Final Year Project Report

Full Unit - Final Report

Offline HTML5 Map

Andreea Gardelean

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Francisco Ferreira Ruiz

Department of Computer Science
Royal Holloway, University of London

April 3, 2024

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 20525

Student Name: Andreea Gardelean

Date of Submission: 05/04/2024

Signature: Andreea G.

Table of Contents

Abstract

Offline applications make a pivotal shift in user experience, internet connectivity is no longer a boundary for a seamless user experience. With an offline-first approach, internet connectivity is rather an enhancement than a necessity.

This report documents the work done to achieve an offline-first Progressive Web Application (PWA) map, using OpenStreetMap data, converted to vector tiles using OpenMapTiles and rendered on the client side as requested by OpenLayers. This approach leverages HTML5 technologies such as Cache API, service worker, and IndexedDB, among others, offering users a native app-like experience over the web.

The report provides an introduction to the project and outlines the objectives of the application. It begins with a literature survey presenting a description of the relevant theory such as technologies and concepts required to reach the application's objectives and demonstrates how they are used in the project.

Following this, the report details the Software Engineering methodology and practices followed throughout the project's development. This includes the methodology adhered to, documentation procedures, testing, and other software engineering practices.

Subsequently, it transitions into a comprehensive description of the development process, describing the implemented features, how they are achieved, as well as exploring alternative approaches and addressing any encountered blockers along the way.

The "End System" chapter elaborates on the application's architecture, starting with a high-level overview of the system, outlining the various components it interacts with. It subsequently offers a more granular perspective by delineating the system through UML use-case diagrams. Proceeds to the "Assessment" chapter, examining privacy in professional issues, along with a self-assessment.

In the "Appendix" section, the "User Manual" outlines the application's functionalities and features, followed by an "Installation Manual" providing steps to run the application locally. The appendix concludes with a diary summarising the tasks carried out throughout the application's development lifecycle.

Chapter 1: **Introduction**

The project aims to develop a Progressive Web Application (PWA) map featuring London area. The application has an offline-first behaviour, enabling users to study their surroundings seamlessly and use the application without relying on an internet connection leveraging the benefits of native applications with the accessibility of the web [1], providing users a native app-like experience over the web.

In addition to providing map data for offline use, the application servers as a platform for users to document their experiences and pinpoint them on the map, simplifying the process of revisiting memories and experiences from a certain location. Adding a pin to the map, the trip data is mapped to the corresponding location, facilitating visual search and trip organisation. Aims at encouraging users to make new experiences, by providing possible experience suggestions such as internet cafes, museums, theatres and malls. Additionally, users have the ability to share their trips by downloading selected trips as a PDF document.

To achieve a PWA the application must work offline, and this is achieved by leveraging HTML5 technologies such as service workers, Cache API and IndexedDB.
The service worker is registered to control one or more pages, can detect requests and send replies to these requests regardless of the user's connection [1], providing the ability to use the application when the internet connection is sparse or absent. The service worker is registered and installed in the browser to cache the necessary assets when the application is first accessed.

The Cache API provides the ability to store application assets, such as files necessary to run the application, in the browser, which can later be served to the user by the service worker as requested. IndexedDB is a transactional database allowing storage of data locally, in the browser in a structured way [9], providing users the full experience.

PWA allows users to seamlessly integrate the application onto their home screen, functionality provided by the browser throughout the provision of a manifest in JSON format [1]. This feature allows users to access the application more conveniently without having to install it from the app store.

The map is vector tile based, providing access to geographical data in a much easier fashion because map objects are represented by vector objects stored in the database [18]. To achieve full offline functionality, the geographical data is stored in the browser's database IndexedDB and rendered as requested using the OpenLayers library.

The project development process significantly enhanced my professional development, promising to benefit my future career prospects in several ways. Firstly, it allowed me to advance and learn new web technologies and skills, deepening my understanding of web application functionality, and how to integrate offline functionality to enhance user experience.
Secondly, the experience improved my problem-solving, time-management and organisational abilities. This experience allowed me to develop modern, efficient and user-friendly web application, be a more efficient and innovative developer in my future career.

Chapter 2: **Literature survey**

This chapter outlines the literature surveyed to aid in understanding the project, how it should be completed, and how the necessary technologies and tools are used to achieve the objectives set above.

2.1 Basic web page development in HTML5

2.1.1 HTML

HTML stands for Hyper Text Markdown Language, is the building block of web development which defines the structure of the web content. HTML has a series of elements which are used to define the content, enclose it or make it stand by itself.

HTML5 is the 5th iteration of the technology, which is built on top of other versions of HTML; introduces new elements, utilities, removed and improved certain functionalities [2], and is supported by all modern browsers.

Below is a basic structure of an HTML file.

```
1   <!DOCTYPE html>
2   <html lang="en">
3       <head>
4         <meta name="Author Name" />
5         <meta charset="UTF-8" />
6         <title>Offline "Hello World"</title>
7         <link rel="stylesheet" type="text/css" href="style.css" />
8         <script src="script.js"></script>
9       </head>
10
11     <body>
12     </body>
13
14  </html>
```

1 - defines the type of the document, and ensures we are using the latest version of HTML.

2 - wraps the contents of the page together, has attributes such as 'lang' which is used to define the language the document is written in.

3 - wrapper which contain metadata about the HTML document, this data is not content of the page.

4:5 - contains metadata that cannot be represented using other elements. Has attributes which can be used to specify information such as author name and document character encoding.

6 - specifies the title of the document, this data is usually displayed as the title of the browser tab.

7 - specifies a relationship between the current document and an external source. Often used to link to an external CSS file which defines the document style.

8 -imports script data into the HTML, can be specified in the head or at the end of the body.

In HTML5 new semantics, media, form, graphic elements and functionalities have been introduced [3]. Below can be found an introduction to the major additions in HTML5 as described in [3].

1. Structural elements

   New structural elements have been defined which organise the content of the page more effectively and introduce new functionality.

   <article> is used to define independent content which is not related to the other content

   <header> contains introductory information which is placed at the beginning of the document, such as page name and menu

   <section> is a section in the document which contains logically connected content

   <aside> contains information which is not directly related to the main content of the page, such as side bars

   <footer> specified before the closing body tag, contains data about the author, copyrights, and other external data related to the document content

   <progress> creates a progress bar which shows the progress of a task

2. Media elements

   <audio> and <video> elements are a major feature, allows to embed audio and video files directly from the HTML document

3. Graphic elements

   <canvas> defines an area which can be used to create different objects such as images and animations via a script

   <svg> allows to draw scalable vector graphics

4. Form elements

   <details> a container which displays the result of an operation performed by the script

   <keygen> generates key pairs used for encryption and decryption of data sent to the server when a form is submitted

Another major feature of HTML5 is the use of cache, web storage and web database which allows to use an application whilst offline. These new functionalities have been explored in more depth in 'Developing Offline HTML5 Application' report.

2.1.2 CSS

CSS stands for Cascading Style Sheets; is a programming language used to define the appearance of an HTML document or its elements when displayed on the screen [4].
Note: in this section, the code snapshots have been taken from 'To-do List' proof of concept.

CSS structure:

Within the style-sheet, a set of rules define the appearance of the document. Each rule consist of a selector, determining which element to be selected. A declaration block, denoted by , encompasses one or more declarations specifying how the element should appear. Each declaration is composed of a CSS property and a value. A basic structure is given below:

```
selector {
  property: value;
}
```

An example of how that might look like in practice:

```
#enter-task {
  padding: 10px;
  border-radius: 5px;
  margin-top: 10px;
  width: 50vw;
  background-color: red;
}
```

CSS styling can be applied to an HTML document in 3 different ways:

1. In line style-sheet
   In line CSS only applies the styling to a single element. The styling is applied directly on the target element in HTML as such:

   ```
   <p style="color:red; font-size: 60px;">Hello World</p>
   ```

2. Internal CSS
   The styling is declared within the HTML document, <style> element is used to place the styling in and is placed in the <head> element. Styling declared here can be applied to multiple elements within the same document.

3. External CSS
   The styling can be declared in an external style with the file extension .css, the file is linked to one or more HTML documents by using the <link> element.

   e.g.:
   ```
   <link rel="stylesheet" type="text/css" href="style.css" />
   ```

Next we will analyse the different ways HTML elements can be selected by the CSS.

1. Selecting by class name
   An HTML element can be selected by its class name as follow:

   ```
   .container { //styling }
   ```

   where *container* is the class name of an element.
   Note: multiple elements can have the same class name, in the case above all elements with the same class name will have the styling applied.

2. Selecting by id

   An HTML element can be selected by the id value using `#` followed by the id value. Each HTML element with an id should have a unique value, thus the styling is applied to only one element.

   ```
   #greeting { //styling }
   ```

3. Selecting by element name

   HTML elements can be selected using the element tag. In this case, all elements with the tag name will be selected, and the styling will be applied. Below, all `<label>` elements are selected and have the styling applied:

   ```
   label { //styling }
   ```

4. Attribute selector
   The element is selected using the element name with a given attribute

   ```
   input[type="radio"] { //styling }
   ```

   above all <input> elements which have the attribute *type* set to 'radio' are selected.

5. Descendant selectors
   This type of selection will select an HTML element which is contained within another element. Given the following HTML elements:

   ```
   <div>
       <p>Hello</p>
       <h1>World</h1>
   </div>

   <div>
       <p>Goodbye</p>
       <h1>Everyone</h1>
   </div>

   <h1>Today is a great day!</h1>
   ```

   if we want to apply styling only to the <h1> elements which are contained in the div we can do the following:

   ```
   div h1 { //styling }
   ```

   this way the outer header will not be affected.

2.1.3 JavaScript

JavaScript is a programming language which is used to add interactivity to a web page dynamically, this is called client-side JavaScript and server-side JavaScript is run on a server, this can be achieved by using Node.js [5].

In JavaScript the order defined in the page is the order the code runs, this is the case when we have blocks of code which are not in a function, will be executed top to bottom [5].

An important feature of JavaScript is that it does not allow to communicate with other pages and websites because the JavaScript code runs completely separate in its own environment, within the browser, for every page. This feature is a good security measure, other websites cannot see the data shared with other websites [5].

JavaScript can be defined internally in the HTML file, or in an external file.

Internal JavaScript
The script is declared in the same HTML file as follow:

```
<script>
    alert("This message comes from internal JavaScript!");
<script>
```

The script is added just before the closing <body> tag, to unsure it is not executed before the HTML has been rendered.

Having the script placed just before the closing <body> tag can have some issues for large websites. A solution to this, and a good practice is to use an event listener which listens to the event of having the HTML document completely loaded [5].

```
<script>
    document.addEventListner("DOMContentLoaded", () => {
        alert("This message comes from JavaScript!");
    });
</script>
```

External JavaScript
The JavaScript code is written in a file with the extension *.js*, the code is added to this file and is linked to the HTML document using the <script> element as shown below:

```
<script src="js/script.js"></script>
```

The script can be placed in the <head> tag, if the JavaScript needs to be executed before the document has been rendered, or before the closing body tags, this way the script will not be executed until the HTML document has been rendered.

Adding the script as shown in the snapshot above before the closing body tags, can cause issues for large websites, with a lot of content, a solution is to add the *defer* attribute inside the script tag [5]:

```
<script src="js/script.js" defer></script>
```

Using the keyword *defer* will tell the browser to continue download the HTML content once the script tag has been reached [6], this way the script and HTML document will load at the same time.

JavaScript client-side
As mentioned above, client-side JavaScript is used to add interactivity to the HTML elements dynamically, and runs on the client's browser.

In order to manipulate HTML and CSS we first need to access the element we want to modify from the document. In JavaScript this can be done by using Document Object Model (DOM) API. DOM represents an HTML document as nodes and objects, this way programming language like JavaScript can interact with the page elements [6].

DOM can be accessed in JavaScript using 'document' or 'window' objects.
Given the minimal HTML page:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>JavaScript interaction example</title>
    </head>

    <body>
        <p id='greeting'>Hello World</p>
        <script src="script.js"></script>
    </body>
</html>
```

The following actions can be performed on the document in JavaScript:

1. Alter element content

   ```
   const greeting = document.getElementById('greeting');
   greeting.textContent = 'This message has been written in JavaScript.';
   ```

   The code above will change the message the paragraph contains.

2. Get an element from HTML document:

   ```
   1  document.getElementById('id');
   2  document.getElementsByClassName('className');
   3  document.getElementsByName('name');
   4  document.getElementsByTagName('tag');
   ```

   1 - gets an element based on its unique id

   2 - returns a list of elements with the specified class name

   3 - returns a list of elements which have the same name i.e: elements with attribute 'name'

   4 - returns a list of elements with the same element value, for example `<p>` tags

3. Add new elements to the HTML

```
const newElement = document.createElement('p');
newElement.textContent = 'Element created in JavaScript';
document.body.appendChild(newElement);
```

   The code snapshot above creates a new paragraph element with text content, and uses *appendChild()* to append the new element to the body of the document.

4. Add CSS from JavaScript

   JavaScript can also be use to alter the CSS of an element as follow:

```
const element = document.getElementById('greeting');
element.style.color = 'red';
```

   To update the CSS the syntax *element.style.property = style* is used. *.style* accesses the style of the element and the *property* is the property of the element we want to update, in the example above the colour.

5. Event listener

   Event listeners can be added to an element, and whenever the added event happens on the element a function is executed. The event is added using the *addEventListener()* method of the EventTarget interface, and takes 2 parameters, the event to be registered and a function with actions it should take when the event occurs.

   For instance:

```
const element = document.getElementById('greeting');
element.addEventListener('click', () => {
    element.style.color = 'red';
});
```

   The code above will add a *click* event listener to the paragraph, and when is clicked the text colour will change.

HTML, CSS and JavaScript are the building blocks of web development. We explored the structure of an HTML file and the new semantics added in HTML5. The CSS section delved into various styling methods and element selection. Lastly, JavaScript section discussed its role and how to add interactivity to web pages, and alter their content.

2.2 Advanced technologies

2.2.1 HTML5 canvas

HTML5 introduced a new element <canvas> is described in [7] as an immediate mode bitmap area on the page which can be manipulated with JavaScript. Canvas allows to draw paths, boxes, circles, text and add images.
The canvas can be defined as follows in the <body> section:

```
<canvas id="canvas" width="1000px" height="600px"></canvas>
```

Must have a width and height defined, a unique id is necessary so that it can be uniquely identified in JavaScript.

The <canvas> elements is not supported by all browsers, if that is the case a fallback text can be specified in between canvas tags, if canvas is not supported the text will be displayed [7]. An example is shown below:

```
<canvas id="canvas" width="1000px" height="600px">
    This is a message in between the canvas tags which will appear in the
    browser page only if canvas is not supported by the current browser.
    Thus if you see this message update the browser version you are using.
</canvas>
```

Next we will have a look at how the canvas can be manipulated in JavaScript.

