Final Year Project Report

Full Unit - Final Report

Offline HTML5 Maps Application

Dylan Maryk

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Gregory Chockler

Department of Computer Science
Royal Holloway, University of London

March 29, 2016

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 9630

Student Name: Dylan Maryk

Date of Submission: March 29, 2016

Signature:

[illegible]

Table of Contents

Chapter 1: **Introduction and Objectives**

1.1 Introduction

The full project is best used on my website using the Google Chrome browser: http://dylanmaryk.com/offlinemaps/main/

My Git repository is on GitHub: https://github.com/dylanmaryk/OfflineMaps

I am doing the Offline HTML5 Maps Application project because I have a keen interest in software, particularly on mobile platforms, that utilises map technology to provide a user with helpful information, especially contextually relevant information based on their location. It is worthwhile to apply this interest using web technologies instead, experimenting with the combination of HTML5 mapping and local storage, while still offering functionality traditionally only offered with online map services, such as providing information about places of interest. The final project is an HTML5 application, able to work similarly both online and offline, that is capable of storing and displaying richly detailed maps using tile images, with information and functionality a user would expect from a native application.

I believe this would be a useful project that currently faces minimal competition. For example, Google's Maps mobile application is a native application, hence only usable on certain platforms, and currently only allows downloading and viewing offline certain areas. Furthermore, there is a maximum downloadable area size [1]. This project has the potential of being hugely beneficial to many. It offers the possibility of being able to use an existing device, with existing web browsing software, to assist with navigational requirements when there is no available method of acquiring the data needed for navigation from a remote source. A user would be able to, for example, view a map to help them get directions when they are in a remote area with no WiFi or mobile signal. Or, a user might live in an area with a slow Internet connection, meaning it would save them large amounts of time to be able to download a map once and then be able to view it again in the future without having to wait for it to be downloaded again. These are just some of many possible use cases for applications of this project.

The implementation section revolves around the sorts of topics I have been researching and how I have used them when developing either proof of concept programs or the final project. These include key background concepts and technologies relevant to web development and mapping tools.

This project will help me in my future career because of two factors: specific elements of the project and general programming experience. Firstly, during development of the project I have focused on topics relating to maps and location, both of which I have focused on in previous projects, meaning they add to my employable experience should I intend to apply for roles involving mapping software. I have learnt much about more specific topics including application caching, IndexedDB and PostGIS. Secondly, I have had the opportunity to extend my overall programming experience, being able to practice using languages such as Javascript and Ruby and libraries such as jQuery and Leaflet. This will be useful for any role for which programming ability is required.

1.2 Objectives

During the first term, I aimed to complete the early deliverables for the project, and was successful in doing so. I developed three proof of concept programs.

1. "Hello world" offline HTML5 application: A simple application built using HTML5 that is viewable offline.
2. To-do list application using an IndexedDB: A simple application to demonstrate the use of an IndexedDB for storing data offline.
3. Webpage that loads and lists raw OpenStreetMap data: A demonstration of retrieving and displaying OpenStreetMap data in a raw format.

I also completed the final deliverables with basic functionality, and without having yet fully optimised in terms of performance and user experience. I successfully implemented a program that works offline, loads and displays map data and allows the user to pan and zoom around the map.

During the second term, I originally planned to extend the program with features such as advanced search functionality and basic directions between two points. However, I spent most of the time on improving how map data is cached, and how that cached data is displayed, and implementing the ability for the user to download however much data they choose for an area of their choice. I then proceeded to add additional features including displaying to the user the name of the area they are currently viewing.

Chapter 2: **System Design**

2.1 High-Level Overview

The project comprises three main areas of development:

1. Map with caching functionality: A browser-based map that automatically caches any data loaded due to user interaction.
2. Download functionality: The user is able to select and download a specific area of the map.
3. Location name display: The user is provided with the name of the location they are currently viewing.

When the user navigates to the URL at which the project is hosted they will see a full-screen map. If this is the first time the user has been to the website in the browser they are currently using, the main elements of the website will be cached, so the user can return to the website at the same URL even when they do not have an Internet connection. Furthermore, any data that is shown on the map, so any images representing areas, will automatically be cached locally in the browser. New data would be loaded by the map as a result of initial page load, the user panning around the map or the user zooming in or out of the map.

At any point while the user is browsing around the map, they can click on the download button in the bottom-right corner of the window, which will begin the process of allowing the user to download a specified amount of data within the area of the map currently visible. The user is presented with a slider representing how many zoom levels deep they want to download data for, so essentially how much detail to download. They can then click “Download” and will be presented with a progress bar informing them of how much the download has progressed. The end result is that the specified amount of data within the visible area of the map is stored locally in the browser.

Whenever the centre point of the map changes due to user interaction, a request is made to an API with the centre point’s coordinates. The API returns the name of the location with the coordinates nearest to the centre point’s coordinates, based on a database of locations worldwide and relevant data for each.

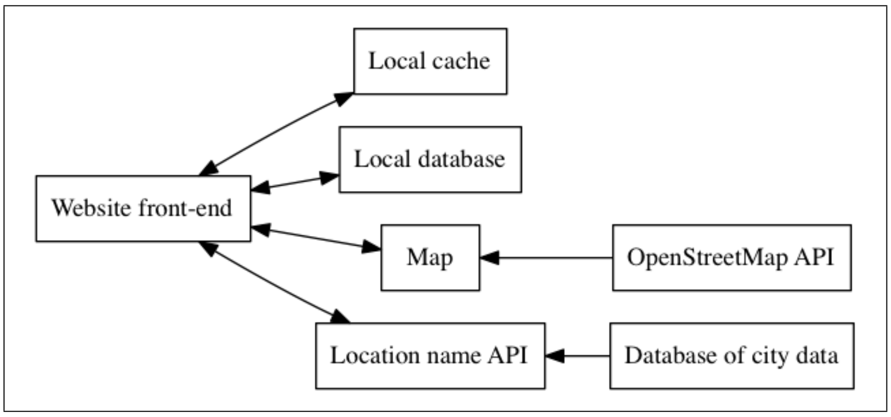


Figure 2.1: UML deployment diagram for the whole system.

2.2 Testing

There are a couple of key scenarios to test in terms of the user‘s ability to use the project successfully. Firstly, the process of map data being cached automatically, and secondly, the ability to select and download an area of the map.

Test case 1: Map data caching

This test tests that any data viewed by the user is cached locally, and can be accessed while their device is offline.

1. Navigate to the URL at which the project is hosted.
2. Wait for all the tiles on the visible portion of the map to load.
3. Disable the device‘s connection to the Internet.
4. Refresh the page in the browser.
5. Check if the tiles that were previously visible when the device was connected to the Internet are visible now.

If the tiles that were previously visible are visible after the refresh, the test has passed, as the data has therefore been successfully cached. If the tiles are not visible, the test has failed.

Test case 2: Map area downloading

This test tests that when the user requests for one zoom level below of map data to be downloaded, that data is successfully downloaded and stored locally, and can be accessed while their device is offline.

1. Navigate to the URL at which the project is hosted.
2. Click the download button in the bottom-right corner of the browser window.
3. Select “1” on the slider.
4. Click the “Download” button.
5. Wait for the progress bar to be full and then disappear, revealing the map again.
6. Disable the device‘s connection to the Internet.
7. Refresh the page in the browser.
8. Zoom in one level. Check if the whole map is populated with tiles, so there is no grey space showing.
9. Zoom in one more level. Check if the whole map is grey space, so there are no tiles.

If on the first zoom the whole map is populated with tiles, and on the second zoom the whole map is grey, the test has passed, as only one zoom level of data has been downloaded. Otherwise, the test has failed.

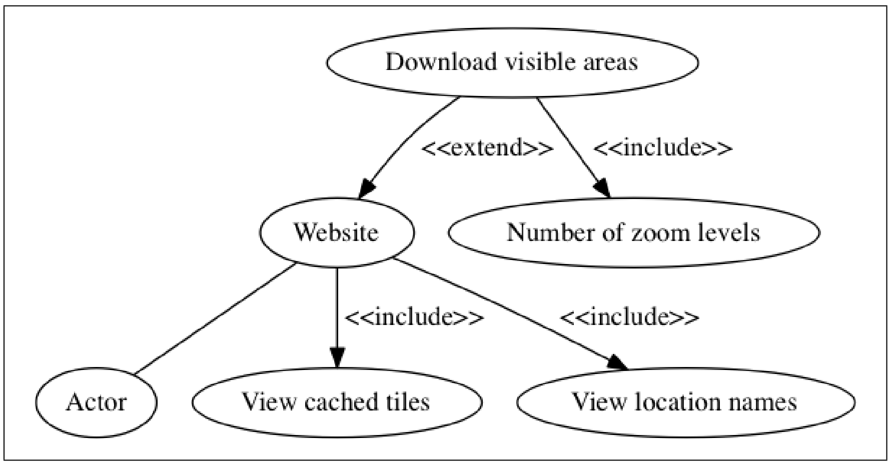


Figure 2.2: UML use case diagram for the usage of the system by the user.

