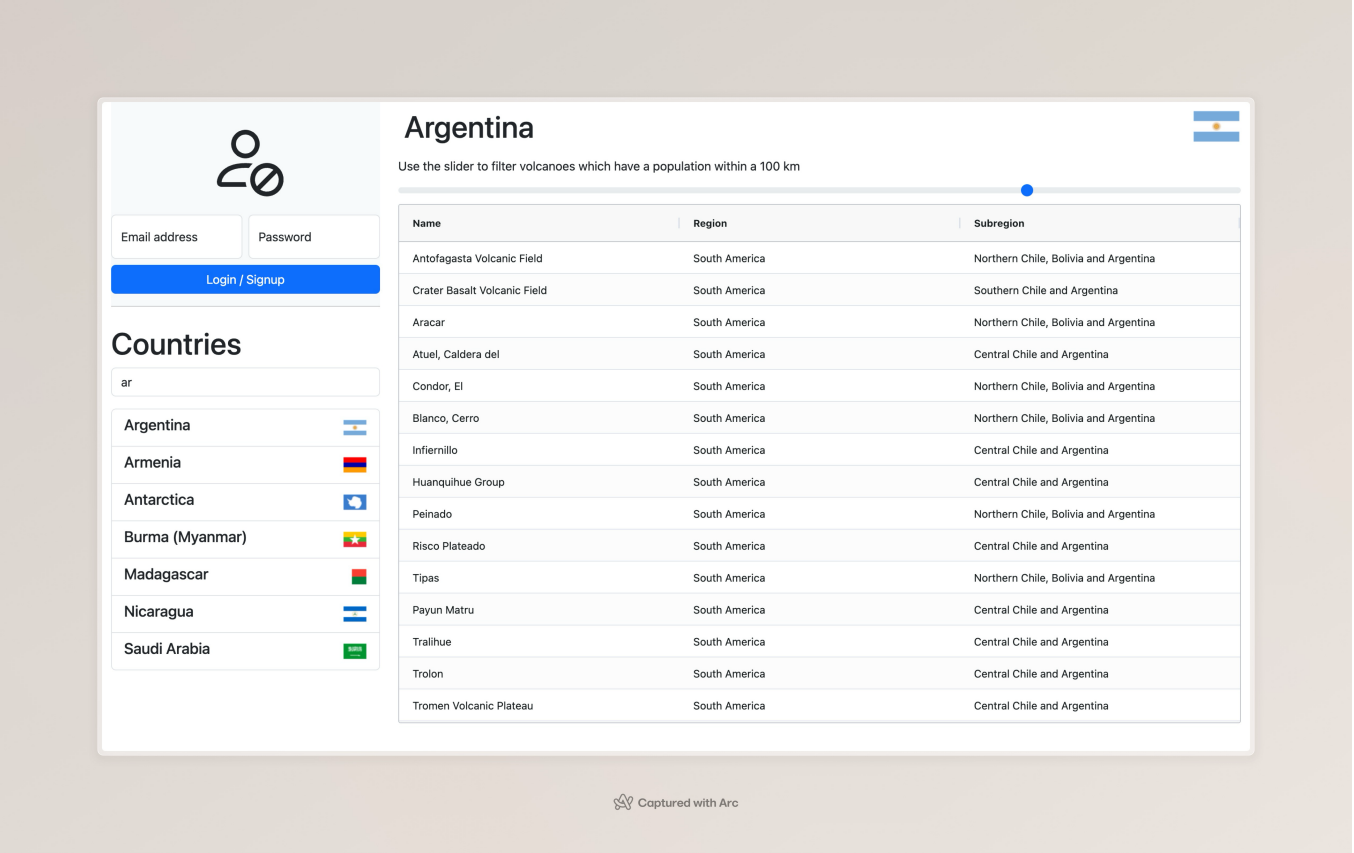
This web application is a single page React website which is designed to display information about various volcanoes. Each volcano holds many datapoints such as its last eruption, the summit height, its population centers, and many others. These volcanoes are sorted by their country.
The volcano information is served by an API with five endpoints. Three of these endpoints return information about countries and volcanoes and are used to source information. The other two endpoints are used to create user accounts and log into user accounts.

The goal of this web app is to display all of the information returned from the API endpoints, utilize all of the path and query parameters of these endpoints, all without the user being aware the website is making requests to the API. To do this, the web app uses appropriate native and third party components, asynchronous programming, and modern routing to allow the user to use the website however they need.

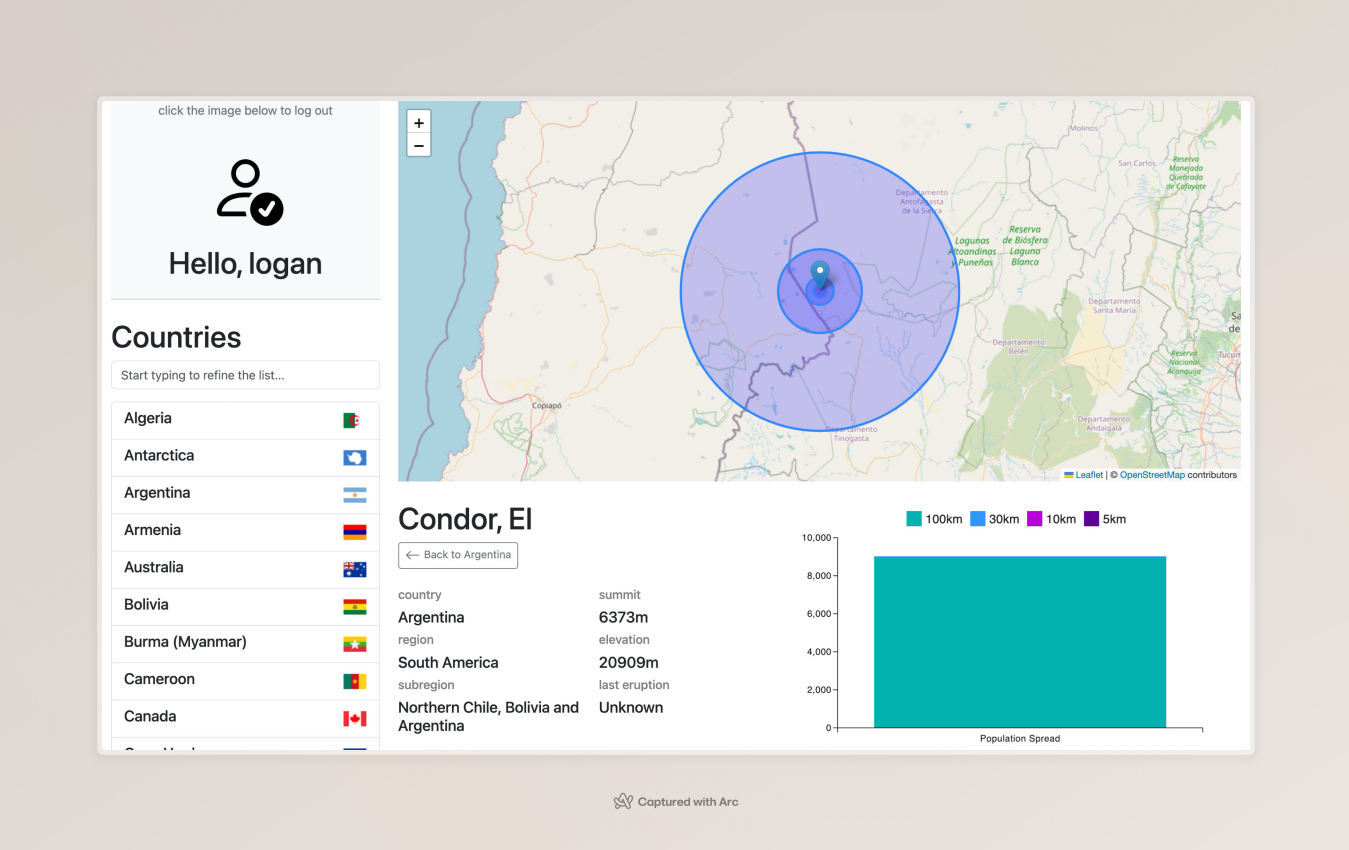
The web app that I developed is made of of two sections, a left panel which the user will always see and a right panel which can display different components based on what the user has selected.
The left panel acts as a nav bar mounted to the side of the screen but as it is vertical, this allowed me to improve the design over a row of text. The top of the left panel houses the user login/signup actions. I have combined the log in and sign up flows into one resulting in the user only have to enter their email and password into one form and the web app taking care of the rest. Below this is the country list. This is a list of countries provided by the API and they extend below the bottom of the page in a scrollable list. At the top of the countries is a search bar to allow the user to filter for a country. Next to each country in the list is an image of its flag which adds to the design of the page. I chose this design with a searchable list over a drop down as I thought it both looked better and allowed for the user to quickly identity their desired country with the flag image.
The right panel is fixed to the height of the page, and can either display a table of volcanoes or a single volcano page view. The table of volcanoes is a Ag Grid Table which allows the user to perform powerful client side filtering and sorting. The use is also able to further filter the volcanoes with a population slider. This uses a query parameter of the volcanoes endpoint to only return volcanoes with a population center within a certain radius. Once the user clicks on a volcano, they will see a single page view of the volcano. This page shows all the information of that volcano as well as a map. If the user is logged in their will also see population data in both a map radius overlay and a bar chart.