```
1    const canvas = document.getElementById("canvas");
2
3    if (!canvas || !canvas.getContext) {
4        console.error("Canvas not supported by current browser,
5        or canvas element does not exist in the HTML.");
6    };
```

1 - the *canvas* variable holds the reference to the canvas object, which was retrieved using DOM
3 - the first condition of the if statement tests if the canvas object exists in the HTML document. The second condition tests if the canvas context exists

Next we need to reference to the 2D context object which has tools for drawing on the canvas:

```
const context = canvas.getContext('2d');
```

In the following sections we will explore how to draw different objects on the canvas.

1. Draw text

```
const drawText = () => {
    context.fillStyle = "#000000";
    context.font = "20px Sans-Serif";
    context.fillText("HelloWorld", 195, 80);
};
```

The function above draws the text 'Hello World' on the canvas, at coordinates (195, 80) with font size 20. *fillStyle* specifies the text colour.

2. Draw image

```
1 const drawImage = () => {
2     const canvasImage = new Image();
3
4     canvasImage.onload = () => {
5         context.drawImage(canvasImage, 10, 500);
6         context.drawImage(canvasImage, 10, 500, 200, 300);
7         context.drawImage(canvasImage, 900, 800, 1000,
                                         2000, 300, 500, 200, 300);
8     };
9    canvasImage.src = "path/to/image/source";
};
```

The function above will draw an image on the canvas from local files on the canvas. Above are three distinct approaches:

On line 5, the image is drawn on the canvas at coordinates(10, 100).

On line 6, the image is drawn on the canvas at coordinates (10, 500), with a width of 200 and height of 300.

On line 7 the image will be cropped starting at coordinates (900. 800) for a widths of 1000 and height of 2000. The resulting image is then positioned at coordinates (300, 500) on the canvas, with a size of width 200 and height 300.

3. Draw rectangular

To draw a rectangle more easily, a dedicated method is available. Below is a complete code snippet for drawing this shape.

```
1 const drawRect = () => {
2     context.strokeStyle = "#FF0000";
3     context.fillStyle = "#FF0000";
4
5     context.strokeRect(0, 0, 980, 580);
6     context.fillRect(0, 0, 100, 100);
7     context.clearRect(10, 10, 50, 50);
8 };
```

2 - sets the colour of the shape's outline

3 - sets the fill colour for the rectangle

5 - draws an outline for the rectangle

6 - creates a filled rectangle using the colour specified by *fillStyle*

7 - clears the specified area by making the background transparent

4. Draw circle

```
const drawCircle = () => {
    context.beginPath();
    context.lineWidth = 5;
    context.arc(400, 200, 80, 0, (Math.PI/180)*360, false);
    context.stroke();
    context.closePath();
};
```

To draw on canvas we must begin a path using *context.beginPath()*. Next, the line width is defined using *lineWidth*.
To draw a circle we use *arc()* method, which takes the following parameters: *arc(x, y, radius, start, end)*,

- x,y: position to place the center of the circle on the canvas

- start, end: angles at which the circle must start and end.

5. Draw lines

```
const drawLine = () => {
    context.strokeStyle = "#000000";
    context.lineWidth = 10;

    context.lineCap = "round";

    context.beginPath();

    context.moveTo(10, 300);
    context.lineTo(181, 300);

    context.stroke();
    context.closePath();
};
```

To draw a line we need to first define the colour of the line by using *strokeStyle* which accepts a hexadecimal value, and a line width using *lineWidth*. These values must be set before calling *stroke()*.

lineCap defines the cap style of the line, with options like round, butt, or square.
moveTo() specifies the staring point of the line
lineTo() specifies the end point of the line
stroke() is used to draw the line from start to end point

6. Draw arc

```
const drawCurve = () => {
    context.lineWidth = 5;

    context.beginPath();

    context.moveTo(100, 100);
    context.quadraticCurveTo(300, 320, 300, 100);

    context.stroke();
    context.closePath();
};
```

The function above draws a quadratic Bezier curve. Similar to drawing a line, can define a line width and colour, begin a path to draw, define a starting point using *moveTo()* and then use the method *quadraticCurveTo()* to draw the curve. The method takes four parameters: 300, 200 as the control points that define the curve and 300, 100 as the end points the drawing should stop.

2.2.2 jQuery

jQuery is a JavaScript library designed for building dynamic web applications, offering a wide range of utilities, visual effects and widgets [8].

An important attribute of jQuery is its cross-browser compatibility. It provides a consistent execution of instructions across all browsers [8], as opposed to vanilla JavaScript which may be interpreted differently across browsers.

Another significant feature is AJAX, enabling seamless behind-the-scenes communications with the server without requiring page refresh [8].
Additionally, jQuery simplifies the access and manipulation of HTML elements, providing extensive functionality for adding animations and visual effects.

In the upcoming section, I will provide insights into the practical usage of jQuery.

1. Selecting elements

   The basic syntax of selecting an element:
   `$`(selector).action()
   where `$` is used to access jQuery, *(selector)* is used to locate the HTML element, selector is an HTML tag such as class name, id etc., and *action()* is the action that needs to be performed on the selector, this will take in a function with the necessary steps that need to be taken on the selector.

   ```
   $('#greeting').text('Hello World');
   ```

   The code above will select an element with id *greeting* and change its text to 'Hello World'.

   There are 3 different ways to select an element, same as in JavaScript:

   ```
   1 $('p')
     $('body section h1')
   2 $('#id')
   3 $('.class')
   ```

   1 - select all <p> elements, alternatively select the element hierarchical, the second option will select the <h1> which is inside a section within the body

   2 - select an element by its id value

   3 - select an element by its class name

   jQuery provides an easier way to select the first and last <p> elements from the document, as shown below.

   ```
   $(p:first)
   $(p:last)
   ```

Above we have explored only a subset of the different ways elements can be selected in jQuery, there is a complete list of different selectors provided in Table 2.1 in [8].

2. jQuery effects

```
$('#greeting').hide();
$('#greeting').slideDown("slow");
```

The code above hides the greeting when the description is clicked and then reveals it again with a slide-down effect. This can be seen in practice by opening 'jQuery.html' in Proof of Concept folder.

Additionally, can specify the duration of the effect by passing a value as a parameter in milliseconds. Other available effects include: show, toggle, fadeOut, fadeIn, slideUp, slideDown.

Can also add animations, the following code snapshot will decrease the size of the <section> element to 50% over 2 seconds.

```
$('section').animate({
    width:  '50%',
    borderWidth: '10px'
}, 2000);
```

3. jQuery HTML and CSS

As mentioned above, jQuery provides the ability to manipulate HTML elements.
For example, can add new HTML elements:

```
$('<p id="after">insertAfter greeting from jQuery</p>').insertAfter('#greeting')
$('<p>insertBefore greeting from jQuery</p>').insertBefore('#greeting');
```

Two new <p> elements have been added using *insertAfter()* and *insertBefore()*, functions which add the new element after or before the greeting.

To update the color of an element directly from jQuery, can use *.css()* function as shown below, or adding a class to the element.
Given the already defined css style for the following classes:

```
.newStyle {
    color: red;
    font-size: 20px;
}

.style2 {
    color: green;
}
```

We can apply the styling to an element as follow:

```
$('p').toggleClass('newStyle');
$('#after').toggleClass('style2');
$('p:last').css({ "font-size": "40px", "color": "orange" });
```

toggleClass() adds the specified class to the selected element or remove it if already exists. Adding the new class with predefined styling will automatically apply the style to it.

Alternatively *addClass()* function can be used to add a class to the element instead without toggling it.

Lastly, the *.css()* method allows to apply custom styling to the element directly. In the above example the last `<p>` element is selected and applied the styling defined inside the brackets.

4. Event handlers

   Event listeners are an important aspect of web development as they enable responses to user interactions. jQuery provides a set of tools for implementing responses to these events.

   In jQuery we need to connect an event to an event handler, and when the event occurs the handler will execute. First we need to bind an element to an event handler, for this *on()* method is used, takes in three parameters: event, data, and handler function, where data parameter is optional [8].

```
$('#description').on('click ', () => {
    $('#greeting').hide();
    $('#greeting').slideDown("slow");
});
```

   Some of the possible events are: click, blur, mouseleave, select, load, resize, scroll amongst others.

   Multiple event handlers can be bound to an element, and a shortcut can be used to achieve this. Instead of explicitly using the *on()* function, the event itself can serve as a binder. For example:

```
$('#description').mouseenter(() => {
    $('#description').css('color', 'purple');
});

$('#description').mouseleave(() => {
    $('#description').css('color', 'orange');
});
```

   Lastly, the *off()* function allow to remove an event handler for a specified element:

```
$('#description').off('click');
```

   HTML canvas is a powerful tools that enables to dynamically draw graphical elements onto the document directly from JavaScript. In conjunction jQuery allows to create a better user experience with fewer steps.

2.3 Developing an offline HTML5 application

This section explores the technologies needed to build an offline application using HTML5. The report includes code snapshots from proof-of-concept programs and knowledge acquired from [9].

2.3.1 Service Workers

This section explores what a service worker is and its practical usage.

A service worker is a JavaScript script, can be registered to control one or more pages of an web application [9]. The Service Worker can listen to different events coming from the registered pages, intercept and modify these events, modify and customise the responses to these events [9].

Lifetime of a service worker: 1. Installing 2. Activating 3. Activated

First lets delve into how to register a Service Worker:

```
const registerServiceWorker = async () => {
    if ("serviceWorker" in navigator) {
        navigator.serviceWorker.register("./serviceWorker.js")
        .then((registration) => {
            console.log("Service Worker registered with scope:",
                                    registration.scope);
        }).catch((error) => {
            console.log("Service worker registration failed:", error);
        });
    };
};
```

First we define an asynchronous function to register the service worker, as the registration process might take some time and we do not want to block other processes from running. The *if* statement checks if the current browser supports service workers, if it does then:

```
navigator.serviceWorker.register("./serviceWorker.js")
```

is called with one argument - the path to the service worker script, and it register the service worker. The *register()* function returns a promise. If the service worker has been successfully registered, the promise is fulfilled and *then* statement is executed, printing the scope of the service worker. If the service worker has not been registered an error occurred and the *catch* block is executed, printing an error message on the console.

After registering the service worker, let's delve deeper into the service worker script. Above we have mentioned:

```
navigator.serviceWorker.register("./serviceWorker.js")
```

where *serviceWorker.js* is a script containing event listeners the service worker should listen to. A simple example of a script is:

```
    self.addEventListener("fetch", (event) => {
        console.log("Fetch request for: ", event.request.url);
    });
```

This script listens for *fetch* events and prints a message in the console containing the URL of the file that was being asked to fetch. Here is where the Service Worker comes into play, and it can be programmed to do anything. For example take the following event listener:

```
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.open(cacheName).then((cache) => {
      return cache.match(event.request.url).then((cachedResponse) => {
        if (cachedResponse) {
          return cachedResponse;
        }
        return fetch(event.request).then((fetchedResponse) => {
          cache.put(event.request, fetchedResponse.clone());
          console.log('added missing resource');
          return fetchedResponse;
        })
      })
    })
  );
});
```

The code above adds an event listener to make the service worker listen to all fetch requests. It then checks if the request URL for the file contains *page.css*, which is a file containing some CSS for the current page. If it does, instead of returning the file returns something else. In the case above it simply returns CSS to change the background colour of all *div* elements on that page. This serves as a simple example of what the service worker is capable of. We will explore more of its potential in the following section.

Note: as we have seen the service worker can intercept requests, modify or replace response contents. To protect users and prevent man-in-the-middle attacks, only pages served over HTTPS can register a Service Worker [9].

2.3.2 Cache API

This section explores the practical usage of the Cache API in conjunction with a service worker.

Cache is a new type of caching layer that is completely under the developer's control, provides persistent storage for request and response objects in the browser, and can store files or other assets necessary for the web page to work offline [9].

In the examples above the service worker was already activated when it was listening to the fetch requests, but we can also make the service worker listen to events before is active. We will use this to our advantage, and cache the necessary assets in between these events.

The service worker has an *install* event, which happens once in the lifetime of each service worker, right after it is registered and before is activated [9]. Caching the items at this point will make caching of the assets dependant on the success of the service worker activation, this way it ensures the assets are not cached when the service worker failed to register.

An install event listener look like this:

```
self.addEventListner("install", (event) => {
    event.waitUntil(
        caches.open("assets").then((cache) => {
            return cache.add("/asset1.html");
        })
    );
});
```

To break down the contents of the event listener above:

waitUntil() is used to extend the lifetime of the install event of the service worker until the function that is passed as argument is completed.

caches.open("assets") is used to create and return a new cache with the name 'assets', or if a cache with the given name already exists will open it and return.

When the above step is complete a cache object wrapped in a promise is returned, we use *.then* to access the returned response.