Chapter 3: **Implementation**

3.1 HTML5 Caching and Storage

3.1.1 Introduction

I have investigated the basics of storing information offline using modern web browser technologies, in order to develop two of the proof of concept programs, and later on the final deliverables.

HTML application caching is vital to allow the user to view the program in their web browser of choice even when they are lacking an Internet connection, and a form of data storage must be used to store the OpenStreetMap data itself while the user is online, so that it can be accessed later even when the user is offline.

3.1.2 Application Caching

The HTML5 application cache provides a simple means by which to locally store the files required to display a website offline automatically, whenever the user loads the page while they are online, ensuring the cached files are updated as frequently as possible. However, it is important to note for the further development of the project that browsers can have different size limits for cached data [2], hence it is preferable to cache only the minimum necessary to display what it makes sense to be viewable offline. It must be noted that the browser's cache for a website will not be updated unless the cache manifest, the file which lists all the files on the server to cache, is updated. Therefore, changes to the online version of the website will not be cached while the user is online unless the developer updates the cache manifest. This can be done as simply as changing a comment e.g. "# v1" to "# v2".

I have written my .appcache file so that every file that is unlikely to be modified frequently is cached, which includes "index.html" and every CSS and JavaScript file. Below the list of files to cache, I have included:

```
NETWORK:
*
```

This ensures that for any files not cached, the app will attempt to retrieve them from the relevant remote locations, meaning the app will always try to get the remote resources before using a fallback option.

3.1.3 Downloading and Caching Required Files

As remote files may be updated frequently, such as those for libraries I use in the project, and because I believe it best to rely on other organisations' servers as little as possible regardless, due to possible downtime for example, I have decided to write a bash script that I can run periodically on the server on which the project is hosted. This bash script is responsible for downloading files required by the project. The user's browser can then download and cache these files directly from my own server. This is the line in the bash script that downloads a

CSS file vital for the program to work:

```
wget -O css/leaflet.css "http://cdn.leafletjs.com/leaflet/v0.7.7/leaflet.css"
```

3.1.4 Data Storage

The bulk of data storage is done using technologies such as the IndexedDB API, which is designed to store larger amounts of data locally. It has a global limit, the maximum amount of storage space that can be used by the browser for all origins (top level domains), of 50% of free space on the device [3]. The group limit, the limit for each group of origins, is 20% of the global limit. This must be handled when considering how much map data can be downloaded onto the user‘s device. The API is interacted with via JavaScript.

I am caching data using an IndexedDB database whereby string representations of OpenStreetMap tile images are stored. This is the cached data that will be used before an attempt to retrieve the resource remotely is made.

3.1.5 Conclusion

When implemented properly, ensuring the correct resources are cached, HTML5 application caching is a simple and effective method of storing key resources of the program offline. The fact it is a standard technology in HTML5 is useful in that it can be used in most modern browsers.

IndexedDB is also a standard web technology, and so is a good choice of data storage because it works in most browsers, is well documented and is capable of storing large quantities of data.

3.2 OSM Data Representation

3.2.1 Introduction

The format of data the program must be able to store offline is tiles as images. In summary, images retrieved from OpenStreetMap via their API represent sections of a map at specified zoom levels. These tiles must then be displayed on a map, the task for which I have decided to the use the JavaScript library Leaflet.

3.2.2 Mapping Library

I opted to experiment with using the open-source JavaScript library Leaflet [4], as it is designed with being mobile-friendly in mind, is lightweight (beneficial in terms of being quick to cache and taking up minimal storage space on the user‘s device) and because it is open-source can be more easily modified. There are therefore also many other open-source projects that build upon Leaflet already in existence.

3.2.3 Displaying and Storing Tile Images

Leaflet handles the basics of loading and displaying tiles. It allows for initialising a tile layer with a URL template that is used to generate the URL for each tile to display, along with a number of optional parameters [5].

```
var osmUrl = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
    osmLayer = new CustomTileLayer(osmUrl);
```

I am using my own extended version of the tile layer class, “CustomTileLayer”, which I will go into detail about later. The URL template has four parameters: “s” for one of the available subdomains to aid possible parallel requests, “z” for the map‘s current zoom level, “x” and “y” for the coordinates of the tile for which an image is being requested. Leaflet handles adding these parameters and adding the resulting URL as the source of the relevant HTML object that represents the tile. This covers displaying the correct map tiles on request, but the next step is to add functionality on top of Leaflet to handle caching these images to an IndexedDB database, so they can be loaded instead if the user is offline.

```
var request = window.IndexedDB.open("TileStorage", 3);
```

This line opens the “TileStorage” database so it can be interacted with. There are three event handlers for this that I make use of [6]. When “onsuccess” is called, the database exists, so the database is initialised.

```
db = event.target.result;
```

When “onerror” is called, it is most likely the user did not give permission for the browser to use IndexedDB, so an error is printed in the console.

```
console.log("Did not allow local data storage.");
```

When “onupgradeneeded” is called, the database does not yet contain an object store, so a new one is created.

```
db = event.target.result;
db.createObjectStore("tile");
console.log("Created new object store.");
```

This handles the structuring of the database, ready for adding data to and retrieving data from the “tile” object store.

Probably the most important part of the process of caching images however is done when certain events are fired by the tile layer instance. I respond to two events. The first is ‘’tileloadstart”, fired before an image source is added to a tile.

```
event.tile.crossOrigin = "Anonymous";
```

This line sets the “crossOrigin” parameter of the tile to “Anonymous”, which is necessary to allow the image, once it has been downloaded, to be retrieved from a canvas once that canvas has been used to transform the image into a Base64 formatted string. Otherwise, when attempting to retrieve the formatted string from the canvas, an error will be raised saying that the canvas has been “tainted”. This is a security measure in place for when, as in this case, images defined by an “img” element are loaded from a foreign source, which “protects users from having private data exposed by using images to pull information from remote web sites without permission” [7].

The second event responded to is “tileload”, fired when an image has been successfully loaded. Below is the original code written for this event.

```
var tileImageString = getBase64Image(event.tile);
var tileImagePoint = event.tile.point;

if (db) {
    storeTileImage(tileImageString, tileImagePoint);
} else {
    tilesToStore.push({image: tileImageString, point: tileImagePoint});
}
```

If the database has been initialised, the tile‘s image, in Base64 format, and the tile‘s position and zoom level are added to the “tile” object store. Otherwise, they are pushed to an array. Once the database has been initialised, the images and positions stored in this array will be added to the object store.

I have now updated the code for the “tileload” event, as I decided later on in development to have the program load cached data before attempting to load remote data, as opposed to attempting to do the reverse. This led to a significant refactor of several parts of the codebase. Below is the updated code written for the event.

```
var tile = event.tile,
    url = event.url,
    tileImagePoint = tile.point;
```

```
if (url.substr(0, 4) == "data") {
    tile.src = this.getTileUrl(tileImagePoint);
} else {
    var tileImageString = getBase64Image(tile);

    if (db) {
        storeTileImage(tileImageString, tileImagePoint);
    } else {
        tilesToStore.push({image: tileImageString, point: tileImagePoint});
    }
}
```

The original code will now only run if the the tile's URL does not begin with "data", which would mean it has been loaded from the cache. Hence, only if the image was retrieved remotely is it added to the object store, either immediately or later on.