I chose to develop this web app using TypeScript. In a full time C# developer so type safety is ver familiar to me. I also believe types prevent a lot of errors when developing a React app across different components and routes.

> User logged out, searching for Argentina with 'ar', filtering volcanoes with a population within 100 kms



> User logged in displaying hello message, viewing single volcano page with population data as overlays and chart




## Completeness and Limitations

Overall I believe I have developed a well thought out and easy to use React application. The user is able to see all information they need to at once, and navigation is smooth and simple. All the endpoints and their parameters have been utilized. All components work as they should. As shown in the bug report below, there is only one outstanding bug remaining in the code. However this does not effect general functionality or presentation.

This is a small React application broken into two sections, a left and right panel. The app uses TanStack Router as its routing solution. This allows the user to reload the page or copy and paste the url into a different browser and pick up where they left off. The left panel is displayed from the 'root' route and TanStacks 'outlet' shows information in the right panel. Navigation is intuitive as the route tree is only two levels deep. Back buttons are clearly disable, and the components used are fit for purpose.

The components I have chosen are always designed for the information they either display or expect as an input. Two examples of this are the country list and the volcano population filter. The available countries are returned as a list, so they are displayed as a list. This allows me to display an image of the countries flag next to the name to make it easily identifiable. This also allows client side filtering of the list with plain text for the user to further refine their search. None of this would be possible with a simple drop down selection. This also allows the user to navigate to a different country just by clicking on it, rather than finding a back button to return to the country selection and then searching through a small drop down list. The volcanoes endpoint has a query parameter which allows for filtering based on populations within a radius. This parameter only allows four options, so instead of allowing plain text and having to account for incorrect inputs, I implemented a range slider. This slider is mapped to the allowed values and the user can slide it left and right to change the results. Other information could be displayed multiple ways such as the population information. I chose to display this information with a map overlay to show the user the actual radius distance and a bar chart to show the relation between the different population numbers.

User registration is handled with a single form. This form allows a user to log in to the web app or first create an account and then be automatically logged in. The code handles this with a single method on the Volcano Api Client class. This method returns a bearer token regardless if the user has an existing account or not. This bearer token lasts 1 hour, and the class is able to detect for this expiry and request a new one if needed.

I use Ag Grid React to perform client side filtering and sorting rather than putting more strain on the endpoints. The radius query parameter filtering could also have been performed on the client side if more information was provided in the volcano list but there is always a trade off between api hits and api latency.

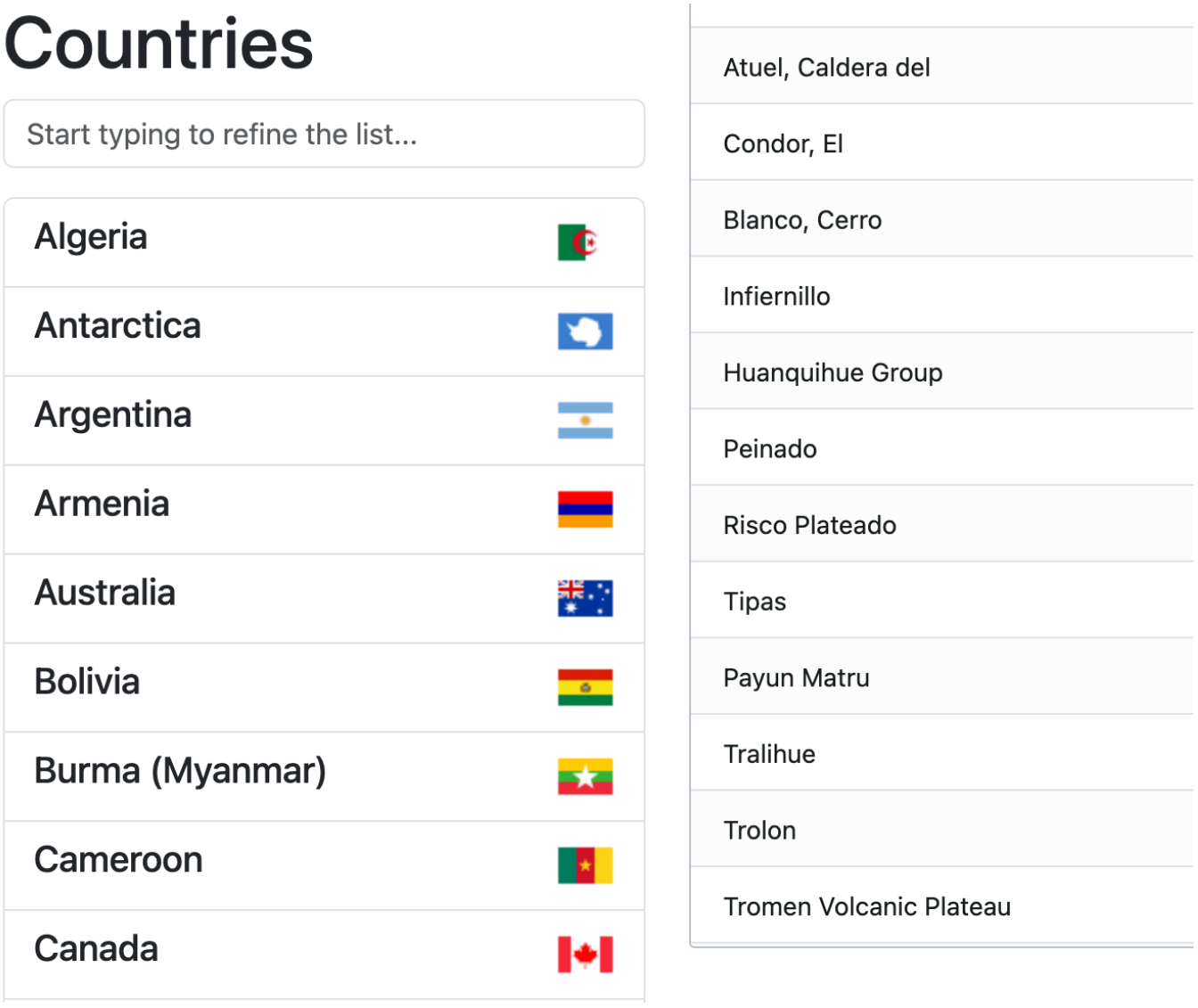
# Use of Endpoints

I built a VolcanoApiClient class with methods to handle requesting data from the different endpoints. This gave me a central place to deserialize the json data to interfaces and ultimately other classes in some cases. This also allowed me to roll up the sign up and log in flows into one method which returns a bearer token.

> as all endpoints are displayed in a single page, I will crop the screen shots to show the relevant section of the screen with some context

## /countries

The countries endpoint is used to populate the list of countries in the side panel. This endpoint returns a list of strings, so no deserialization was needed. The list of countries is then filtered on the client side.




```
public async getCountries() {
        const response = await fetch(`${this.baseUrl}/countries`);
        const data = await response.json() as string[];
        return data;
}
```

## /volcanoes

The volcanoes endpoint is used by the getVolcanoes method in the volcano client.