```
cache.add(/asset1.html)
```

adds the file *asset1.html* to the cache.

A new fetch event listener is created to handle fetch requests. If fetching a file from the server fails, the service worker catches the error and retrieves the requested file from the browser cache. This feature enables to run the application offline.

Before demonstrating this in practice is important to note that an application can have online-first or offline-first behaviour.
Online-first attempts to run the application online first, fetching the required files from the server. If this fails, because there is no internet connection, the files will be served from the cache.
While offline-first will fetch the required assets directly from the cache, without trying to connect the server.
This is an important aspect because it shapes how the service worker should behave.

1. Online-first

```
self.addEventListener("fetch", (event) => {
  event.respondWith(
    fetch(event.request).catch(() {
      return caches.match("/asset1.html");
    })
  );
});
```

The service worker will only fetch the assets from the cache when an error occurred whilst connecting the server, which is caught by the *catch()* block.

2. Offline-first

```
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request)
    })
  );
});
```

Offline-first behaviour changes how the service worker works, the fetch request is intercepted by the service worker and it searches the cache for the requested asset and returns it, without giving a chance to connect the server. If the requested resource does not exist in the cache it will fetch the request from the server.

2.3.3 IndexedDB

This section explores what IndexedDB is and its practical usage.

IndexedDB is a transactional database where data is stored in the browser on client-side rather than a remote server, this allows offline data access. IndexedDB is transactional, meaning that the transactions within a transaction either all fail or all succeed. IndexedDB is a NoSQL database meaning that data is stored in key-pair values, similar to JSON, and is indexed allowing easy data retrieving by keys [9].

In IndexedDB we can create multiple databases, each database can contain multiple object stores (these are similar to tables in SQL), each object store contains one data types in key-value pairs [9].

An important aspect is that databases are versioned, if you want to create a new object store or modify an existing one you need to open a database connection with a new version number.

Let's have a look at some tasks that can be performed in IndexedDB: Note: the code snapshots are from 'To-do List' proof of concept program, completed using the knowledge from [9].

1. Opening database

   First we need to create a database and establish a connection as shown in the function below:

```
1 const openDatabase = () => {
2  const request = window.indexedDB.open(dbName, dbVersion);
3
4  request.onerror = (event) => {
5  console.error("Database error: ", event.target.error);
6  };
7
8  request.onsuccess = (event) => {
9    console.log("Database opened successfully!");
10  };
11
12  request.onupgradeneeded = (event) => {
13    const db = event.target.result;
14    const objectStore = db.createObjectStore("tasks",
15      { keyPath: "id", autoIncrement: true }
16    );
17
18    objectStore.createIndex("name", "name", { unique: false });
19    console.log("Database upgraded successfully");
20  };
21 };
```

   The code will be analysed line by line:
   2 - a request to open a database. It does not return a database connection but a request to open a database connection and we can listen to events on this request

   4:6- if an error occurred during the request, an *onerror* event will be triggered and an error message will be printed to the console

   8:10 - event will be triggered if the request was fulfilled

12:20 - The event declared here is triggered when the database version is upgraded to a higher version, the browser will detect this change and trigger an upgrade needed event. We can now modify the database when an upgrade happens by listening to the above event.

2. Storing items into database

   Below is the complete function for adding an item to the database. Note: in practice we always listen to *onerror* and *onsuccess* events happening on the *objectStore*, this gives us information on a particular request, as shown in the previous example. For the example below they have been removed.

```
1 const addItem = () => {
2  const task = { task : "Buy milk" };
3
4  const transaction = db.transaction(["tasks"], "readwrite");
5  const objectStore = transaction.objectStore("tasks");
6  const addItem = objectStore.add(task);
7 };
```

   4 - creates a new transaction which operates on the object store ”tasks” with a read-write scope, allowing to read from the object store and write through this connection.

   5 - creates a reference to the object store ”tasks” from within the transaction, allowing to perform different tasks on the object store.

   The event handlers below are for the transaction, this occurs when a transaction is completed successfully or fails. It is good practice to listen to these events because it gives us a high-level overview of what happens in the database.

3. Reading items from database

   Reading items from database requires to open a transaction and a connection to the object store we want to read.
   Below is a complete function for reading data from database.

```
1 const readTasks = () => {
2  const transaction = db.transaction(["tasks"], "readonly");
3  const objectStore = transaction.objectStore("tasks");
4  const getAllData = objectStore.getAll();
5
6  getAllData.onsuccess = (event) => {
7    const tasks = event.target.result;
8    console.log("Tasks retieved from db: ", tasks);
9
10    tasks.forEach((task) => {
11      console.log(task);
12    })
13  };
14
15  getAllData.onerror = (event) => {
16    console.error("Error retrieving tasks: ", event.target.error);
17  };
18 };
```

   2 - creates anew transaction which operates on the object store ”tasks” with read-only scope, only allows to read from the database through this connection

4 - retrieves all the data from the object store and store it in the variable *getAllData*.

We can then listen to events happening on variable.

6:13 - when listening to the event *onsuccess* we can iterate through the collection of records from the object store and print each record individually or perform any task with that information.

4. Deleting items from database

   Below is a complete function for deleting an item from the database. We create a delete request on the object store, to delete the record with the given taskId, which is a unique key. We can then listen to events happening on the request, this inform us if the request has been successful or not.

```
1 const deleteTaskFromDB = (taskId) => {
2  const transaction = db.transaction(["tasks"], "readwrite");
3  const objectStore = transaction.objectStore("tasks");
4  const deleteRequest = objectStore.delete(taskId);
5
6  deleteRequest.onsuccess = (event) => {
7    console.log("Task deleted successfully!")
8  };
9
10  deleteRequest.onerror = (event) => {
11    console.error("Error while deleting task: ", event.target.error);
12  };
13 };
```

   As shown previously we need to open a database and object store connection (2:3)

   4 - we can use *delete(key)* to delete the entry with the given unique id.

5. Updating items in database

   Below is a complete function for updating an item in the database. Again, a connection to the object store needs to be created, then prepare the item we want to update and then create a request to update the item. The method *put* is used, because taskID is unique, this will update the record with the given id.

```
const updateDB = (newValue, itemID) => {
    const transaction = db.transaction(["tasks"], "readwrite");
    const objectStore = transaction.objectStore("tasks");

    const updatedTask = { name: newValue, id: itemID }
    const updateRequest = objectStore.put(updatedTask);

    updateRequest.onsuccess = () => {
        console.log("Task updated!");
    };

    updateRequest.onerror = (event) => {
        console.error("Task could not be updated: ", event.target.error)
    };
};
```

2.3.4 IndexedDB indexing

As the name suggests, IndexedDB is an indexed database making use of indices, allowing faster retrieval of data. An index can be created on an object store that can then be used to retrieve only the needed objects [9].

For the map an index is created for every object store on the x, y keys of every tile.

```
objectStore.createIndex('xyIndex', ['x', 'y'], { unique: true });
```

The *createIndex()* function takes as the first argument the index name, then the path to the key should be used as an index, in the case above the x, y coordinates of the tile. Lastly, an optional array of options, in the case above, it is mentioned that the keys used for the index are unique [9].

Having an index allows to open a cursor to iterate through the objects with matching criteria. In the project the code to retrieve the tile from the database looks as follow:

```
1.    const transaction = db.transaction([z], 'readonly');
2.    const objectStore = transaction.objectStore(z;
3.    const objectIndex = objectStore.index("xyIndex");
4.    const request = objectIndex.openCursor([x, y]);
```

1: start a transaction on the object store specific to the z coordinate
2: open the object store
3: get the index from the opened object store
4: open the cursor, which iterates through the data, on the index rather than the object store. We pass the values used in the index to identify the entry.

As shown in the previous chapter, the *onsuccess* event can be used to retrieve the requested tile.

2.3.5 Web Storage

This section explores the practicality and types of the Web Storage utility.

Web storage is an API which allows to securely store key/value pairs in the browser [10]. Web storage API is a more secure approach than cookies, the data stored in the browser is stored securely locally, can store large amount of data (5MB) without performance issues, and information is never transferred from the browser [11]. The information stored in the web storage is accessible to all pages from the same origin [11].

Web storage provides 2 ways of storing objects in the browser:

1. Session storage

   Maintains a storage area for the current web page, which is available as long as the web browser tab is open, the data persists even if the page reloads. The data stored here only persists while the current tab is open, when the tab is closed all data will be deleted. The data stored in the session storage is never sent to the server [12]. The session storage is accessible through the window object as: **window.sessionStorage**

2. Local storage

   Local storage behaves similarly to the session storage, the only difference is that the data stored for a given session persists even after closing and reopening the browser or tab for a given page. The stored data has no expiration date, and the only way to delete the information from local storage is through JavaScript, or clearing the browser cache [12]. The local storage is accessible through the window storage as well: **window.localStorage**

In the following example, we will use session storage to demonstrate how the API works in practice. Local storage works in the exact same way, the only difference is using *localStorage* or *sessionStorage* methods to store and access the data.

It is important to note that not all web browsers support local and session storage. To ensure no errors occur whilst using them we can check if the browser supports them as follows:

```
if (window.sessionStorage !== 'undefined') {
    console.log("Local storage supported the browser");
} else {
    console.log("Local storage not supported by the browser.");
}
```

If session storage is supported, we can proceed to use its full functionality. Below, we will demonstrate the complete capabilities of session storage:

```
1 if (window.sessionStorage !== 'undefined') {
2  console.log("Local storage supported the browser");
3
4  const now = new Date();
5  const year = now.getFullYear().toString().padStart(2, "0");
6
7  sessionStorage.setItem("surname", "Smith");
8  sessionStorage.setItem('year', year)
9
```

```
10  const greeting = `Hello ${sessionStorage.getItem('surname')}`;
11
12  const element = document.getElementById('greeting');
13  element.innerHTML = greeting;
14
15  sessionStorage.removeItem("surname");
16  sessionStorage.clear();
17 } else {
18  console.log("Local storage not supported by the browser.");
19 };
```

In the code above the following happen:

1 - we check if session storage is supported by the browser
7:8 - add a new key-value pair to the storage where the key is 'name' and value is 'Smith' and 'year' and current year
10 - we then retrieve the value with key 'surname' from the session storage and use it to create a custom greeting which is the displayed on the HTML page
15 - to remove an item from the session storage we use *removeItem* function, which will remove an entry with the given key
16 - alternatively if we want to delete all the entries in the session storage we use *clear()* method

We can also use developer tools provided by the browser to check what items are stored in the web storage as seen below:

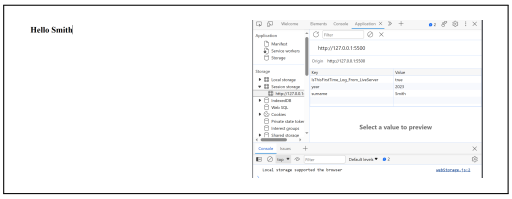

Figure 2.1: Cached assets.

Web storage is particularly helpful when we want to store values which are used by multiple pages often, this will provide a quick and secure response, without the need of making server requests often.

In this section, we explored HTML5 technologies that provide offline capabilities for web pages, covering service workers, cache, local databases for larger structured data, and web storage for smaller data.

2.4 Open Street Map data representation

This section provides an overview of how OpenStreetMap data is represented, showing examples of data returned by the proof of concept.

OpenStreetMap data is saved in a .osm file format, which is specific to Open Street Map, is coded in XML and contains raw geographical data in a structured and ordered format [13]. This file type is designed to be sent and received over the internet easily and fast in a standard format [13].

Files with extension .pbf and .bz2 are also .osm files, however these are .osm compressed files [13].

The following code snippets are raw OSM data which were retrieved though the Overpass API in OSMData proof of concept:
Note: some of the output has been truncated because it was very large, and for illustration purposes the data below suffices. Where the output has been truncated is denoted by '...'.

A query to retrieve OSM data of London museums has returned the data in different elements such as: node, way and relations all which have tags.

Let's see what they represent:

1. Node

   A node is used to represent a specific point in space, is defined by its latitude and longitude, and a unique node id [14].

2. Way

   A way is an ordered list which contains between 2 and 2000 nodes used to define a polyline [14].

   There are 3 different ways:

   1. Open-way is a linear representation of a feature where the first and last node are not identical [14]. Often used to represent roads, rivers train tracks etc.

   2. Closed-way polyline is where the first and last node are identical [14].

   3. Area which is also knows as a polygon, 'is an enclosed filled area or territory' [14], they often have a tag 'area=yes', however there are some exceptions. Used to represent closed areas such a large building, parks etc.

3. Relations A relation is used to define geographic relations between different objects. The objects are represented by a group of members which contain multiple nodes [14].

4. Tags

   Tags are used to define the meaning of a particular element (node, way, relations) [14]. Every tag has 2 fields: key 'k' and a value 'v', where the key must be unique.

   After retrieving all museums from London not all museums have been represented the same.

   For example The Sherlock Holmes Museum and shop was represented by a node:

```
<node id="3916613190" lat="51.5237629" lon="-0.1584743">
  <tag k="addr:city" v="London"/>
  <tag k="addr:housenumber" v="221b"/>
  <tag k="addr:street" v="Baker Street"/>
```