I have created a function "loadStoredTileImage" that sets the image of a tile.

```
function loadStoredTileImage(tile, remoteUrl) {
    if (db) {
        var tileImagePointString = getPointString(tile.point),
            request = getObjectStore().get(tileImagePointString);
        request.onsuccess = function(event) {
                var tileImageString = request.result;

                if (tileImageString) {
                    tile.src = tileImageString;
                } else {
                    tile.src = remoteUrl;
                }
        };
    } else {
        tilesToLoad.push({tile: tile, remoteUrl: remoteUrl});
    }
}
```

Tile images and positions are stored in the object store using a key-value relationship: the position and zoom level is the unique key and the image as a Base64 string is the value. If the database has been initialised, the value for the tile‘s position is attempted to be retrieved from the database. If successful, this value is set as the tile‘s image source. Otherwise, the remote URL for the tile‘s image is set as its image source. This means that the program determines when to load images from the local database instead of retrieving the relevant images remotely. Originally to achieve this, I opted to implement timeout functionality. To do this, I had overridden the "_loadTile" function of Leaflet‘s "TileLayer" class in order to give each tile a number of seconds after which a function is called that checks if the image was successfully loaded.

```
tilesToCheck.push(t);
setTimeout(checkImageLoaded, 5000, this);
```

If the program had failed to get the tile‘s image from OpenStreetMap‘s API, the "checkImageLoaded" function would fire the "tileerror" event, which originally handled the functionality now present in my "loadStoredTileImage" function.

```
function checkImageLoaded(layer) {
    var tile = tilesToCheck[tilesToCheckCounter];

    if (!tile.complete) {
        layer.fire("tileerror", {
            tile: tile
        });
    }

    tilesToCheckCounter++;
}
```

Now, while I still override the "_loadTile" function, inside it I simply call "loadStoredTileImage", in order to attempt loading the stored image for the tile before trying to get the remote image, passing the tile itself and its associated URL for the image.

```
loadStoredTileImage(t, this.getTileUrl(e));
```

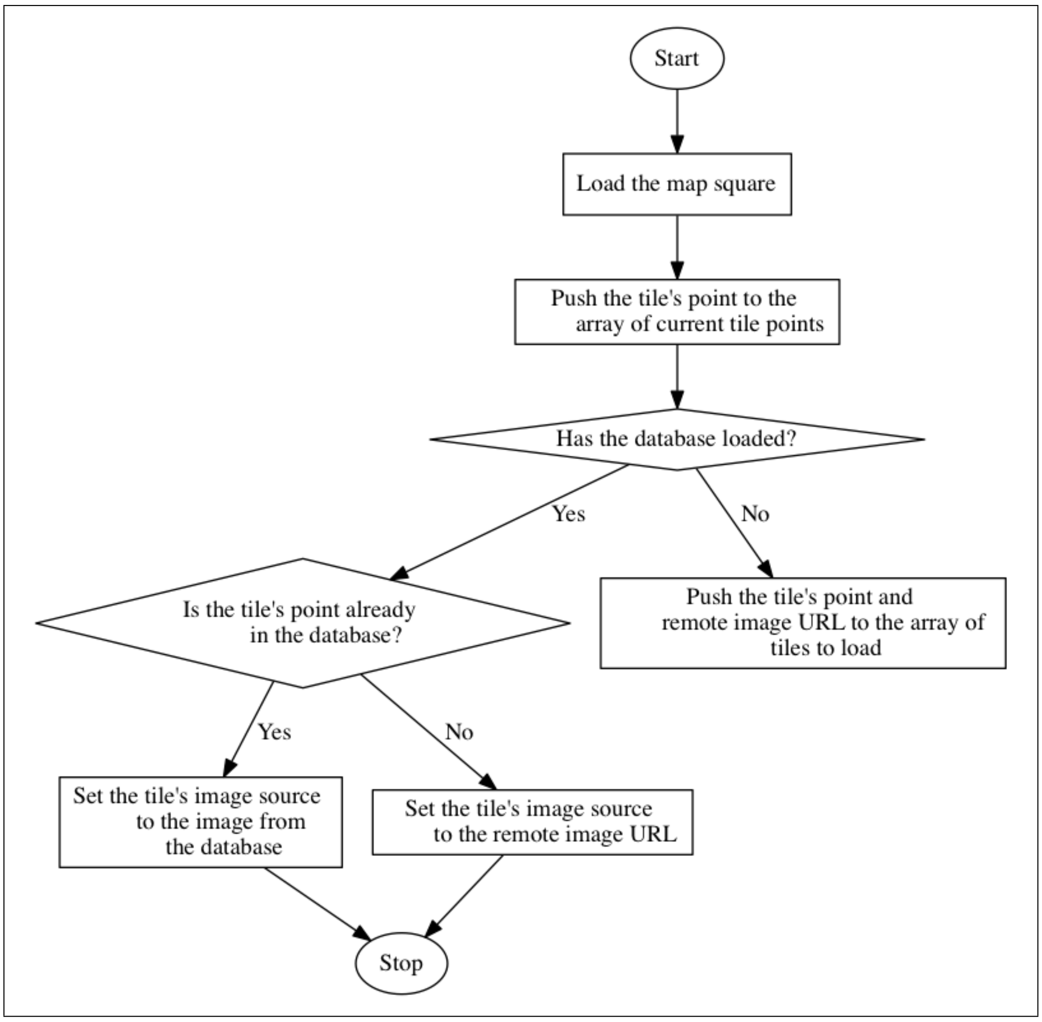

Start
Load the map square
Push the tile's point to the array of current tile points
Has the database loaded?
Yes
No
Is the tile's point already in the database?
Push the tile's point and remote image URL to the array of tiles to load
Yes
No
Set the tile's image source to the image from the database
Set the tile's image source to the remote image URL
Stop

Figure 3.1: Control flow diagram for loading an image for a tile on the map.

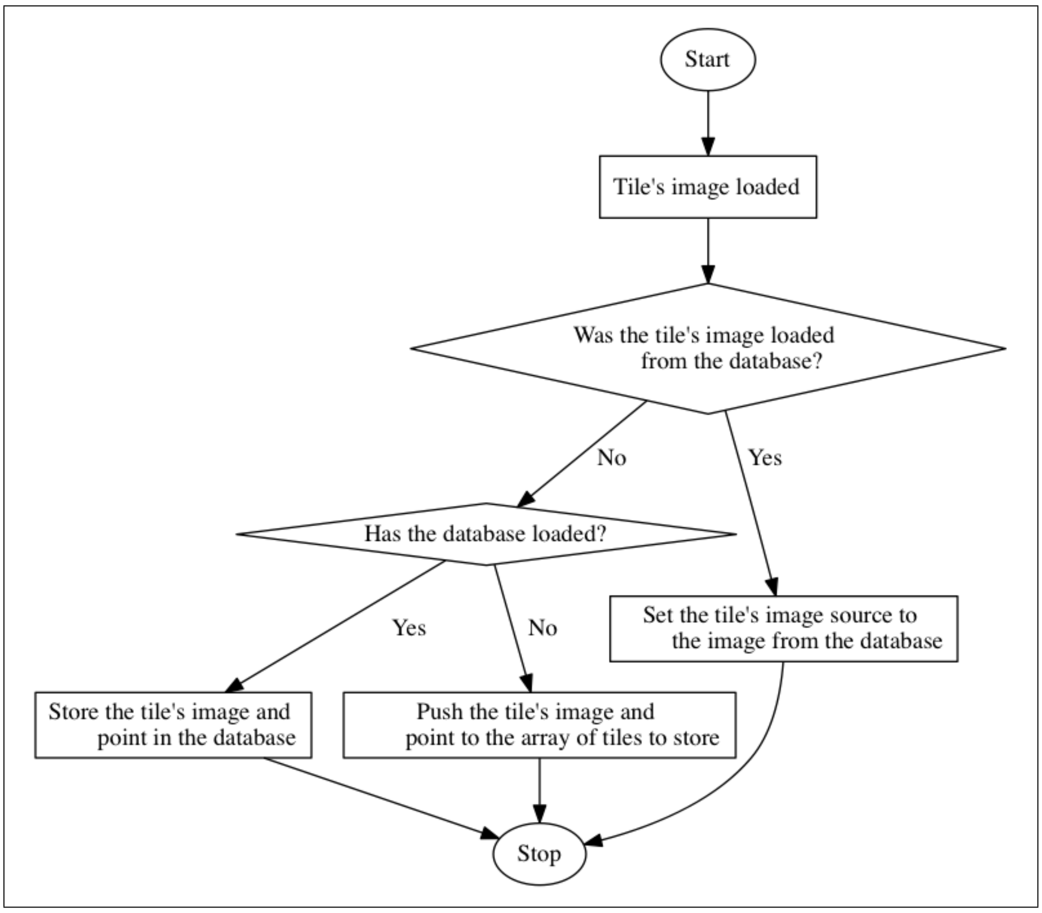

Start
Tile's image loaded
Was the tile's image loaded from the database?
No
Yes
Has the database loaded?
Yes
No
Set the tile's image source to the image from the database
Store the tile's image and point in the database
Push the tile's image and point to the array of tiles to store
Stop

Figure 3.2: Control flow diagram for saving an image of a tile on the map.