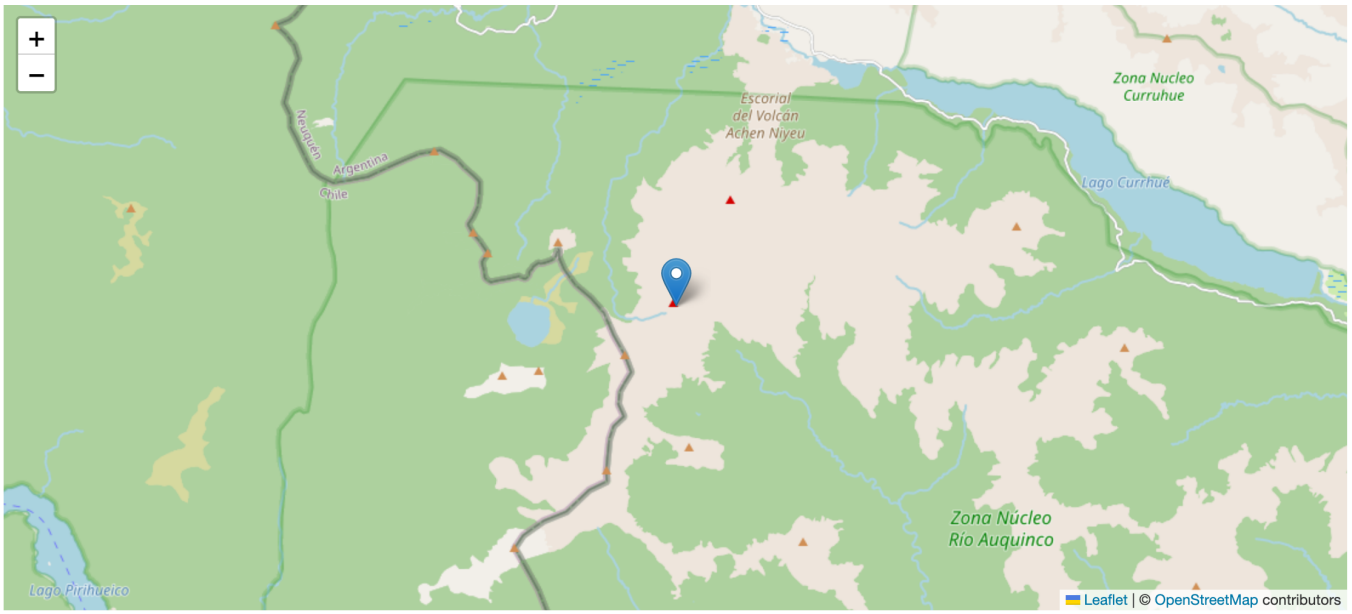
| Name | Region | Subregion |
|---|---|---|
| Antofagasta Volcanic Field | South America | Northern Chile, Bolivia and Argentina |
| Crater Basalt Volcanic Field | South America | Southern Chile and Argentina |
| Aracar | South America | Northern Chile, Bolivia and Argentina |
| Atuel, Caldera del | South America | Central Chile and Argentina |
| Condor, El | South America | Northern Chile, Bolivia and Argentina |
| Blanco, Cerro | South America | Northern Chile, Bolivia and Argentina |
| Infiernillo | South America | Central Chile and Argentina |
| Huanquihue Group | South America | Central Chile and Argentina |
| Peinado | South America | Northern Chile, Bolivia and Argentina |
| Risco Plateado | South America | Central Chile and Argentina |
| Tipas | South America | Northern Chile, Bolivia and Argentina |
| Payun Matru | South America | Central Chile and Argentina |
| Tralihue | South America | Central Chile and Argentina |
| Trolon | South America | Central Chile and Argentina |
| Tromen Volcanic Plateau | South America | Central Chile and Argentina |

```
public async getVolcanoes(country: string, radius: number) {

        this.setRadiusFilter(radius);

        let queryParameterString = `?country=${country}`;
        if (this.radiusFilter !== "") {
                queryParameterString += `&populatedWithin=${this.radiusFilter}`;
        }

        const response = await fetch(`${this.baseUrl}/volcanoes${queryParameterString}`);
        const data = await response.json() as IVolcano[];

        // Convert the data array to an array of Volcano objects
        const volcanoes = data.map((volcano) => {
                return new Volcano(volcano);
        });
        return volcanoes;
}
```

This method can accept a radius filter from either the class or as a parameter on the method. The json is deserialized to an interface called IVolcano. This interface is used in the constructor of the Volcano class. The list of volcano classes is returned.

## /volcano/{id}

The get volcano by id endpoint is used by the getVolcanoById method. This method returns a single volcano as a volcano class type. If the user is logged in but the returned data does not contain population data, the method will call the getToken method to refresh the expired bearer token and then call the endpoint again. If the returned data does not contain population data after five retries, it will fall through to the unauthenticated api call and return that volcano.

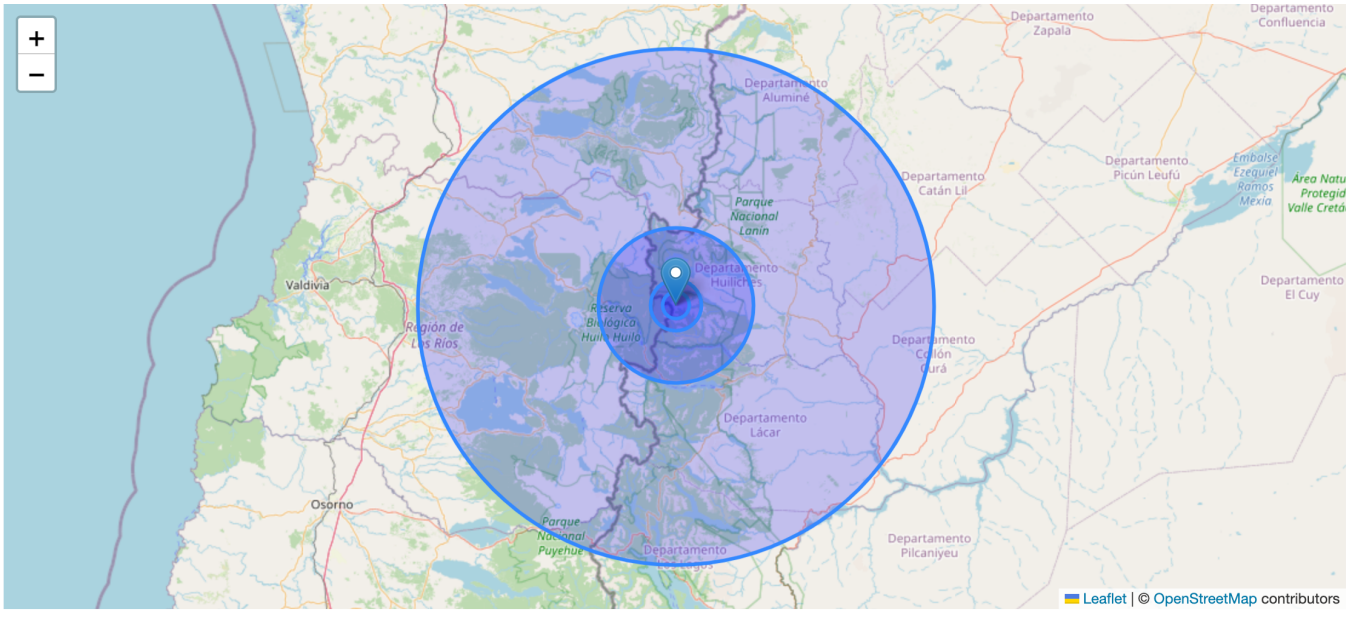



# Huanquihue Group

← Back to Argentina

country
**Argentina**
region
**South America**
subregion
**Central Chile and Argentina**

summit
**2189m**
elevation
**7182m**
last eruption
**1750 CE**

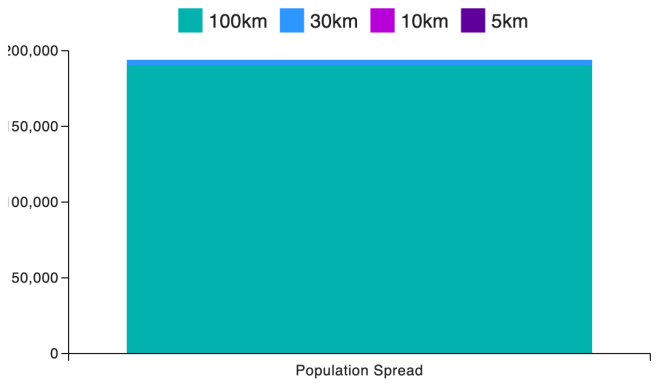



# Huanquihue Group

← Back to Argentina

country
**Argentina**
region
**South America**
subregion
**Central Chile and Argentina**

summit
**2189m**
elevation
**7182m**
last eruption
**1750 CE**

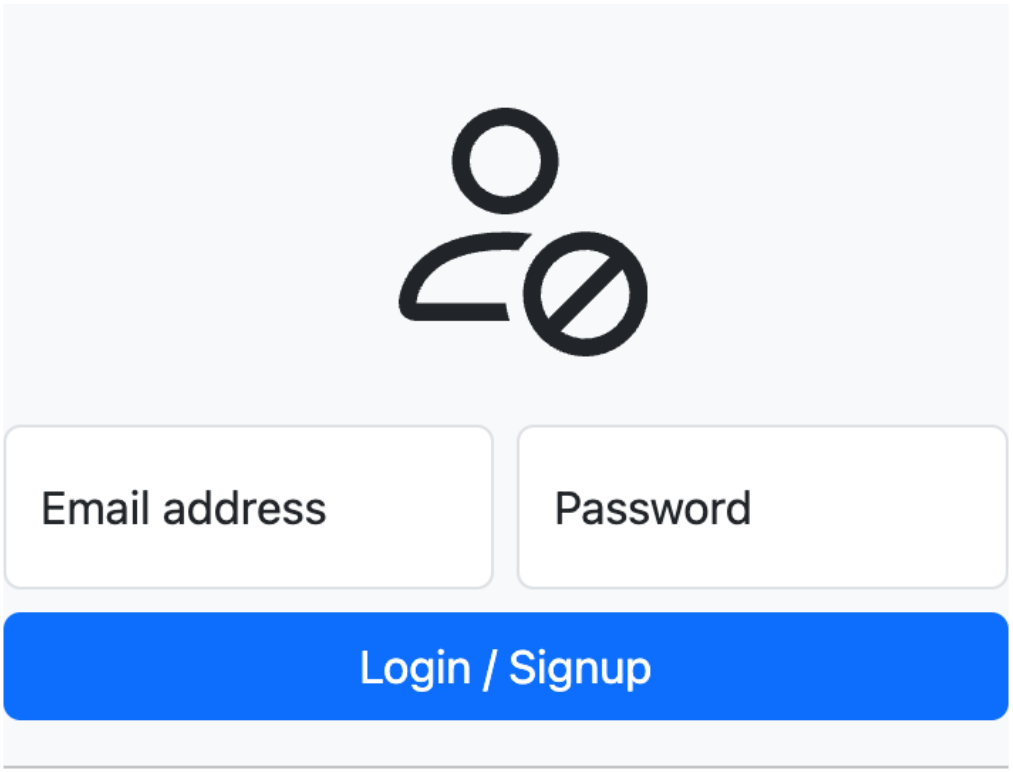


```
public async getVolcanoById(id: number) {

        let continueLoop = true;
        let loopCount = 0;

        if (!this.loggedIn) {
                continueLoop = false;
        }

        while(continueLoop) {
                loopCount += 1;

                if (loopCount > 5) {
                        continueLoop = false;
                }

                const response = await fetch(`${this.baseUrl}/volcano/${id}`, {
                        method: 'GET',
                        headers: {
                                'Authorization': `Bearer ${this.bearerToken}`
                        }
                });

                if (response.status === 401) {
                        console.log("Token expired, getting new token");
                        this.bearerToken = await this.getToken(this.username, this.password);
                } else {
                        const returnedData = await response.json() as IVolcano;
                        const volcano = new Volcano(returnedData);
                        return volcano;
                }
        }

        const response = await fetch(`${this.baseUrl}/volcano/${id}`);
        const returnedData = await response.json() as IVolcano;

        return new Volcano(returnedData);
}
```