```
  <tag k="fee" v="yes"/>
  <tag k="level" v="0-1"/>
  <tag k="name" v="The Sherlock Holmes Museum and shop"/>
  <tag k="name:en" v="Sherlock Holmes Museum"/>
  <tag k="opening_hours" v="Mo-Su 09:30-18:00"/>
  <tag k="phone" v="+44 20 7224 3688"/>
  <tag k="tourism" v="museum"/>
  <tag k="website" v="https://www.sherlock-holmes.co.uk/"/>
  <tag k="wheelchair" v="no"/>
  <tag k="wikidata" v="Q1990172"/>
</node>
```

And has multiple tags which describe the location, such as name (which is available in multiple languages), website, opening hours and others.

Whilst the Science Museum was represented by an open-way, notice the first and last node are not identical:

```
<way id="27765411">
  <bounds minlat="51.4969551"
  minlon="-0.1783304" maxlat="51.4978392" maxlon="-0.1741311"/>
  <nd ref="807945751" lat="51.4969551" lon="-0.1782508"/>
  <nd ref="304884794" lat="51.4973377" lon="-0.1783304"/>
  ...
  <nd ref="807945751" lat="51.4969551" lon="-0.1782508"/>
  <tag k="addr:housename" v="Science Museum"/>
  <tag k="building" v="public"/>
  <tag k="building:levels" v="3"/>
  ...
  <tag k="name" v="Science Museum"/>
  ...
  <tag k="name:uk" v=" "/>
  <tag k="opening_hours" v="Mo-Su 10:00-18:00"/>
  <tag k="roof:levels" v="2"/>
  ...
  <tag k="website" v="https://www.sciencemuseum.org.uk/"/>
  <tag k="wheelchair" v="limited"/>
  <tag k="wikidata" v="Q674773"/>
  <tag k="wikipedia" v="en:Science Museum (London)"/>
</way>
```

The British Museum was represented by a relation, most probably because is such a large building, and in order to define the outline properly was required to be represented by a relation:

```
<relation id="177044">
  <bounds minlat="51.5180530" minlon="-0.1289022"
      maxlat="51.5205483" maxlon="-0.1248778"/>
  <member type="way" ref="37909715" role="inner">
    <nd lat="51.5192654" lon="-0.1261622"/>
    <nd lat="51.5191322" lon="-0.1264629"/>
    ...
    <nd lat="51.5199606" lon="-0.1269526"/>
```

```
    <nd lat="51.5192654" lon="-0.1261622"/>
  </member>
  ...
  <member type="way" ref="37909796" role="outer">
    <nd lat="51.5196390" lon="-0.1269850"/>
    <nd lat="51.5196283" lon="-0.1268722"/>
    ...
    <nd lat="51.5196237" lon="-0.1270963"/>
    <nd lat="51.5196390" lon="-0.1269850"/>
  </member>
  <tag k="addr:city" v="London"/>
  <tag k="addr:country" v="GB"/>
  <tag k="addr:postcode" v="WC1B 3DG"/>
  <tag k="addr:street" v="Great Russell Street"/>
  ...
  <tag k="historic" v="yes"/>
  ...
  <tag k="listed_status" v="Grade I"/>
  <tag k="museum" v="archaeological"/>
  <tag k="name" v="British Museum"/>
  ...
  <tag k="opening_hours" v="Mo-Th,Sa-Su 10:00-17:30; Fr 10:00-20:30"/>
  ...
  <tag k="wikidata" v="Q6373"/>
  <tag k="wikimedia_commons" v="Category:British Museum"/>
  <tag k="wikipedia" v="en:British Museum"/>
  <tag k="year_of_construction" v="1823 - 1847"/>
</relation>
```

As seen in the examples above even though the locations are the same type they are not represented the same, and factors such as size can affect what elements are used to represent these locations.

2.5 Vector vs. Image tile maps

This section gives an overview of tiled maps, vector tile maps, raster tile maps, the main differences, and similarities between the two.

2.5.1 Tiled maps

Tiled web maps is a tiling technique used to enhance user interactivity. The map is divided into a fixed grid of images, referred to as tiles, which are served to the user as requested [15]. Maps employing this approach utilize multi-resolution map image pyramids, as shown in the figure below.

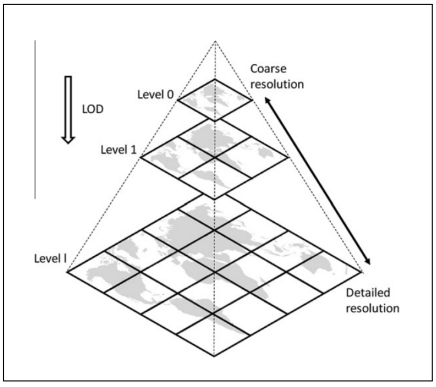

Figure 2.2: Tile pyramid [15].

The top of the pyramid consists of low-resolution tiles, displaying low-levels of details and as the zoom level increases the tiles are higher resolution with higher level of detail. The information displayed on the map is reduced as the zoom level decreases through progressive generalization [15].

As it can be seen from the figure above, at the higher zoom level the map is composed of multiple tiles joined together to form the map. The tiles are not loaded all at once but as requested. The tiles are requested based on the XYZ tiling scheme.

The tiling scheme organises tiles using a 3-dimensional coordinate system x/y/z where x and y are axes value and z is the zoom level [20]. At zoom level 0 the map is divided into 4 tiles, and as the zoom level increases the zoomed in tile will be replaced by other 2^z tiles. The figure below depicts this behaviour.

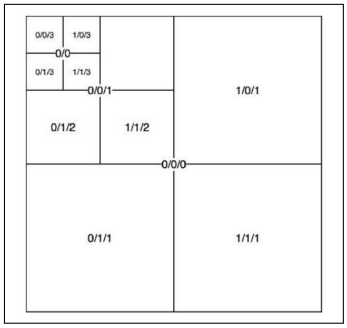

Figure 2.3: XYZ coordinate schema [20].

2.5.2 Vector tile maps

In vector tile maps, the geospatial data is represented by vectors objects such as points, lines and polygons [17], Open Street Map encodes this data as node, ways and relations as seen above. The geospatial data needs to be processed to convert nodes to geometric points and ways to lines or polygons, and then convert the vector objects to vector tiles.

Vector tiles can be generated from the OSM raw data using OpenMapTiles, an open-source project which allows to create vector tiles, which can be used for hosting, self-hosting or offline use [18].
The vector tiles are then saved into .mbtiles file format, this format stores the vector tiles in SQLite database which is perfect for immediate usage and efficient transfer [19].

The vector tiling principle is to decompose the vector data into multiple tiles each corresponding to a certain area as shown below, if an object belongs to more than one tile, is decomposed into several pieces and assigned to the corresponding tile [21].

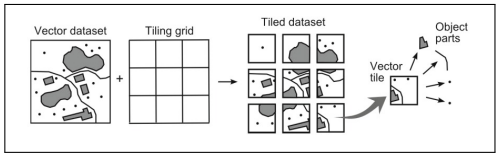

Figure 2.4: Principle of vector tiling [21].

The generated tiles are stored on the server and then rendered on the client side, however is important to note that this data does not contain any information for rendering and tools such as OpenLayers can be used to render the vector tiles on the client side.

2.5.3 Image tile maps

Image tile maps are commonly knows as raster tiles because they come in raster format (as images). Raster tile maps also use the tiling technique described above.

Raster tile maps are composed of images where each image, commonly, is of size 256x256 pixels [17]. Each image needs to be generated individually, process which takes a lot of computational power and memory to store them. For this reason the tiles are rendered on the server-side and served to the end-user as requested [17].

2.5.4 Vector vs. Image tile maps

Vector and raster tiled maps use the same tiling scheme described in section 2.5.1, the main difference between the two is how the tiles are represented and this aspect brings its advantages and disadvantages. Below we will explore such aspects.

Raster tiles are time and resource consuming because it requires to generate a new tile for each zoom level, if data, labelling or styling changes for a certain area then the tiles need to be re-generated for that specific area, it was estimated that for a map of zoom level 20 the tiles would be 20,480 TB of data [17].
Geospatial calculations such as route mapping cannot be performed because it does not contain any geographical data, such data needs to be stored separately to achieve this behavior.

In contrast, vector tiles are more flexible, allow a smoother transition between the zoom levels

comparing to raster tiles, and the size of the vector tiles is nearly a quarter of the size of raster tiles [16].

Because vector tiles are composed of geographical data, making the necessary data updates would update the tiles automatically, without having to re-generate the tiles. The styling of the map can be manipulated on client-side, and having customised design is more affordable comparing with vector tiles [16]. In contrast, raster tiles need to be re-generated if styling or geospatial data updates occur.

Route mapping and other geospatial information is available from the database [16], which makes it easier to access this information without the need to store extra information.

2.6 OpenLayers

This section provides a brief overview of OpenLayers library and its application within the project, focusing on two key features important in reaching the application's objectives.

OpenLayers is an open-source JavaScript library for displaying dynamic map in the browser with the ability of displaying different types of maps such as vector tile maps and raster tile maps from a source [23]. The source can either be a server or a function that returns the tile in a specific format and OpenLayers renders this data in the browser seamlessly.

In the project, OpenLayers is used to render the vector tiles in the browser, operating on the client side. The library displays node, points and ways as lines, polygons and points, while offering zoom in/out functionality, map navigation and map styling functions. Additionally, the library is responsible for requesting the necessary tile from the source as requested. For example, when the map is zoomed in or out OpenLayers dynamically requests the relevant tiles from the source, ensuring the appropriate data is displayed.

In the project, there are two sources for map data provision. Online, data is sourced from a tile server implemented using Node.js and Express.js, which retrieves data from an SQLite database. Conversely, offline, map data is sourced from IndexedDB, requiring adjustments to OpenLayers' data loading approach.

2.6.1 Map Component

The core component of OpenLayers is the map, facilitating rendering of the map data on the page. This task needs a view, a layer and a container.
In the application, the OpenLayers map is implemented as follows:

```
1 var map = new Map({
2   target: 'map',
3   view: new View({
4     center: fromLonLat([-0.489, 51.28]),
5     maxZoom: 14,
6     zoom: 6,
7     extent: transformExtent([-0.489, 51.28, 0.236, 51.686], 'EPSG:4326', 'EPSG:3857');
8   }),
9   layers: [],
10   controls: defaultControls({attribution: false}).extend([attribution])
11 });
```

1: creates a new map object
2: the target "map" targets the container of the map, which is a *div* element in the HTML document with id "map"
3: the map's view represents a 2D view of the map
4: specifies the initial center of the view; when the application is open the map will be centered around the specified coordinates (in the case above the coordinates correspond to London area)
5: the maximum zoom level of the map. The map cannot be zoomed in beyond this point, value used to determine the map resolution
6: the initial zoom level of the map. The map cannot be zoomed out beyond this zoom level, value used to calculate the initial resolution
7: the extent is a constraint of the view. The map cannot be navigated beyond these

coordinates. Used to make sure the user does not navigate parts of the map for which tiles do not exist, thus constraining the user to only navigate London
9: an array containing the layers of the map. Initially the array is empty, however as the map is navigated new layers are created and added to the map as requested.
10: controls added to the map, in the case above map attributions
This is the first step in creating the map, and plays a crucial role as without the map element, the data could not be rendered.

2.6.2 Layers

As mentioned in the API documentation, a layer is a container which groups together the properties of the data from the specified source [25]. There are multiple sub-classes of the layer class, the one used in the project belongs to the sub-class *VectorTile*. The *VectorTileLayer* is specifically design to contain vector data, and is rendered on the client-side.

```
1  const layer = new VectorTileLayer({
2    source: new VectorTileSource({
3      format: new MVT(),
4      url: url,
5    })
6  });
```

1: construct a new vector layer object
2: specify the source which provides the map data; *VectorTileSource* provides the data from the source to the *VectorTileLayer* into a grid, optimised for rendering
3: the map data format
4: URL to the server which serves the vector data

2.6.3 tileUrlFunction

The function *tileUrlFunction* allows to customise the URL of the requested tile and gives access to the coordinates of the tile. This function was implemented to redirect OpenLayers to a different source when offline. The implementation looks as follow:

```
1    tileUrlFunction: (coordinates) => {
2        const [z, x, y] = coordinates;
3        if(!navigator.onLine) {
4          return retrieveTile(z, x, y);
5        } else {
6          return url + z + '/' + x + '/' + y + '.pbf';
7        }
8    };
```

The above function redirects OpenLayers to a different source depending on the internet connectivity. When online the source is an URL, whilst offline the source is a function retrieving the tile from IndexedDB.

2.6.4 tileLoadFunction

To achieve offline functionality with OpenLayers, it is necessary to customise how the tiles are requested from the source and loaded by OpenLayers. The function *tileLoadFunction* allows to customise how a tile is loaded and displayed [24].

By default, OpenLayers uses the source passed as a URL to request and retrieve map tile data from the server. However, in offline scenarios, the source becomes a function retrieving map data from IndexedDB. This function does not provide the same response format as the URL source, therefore is imperative to modify the tile loading process to accommodate this difference, ensuring adaptability based on available connection.
In the project the *tileLoadFunction* is implemented as follows:

```
1  tileLoadFunction: (tile, url) => {
2    tile.setLoader(async (extent, projection) => {
3      try {
4        const coords = tile.getTileCoord();
5        const [z, x, y] = coords;
6        var data = await retrieveTile(z, x, y);
7        if (!data) {
8          const response = await fetch(url);
9          data = await response.arrayBuffer();
10         await addTile(data, z, x, y);
11       }
12       const format = tile.getFormat();
13       const features = format.readFeatures(data, {
14         extent: extent,
15         featureProjection: projection,
16       });
17       tile.setFeatures(features);
18     } catch (error) {
19       tile.setState(TileState.ERROR);
20     }
21   });
22 };
```

2: set feature loader for reading the features of the tile
4: get the coordinates of the requested tile
6: retrieve tile from IndexedDB
7-9: if IndexedDB returned null, request the tile from the server
10: add the missing tile to IndexedDB, completing the offline functionality
12-13: read the extent and projection of the requested tile
13-17: sets the extent and projection for the layer
18-19: Sets the state of the tile if the tile cannot be loaded, this allows the tile to be removed from the queue. If this property is not set the tile will block other tiles from loading [24].
The function is implemented following the API documentation found at: [24].

2.6.5 Drawing on the map

OpenLayers provides the ability to draw geographic features on the map. This is achieved by using a *Feature*, which is a vector object. In the project a circle is drawn on the map, indicating the position of the user on the map as follows:

```
1 const circle = new Feature({
2     geometry: new Circle(fromLonLat([longitude, latitude]), calculateSize()),
3 });
```

2: *geometry* is used to specify the shape of the vector object to be drawn. In the case above a circle, takes as arguments the coordinates where the center of the circle should be drawn on the map, and a radius which is dynamically calculated based on map resolution. The calculations are performed by *calculateSize()* function. The rationale behind dynamic radius calculation is to ensure that as the map is zoomed in, the circle size decreases for visibility, and conversely, as the map is zoomed out, the circle size increases to maintain visibility.

2.6.6 Map Styling

When rendered, by default the map is drawn by blue lines, has no styling and any feature drawn on the map will inherit the styling of the map thus having colour blue. The *style* class provides the ability to customise the styling of the map.
Firstly, the styling is specified by using *Style* vector feature container, and the styling is specified as JSON, having key-value pairs. Below the colour of the location point is updated by using the *fill* option.