3.2.4 Conclusion

The combination of the Leaflet library, the IndexedDB API and some additional custom implementations allows for a functional program suited for the purpose of downloading, storing, loading and displaying map data from the OpenStreetMap API.

3.3 Downloading Map Areas

3.3.1 Introduction

Having implemented locally storing any map tiles loaded by the map, I could now build on top of this existing functionality in order to implement the ability for the user to download a specific area of the map, and with a specified amount of detail. I did this mainly by constructing an algorithm that uses existing functions in Leaflet in a custom manner, so as to iterate through zoom levels and tiles to download the desired set of images that make up an area of the map, so they can be stored and later viewed offline.

3.3.2 Calculating, Downloading and Storing Tile Images

The bulk of the process of downloading the images for all the tiles within a certain area of the map is done in the “downloadPoint” function. This function iterates through every “theoretical” tile within a certain area of the map using nested loops with time complexity O(N^2) and calls itself recursively within these two loops.

```
function downloadPoint(tilePoint, zoomLevel, originalZoomLevel,
    zoomLevelsToDownload, isLastTile, layer) {
    var map = layer._map,
        tileLatLng = getPointToLatLng(tilePoint),
        mapTopLeftPoint = map._getNewTopLeftPoint(tileLatLng, zoomLevel),
        tileSize = layer._getTileSize(),
        mapPixelBounds = new L.Bounds(mapTopLeftPoint,
            mapTopLeftPoint.add(map.getSize())),
        mapPointBounds = new L.bounds(mapPixelBounds.min.divideBy(tileSize)._floor(),
            mapPixelBounds.max.divideBy(tileSize)._floor());

    for (var pointX = mapPointBounds.min.x; pointX <= mapPointBounds.max.x;
        pointX++) {
        for (var pointY = mapPointBounds.min.y; pointY <= mapPointBounds.max.y;
            pointY++) {
            var newTilePoint = new L.Point(pointX, pointY);
            newTilePoint.z = zoomLevel;
            var tileImageUrl = layer.getTileUrl(newTilePoint);
            var tileImage = new Image();
            tileImage.crossOrigin = "Anonymous";
            tileImage.point = newTilePoint;
            tileImage.onload = function() {
                var tileImageString = getBase64Image(this);
                downloadedTilesToStore.push(
                    {image: tileImageString, point: this.point});

                $("#downloadProgress").attr("value", tilesDownloaded++);

                if (downloadedTilesToStore.length >= tilesToDownloadFinalCount) {
                    storeDownloadedTiles();

                    tilesToDownloadFinalCount = Number.MAX_VALUE;
                    tilesDownloaded = 0;
                }
            };
            tileImage.src = tileImageUrl;

            tilesToDownloadCount++;

            var newTileIsLastTile = isLastTile && pointX == mapPointBounds.max.x &&
                pointY == mapPointBounds.max.y;

            if (zoomLevel - originalZoomLevel < zoomLevelsToDownload &&
                zoomLevel < map.getMaxZoom()) {
                downloadPoint(newTilePoint, zoomLevel + 1, originalZoomLevel,
                    zoomLevelsToDownload, newTileIsLastTile, layer);
            } else if (newTileIsLastTile) {
                tilesToDownloadFinalCount = tilesToDownloadCount;
                tilesToDownloadCount = 0;

                $("#downloadProgress").attr("max", tilesToDownloadFinalCount);
            }
        }
    }
}
```

I use the term “theoretical” to describe certain tiles to mean that these tiles are not generated and displayed by the map itself, but that the properties of would-be tiles are created by the this function, partly using Leaflet functions, in order to perform necessary calculations. The end-goal of these calculations is to determine a URL from which can be downloaded an image representing data on the map. Since OpenStreetMap tile URLs require coordinates and a zoom level to successfully retrieve an image, these are the parameters that must be calculated.

For calculating the zoom level for a tile, the zoom level is simply passed through recursion to the next instance of “downloadPoint”, being incremented by one on each pass. The X and Y coordinates are calculated by iterating through the X and Y values between the minimum and maximum bounds of the map using two nested loops. The map bounds are calculated using the top-left point of the map and the size of the map, this top-left point coming from the current tile’s point and the current zoom level.

Once a tile’s URL has been determined, the image is downloaded from that URL and stored in an array along with the coordinates of the tile. Once every image has been downloaded, the function “storeDownloadedTiles” is called, which in turn calls the function “storeDownloadedTileAtPos”, passing “0” as a parameter.

```
function storeDownloadedTileAtPos(downloadedTilePos) {
    if (downloadedTilePos < downloadedTilesToStore.length) {
        var tileImageString = downloadedTilesToStore[downloadedTilePos].image;
        var tileImagePointString = getPointString(
            downloadedTilesToStore[downloadedTilePos].point);
        var objectStore = getObjectStore();
        objectStore.transaction.tilePos = downloadedTilePos;
        objectStore.put(tileImageString, tileImagePointString).onsuccess =
            function(event) {
            storeDownloadedTileAtPos(this.transaction.tilePos + 1);
        };
    } else {
        downloadedTilesToStore = [];

        $("#downloadProgress").attr("value", 0);
        $("#progressLabelContainer").css("visibility", "hidden");
        $("#disableBox").css("visibility", "hidden");
    }
}
```

"storeDownloadedTileAtPos" uses recursion to store each downloaded tile image in the database. Putting data into an IndexedDB database is an asynchronous process, so I used recursion to ensure the next image is stored only once the current image has been put into the database. The function calls itself within the "onsuccess" callback of the object store's put function, called once the image and its relevant coordinates have been successfully put into the database.

For the above functions to ever be called, "downloadPoint" of course needs to initially be called by another function, this function being "downloadVisibleArea".

```
function downloadVisibleArea(zoomLevelsToDownload, layer) {
    $("#sliderButtonContainer").css("visibility", "hidden");
    $("#progressLabelContainer").css("visibility", "visible");
    $("#downloadProgressLabel").text("Downloading...");

    var zoomLevel = layer._map.getZoom();

    $.each(tilePoints, function(index, tilePoint) {
        downloadPoint(tilePoint, zoomLevel + 1, zoomLevel,
            zoomLevelsToDownload, index == tilePoints.length - 1, layer);
    });
}
```

This function is passed the number of zoom levels for which to download tiles, as specified earlier on in the download initialisation process by the user, and iterates through every tile currently visible on the map. For each of these, "downloadPoint" is called, beginning the iterative and recursive process to download every tile for the zoom levels specified.

The download process starts with the user clicking the "Download" button after selecting for how many zoom levels they want to download tiles.

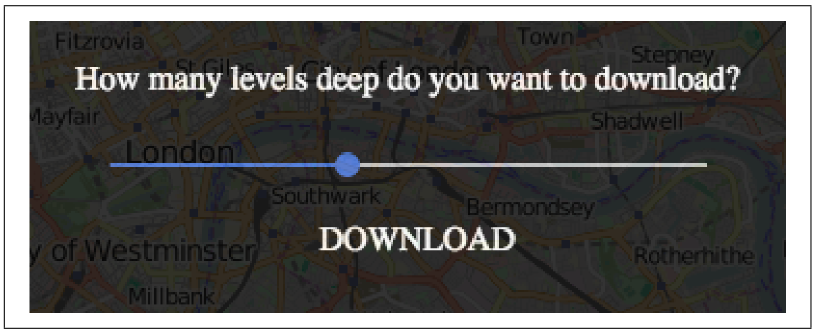


Figure 3.3: Zoom level selection slider and "Download" button.