## /user/register

## /user/login

In my app, the user is able to log in or sign up via the same form. This form calls the getToken method of the volcano client which handles the creation of the user and logging in.
The client has private register and logIn methods which the getToken method calls. Each of those private methods first deserialize the json into interfaces and those interfaces are used in the constructor of the RegisterResponse and LoginResponse classes. These classes have methods to surface important information such as .hasError() and .userCreated(). This makes the getToken method flow easier to understand.




```
public async getToken(username: string, password: string) {

        const registerResponse = await this.register(username, password);

        // If there was an error registering the user, return an empty string
        if (registerResponse.hasError()) {
                console.log("There was an error registering the user");
                console.log(registerResponse.message);
                return "";
        }

        // Some logging
        if (registerResponse.userAlreadyExists()) {
                console.log("User already exists");
        } else if (registerResponse.userCreated()) {
                console.log("User created");
        }

        const loginResponse = await this.logIn(username, password);

        // If there was an error logging in, return an empty string
        if (loginResponse.hasError()) {
                console.log("There was an error logging in");
                console.log(loginResponse.message);
                return "";
        } else if (loginResponse.invalidCredentials()) {
                console.log("Invalid credentials");
                return "";
        }
```

```
        const token: string = loginResponse.getToken();

        return token;
}
```

I chose to roll these endpoints into one method because the /register endpoint returns helpful information which can be used regardless of if the user exists or not. The user is then logged in.

# External components

1. React Leaflet
   Leaflet is a Javascript mapping library which has been ported to React with React-Leaflet. The React port is the library I use. Its a simple mapping library with map overlays.
   - React Leaflet https://github.com/PaulLeCam/react-leaflet
   - Leaflet https://github.com/Leaflet/Leaflet
2. Ag Grid React
   Ag Grid React is a powerful chart and table library. I used this for the list of volcanoes.
   - Ag Grid React https://github.com/ag-grid/ag-grid
3. Bootstrap
   I used Bootstrap for styling.
   - Bootstrap https://github.com/twbs/bootstrap
4. Flags API
   This is a simple api which returns an image of a flag. The ISO country code is included in the request uri and the image is returned. This is really fast.
   - Flags API https://flagsapi.com
5. TanStack Router
   I used TanStack Router instead of the native React Router. TanStack supports both code based routing and file based routing, and I chose to go with file based. I chose TanStack because it is very modern and simple to understand and adopt.
   - TanStack Router https://github.com/tanstack/router

# Application design and usability

## Design and layout decisions

My single page web app is comprised of a left and right panel, with the left panel being much thinner than the right. The left panel containers user information and actions at the top with a searchable, scrollable list of countries at the bottom. While the right panel can either show a table of volcanoes or a detailed single volcano view.

I made the decision to house the entire page in a single view as this is a very simple web app that doesnt need to move the user around pages. The user is always able to see the left panel with the list of countries and user information, and this acts as a sort of 'home' for the user. They are always able to make actions related to their user and switch countries regardless of where they are in the app. The user will never get lost in a deep tree of page navigation flows.

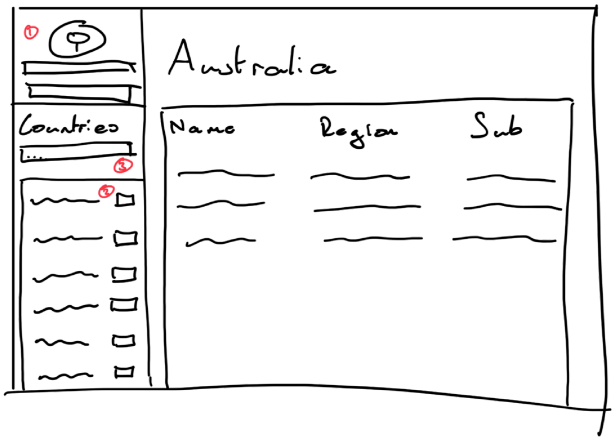
The left panel has an icon at the top to quickly show if the user is logged in or not. If the user is not logged in, they will see a combined login/sign up form with a action button below. When they enter their email address and password, the app will first check if they have an existing account. If they do, it will log them into that account, if they dont, an account will be created and then they will be logged into that. I made the decision to combine the log in and sign up forms to save space in the app. As I made the decision to have all information visible on the screen at once it was important that each component did not take up more space than was needed. Once the user is logged in, they will see a slightly different icon and a welcome text with their name.

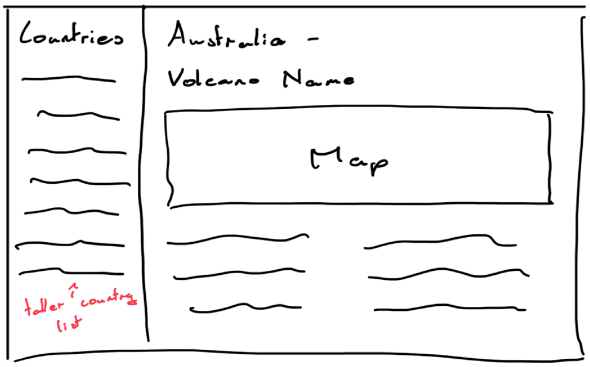
Below the user information component is a searchable and scrollable list of countries. I wanted the list of countries to be showcased, large, and easily searched. This was done to help the user find the country they are looking for quickly. The other options were a drop down list of a free text box. The drop down list would make the countries hard to find in small text and the free text input would have to account for spelling errors and missing countries. I also display an image of the flag of each country next to their name in the list for added visual interest.

The right panel either shows a table with a list of volcanoes, or a detailed view of a single volcano. The table utilizes the Ag Grid React library which made is fast and easy to make a nice looking table. There is a horizontal slider which acts as a range input for the volcanoes endpoint's range filter query parameter. The levels of this range slider are mapped to the values accepts by this query parameter, and the final level removes the query parameter all together. This eliminates the need for error handling that a plain text input field would need for this application.

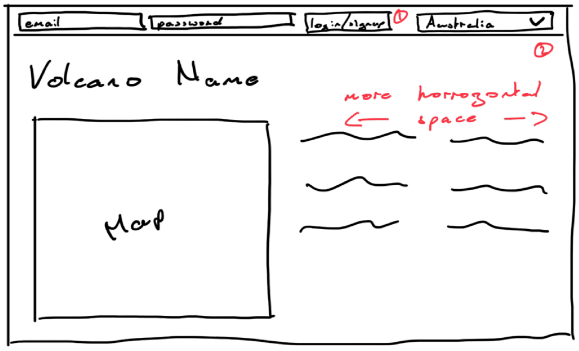
When the user clicks on a volcano, they are presented with a detailed view. This detailed view takes the same position on the screen as where the table was. This single view shows a map with a marker of where the volcano is as well as some other information below the map. If the user is logged in they will see a bar chart displaying population information within certain radii of the volcano. The map will also show an overlay of these radii to give the user more context.

During the design faze of this web app, I explored having a top nav bar instead of a side panel. This would give the grid and single page volcano components more horizontal space, but I would have to scarifies the list of countries which I wanted to retain. In this alternate nav bar design, the countries list would be hidden in a drop down selection. I also explored housing the log in and sign up pages in separate pages all together, without being combined. I ultimately did not go with this decision when I was able to combine the logic into one method.