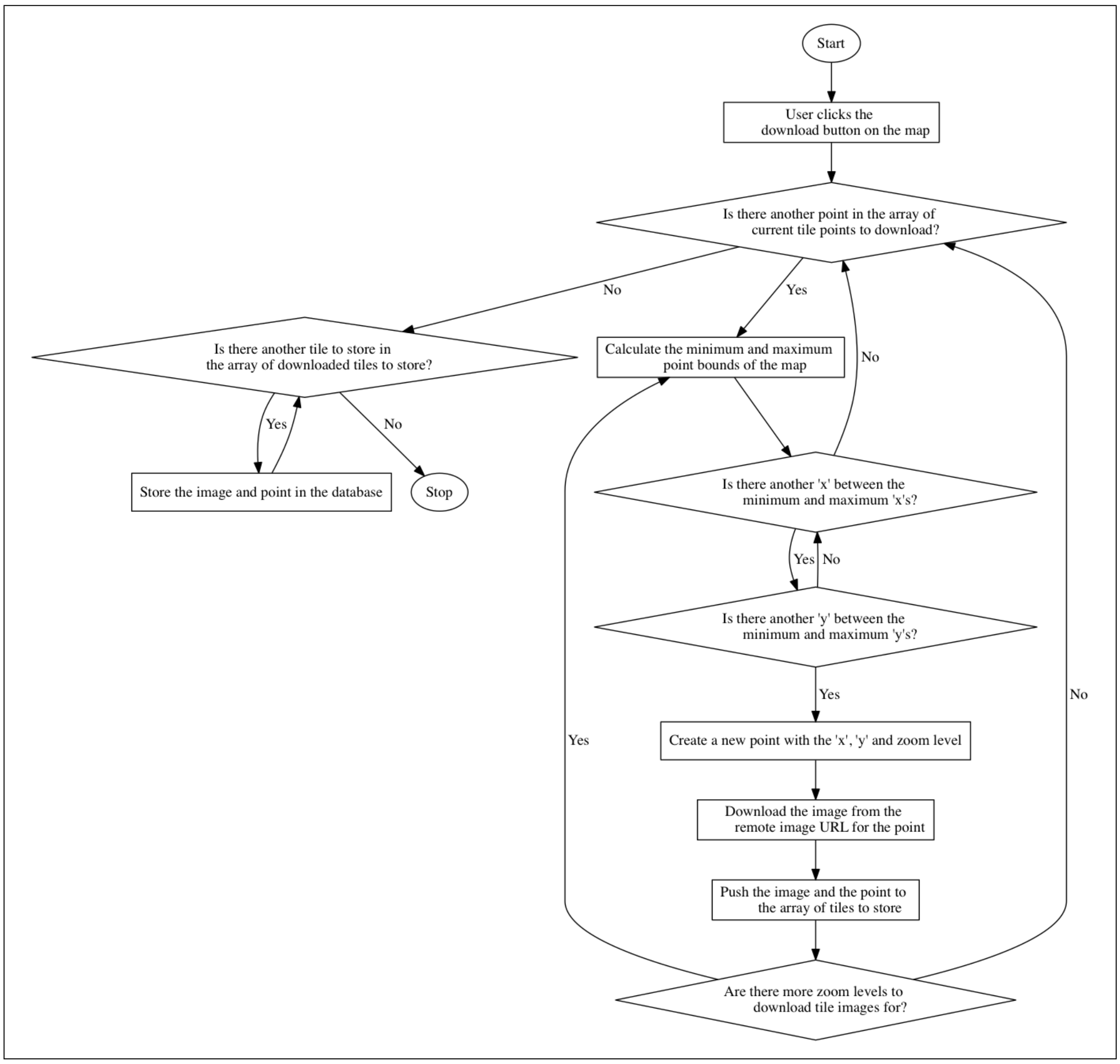

Start
User clicks the download button on the map
Is there another point in the array of current tile points to download?
No
Yes
Is there another tile to store in the array of downloaded tiles to store?
Calculate the minimum and maximum point bounds of the map
No
Yes
No
Store the image and point in the database
Stop
Is there another 'x' between the minimum and maximum 'x's?
Yes
No
Is there another 'y' between the minimum and maximum 'y's?
Yes
No
Yes
Create a new point with the 'x', 'y' and zoom level
Download the image from the remote image URL for the point
Push the image and the point to the array of tiles to store
Are there more zoom levels to download tile images for?

Figure 3.4: Control flow diagram for downloading the currently visible area of the map.

3.3.3 Issues and Complexity

The development of the algorithm to calculate tile image URLs was subject to many difficulties, often revolving around the fact I was making use of many functions in the Leaflet library not designed to be used outside of Leaflet itself, resulting in sparse documentation. Leaflet is not designed for the purpose of downloading large amounts of data at once.

I started looking into how to download tiles at a higher zoom level than the current one, exploring functions in Leaflet like ‘pointToLatLng‘, which could be used to get the latitude and longitude of a point, the ‘x‘ and ‘y‘ of a tile. I considered that this latitude and longitude in conjunction with a higher zoom level could be used in a similar way to how Leaflet‘s code uses them, so that where Leaflet would produce the URLs for the images of the new tiles it creates, my program would simply download the images for newly calculated positions, without actually creating new tiles. This process could repeat until it has been completed for every zoom level selected by the user, resulting in the download of all available images at all selected zoom levels for the area visible to the user. Based on this rough idea of where to start, I wrote pseudocode for the algorithm I wanted to implement, while looking into the Leaflet JavaScript file to attempt to understand what certain functions did, how they interacted with each other and if they could be useful in building the algorithm.

```
for currentZoomLevel to maxZoomLevel
    zoomLevel = currentZoomLevel + 1

    if zoomLevel <= maxZoomLevel
        for each tile point
            latLng = pointToLatLng(point)
            topLeftPoint = getNewTopLeftPoint(latLng)
            pixelBounds = get pixel bounds for top left point
            tileSize = get tile size for zoom level
            pointBounds = calculate using pixelBounds and tileSize
            centrePoint = get centre of pointBounds

            for pointBounds minimum Y to maximum Y
                for pointBounds minimum X to maximum X
                    point = new Point(X, Y)
                    imageURL = getTileUrl(point)
                    download and store image at imageUrl
                end
            end
        end
    end
end
```

After implementing my first version of the algorithm, it was not successfully obtaining URLs for the correct images i.e. the expected area on the map. I discovered part of the reason for this is that the ‘pointToLatLng‘ function does not work correctly without some additional work first. The ‘x‘ and ‘y‘ of a point is not the same for all services providing map images, OpenStreetMap in my case, hence some mathematics is required to manually convert them into the correct latitude and longitude respectively. I found an example of the code required to do this on OpenStreetMap‘s wiki [8], and implemented a function to convert a point to a new set of coordinates based on this.

```
function getPointToLatLng(point) {
    var lng = point.x / Math.pow(2, point.z) * 360 - 180,
        n = Math.PI - 2 * Math.PI * point.y / Math.pow(2, point.z),
        lat = 180 / Math.PI * Math.atan(0.5 * (Math.exp(n) - Math.exp(-n)));

    return new L.LatLng(lat, lng);
}
```

The main ongoing issue with downloading mass amounts of OpenStreetMap data is with the time and space complexity. Downloading large amounts of data requires a long wait for the user and enough remaining storage space on their device. This was my reasoning for implementing the ability for the user to choose how much data they want to download. As well as taking a prolonged period of time to run, because the algorithm goes iteratively through every point and recursively through every zoom level, it is highly resource intensive. For example, if it is run for zoom levels 13 through 18, so for 6 zoom levels in total, there are almost 3 million images to download: assuming 12 tiles are originally visible, 12^6 = 2,985,984. This is one reason it is important to provide the user with a progress bar: so they can see how much data has been downloaded so far of the total data to be downloaded.

3.3.4 Conclusion

Implementing functionality using a library not designed to be used to perform said functionality was a difficult but enjoyable challenge, and has become the most important and interesting aspect of the project. A possible future development would be to explore methods of improving the efficiency of the mass download process, in terms of both time and space.

3.4 Getting Location Names from Coordinates

3.4.1 Introduction

I have researched and implemented a method for getting a name for the location currently visible on the map. This method uses an API I have built server-side to convert a latitude and longitude provided by the map to a location name, by querying a database of locations and their coordinates.

3.4.2 Converting Coordinates to Names

I started looking at how to convert coordinates to country and/or city names for the purpose of displaying to the user where the map is currently focused geographically. I considered that this could be done using reverse geocoding, the process of using geographic coordinates to find a description of a location, usually a place name [9]. Google provides an API for geocoding, but I decided against using it as it would require a dependence on their services, and my project could be affected by their rate limit of only “2,500 free requests per day”, especially during testing [10]. Instead, I looked at using a daily dump of city data provided by GeoNames [11]. From a dump, my server could update a table in a SQL database. This database could then be accessed client-side in the browser via an API.

The server needs a means by which to quickly and efficiently query the data, in terms of putting in a set of coordinates and getting out the name of a location. To accomplish this, I opted to use a PostGIS database [12]. PostGIS is an extension for PostgreSQL that adds support for geographic objects, allowing location queries to be run in SQL. Setting up PostGIS on a PostgreSQL database requires installing multiple prerequisites from source, and I found parts of a blog post on installing and configuring PostgreSQL and PostGIS to be a useful resource to help with this process [13].

Before I could query the database, I needed to create a table in which to store the city data and a way to load the data into it from the dump text file. I did this by writing and running two SQL scripts.

```
CREATE DATABASE offlinemaps;
\c offlinemaps
CREATE EXTENSION postgis;
CREATE TABLE geoname (
    geonameid INT,
    name varchar(200),
    asciiname varchar(200),
    alternatenames varchar(5000),
    latitude DECIMAL(10,7),
    longitude DECIMAL(10,7),
    featureclass char(1),
    featurecode varchar(10),
    countrycode char(2),
    cc2 char(60),
    admin1code varchar(20),
    admin2code varchar(80),
    admin3code varchar(20),
    admin4code varchar(20),
    population bigint,
    elevation INT,
    gtopo30 INT,
    timezone varchar(100),
    modificationdate date
);
```

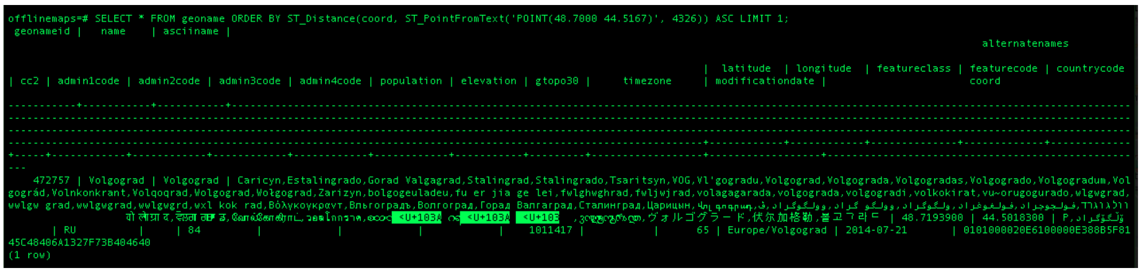
This SQL starts by creating a new database. It then creates a table with columns into which the data from the dump file can be put, using appropriate data types.

```
ALTER TABLE geoname DROP COLUMN IF EXISTS coord;
\COPY geoname FROM 'cities1000.txt' NULL AS '';
SELECT AddGeometryColumn('geoname', 'coord', 4326, 'POINT', 2);
UPDATE geoname SET coord = ST_PointFromText(
    'POINT(' || latitude || ' ' || longitude || ')', 4326);
```

This SQL starts by copying the data from the dump text file into the table. Each location (each row of the table) now needs to have a point based on its latitude and longitude that allows for the table to be queried using PostGIS. This point is of type "geometry", a type specified by PostGIS that can be used for performing queries on a point. I use the PostGIS function "AddGeometryColumn" [14] to add a column to the table that holds these points. The points are created from the latitude and longitude of each location. A query to select the name of a location based on a set of coordinates can now be written.

```
SELECT * FROM geoname ORDER BY ST_Distance(coord,
ST_PointFromText('POINT(48.7000 44.5167)', 4326)) ASC LIMIT 1;
```

The example SQL query above selects the location in the table nearest the coordinates specified, by calculating the distance between every location and the coordinates, ordering the locations by this value in ascending order with "ORDER BY ... ASC" and limiting the number of results returned to one with "LIMIT 1". For calculating the distance, first, a new point is created from the specified coordinates using the PostGIS function "ST_PointFromText" [15], which returns a geometry object. Now, the distance between a location's point and the point that has just been created can be determined. This is done using the "ST_Distance" function [16].



```
offlinemaps=# SELECT * FROM geoname ORDER BY ST_Distance(coord, ST_PointFromText('POINT(48.7000 44.5167)', 4326)) ASC LIMIT 1;
 geonameid |   name    | asciiname |
                                                                                                          alternatenames
                                                                                  |  latitude  | longitude  | featureclass | featurecode | countrycode
| cc2 | admin1code | admin2code | admin3code | admin4code | population | elevation | gtopo30 |     timezone     | modificationdate |                        coord

-----------+-----------+-----------+------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+------------+------------+--------------+------------
+-----+------------+------------+------------+------------+------------+------------+---------+------------------+------------------+----
---
    472757 | Volgograd | Volgograd | Caricyn,Estalingrado,Gorad Valgagrad,Stalingrad,Stalingrado,Tsaritsyn,VOG,Vl'gogradu,Volgograd,Volgograda,Volgogradas,Volgogrado,Volgogradum,Vol
gográd,Volnkonkrant,Volqoqrad,Wolgograd,Wołgograd,Zarizyn,bolgogeuladeu,fu er jia ge lei,fwlghwghrad,fwljwjrad,volagagarada,volgograda,volgogradi,volkokirat,vu~orugogurado,wlgwgrad,
wwlgw grad,wwlgwgrad,wwlgwgrd,wxl kok rad,Βόλγκογκραντ,Вльгоградъ,Волгоград,Горад Валгаград,Сталинград,Царицын,Վոլգոգրադ,ف,ولگوگراد,وولگو گراد,وولگوگراد,ولگوگراد,فولغوغراد,فولجوجراد,פולגוגראד,ווֹלגאגראד
              वोल्गोग्राद,ਵੋਲਗੋਗਰਾਡ,வோல்கோகிராட்,วอลโกกราด,ဗော <U+103A ဂ <U+103A <U+103 ,ვოლგოგრადი,ヴォルゴグラード,伏尔加格勒,볼고그라드 | 48.7193900 | 44.5018300 | P,ولگوگراد
       | RU          |     | 84         |            |            |            |    1011417 |           |      65 | Europe/Volgograd | 2014-07-21       | 0101000020E6100000E388B5F81
45C48406A1327F73B404640
(1 row)
```

Figure 3.5: Result of a sample query.

3.4.3 Requesting Location Names Client-Side

I thought it would be best to create an API which the browser can send request to, passing a latitude and longitude and having returned the name of the nearest location to those coordinates. I decided to build this API using Ruby and Sinatra, as I have experience using the technologies.

Sinatra is a domain-specific language for easily creating web applications in Ruby [17]. At the top of my Ruby file, I set up Sinatra.

```
require 'sinatra'

set :port, 8110
set :environment, :production
```

This sets the port on which to run the Sinatra instance to 8110. As the domain name "dylanmaryk.com" points to my server, the URL to which to send requests is therefore "http://dylanmaryk.com:8110". I can now determine what to do and what data to return when a request is sent to this URL.

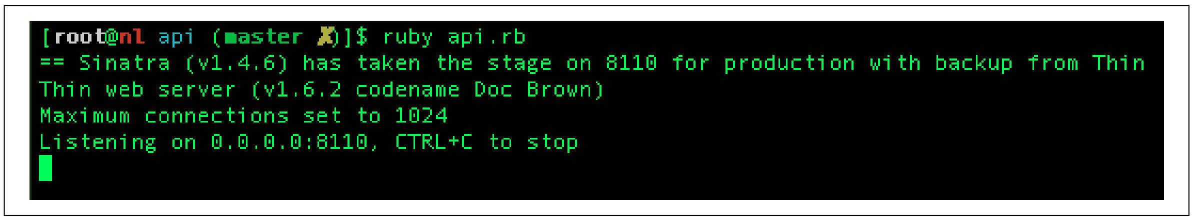
```
require 'pg'

...

get '/name' do
    conn = PG::Connection.open(:dbname => 'offlinemaps', :user => 'postgres',
        :password => 'offlinemaps')
    res = conn.exec_params('SELECT * FROM geoname ORDER BY ST_Distance(
        coord, ST_SetSRID(ST_MakePoint($1, $2), 4326)) ASC LIMIT 1',
        [params[:lat], params[:long]])
    res[0]['name']
end
```

The code inside "get '/name' do" is run when a request with parameter "name" is received: "http://dylanmaryk.com:8110/name". I make use of the PG library, Ruby's library for interfacing with PostgreSQL. This allows me to run a SQL query whenever the API receives a request, the result of which can then be returned to the client. My main source of help when utilising PG was the documentation for the "Connection" class [18], for connecting to and querying the database with the parameters passed in the request. "$1" and "$2" represent the values of the parameters passed, which are the latitude and longitude from the map client-side. Once the query has returned a result, the "name" of the first row of the results is returned to the client. The Ruby file is then run.

```
ruby api.rb
```



```
[root@nl api (master ✗)]$ ruby api.rb
== Sinatra (v1.4.6) has taken the stage on 8110 for production with backup from Thin
Thin web server (v1.6.2 codename Doc Brown)
Maximum connections set to 1024
Listening on 0.0.0.0:8110, CTRL+C to stop
```

Figure 3.6: Running Sinatra process.