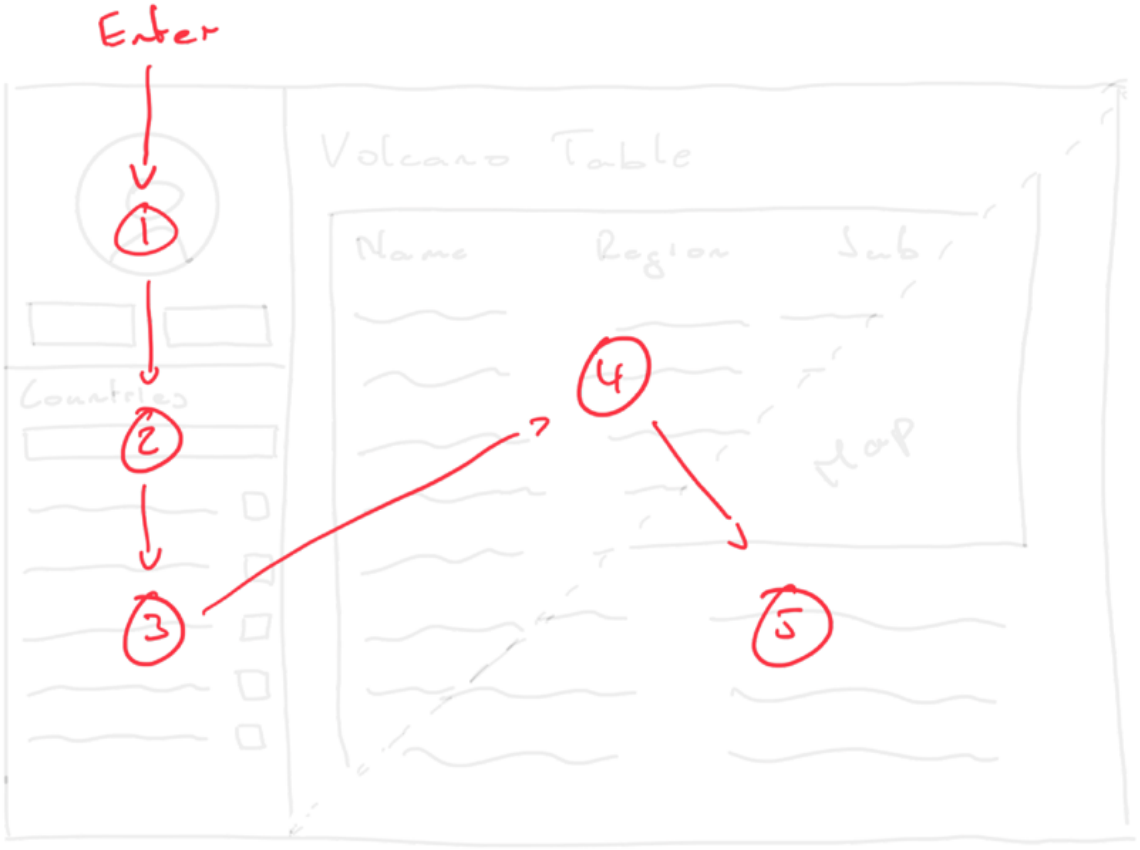



① Combined log in / Sign up
② Country list drop down

## Usability and navigation flow

My single page web app is simple and easy to use due to its persistent side panel and uncluttered layout. When the user see the web app for the first time, their eyes move around the page in order of the buttons they have to click. They first could decide to log in or create an account. This information is presented in the top left of the screen where their eyes start. They then have to decide which country to search within and their eyes are drawn to this list as it is right below the user information screen in the left side panel. After they select a country, their eyes will be drawn to the right side of the screen when the table of volcanoes is shown. After they pick a volcano, the same area of the screen they are looking at is replaced with their selection; the single volcano page. From the single volcano page, the user can either return to the list of volcanoes with a back button, or switch countries by using the side panel.

Their eyes naturally flow to different components in the order of decisions they have to make. They first see the log in screen, then the country search field, then the country list, then the volcano table, then the single volcano page.




## Accessibility

### Pros

1. The app layout looks fantastic on my 14" MacBook pro. The layout also responds well to larger screens like external monitors and smaller screens when I have Arc's dev tools open.
2. The app uses Bootstrap buttons and components which are large and easy to interact with.
3. The navigation flow is intuitive and the user never gets buried in pages.

### Cons

1. There is no log out button, instead the user has to click to profile icon to log out. There is some text near this icon explaining this functionality, but a log out button would be far better
2. On desktop, if the user's screen is too narrow and the user is logged in, when they view the single page volcano screen, the chart showing population will fall off the bottom of the screen. Due to the limitations with my CSS skills, the right panel does not scroll regardless of the height of its content.
3. The app is unusable when being viewed on a mobile browser. The left panel cuts off most of the screen and the information in the right panel is too narrow. This could be fixed by making the left panel collapsable and revealed by a hamburger button at the top. The right panel should show a compressed version of the table which scrolls with the page rather than in itself. The single page volcano view should be redesigned to show a hero image at the top, possibly replaced by the map. The information should be organized and the bar chart should be placed at the bottom.

# Technical description

This application is split into two side and that is reflected in the source code. The components for the user information components and the country list are stored in a left container component and the volcano table and single page components are stored in a right container.

```
<div className="row">
        <div className="col" id="leftPanel">
                <LeftPanelContainer />
        </div>
        <div className="col-9">
                <Outlet />
        </div>
</div>
```

I have used plain CSS to allow the user to scroll the left container as the volcano list will almost always be taller than the users screen. The right side of the screen has a fixed height. This is sometimes a drawback as if the user has a small screen and is logged in, the volcano population bar chart can clip below the bottom of the screen.

```
.App {
        height: 100dvh; /* Fill the height of the container */
        overflow-y: hidden; /* Add vertical scroll if content exceeds the height */
}

#leftPanel {
        height: 100dvh;
        overflow: auto;
}

#userInfo {
        position: sticky;
        top: 0;
}
```

The left container is displayed with TanStack Router's root route. The root route is always displayed which fit well with my web app as the left panel acts as a persistent home bar. The right panel is displayed via TanStack's 'Outlet' component. All other routes are shown from this component. There are two routes: /country and /country_/volcano. The underscore which trails the country in the second route is TanStack's method of preventing this route from also being matched with the /country route.

```
export const Route = createFileRoute('/$country')({
        component: VolcanoTable
})

function VolcanoTable() {
        const { country } = Route.useParams();
        const { setSelecteCountry } = useContext( CountryContext ) as CountryContextType;
        setSelectedCountry(country);

        return (
                <CountriesVolcanoesContainer />
        )
}
```

My web app uses React's context and state api extensively. The selected country, volcano api client, selected volcano, and logged in user are all contexts. While state is used for information local to the component which may change like the radius filter. The volcano api client was initialized as a context as when the user is logged in, a new client is created with the bearer token. This forces all dependent components to automatically reload with the logged in information.

```
export interface VolcanoSelectedContextType {
        volcanoSelected: boolean;
        setVolcanoSelected: Dispatch<SetStateAction<boolean>>;
}

export cons VolcanoSelectedContext = createContext<VolcanoSelectedContextType | null>(null);
```

```
export interface VolcanoClientContextType {
        volcanoClient: VolcanoApiClient;
        setVolcanoClient: Dispatch<SetStateAction<VolcanoApiClient>>;
}

export const VolcanoClientContext = createContext<VolcanoClientContextType | null>(null);
```

I created a volcano api class to interact with the api and and house some helpful methods as explained in the endpoints section. This class is used throughout the code and I found it very helpful. One drawback is the way refreshing bearer tokens is handled. As the endpoint does not return a refresh token, the only way to request a new bearer token is with the user's log in credentials. These are stored in the client as to not have to ask the user to log in again once the bearer token expires. If the endpoint returned a refresh token, this could be saved instead and used to request a new bearer token.

```
import { ILoginResponse, IRegisterResponse, IVolcano, LoginResponse, RegisterResponse } from './Interfaces';
import Volcano from './Volcano';

export default class VolcanoApiClient {

// Base URL for the API
private baseUrl: string = "http://4.237.58.241:3000";

private bearerToken: string = "";
```

## Difficulties / Exclusions / Unresolved Issues

All requirements have been implemented as well as other extra components such as the map overlays and flag images.

There is a remaining unresolved issue in the web app. This is a small bug and does not effect the overall usability of the web app. If the user is looking at the single volcano page and decides to log out, they will lose some information in the page. They will still see all information returned from the volcanoes endpoint, but wont see last eruption or location information. This results in the map reloading and returning to 0 latitude 0 longitude.