I have written a JavaScript function client-side that handles sending a request to the server with the latitude and longitude of the current centre of the map as parameters. I used the documentation for jQuery to help me implement a GET request [19].

```
function displayLocation(map) {
    var mapCenter = map.getCenter();
    var mapLat = mapCenter.lat;
    var mapLng = mapCenter.lng;

    $.get("http://dylanmaryk.com:8110/name?lat=" + mapLat + "&long=" + mapLng,
        function(data) {
        $("#locationLabel").text(data);
    });
}
```

When the client receives a response from the server, it displays the response text to the user.

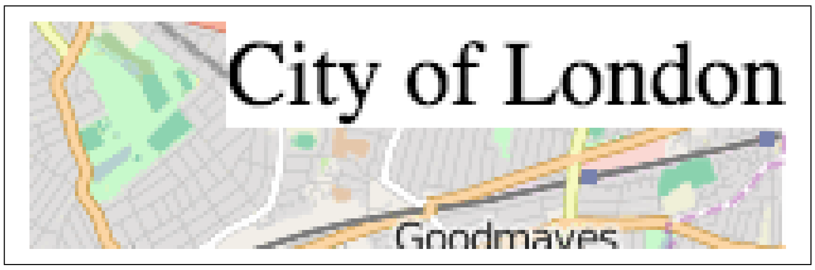


Figure 3.7: Display of location name.

3.4.4 Conclusion

Using a server to find a location name for a given set of coordinates allows for improved computational efficiency, as it means the process of performing a lookup on a vast quantity of data is delegated to a computer that is more powerful than a browser, especially on a device with less computational power such as a smartphone. This ensures searches are faster and do not interfere with user interaction and other functions being carried out by the browser, such as caching tiles on the map.

Chapter 4: **Installation Manual**

Note: Following the instructions below is optional. The project can also be used at the following URL, without any setup required: **http://dylanmaryk.com/offlinemaps/main/**. Note that **Chrome is the preferred browser** to use, as this is the browser in which the project has been tested most.

4.1 Front-End

The front-end of the website works without any modifications if run on a web server. Simply upload all the files in the "program" directory to your web server and navigate to "index.html". However, if you run it on a local machine, on Chrome at least, you will encounter the following error in the console.

```
Imported resource from origin 'file://' has been blocked from loading by
Cross-Origin Resource Sharing policy: Invalid response.
Origin 'null' is therefore not allowed access.
```

This occurs because the resources for the slider and button UI elements for the download process are being imported from a URL beginning with "file://" when the page is loaded on a local machine, which is considered an invalid origin. The workaround for this is to open Chrome using the following command on Windows, after navigating to the directory in which the Chrome executable is located.

```
chrome.exe --allow-file-access-from-files
```

Or, the following command on Mac.

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome
--allow-file-access-from-files
```

Now navigate to "index.html" and the website will be usable locally on your machine.

4.2 Back-End

The API for providing the name of a location for a given set of coordinates can be set up on a Linux server using the following instructions.

First, follow the instructions at http://codingsteps.com/installing-and-configuring-postgresql-in-amazon-linux-ami/ to install and configure a standard installation of PostgreSQL and PostGIS. Then, login to PostgreSQL. Make sure you are in the "program" directory.

```
sudo -u postgres psql
```

Enter the following commands to enable PostGIS, create the database and table into which the location data will be inserted and quit PostgreSQL.

```
\i scripts/creategeonames.sql
\q
```

Make sure you are still in the "program" directory. Run the bash script that gets the latest location data and inserts it into the table. You will be prompted to enter your PostGIS password during the process.

```
bash updatedata.sh
```

Now that the database has been configured and populated, open "api/api.rb". Modify the password field so that your own PostGIS password is used when connecting to the database.

```
conn = PG::Connection.open(:dbname => 'offlinemaps', :user => 'postgres',
    :password => '<YOUR PASSWORD>')
```

Assuming you have Ruby installed on your system, you still need to install the dependencies required for the API to run.

```
gem install pg
gem install sinatra
gem install sinatra-cross_origin
```

To ensure your front-end is sending requests to your own Sinatra instance on your web server, open "js/offlinemaps.js" and modify the URL to which GET requests are sent.

```
$.get("http://<YOUR DOMAIN NAME>:8110/name?lat="
    + mapLat + "&long=" + mapLng, function(data) {
```

Finally, run your Sinatra instance.