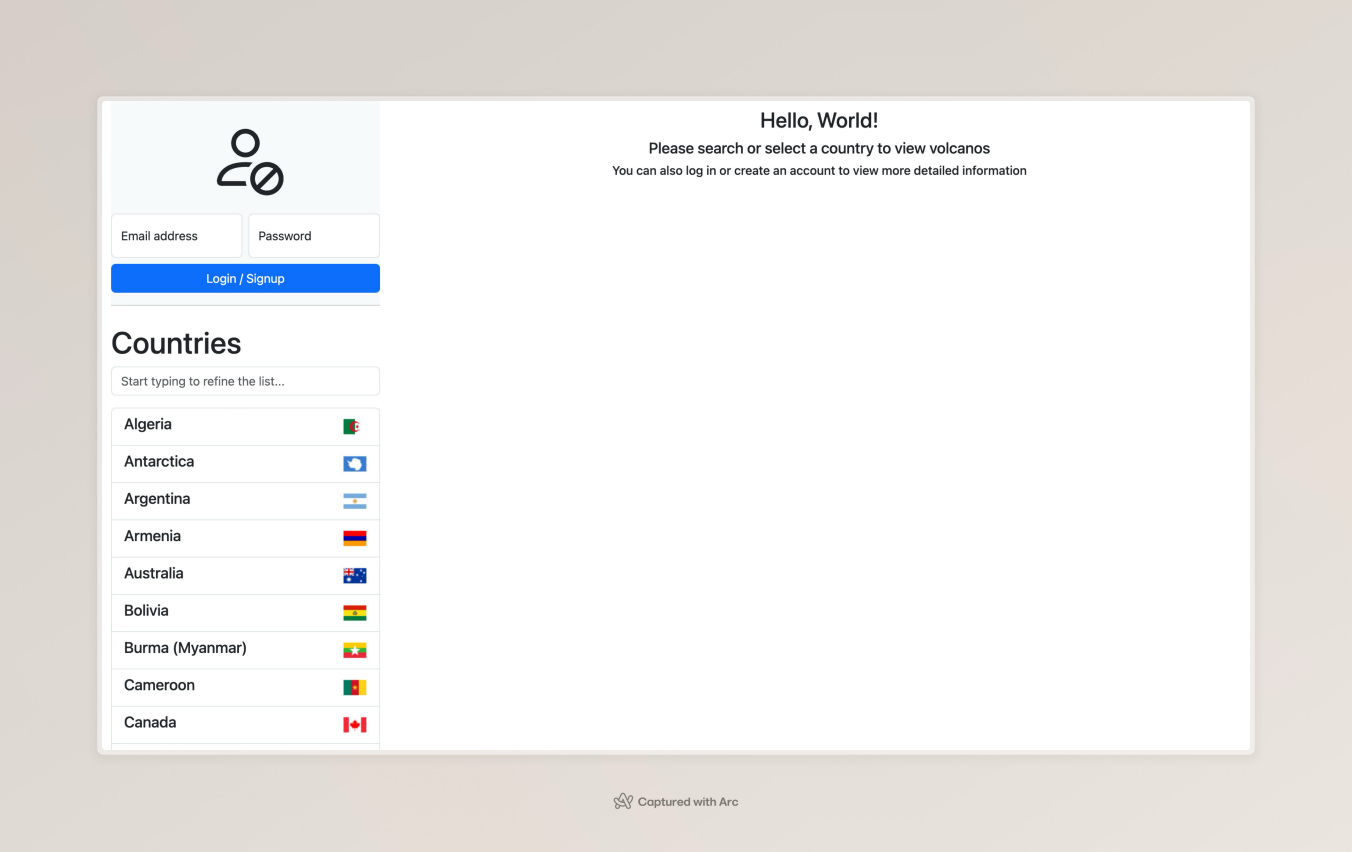
To resolve this, the user can return to the volcano list and click on the volcano again which will load the correct information. This bug is due to the order I developed this application. I mistakenly chose to implement the router last. Before the app sat on top of TanStack Router, navigation was perfect and there were no bugs. After I moved the app to sit on top of the router and moved sone context api calls to within the routes, bugs started to appear. I was able to remove all but one. As with any application, the correct solution to this problem is a complete refactor of the context and state api calls.

## Tests

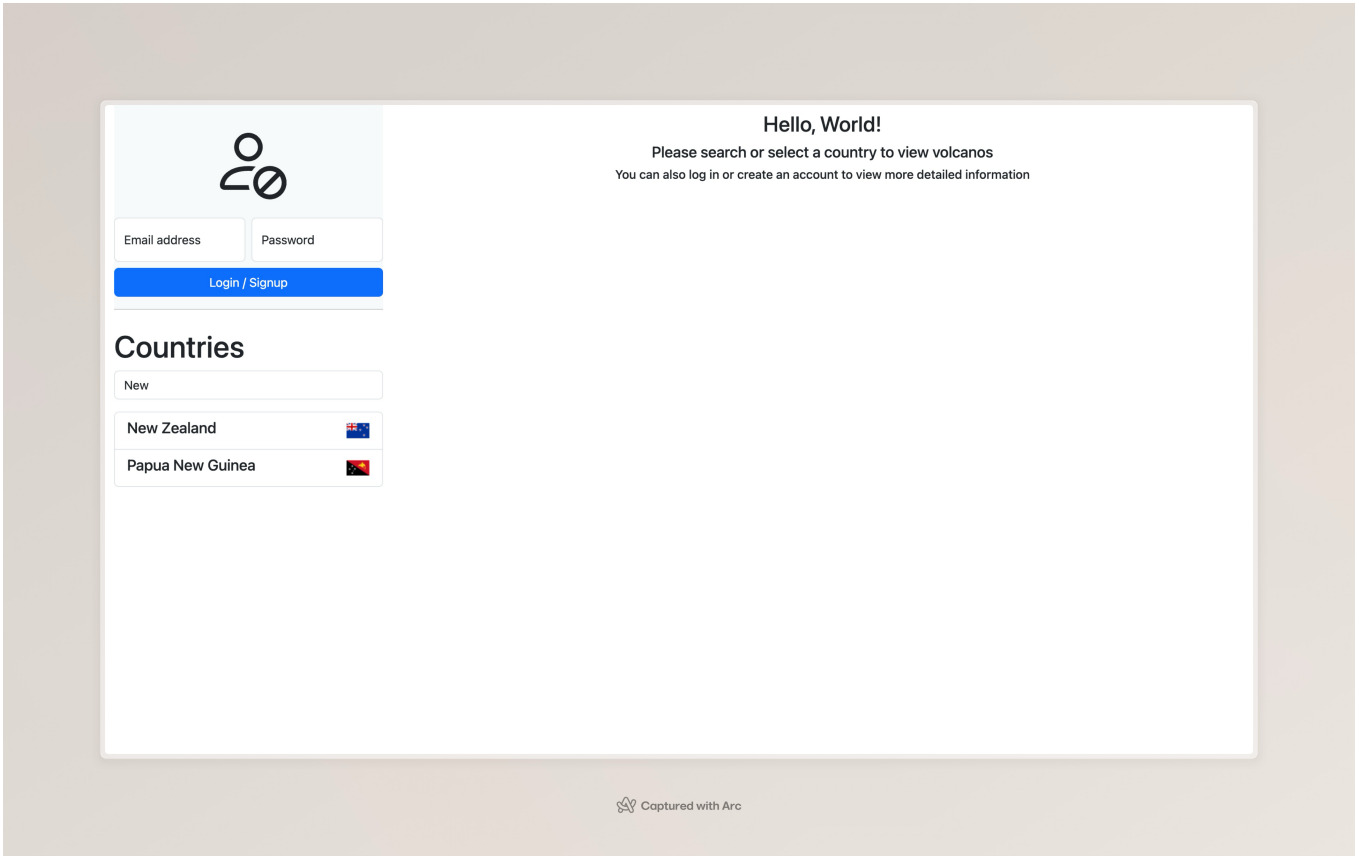
| Action | Type | Outcome | Appendix | Comments |
|---|---|---|---|---|
| Website initially loads correctly | Normal | PASS | | |
| User can create an account | Normal | PASS | | |
| User is then auto logged in | Normal | PASS | | |
| User can log into an account | Normal | PASS | | |
| Bearer token is auto refreshed | Normal | PASS | B | This appendix shows the code to do this |
| User can filter country list with search | Normal | PASS | | |
| Search fuzzy filters correctly | Normal | PASS | | |
| User can select a country | Normal | PASS | | |
| Country component in list displays correct flag | Normal | PASS | | |
| Volcano table for country displays correct flag | Normal | PASS | | |
| User can use slider to filter volcanoes by population radius | Normal | PASS | | |
| User can switch to another country from volcano table | Normal | PASS | | |
| User can select a volcano from the table | Normal | PASS | | |
| Single page volcano displays correctly | Normal | PASS | | |
| User can log in while on the single volcano page to see population information | Normal | PASS | | |

| Action | Type | Outcome | Appendix | Comments |
|---|---|---|---|---|
| User can select a single volcano while already logged in and see population data | Normal | PASS | | |
| User can log out which removes population data | Edgecase | FAIL | A | Sometimes when the user logs out, the volcano page will show bad infromation |
| User can go back to volcano list from single volcano page | Normal | PASS | | |