```
ruby api/api.rb
```

Chapter 5: **Professional Issues**

5.1 Privacy

Arguably the most important ethical issues regarding mapping software revolve around privacy concerns. Since Edward Snowden leaked classified documents on the United States intelligence organisation the National Security Agency (NSA), there has been growing public interest in the collection and possible misuse of private data, both by governments and private companies. In 2013, certain documents released by Snowden revealed that the NSA was "gathering nearly 5 billion records a day on the whereabouts of cellphones around the world" [20], showing their mass collection of location data. Google openly tracks the locations of users' devices, providing their users with a detailed map of where they were at specific dates and times [21]. Many including myself believe that collection of people's private data is becoming overly and unnecessarily intrusive, and I have followed through with this ethical stance in the development of my project. I do not have access to information on what map data the user has downloaded or where on the map they have viewed, nor to any general metadata that could help to identify a user, such as the operating system or browser they are accessing the website on. I have not implemented any means of capturing this data, nor of uploading it to my server. Originally, when a request to the location name API was made, the IP address from which the request originated and the values of the latitude and longitude parameters were printed in the terminal. While this data was never stored permanently on the server, it is now not even printed to the terminal, so it cannot be viewed temporarily.

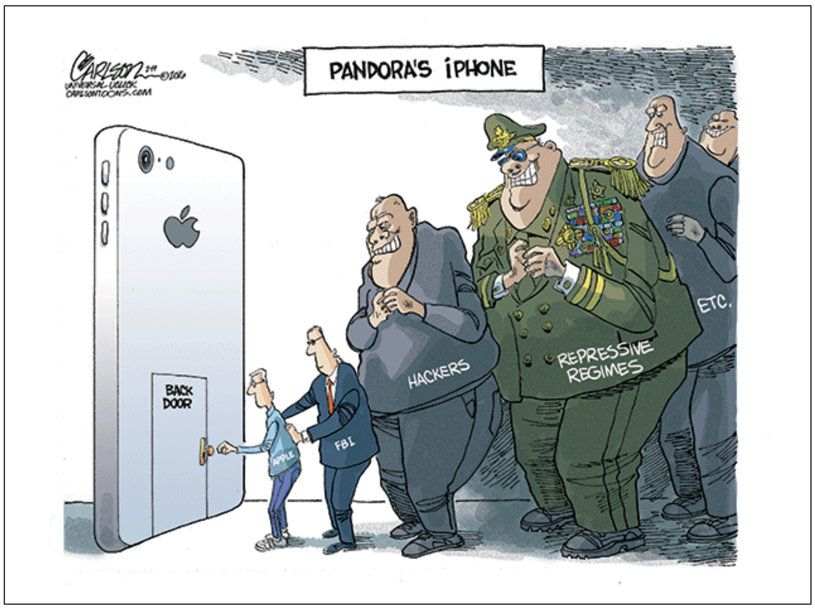


Figure 5.1: Stuart Carlson's "Trap Door" [22].

The above cartoon, commenting on the story of the Federal Bureau of Investigation (FBI) requesting that Apple unlock a suspect's iPhone for them [23], represents one of my favourite arguments for keeping the user's privacy as intact as possible. If a service collects too much data for itself, that data then becomes more easily accessible for outside organisations such as governments. One of the easiest methods of protecting users from having their data retrieved by unauthorised parties is to not store any data not necessary for the running of the service in the first place.

5.2 Reliability

It is important that a service which users depend on is considered a reliable product. Otherwise, not only will users begin to lose confidence in the service, and therefore use it less, but it is arguably simply a matter of principle that a developer should care that their users are satisfied with their experience of a product. One key aspect of a reliable service is minimal downtime, which is something I have tried to achieve by relying on third-party services as little as possible. Indeed, the concept of caching remote data locally itself encourages minimal dependence on even a connection to the Internet, let alone third-party data providers.

One decision I made regarding reliability was not to rely on Google‘s geocoding API for converting coordinates into addresses. I thought that this would require too heavy a dependence on Google for functionality fundamental to my project. Instead, I designed my own API to perform similar functionality, using data stored in a database on my own server. This means should there be any server-side failures in the system, I can address them directly myself, without having to contact a third-party to assist in rectifying an issue.

5.3 Licensing

Developers of any software are able to include a license, a “legal instrument governing the use or redistribution of software”. Not only is there a legal necessity to use developers‘ software in a manner in which they have permitted, but an ethical one too. It should be up to the creator and owner of any piece of work to determine how they want anyone else to use that work. Hence, I have used any third-party libraries and assets as permitted and have included the license itself where necessary.

OpenStreetMap says that you are “free to copy, distribute, transmit and adapt our data, as long as you credit OpenStreetMap and its contributors” under the Open Data Commons Open Database License (ODbL) [24]. They require the use of the credit “ OpenStreetMap contributors”, and if you are using their map tiles, as I am, you must link to their copyright page. Hence I have added this attribution with a link to the copyright page in the bottom-right corner of the website.

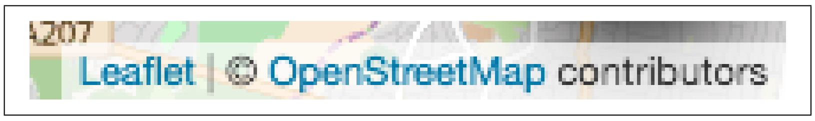


Figure 5.2: OpenStreetMap attribution text.

Leaflet permits the “redistribution and use” of the library “with or without modification”, providing that the “copyright notice, this list of conditions and the following disclaimer” are retained in redistributions of the source code. I have therefore included the license in my distribution of the library.

jQuery uses the MIT license, stating that “you are free to use any jQuery Foundation project in any other project (even commercial projects) as long as the copyright header is left intact” [25]. I have therefore simply left the copyright header intact in my copy of the jQuery source file.

Finally, for my slider and button UI elements, I am using the Paper Elements library from the Polymer Element Catalog, a set of visual elements that implement Google‘s Material Design. They use the BSD license, which just like Leaflet‘s license says that the “copyright notice, this list of conditions and the following disclaimer" must be retained, hence I have included their license in my project.

5.4 Plagiarism

It is vital to give credit to the author of a piece of code for their work. Passing off another developer‘s work as one‘s own would be highly unethical, as this equates to theft. You are taking something that was not originally yours and claiming it as your own.

In this report, I give credit where it is due for snippets of code I did not write. For example, I give credit to the OpenStreetMap wiki for a JavaScript function I implement for converting X and Y coordinates to latitude and longitude coordinates.

Chapter 6: **Self-Evaluation**

I would consider the project to have been an overall success. I was able to complete the original specifications, both the early and final deliverables, and go on to exceed them by implementing additional functionality and making continuous improvements throughout development. In terms of functionality, the project works as I intended, with data being cached while the user navigates the map plus the added ability to download a specific area of the map in as much detail as the user desires. I was able to improve aspects of the project between adding new features, such as when refactoring much of my original caching code. While originally I was attempting to get remote OpenStreetMap data before loading locally cached data, I reversed the two processes as recommend while I was presenting my project during the first term, so that it now attempts to load a local copy of the data first.

There were some additional features and optimisations I had intended to implement, but unfortunately was unable to due to time constraints, such as the ability for the user to enter the name of a location and have the map move to that location, based on the coordinates of the location. These coordinates would have been retrieved via an extension to the location name API I built, which would have done the reverse of the current implementation of the API, so instead of searching for a location by its coordinates, the API would also allow for searching for a set of coordinates based on a name. I still intend to implement this functionality at some point in the future, as well as to make my project public and publish the source code.

One thing in particular I believe I did right was to allow the user to select how many zoom levels deep into the area they are currently focused on they want to download data for, as this provides the user with more choice as to how they want to use the service. At the same time however, it would have been a good idea to put some sort of restriction on how many zoom levels the user can select at a maximum, as as I explain in my implementation section, there are efficiency problems as the amount of data to download increases. It may therefore have also been preferable to dedicate more time to investigating how to combat these problems with the amount of time downloading takes.

One particular thing I do not believe I did right was that I failed to dedicate enough time to testing my project in multiple browsers. While the project works fine in Chrome, it often fails to work properly in browsers such as Firefox and Safari when it comes to functionality like caching data offline. This is probably the most serious flaw of my project, as it reduces the potential number of users and diminishes the user experience.

I learnt much from the experience of working on a large project over an extended period of time. For one, I realised that time management is key. While I was able to achieve a large amount of work, I should have better planned exactly what aspects of the project I would work on at what times in advance, considering time estimates for each part. This many have resulted in me having time to add some more functionality and polish. I also learnt the need to review my own work frequently, revisiting code I have written and improving it often, as there is always something that can be made better, such as in terms of efficiency or just cleaner code. Additionally, there is the fact that this piece of work will be viewed by others, and hence it is especially important to acknowledge the work of others, within the codebase and the report, and to make sure both the code and written report elements of the project are well laid out, follow standards and are readable.

Chapter 7: **Bibliography**

[1] Maps for mobile: Download a map and use it offline

https://support.google.com/gmm/answer/3273567?hl=en-GB

[2] W3Schools: HTML5 Application Cache

http://www.w3schools.com/html/html5_app_cache.asp

[3] Mozilla IndexedDB documentation: Browser storage limits and eviction criteria

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria#Storage_limits

[4] Leaflet: Open-source JavaScript library for mobile-friendly interactive maps

http://leafletjs.com/

[5] Leaflet Documentation: TileLayer

http://leafletjs.com/reference.html#tilelayer

[6] Mozilla IndexedDB documentation: IDBOpenDBRequest

https://developer.mozilla.org/en-US/docs/Web/API/IDBOpenDBRequest

[7] Mozilla HTML documentation: CORS enabled image

https://developer.mozilla.org/en-US/docs/Web/HTML/CORS_enabled_image

[8] OpenStreetMap wiki: Slippy map tilenames

https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#ECMAScript_.28JavaScript.2FActionScript.2C_etc..29

[9] Wikipedia: Geocoding

https://en.wikipedia.org/wiki/Geocoding

[10] Google Maps Geocoding API: Usage Limits

https://developers.google.com/maps/documentation/geocoding/usage-limits

[11] GeoNames: City data dump

http://download.geonames.org/export/dump/

[12] PostGIS: Spatial and Geographic Objects for PostgreSQL

http://postgis.net/

[13] Coding Steps: Installing and Configuring PostgreSQL and PostGIS on Amazon Linux

http://codingsteps.com/installing-and-configuring-postgresql-in-amazon-linux-ami/

[14] PostGIS documentation: AddGeometryColumn

http://postgis.net/docs/AddGeometryColumn.html

[15] PostGIS documentation: ST_PointFromText

http://postgis.net/docs/ST_PointFromText.html

[16] PostGIS documentation: ST_Distance

http://postgis.net/docs/ST_Distance.html

[17] Sinatra: README

http://www.sinatrarb.com/intro.html

[18] PG documentation: Connection

http://deveiate.org/code/pg/PG/Connection.html

[19] jQuery API Documentation: jQuery.get()

https://api.jquery.com/jquery.get/

[20] The Washington Post: NSA tracking cellphone locations worldwide, Snowden documents show

https://www.washingtonpost.com/world/national-security/nsa-tracking-cellphone-locations-worldwide-snowden-documents-show/2013/12/04/5492873a-5cf2-11e3-bc56-c6ca94801fac_story.html

[21] Google Maps: Edit or delete your timeline

https://support.google.com/maps/answer/6258979

[22] Stuart Carlson: Trap Door

http://www.carlsontoons.com/uncategorized/trap-door/

[23] Wikipedia: FBI-Apple encryption dispute

https://en.wikipedia.org/wiki/FBI-Apple_encryption_dispute

[24] OpenStreetMap: Copyright and Licence

http://www.openstreetmap.org/copyright

[25] jQuery Foundation: License

https://jquery.org/license/