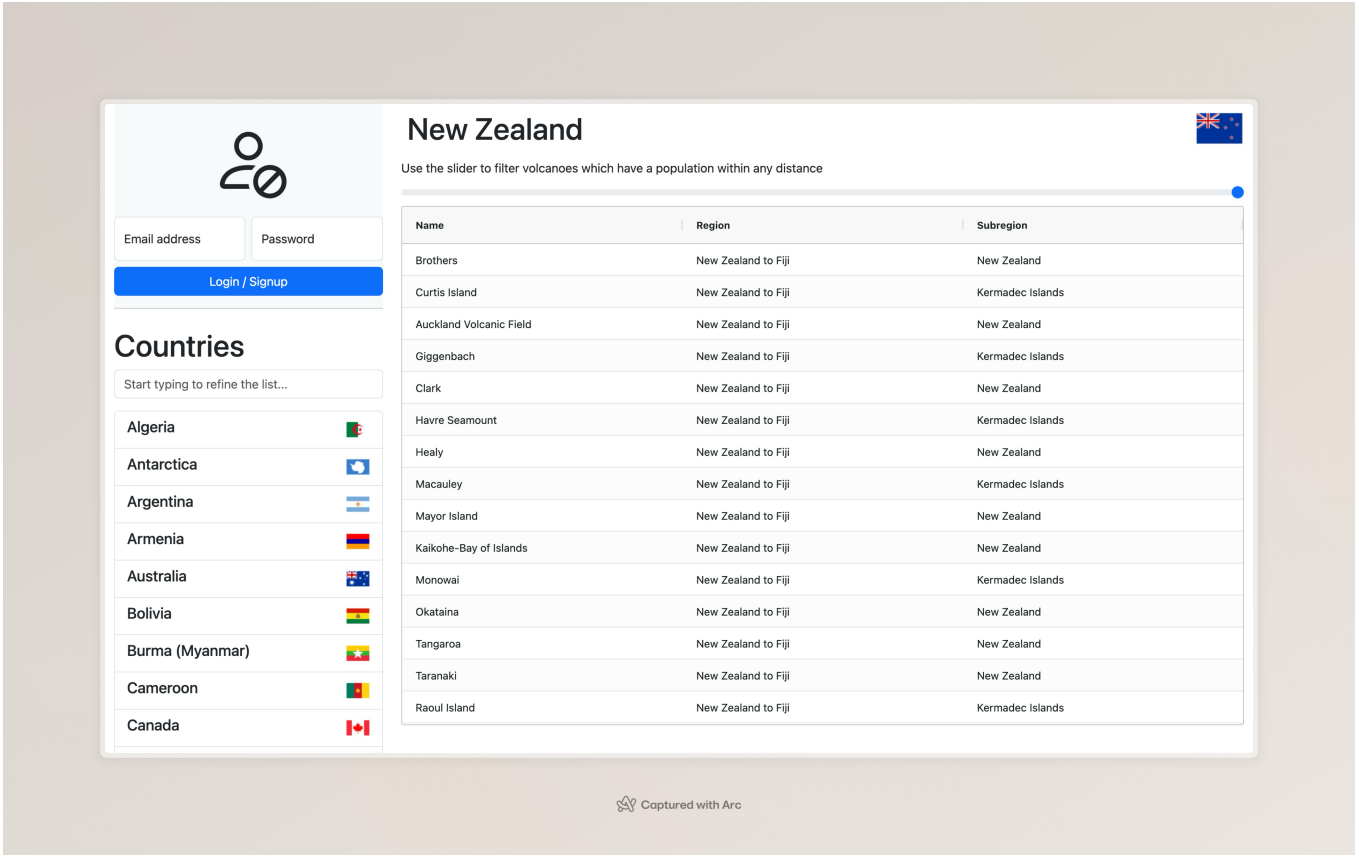
# Appendix (user guide)



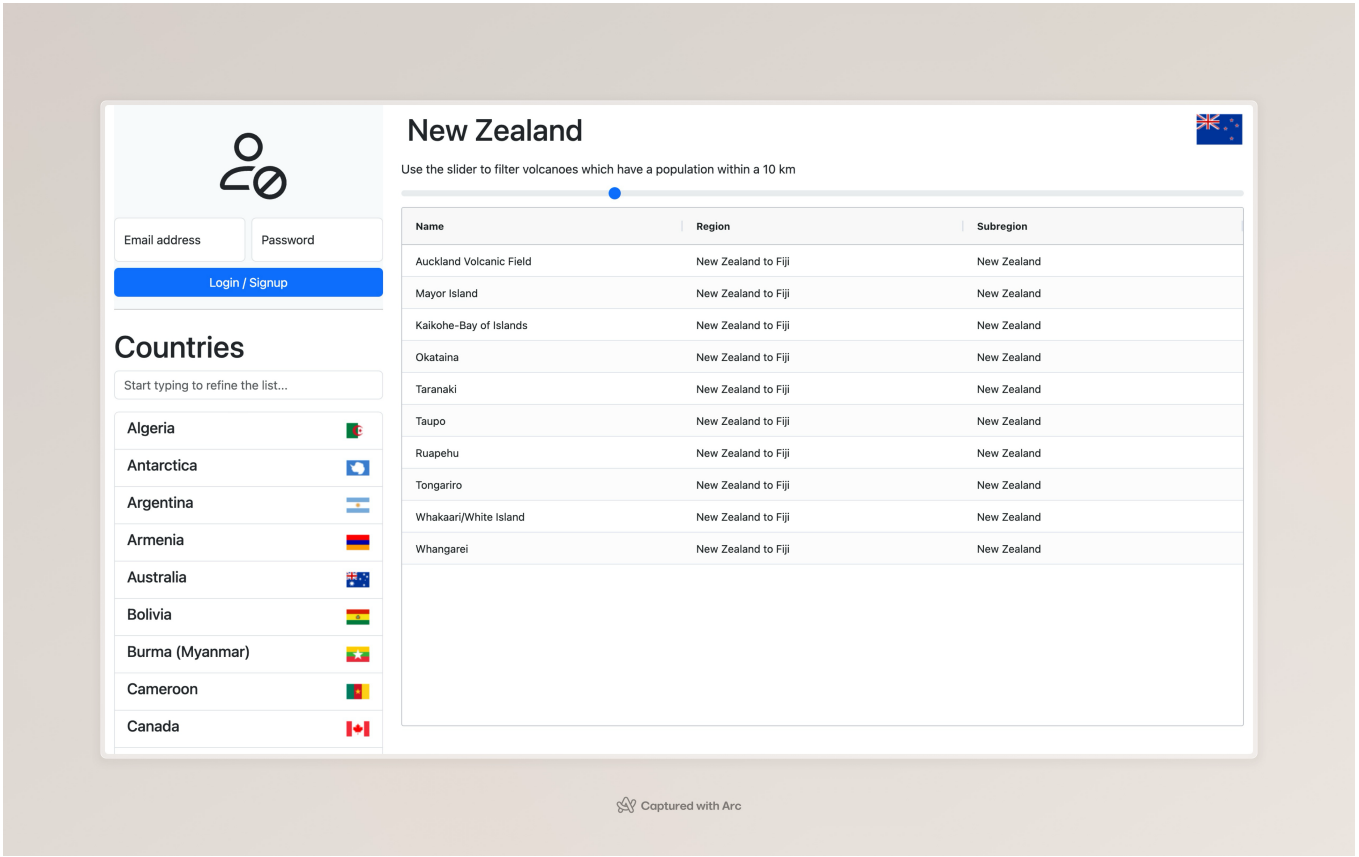

When you launch the app, this is what you see. You can log into the app at this point, or create an account. You can also search for or click on a country. These actions are performed on the left hand side.



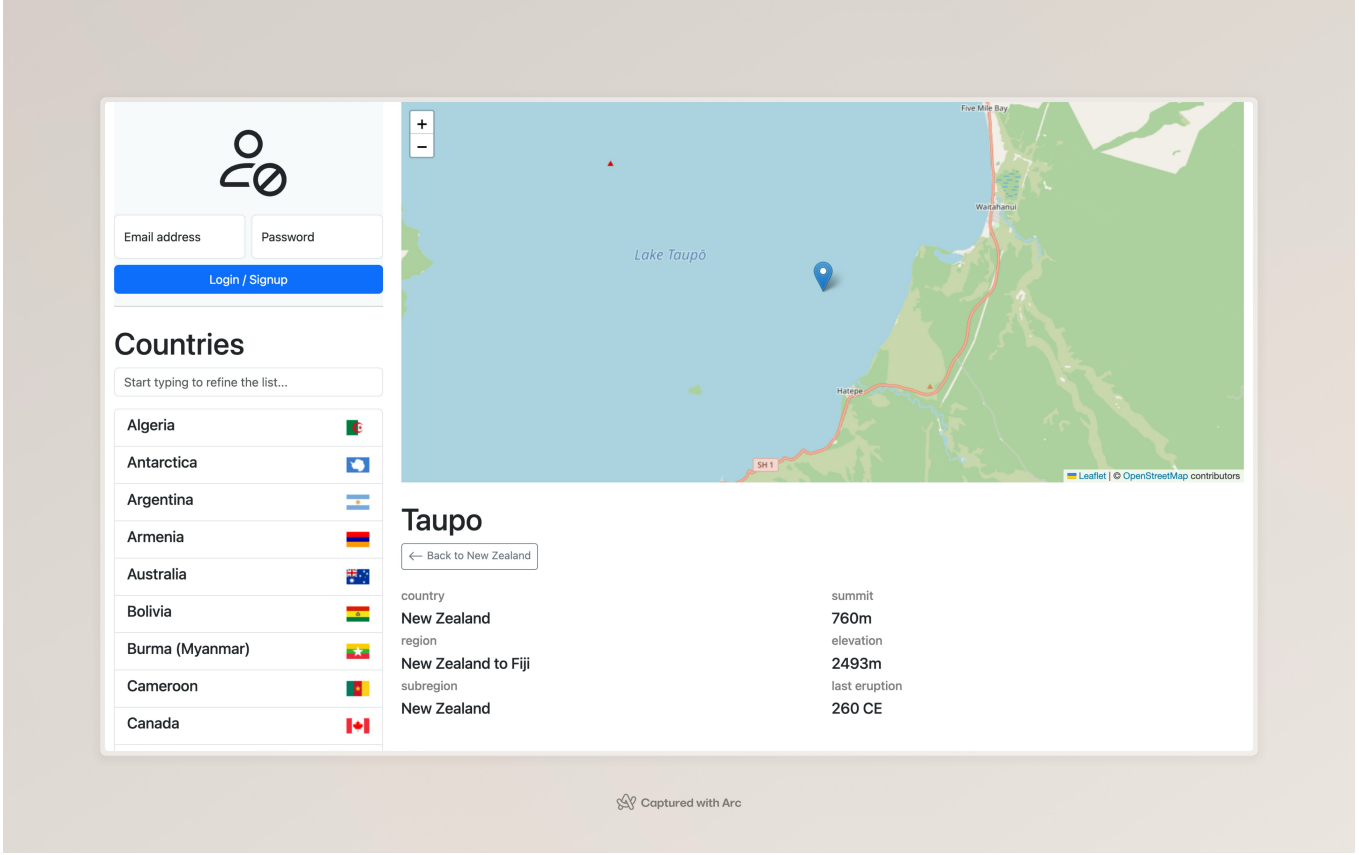

This is what you see when you start searching for a country. The list will only show matches to your search term if it appear anywhere in the name.



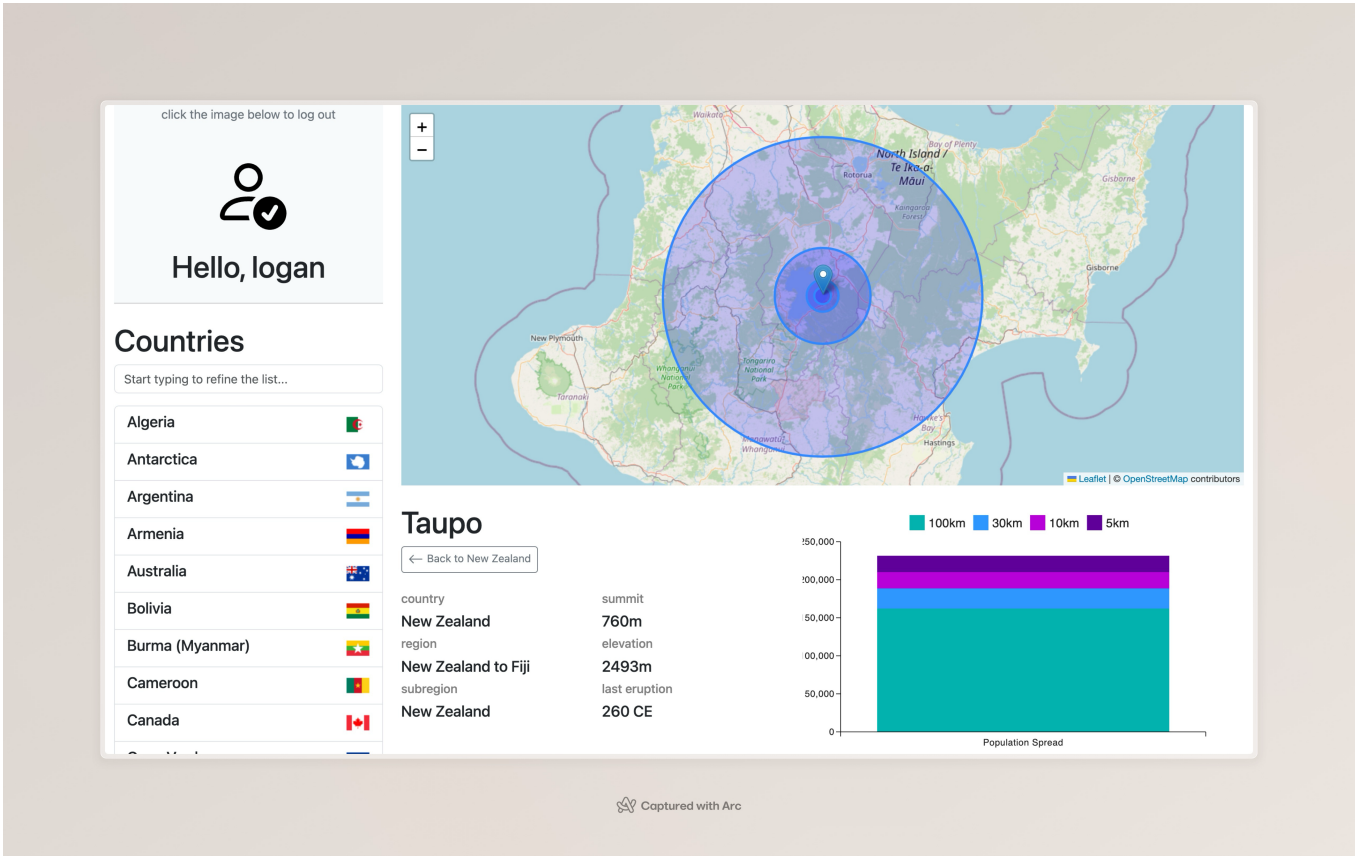

This is what you see when you click a country. You can either click a volcano or filter them by if they have a population within a certain radius. To do this, move the slider at the top.




This is the shorter list shown after the radius filter is made smaller.

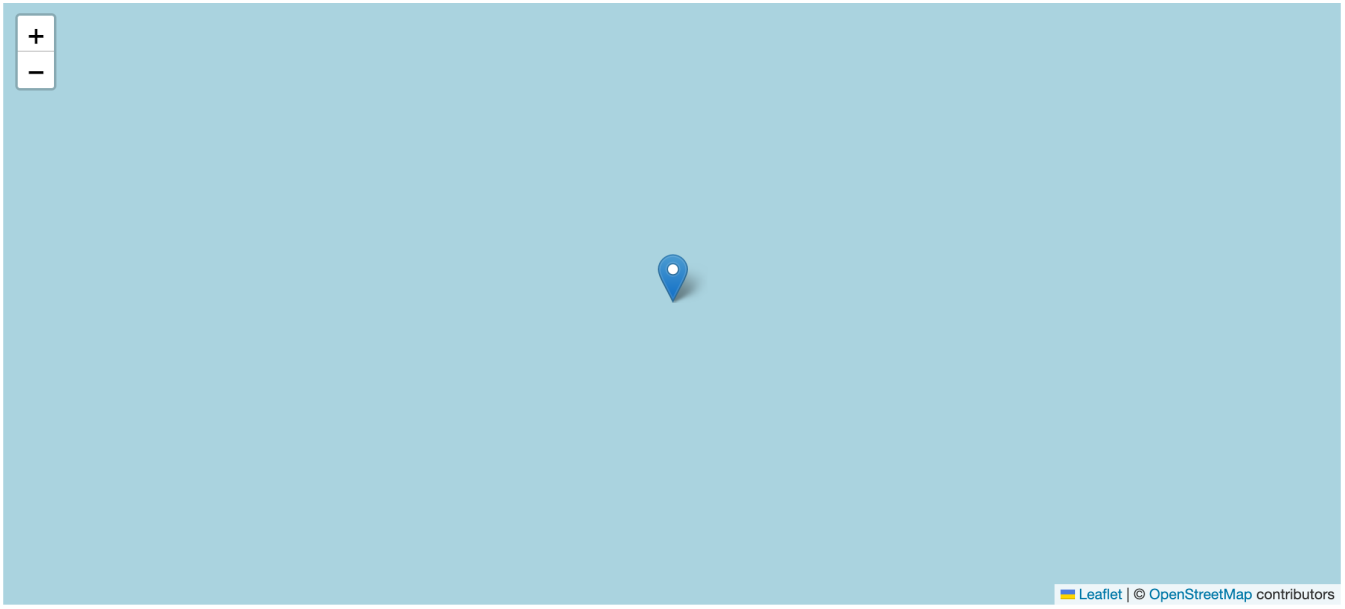



This is the view of a single volcano after you click a volcano in the list. In this case, the volcano is contained within Lake Taupo, New Zealand. You can see basic information about the volcano such as summit and last eruption.




If you log into the app or create an account while looking at the single volcano page, you will see more information about the volcano. This population information is displayed in two ways, the map radius overlay and the bar chart. You can either go back to the volcano list with the back button, or switch countries by selecting another one in the side panel.

## A




# Huanquihue Group

← Back to Argentina

country
Argentina
region
South America
subregion
Central Chile and Argentina
summit
Unknownm
elevation
Unknownm
last eruption
Unknown

**B**

```
public async getVolcanoById(id: number) {

        let continueLoop = true;
        let loopCount = 0;


        if (!this.loggedIn) {
                continueLoop = false;
        }

        while(continueLoop) {
                loopCount += 1;

                if (loopCount > 5) {
                        continueLoop = false;
                }

                const response = await fetch(`${this.baseUrl}/volcano/${id}`, {
                        method: 'GET',
                        headers: {
                                'Authorization': `Bearer ${this.bearerToken}`
                        }
                });

                if (response.status === 401) {
                        console.log("Token expired, getting new token");
                        this.bearerToken = await this.getToken(this.username, this.password);
                } else {
                        const returnedData = await response.json() as IVolcano;
                        const volcano = new Volcano(returnedData);
                        return volcano;
                }
        }

        const response = await fetch(`${this.baseUrl}/volcano/${id}`);
        const returnedData = await response.json() as IVolcano;

        return new Volcano(returnedData);
}
```