```
circle.setStyle(
  new Style({
    fill: new Fill({
      color: 'rgba(46, 204, 113, 1)',
    })
  })
);
```

When dealing with the whole map, the styling of every feature takes more than 5 lines of code. The map styling can be declared in an external file with the extension *.json* and applied to the map using *styleFunction* function. In the application this is used as follows:

```
1. fetch('style.json').then((response) => {
2.   response.json().then(function(style) {
3.     stylefunction(londonLayer, style, 'openmaptiles');
4.   });
5. });
```

1: the map styling is retrieved from an external file by making a fetch request
3: the map styling is applied using *styleFunction()*, taking as arguments the layer to be styled, the style sheet, and the style source.
In the project the map styling is provided by OpenMapTiles, available on GitHub for public use.

2.7 Asynchronous JavaScript

JavaScript has a single thread of execution, thus when a function starts executing it runs until completion before executing the next one, this feature protects the data from being corrupted during the process [27]. However, when making certain API calls such as to IndexedDB or a server, where most actions are asynchronous [9], it may take a while until the task completes, process which will block other code from running. For this reason operations which are time-consuming are asynchronous [27].

When working with asynchronous functions, we specify what the function should do and provide a callback function which is invoked when the data is available or an error occurred [27]. "Callbacks are the cornerstone of asynchronous JavaScript programming" and are used to manage the execution order of asynchronous functions as described by D. Parker in [26].

Is important to note that the code implementing the asynchronous task is not part of the JavaScript thread and runs on its own thread [26]. When the asynchronous task is finished, it will return a callback which is placed in the JavaScript queue; when a function finished executing, JavaScript will get the next item from the queue and if this happens to be a callback it will be invoked [26].

2.7.1 Promises

When working with asynchronous functions it can be challenging to manage the order of execution and error handling. Each callback needs to be managed separately with *.then()* and *.catch()*, and if it happens that an operation makes multiple asynchronous operations then it will result in multiple levels of processing with deeply nested callbacks which are hard to read and understand, this is also known as "callback hell" [27].
A solution to this challenge is to use promises, allowing to organise callbacks into easy to read, maintain and understand steps [26]. A promise is an object serving as a placeholder for a value that is promised to be produced eventually if is not available right away, or never if an error occurred as described in [26] [27].
A promise is created by invoking the *Promise* constructor:

```
new Promise()
```

The constructor takes one argument, a function called "executor function" with two arguments which are handlers for the success and failure outcomes [27]:

```
1    new Promise(function(resolve, reject) {
2      let result =  task which produces the desired result
3      if (success) {
4        resolve(result);
5      } else {
6        reject(errorMessage);
7      }
8    });
```

When the result is available is passed to the *resolve* handler or if an error occurred the error is passed to the *reject* handler.

2.7.2 Asynchronous JavaScript in practice

This section provides an overview of how asynchronous functions and Promises work together by using two, simplified, functions implemented in the project.
Function responsible for adding a tile to IndexedDB.

```
1 const addTile = (tile, z, x, y) => {
2   return new Promise((resolve, reject) => {
3     const transaction = db.transaction([z], "readwrite");
4     const objectStore = transaction.objectStore(z);
5     const data = { 'x': x, 'y': y, 'tile': tile};
6     const addItem = objectStore.add(data);
7     addItem.onsuccess = (event) => {
8      resolve(`layer with coordinates ${z}-${x}-${y} added`);
9     };
10     addItem.onerror = (event) => {
11       reject(`Failed to add tile ${z, x, y}: `, event.target.error);
12     };
13   })
14 };
```

The above function returns a promise because the process of adding an item to IndexedDB is asynchronous. When the *onsuccess* event is triggered the promise is fulfilled and is resolved by calling the *resolve* handler. If an error occurred whilst adding the item, the promise is rejected by calling the *reject* handler which is passed as argument the reason for rejection.

Function responsible for fetching a tile from the server with the provided coordinates, and adding the data to IndexedDB by calling the function above.

```
1 const getAndAddTile = async (z, x, y) => {
2   const url = `http://localhost:3000/data/tiles/${z}/${x}/${y}.pbf`;
3   try {
4     const tileResponse = await fetch(url);
5     if (!tileResponse.ok) {
6       return;
7     }
8     const tile = await tileResponse.arrayBuffer();
9     const addTileResponse = await addTile(tile, z, x, y);
10     updateProgressBar();
11   } catch (error) {
12     console.log('ERROR', error);
13   }
14 };
```

The above function is used to fetch, from the server, a vector tile and add it to IndexedDB. The function has the *async* keyword indicating that is asynchronous and finishing might take a while. The reason it will take some time to complete is because the function calls 3 other asynchronous functions: *fetch*, *addTile* and *arrayBuffer()*.
The *await* operator is used to wait for the promise to return before executing the rest of the code, this feature allows to manage asynchronous code in a synchronous manner, and is specifically important when the next operations depend on the success of the asynchronous function.

2.8 Geolocation API

To improve user experience and make use of the full potential of HTML5 I have decided to use Geolocation API to indicate the user's location in space.
The Geolocation API specification provides scripted access to the geographical location of the browser of the hosting device via longitude and latitude values [28] [29]. The Geolocation API is supported on most modern web browsers on mobile and desktop devices; full support is shown in the image below.

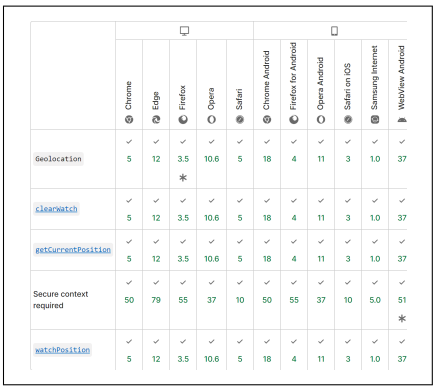

Figure 2.5: Geolocation support [30].

In the project, the Geolocation API is used to get the user's location and show it on the map as an extra feature. Geolocation is implemented as follows:

```
1 if("geolocation" in navigator) {
2   navigator.geolocation.watchPosition(
        updatePosition, catchError, { enableHighAccuracy: true });
3 } else {
4   console.log("Location is not supported by your browser.");
5 };
```

Because not all web browsers support this feature, it first needs to be checked for support, and this is achieve in line 1.
Geolocation API provides two methods for retrieving user's location:
1: *getCurrentPosition(successCallback, [errorCallback, [options]]*
2: *watchPosition(successCallback, [errorCallback, [options]]*
Both methods attempt to gather the user's geolocation information, and when this succeeds it calls the *successCallback* function, when it fails calls the *errorCallback* function [29]. The main difference between the two methods is that *watchPosition()* retrieves the user's location at regular intervals, this way if the location updates the application receives the update in real time without having to query that manually, while *getCurrentPosition* will only get the user's geolocation information once, when is executed.
In the project I have decided to use *watchPosition()* method because it provides a more flexible way of displaying the user's locations is real time.
updatePosition(position) is a callback function, gets passed a *Position* object as parameter from the API. This object contains all the geolocation information returned from the Geolocation API such as *coords* a *Coordinates* object containing geographic coordinates and *timestamp* which is a DOMTimeStamp containing information about the time when the *Position* object was obtained which contains the longitude and latitude coordinates.
updatePosition() is also responsible for creating a vector layer containing as feature a green dot representing the user's location on the map at the given location. The *Coordinates* object

contains information such as latitude, longitude, altitude, accuracy, heading and speed. In the project only latitude and longitude is used for drawing the location dot at the correct location on the map.

catchError(error) callback function gets passed as argument the error that occurred. *enableHighAccuracy* gives the API a flag to attempt to get as close to the exact location of device as possible [29]. By default this is disabled because it may increase the power consumption and slower response times.

As with any service which uses sensitive information comes with its privacy and security concerns. For this reason the Geolocation API specification specifies that the user's location will not be sent to the application without their explicit consent [29]. The permission is integrated in the browser and when the user first visits the application a pop-up will show asking for the permission as shown in the figure below.

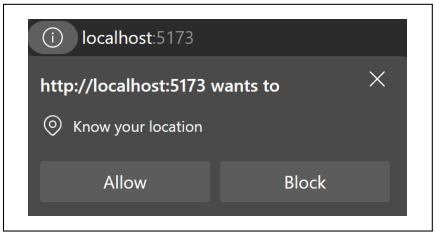

Figure 2.6: Location consent

The consent can be retracted at any point in time from the browsers tab.

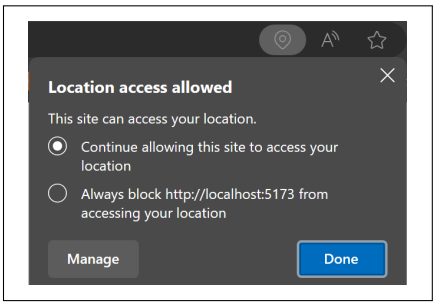

Figure 2.7: Retract location consent

Chapter 3: **Software Engineering**

3.1 Methodology

During project development I have employed the Agile Scrum methodology, and is adapted to a one-person project.

Agile is a set of methods, methodologies and a mindset that help think more effectively, efficiently and make better decisions, as described in [33]. Agile methodologies put emphasis on high-quality working software, customer collaboration, responding to change and team collaboration in alignment with the Agile Manifesto [33]. Scrum is an agile methodology, which provides a framework with defined roles (product owner, scrum master, team members), artifacts (product backlog), and events (sprints, sprint planning, sprint retrospective and daily scrum meetings) [33].

Prior to initiating project development, both in the first and second phase of project development, I have established a project backlog in Trello. The backlog contains a comprehensive list of features imperative for achieving project success, each feature is assigned a priority level such as high, medium, and low.

The project is organized into weekly sprints, each spanning with a total workload of 16-20 hours. Within each sprint, two primary development days are allocated, each consisting of approximately 8 hours of work, with additional smaller tasks and refinements being addressed over the week. The work encompasses research and feature implementation.

On the first day of the sprint, sprint planning is conducted, involving selecting tasks to complete from the backlog and planning for their completion. Task selection prioritized high-priority tasks that contribute significant value to the end product and serve as dependencies for other tasks. Task planning involved breaking down tasks into smaller components, facilitating the estimation of time required for completion and identifying additional research needs. The breakdown often involves investigating new concepts and technologies, ensuring the successful completion of the main task.

The daily scrum meetings are adapted for a one-person project. At the start of each workday, the usual team stand-up served as an opportunity to review the progress made on the previous working day, identify challenges and pinpoint areas of improvement. The remaining tasks are reassessed and established a new time frame based on the progress achieved the previous working day. Each of the completed tasks are updated in the backlog.

At the end of each sprint, a sprint review is conducted. This provides an opportunity to assess both successful and unsuccessful aspects of the sprint, facilitating improved planning for subsequent sprints. Next, I revise the project progress, update the backlog and re-evaluate my goals and objectives for the end-product.

Employing an agile scrum methodology aides in maintaining focus, tracking progress achieved, think more effectively, work more efficient and make better decisions for project success. This methodology enables me to review mistakes and identify areas of enhancement in future sprints, thereby reducing errors and effectively monitoring the progress of project development.

3.2 Documentation

Throughout the project development, emphasis is placed on code clarity through meticulous documentation of every method using JSDoc standards. Additionally, comments are inserted where necessary to enhance clarity. The code documentation ensures the code remains easy to understand, facilitating debugging, maintenance and understanding the methods' aims.

As outlined in the JSDoc API documentation [31], JSDoc is an API documentation generator designed to generate JavaScript documentation. The API scans the source code and generates an HTML documentation website containing the information present in the documentation.

The JSDoc comments are placed before the code being documented. Each documentation must start with a comment in the form of /** to be recognised by the JSDoc parser. Subsequently, JSDoc analyses the source code and incorporates parameters and return values, if any. A basic documentation generated with JSDoc for a function with the signature

```
const retrieveTile = (z, x, y) => {...}
```

looks as follow:

```
/**
 *
 * @param {*} z
 * @param {*} x
 * @param {*} y
 * @returns
 */
```

The basic structure outlined above represents the auto-generated documentation by JSDoc. To complete the documentation, the first line requires a function description - a concise sentence describing the function's purpose. Instead of *, the documentation specifies the type of parameter the function expects to take, followed by a description emphasising the intended representation of the parameter. The complete documentation takes the following form:

```
/**
 * Retrieve the vector tile with the given coordinates from IndexedDB.
 *
 * @param {int} z zoom level
 * @param {int} x tile x-coordinate
 * @param {int} y tile y-coordinate
 * @returns returns a Promise with the vector tile or rejection reason
 */
```

JSDoc tool is used to generate an HTML website from the source files by executing the command *jsdoc filename.js* in the command line.

Following Software Engineering best practices, the addition of code documentation enhances code reusability and readability. This practice proved valuable during development, offering insight into function actions solely through documentation, thus aiding testing and debugging efforts. Moreover, it fosters collaboration by providing clarity to potential reviewers.

3.3 Testing

To ensure the functionality of the application, automated unit tests are written using Mocha and Jest testing frameworks. Additionally, manual testing is carried out to ensure the end product works as expected for the end user. Faults or errors discovered during testing are promptly addressed.

Mocha testing framework is used to test the service worker, ensuring its successful registration and assets caching. Before and after each test case, the service worker is unregistered and the cache is cleared to maintain a clean testing environment and uphold test integrity.
Testing with mocha requires setting up an HTML page that is opened in the browser and displays the failed and successful test cases. The tests first require to register the service worker, after installation the tests can be performed. Tests are executed to verify if the pre-cache files from *files.json* are cached.

Essential front-end components, including IndexedDB operations, are rigorously tested to ensure proper set-up with Jest. Additionally, the tile server is thoroughly tested with Jest.
Due to Jest being a Node.js testing framework, front-end technologies such as window, DOM, jQuery, and IndexedDB need to be mocked to have the ability to test the components which use them. Each testing suite include *beforeEach* and *afterEach* methods responsible for cleaning the tests before and after each test, ensuring each test begins with a fresh environment. Front-end components are tested to ensure that event listeners are attached as expected, and the respective functions are triggered when the event occurs.

Manual testing is performed at every stage of development by executing all existing use cases manually, mimicking the actions of end users. This approach is employed to identify bugs, issues and defects in the application early in the development process.

The system undergoes three types of manual testing:
1. Unit testing to validate each component individually and ensure it works as expected.
2. Integration testing to check the behaviour of the application when the different parts of the app are integrated logically and tested as a group. This is performed whenever a new feature is updated or integrated, ensuring the system components work together seamlessly.
3. System testing is performed to evaluate the performance and usability of the system by executing all possible use cases, allowing assessing the end-to-end system specifications.

Compatibility testing is conducted to evaluate and compare the functionality of the application across various browsers, devices and operating systems, aiming to identify discrepancies in behaviour and design. The application is tested on four distinct operating systems: Apple, Android, Windows and Linux ensuring full compatibility across all platforms.
Additionally, the application is tested on three major browsers: Chrome, Safari and Microsoft Edge, ensuring consistent and expected behaviour and design across browsers.
The results of compatibility testing contribute to the enhancement of the user interface design, responsiveness, overall user experience providing a flexible and mobile-friendly design.

3.4 Other Software Engineering Practices

A fundamental aspect of project development entails the use of Git version control, which effectively helps track changes and safeguards project development progress at all stages. This helps prevent loss of progress in unforeseen circumstances, such as hardware failure. To ensure the code remains current and secure, it is regularly uploaded to GitLab on the university's CIM server every development day, with multiple commits daily.

Meaningful commit messages are employed to ensure they accurately describe the changes made, facilitating efficient tracking of issues and development stages.

Branches play a significant role in organising project features efficiently. A new branch is created from the main branch for every new feature, guaranteeing safe development without disrupting the stability of the working code. Upon successful implementation and validation of a feature, the feature branch is merged with the main branch to ensure the code is most up-to-date. Testing ensures the integrity of the main branch is not compromised, after which the functional code is fetched, merged and pushed to the main branch. This workflow helped maintain all working, and most up-to-date code into the main branch.

Releases are made to mark stable, improved versions of the project at some point in time. Every release is assigned a number version to help organise releases. Candidate release branches are used for testing and bug fixing purposes. Throughout the project's development, five major releases are issued, with four categorised as major releases and one minor release.

The project's final report is also stored in version control on Overleaf online editor, preventing the loss of completed work.

To enhance code readability, the Visual Studio Code extension Prettier is used. Prettier is a code formatter with support for many languages, standardises the code style by removing original formatting and applying configurable rules to files [32]. This practice ensures consistent formatting throughout the project, encompassing aspects such as tab size, semicolon usage and line length.

During development, code smells are eliminated; this practice facilitated the creation of easily maintainable and readable code by addressing issues such as long methods, repeated code, and dead code smells. Consequently, code refactoring led to enhanced organization, structure, and simplification of the code.

Chapter 4: **Development Process**

4.1 Initial development phase

This section outlines the achievements made in the first phase of project development during term 1.

The project is initiated by creating proof-of-concept programs, exploring relevant technologies, and documenting my studies in the reports found in the *Literature survey* section. This factor plays an important role in understanding how an offline application works.
Proof-of-concept programs and reports cover the following topics: service workers, cache API, IndexedDB, web storage, raw OpenStreetMap data, HTML5 canvas, jQuery, development in JavaScript, and building blocks of web development.

In addition to the tools mentioned above, the examination encompassed the following tools and technologies to accomplish the first development phase of the project: generating vector tiles using the open-source project OpenMapTiles, rendering vector tiles using OpenLayers, applying styling to the map using a predefined *style.json* file, understanding the functioning of the Node.js server, and using Overpass API and Overpass Query Language to retrieve raw OSM data over the web.

The application exhibits an online-first behaviour, fetching map data in vector tile format from a locally hosted tileserver-gl managed via a Docker container. Map styling is provided by OpenMapTiles, applied from a JSON file. When offline, the map data is available in GeoJSON format and is rendered from a local file without any applied styling.
The map data is rendered on the client side using OpenLayers and displays the world map at zoom level 0, and London at maximum zoom level 14.

Located on the right side of the map, the panel displays the current time, a compass indicating the orientation of the device, and a download button. While the button offers visual feedback upon clicking, the download functionality is currently inactive.
The user's location is indicated on the map by a green dot, which updates in real time as the user moves in space. This feature is implemented using Geolocation API, is available only if permission is given from the browser, and relies on an Internet connection.

Additionally, the map features a search bar for users to locate specific areas; however, it currently lacks the ability to perform searches. The map includes appropriate attribution tags for using OpenMapTiles and OpenStreetMap data in generating the vector tiles. The user has the ability to zoom in, zoom out and move around the map; functionality provided by OpenLayers.

Upon accessing the application, a service worker is registered to cache essential assets for offline functionality. These assets include the HTML document, scripts for rendering the map, and other components of the panel. The service worker is tested with Mocha to confirm proper installation and caching.
While in offline mode, the application assets are retrieved from the browser cache.

IndexedDB is initialised, and separate object stores are created for each zoom level of the map. This setup is designed to store critical geographical data necessary for offline map display. At this stage, no information is saved in the database; I plan to address this issue in the next development phase.

Following Progressive Web App (PWA) methodology, it is imperative to configure a manifest enabling users to save the application to the device's home screen. Users can add the application to their home screen by selecting the download button from the window tab, as illustrated below:

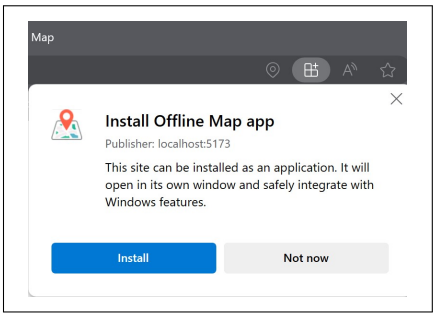

Figure 4.1: Download application.

4.2 Tile server

The choice of serving tiles from tileserver-gl proved impractical for implementing the offline functionality and for deploying the application. Consequently, I proceeded to develop a custom tile server using Node.js and Express.js to address these limitations.

The tile server acts as a source for providing vector tiles and tile coordinates. The vector tiles, generated with OpenMapTiles, are stored in files with the extension *.mbtiles*, which, as discussed in section 2.5.2, is a SQLite database for storing the vector tiles for immediate usage and efficient transfer. Using this knowledge, the server would be able to read the tile data directly from the file using *sqlite3* library.

The server has two endpoints:
1. The first endpoint retrieves a tile with specified coordinates from the SQLite database and returns the tile to the client in x-protobuf format over HTTPS. Is defined by the following signature:

```
app.get('/data/tiles/:z/:x/:y.pbf, (req, res) => {...});
```

A database query is performed to retrieve the tile corresponding to the coordinates specified in the URL, which represent the z, x, and y coordinates. The query format for fetching tiles is as follows:

```
const query = `SELECT tile_data FROM tiles WHERE zoom_level= ?
            AND tile_column = ? AND tile_row = ?`;
```

In the context of the database query, `tile_column` corresponds to the x coordinates, while `tile_row` corresponds to the y coordinates of the tile. If the tile does not exist in the database, the server will respond with a message and a 404 status. If an error occurs while opening the database, a 500 response status will be returned to inform the client of the issue. Otherwise, the server will transmit the tile to the client with a 200 response status.

An important consideration is that the y coordinate must be flipped before querying the tile from the database. This adjustment is necessary because the mbtiles are in TMS format, as identified by [34]. The flipping is achieved through the formula:

```
y = (1 << z) - 1 - y;
```

2. The second endpoint is responsible for retrieving all the existing tile coordinates from the database for a specific zoom level and returns an array containing the retrieved coordinates. Is defined by the following signature:

```
app.get('/offline/tiles', (req, res) => {..});
```

The query for retrieving the coordinates is:

```
const query = `SELECT tile_column, tile_row FROM tiles WHERE zoom_level=${z}`;
```

This feature facilitated the deployment of the application without relying on a third-party service. It also provided an opportunity to enhance my proficiency in Node.js and was particularly valuable during the implementation of the offline functionality.

4.3 Offline functionality

A major step in application development is implementing the offline functionality. This feature has multiple levels and the first and most important is to make the map data available on the client-side whilst offline.

The initial step involves the client requesting existing tile coordinates for a specific zoom level from the server. This is achieved by initiating a *for* loop starting from 6 to 14 and contacting the server via the designated endpoint */offline/tiles/* as outlined in the previous chapter. Subsequently, the server responds with an array containing the existing tile coordinates for the requested zoom level. The client first needs to request the existing coordinates of the specified zoom level because it requires the x and y coordinates when adding data to the database. Without these coordinates, if the tiles are simply sent from the server to the client it would be impossible to determine which tile to retrieve from IndexedDB.
Next, the client iterates over the coordinates array and retrieves each tile coordinate individually. It then contacts the server endpoint */data/tiles/*, passing the z, x, and y coordinates of the tile. In response, the server provides the tile in x-protobuf format, which the client reads as an array buffer. The retrieved tile is subsequently added to IndexedDB in the object store corresponding to its zoom level. Each tile is added to its corresponding object store to speed up lookup when a tile is requested.

There exist over 1000 tiles for zoom level 14, and the browser cannot handle many requests made in a short period of time, reason for which an *ERR INSUFFICIENT RESOURCES* error is thrown, resulting in tiles missing form the database. To address this, a recursive approach is employed when fetching the tile from the server. If a tile request fails due to this error, the function is called again up to three more times. The recursive behavior ensures the request eventually succeeds while preventing infinite loop, thus maintaining code integrity. Below is a simplified representation of the function, illustrating the mechanism:

```
export const fetchAndAddTile = async (z, x, y, retry) => {
  const url = `http://localhost:3000/data/tiles/${z}/${x}/${y}.pbf`;
  try {
    const tileResponse = await fetch(url);
    const tile = await tileResponse.arrayBuffer();
    await addTile(tile, z, x, y);
    updateProgressBar();
  } catch (error) {
    if (error.name === 'TypeError') {
      if (retry < 3) {
        fetchAndAddTile(z, x, y, retry + 1);
      }
    }
  }
};
```

After the tile is successfully added to the database, the progress bar is updated, allowing users to receive real-time updates on the download progress. This feature is implemented using the *progress* HTML element.

Alternative 0: The initial alternative to adding the map data to IndexedDB involves converting the tiles to GeoJSON format and storing in the database. However, this approach is dismissed because it consumes more space in the database compared to vector tiles. This approach is not ideal due to inefficient storage behavior.

Alternative 1: An alternative approach is to create a server endpoint returning, to the client, an array containing all existing tiles in the following format:

```
[{'x' : x, 'y' : y, 'z':z, 'tile' : tile}, ...]
```

However, this solution is not practical because it involves sending over 2000 tiles in the array. This leads to *Invalid string length* or *Heap out of memory* errors, due to the excessive amount of data being sent.

Alternative 2: An improved solution of the first alternative is to request from the server all existing tiles for a specific zoom level, and send the tiles in the format mentioned above. This solution proved effective for the first 13 zoom level. However, for zoom level 14, where are over 1000 tiles, this approach does not work as it leads to the same errors encountered in the first alternative.

Alternative 3: An alternative solution to address the issues encountered in alternatives 1 and 2 would be to use a web socket to send the data into smaller manageable chunks. However, this approach is disregarded as using web sockets in this manner would deviate from its intended use case, leading to poor code practice.

Having successfully added the data to IndexedDB, the subsequent task is to implement the retrieval of tiles from the database. This process requires to create an asynchronous function *retrieveTile(z, x, y)*, responsible for creating a database query to retrieve the tile with the specified coordinates and returns it, as outlined in section 2.3.3.

Another crucial step involves understanding how the data retrieved from the database can be passed to OpenLayers for rendering. This proved to be a challenge because OpenLayers typically accepts a URL with placeholders for coordinates. However, when offline, this URL would no longer function and becomes necessary to inform OpenLayers that the data retrieval needs to be achieved through a function rather than a URL.

To accomplish this objective, the map source is updated in OpenLayers during the layer creation process. This is achieved by creating a custom *tileLoadFunction*, which enables customization of how the tile is loaded and displayed, as outlined in section 2.6.4. The implementation of this function is detailed in the same section, providing a thorough analysis of its behavior. This solution facilitates the transition to an offline-first approach, where the tile is first retrieved from the database, if the tile is not found, it falls back to the network, and the missing tile is subsequently added to the database for completeness.

Alternative 0: An alternative approach is to create a custom *tileUrlFunction*, as described in section 2.6.3, allowing to customise the source URL. This solution is disregarded because it does not offer the optimal solution for an offline-first approach.

The final step in achieving full offline functionality is the registration of a service worker. Section 2.3.1 provides detailed information into the functionality of a service worker and its registration process. In the final product, the service worker is created using *vite-plugin-pwa* plugin, with ability to generate a web app manifest, facilitating the addition of the application as a shortcut to the device; creates a service worker and a script for its registration in the browser [35]. Additionally, the plug-in uses the *workbox-build* library to traverse the build output folder of the application and cache the resources found in the folder [36]. The configurations of the service worker, manifest and pre-cache are declared in the file *vite.config.js*.

Alternative 0: The initial solution implemented uses the service worker defined in the file *serviceWorker.js*. This service worker fetches and caches the pre-cached files listed in *files.json*. While this efficiently handles all assets listed in the pre-cache, it overlooks the need of caching the OpenLayers library, essential for rendering the map. Consequently, it became necessary to manually add all file paths related to OpenLayers functions to the pre-cache. This approach proved inefficient due to the large number of files and dependencies within the library. To address this challenge, an alternative approach to caching the library is to update the service worker to fallback to the network when a resource is missing and add it to the cache. However, this solution requires the service worker to be registered before the page loads to ensure the app is fully functional after is first accessed. In practice, the service worker works asynchronously and the page loads before the service worker is registered. To overcome this a solution is to reload the page after the service worker is registered, however this solution relies on the user to perform this action.

4.4 Adding trip

The next milestone focuses on adding trips to the map, consisting of two main components. Firstly, saving the trip data to IndexedDB and displaying it in the panel. Secondly, placing a pin at the trip's location on the map, offering users the ability to view the trips and highlight them by selecting the corresponding pin on the map.

The first part of this feature involves opening a new database in IndexedDB to store the trips, implementing appropriate methods for adding, deleting, and retrieving trip information as described in section 2.3.3. Once the trip is successfully added to the database, a trip container is created with the trip data and added to the panel, providing users a visual representation of their trips.

The second phase involves adding a trip pin to the map to represent the trip's location. To achieve this, a new vector layer is created specifically for containing pins corresponding to the trip location, these pins are represented as features within the vector layer. When a trip is successfully added to the database, the onsuccess event resolves the promise with the ID of the trip. This ID, is a unique identifier generated when the trip is added to the database by IndexedDB, is used to associate the trip pin with the corresponding trip, accomplished by assigning the trip pin the ID of the trip as a property of the feature.

Upon clicking the pin on the map, the corresponding div in the panel, identified by the same ID as that of the pin, is highlighted. The highlighting is achieved by adding a box shadow around the trip container and bringing it to the top of the panel, and by increasing the size of the selected pin on the map. The highlight functionality of the pin is achieved by attaching an event listener to the layer containing the trips, when a feature of this layer is clicked, the event is triggered, facilitating the highlighting of the corresponding trip in the panel and that of the pin.

To maintain clarity and ensure that only one pin-trip is highlighted at any given time, selecting a pin triggers the resizing of all other pins to their initial size and the box shadow is removed from all trip panels except for the selected one.

To make it easier for the user to distinguish between trip ratings, six distinct pin colors have been introduced, each color corresponding to a different rating possibility. For example, if no rating is provided, a black pin is assigned, a trip rated 1 receives a red pin, 2 receives orange, and so forth, culminating in a green pin for a 5-star rating.

4.5 Displaying attractions

The map displays various attractions, including internet cafes, malls, museums, and theaters, allowing users to select the attractions of interest.

To achieve this functionality, a vector layer is created containing, as features, icons positioned at the location of the attraction.
The following method illustrates how to create a feature in OpenLayers:

```
const addItem = (longitude, latitude, attraction, layer) => {
  const icon = new Feature({
    geometry: new Point(fromLonLat([longitude, latitude]))
  });

  icon.setStyle(new Style({
    image: new Icon({
      src: `../icons/${attraction}.png`,
      scale: 0.05
    }),
  }));
  layer.getSource().addFeature(icon);
};
```

The location data for each attraction is obtained from Overpass API, using Axios to make a HTTP request. Every request requires a query written in Overpass Query Language, generated with the help of Overpass Turbo.
A query has the following form:

```
const cafeQuery = `
area(id:3600065606)->.searchArea;
node["amenity"="internet_cafe"](area.searchArea);
out geom;
`;
```

The data is returned in XML format, adhering to the specifications outlined in section 2.4. Latitude and longitude data for each attraction is extracted from the XML, and used to position the icon at the respective location. Users can manually select which attraction to be displayed on the map within the panel.

4.6 Download trips in PDF

The concluding feature, completing the final product, involves downloading trips for sharing with others at the user's discretion. This is accomplished by enabling the user to download the trip data as a PDF document. This task is achieved using *jsPDF* library.
An overview of how the library methods used to generate the PDF are shown in the code snippet below:

```
1    const doc = new jsPDF('p', 'pt', 'a4', true);
2    doc.text(text, 10, y);
3    const description = doc.splitTextToSize(string, pageWidth - 10);
4    doc.addImage(src, 'JPEG', 100, y, pageWidth - 200, 200, undefined, 'FAST');
5    doc.addPage();
6    doc.save('trips.pdf');
```

1 - *new jsPDF()* initialises a new document in portrait mode, with size unit *pt*, an A4 page size, and *true* parameter indicates support for Unicode characters

2 - *text()* method adds a line of text to the document, taking as arguments the text content, and the x y coordinates for its placement

3 - *splitTextToSize()* method splits the string *string* into an array of strings, where the length of each split string is specified by *pageWidth - 10*. This method is particularly useful for breaking lengthy strings, such as trip description that do not fit on a single line

4 - *addImage()* method inserts an image into the document. Takes as parameters the image source, format, x and y coordinates for placement, width and height of the image, and the image compression. In the project, the compression level *FAST* is used to speed up the process of generating the PDF

5 - *addPage()* method adds a new page to the document. In the project is used to add a new page after each added trip, ensuring each trip is added to a new page. Also used when the information of a trip cannot fit on a single page

6 - *save()* method is used to save the document, taking as argument the name the downloaded file should have

Alternative: An alternative is using pure JavaScript for PDF downloading. This method requires to create a new HTML page containing the desired data and use the method *window.print()*. However, this solution prompts the user with a print dialog, requiring manual document saving, adding extra steps for the user to complete the task. The additional steps negatively impacts the usability and user experience, reason for which I opted to use the library *jsPDF*, allowing automated document download, providing a smoother and more user-friendly experience.

4.7 Responsive design

To achieve a Progressive Web App (PWA) and provide users a native app-like experience over the web, it is imperative to develop a responsive design capable of adapting to various devices and screen sizes.

The first snapshot in Figure 4.2 illustrates the application's appearance before implementing full responsiveness. Despite most element sizes being relative to the viewport, it does not have the desired outcome for smaller devices.
In contrast, the final two illustrations showcase the responsive design, offering users a more legible interface. In these designs, the data panel is enlarged for improved readability, and on mobile devices, the panel extends to full screen.

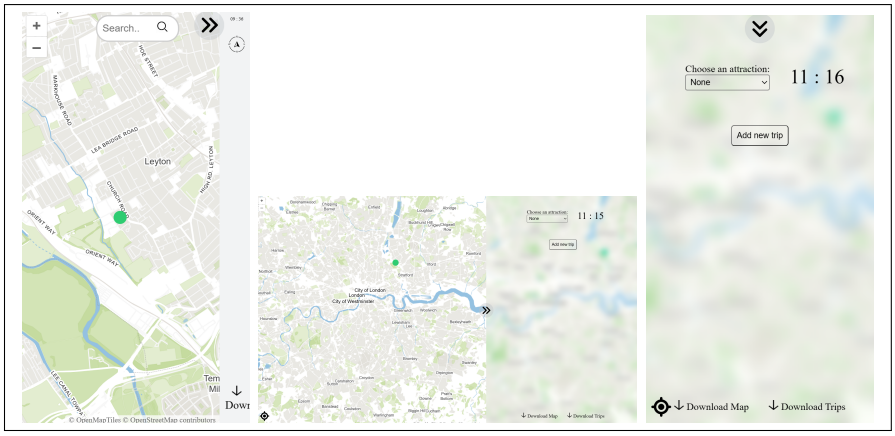

Figure 4.2: Design history.

The responsive design is achieved using CSS, specifically through the *@media* query. The query allows to apply different style declaration based on device qualities such as viewport width and height [37].

When implementing a responsive design, a global styling is defined with CSS, which is applied across the entire system. Through the use of *@media* query, different components of the application are adjusted and modified based on the device's specified qualities. In this project, CSS design is changed for devices with a maximum width of 800 pixels. The width of the panel is adjusted to occupy the entire screen, the trip container width is updated, and the position of the button for closing the panel is shifted.
This is accomplished utilising the *@media* rule as shown below:

```
@media screen and (max-width: 800px) { //css rules }
```

Another component of a responsive design is flexibility, which requires the use of relative length units. Absolute units, such as cm, mm, in, px, pt, and pc, maintain fixed sizes and will appear consistent across various screen dimensions [37]. On the other hand, relative length units are proportionate to the size of the viewport [37], these units include em, rem, vw, vh.

Throughout the project, rem or em units are consistently used to ensure font sizes and other dimensions scaled appropriately, guaranteeing visibility across various screen sizes.

Chapter 5: **End System**

5.1 End System Architecture

Figure 5.1 provides a high overview of the system's interaction with various components. The system engages with five distinct entities.
1. Express JS - the tile server, serving map data. The system initiates requests for map coordinates and map vector tiles data, with the server retrieving this information from the *tiles.mbtiles* file, which is an SQLite database.
2. IndexedDB - the system interacts with IndexedDB, facilitating the storage or retrieval of data such as vector tiles, trips data, and attractions data.
3. OpenLayers - the interaction with OpenLayers is to render the map and layers.
4. Overpass API - the communication with Overpass API is to retrieve OSM data for the specified attractions.
5. Cache Storage - the interaction between the system and cache storage is to store essential assets for running the application and for retrieving these assets when offline, enabling the application to run smoothly offline.

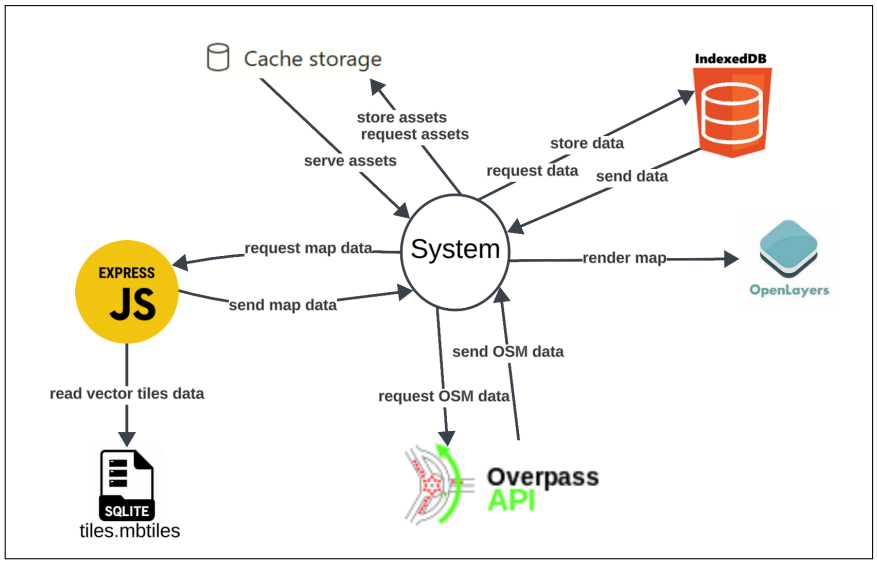

Figure 5.1: System architecture

Figure 5.2 illustrates the use case of rendering the map and the requested tile is not found in IndexedDB.

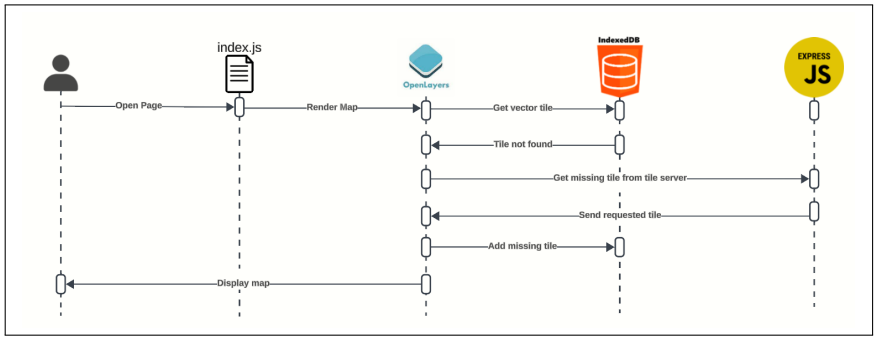

Figure 5.2: Map rendering online

Figure 5.3 illustrates the use case of downloading map data, event triggered by the 'Download' button.

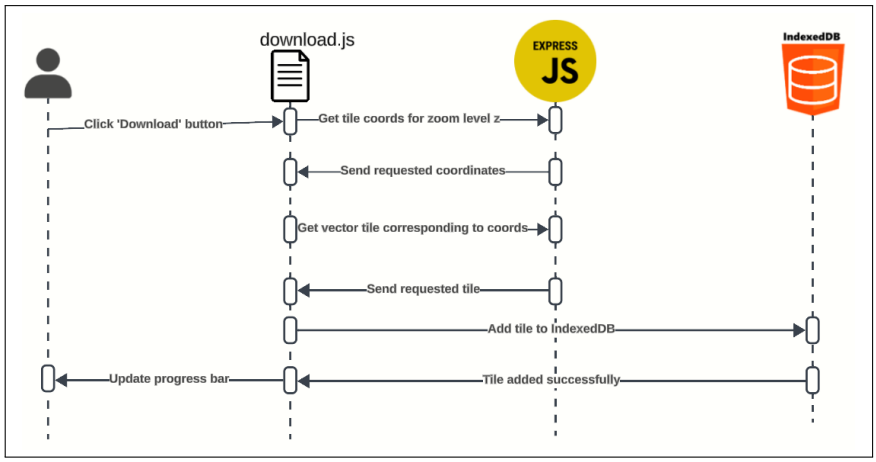

Figure 5.3: Map data download

Figure 5.4 illustrates how the application renders the map when the requested vector tile data is available in IndexedDB. This depicts the standard behavior of the application once all map data has been downloaded.

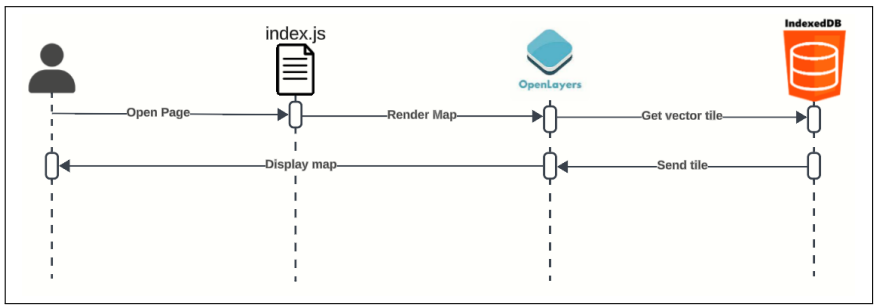

Figure 5.4: Map rendering offline

Figure 5.5 illustrates the use case of adding a new trip to the map, event triggered by clicking the "Add new trip" button.

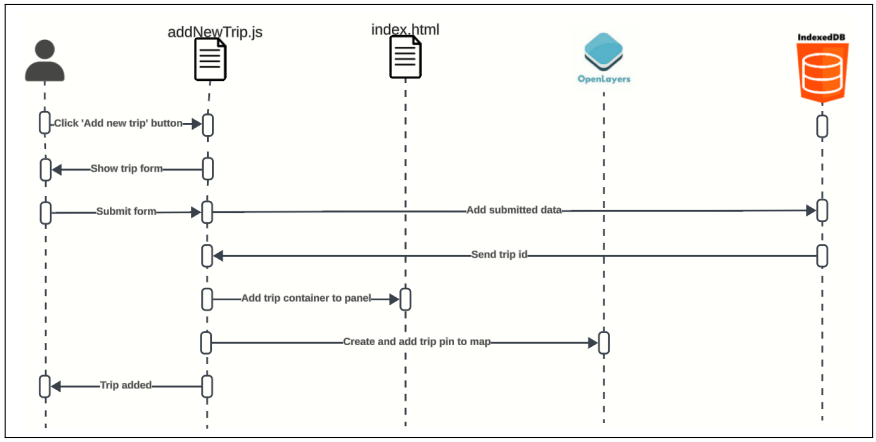

Figure 5.5: Add new trip

Figure 5.6 illustrates the use case of deleting a trip, event initiated by clicking the bin icon from the trip container.

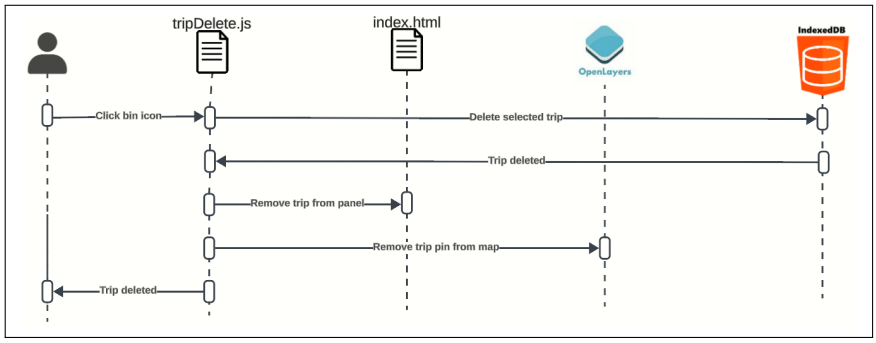

Figure 5.6: Delete trip

Figure 5.7 illustrates the use case of retrieving attractions data from Overpass API and storing the data in IndexedDB. This event is triggered when the application is accessed.

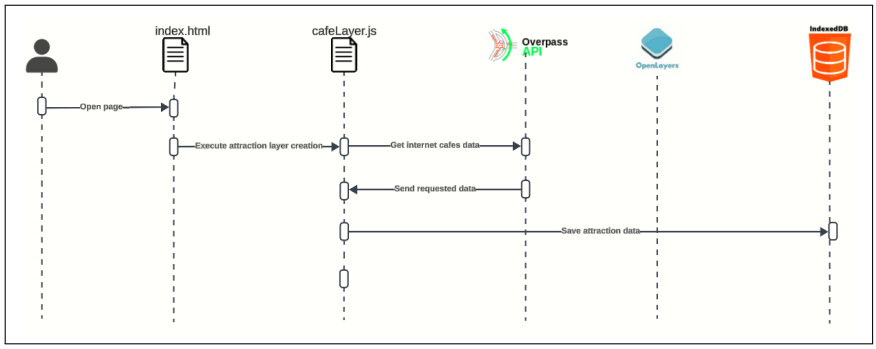

Figure 5.7: Get OSM attractions data

Figure 5.8 illustrates the use case when an attraction is selected to be displayed, event triggered by clicking the "Select attraction" button.

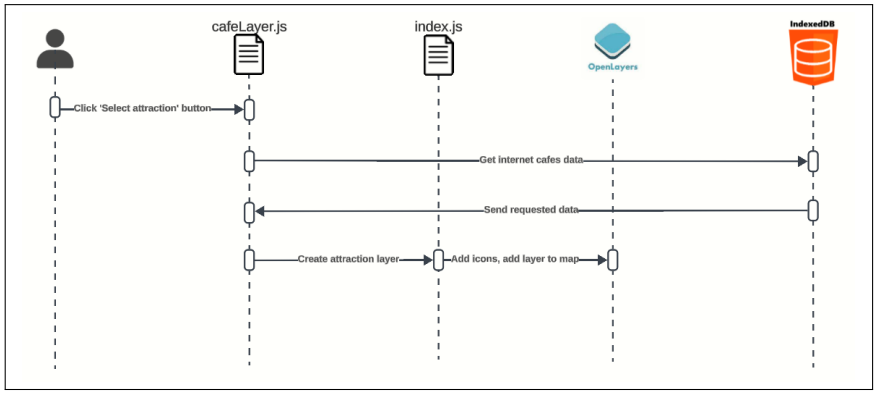

Figure 5.8: Display attraction

Chapter 6: **Assessment**

6.1 Professional issues

This section outlines the consequences of not properly addressing privacy issues such as web privacy and the legal implications concerning data usage and privacy. It analyses the Facebook-Cambridge Analytica event from the public domain, and present concerns to my project regarding privacy and data usage.

Privacy is a concept with different definitions as identified in [39]. Privacy is defined in the English dictionary [38] as: "the state of being alone, or the right to keep one's personal matters and relationships secret". Despite its varying interpretations, privacy is an important concept for preserving one's autonomy, sensitive information, and rights.

Web privacy refers to the protection of personal information on the internet. As outlined in [39], Weston's description of informational privacy resonates well with the digital era: " the claim of individuals, groups, or institutions to determine for themselves when, how, and to what extent information about them is communicated to others". However, various types of information can fall under the umbrella of web privacy, including personal information, private information, personally identifiable information, anonymised and aggregate information as outlined in [39].

Web privacy concerns emerged as soon as the internet became available to the public, as it became evident that user data is being collected, stored, and utilised for other purposes than the explicitly specified when data was collected [40]. This raised concerns regarding unauthorized access and malicious use of this data such as blackmail and impersonation.

In response, the UK introduced the first Data Protection Act in 1984 and is still in use today, to safeguard individuals against the misuse of personal data, to be used other than the intended purposes [40]. Today, the Data Protection Act continues to regulate how an individual's personal data is used by organisations, businesses, and government, ensuring that is used fairly, lawfully, and transparently, used for only the explicitly specified purposes [41]. These regulations have been updated over the years to reflect the ever-evolving landscape of the internet. While regulations exist in all countries, they may vary in their specified rules and requirements. Additionally, professional bodies such as the British Computer Society and Association for Computing Machinery provide standards and a code of conduct to ensure ethical and moral behaviour of professionals at an individual level [46].

As previously mentioned, regulations are continuously updated to keep pace with the evolving landscape of the internet. Unfortunately, updates to privacy regulations tend to occur when major events or mistakes that compromise user privacy take place. This pattern shows the historical tendency to treat privacy, as security has been, as an afterthought.

One such event is the Facebook-Cambridge Analytica scandal. This scandal revealed that users' data was utilised without their explicit consent, and for purposes other than the originally intended. The event started with the application "thisisyourdigitallife", which circulated on Facebook as a personality test. Users would answer questions about themselves, granting the application access to their personal information such as likes, contact lists, and others and provided right to use their personal information for academic purposes [42]. It was revealed that the app not only gathered data from its participants but also from their friends, including data such as date of birth, name, location, and gender [45], collecting data from a total of 87 million Facebook users [42]. Cambridge Analytica, a political consultancy firm

acquired this data to "model the personality" of U.S. voters, aiming to predict and influence individuals' political views during the 2016 elections [42].

The Facebook-Cambridge Analytica scandal without question breached users' privacy rights. The collected data was not used for academic purposes but rather to influence people's political choices, introducing ethical considerations beyond privacy concerns. Cambridge Analytica's actions not only invaded users' privacy but violated international law, specifically Article 19 of the Universal Declaration of Human Rights which states that everyone has the right to hold opinion without any interference and get through any media, as identified in [44].

The app's practice of harvesting private information from Facebook without the explicit consent of the individual but rather through the consent of their friends violates any standard privacy policy, ethical principle, or moral code. As a consequence, legal actions were taken against Facebook which faced a fine of $5billion for sharing private data with third parties [42], while Cambridge Analytica has filed for bankruptcy [45]. Additionally, a new order has been issued to add new privacy standards [42]

This incident had a significant impact on establishing new privacy standards [45]. As expected, the event had an impact on Facebook users and the victims. Following this event, users have initiated campaigns on other social media platforms to delete Facebook, expressing concerns over privacy invasion. Additionally, new reports found that about 74% of Facebook users have updated their privacy settings, taken a break or deleted the app [43].

An aspect of concern in the project revolved around the storage of trip data. Originally it was planned to store users' data in the cloud database MongoDB synchronized with data from IndexedDB when online, aiming to prevent loss of data in case users accidentally deleted the data from the browser. I have decided to not implement this feature to respect users privacy, particularly considering the sensitive nature of the personal information involved, and to respect the privacy of the potential users during demonstrations. This decision also aligns with the primary goal of the application: to enable users to save their memories privately. I opted for the decision to leave it entirely to the users decision how this data would be stored, despite the inherent risk of potential data loss. To ensure the protection of users' privacy, compliance and adherence with regulations, other critical security measures would have been necessary if data were to be securely stored in the cloud.

In conclusion, disregarding professional issues such as privacy can lead to significant repercussions, affecting not just the involved parties but also society at large. The highlighted event not only altered individuals' behaviours and attitudes toward their online privacy but also reshaped societal perceptions, laws and regulations regarding privacy. This emphasizes the profound impact that privacy breaches can exert on individuals, communities, and the regulatory framework.

6.2 Self-assessment

1. How did the project go?
Overall, the project advanced successfully and as planned. I have effectively created an offline map application utilizing HTML5 technologies and introduced a distinctive feature that sets it apart. The implementation phase presented various challenges of differing complexities, contributing to the refinement of my skills and the acquisition of invaluable experience. Despite encountering ups and downs along the way, the project has been a valuable learning journey.

2. What are the next steps?
Moving forward, the next steps involve conducting user testing to gather feedback, refining the design and user experience based on the insights gained, and enhancing current functionalities as necessary. Additionally, plans include expanding the map to enable users to download any location worldwide, saving data in a remote database to ensure trip data remains safe and synchronized with IndexedDB, and implementing user account creation feature.
Other plans include enabling automatic map updates when data becomes outdated, incorporating a search functionality to streamline the process of selecting trip locations, and integrate image cloud storage functionality. These developments aim to enhance the overall usability and functionality of the application, providing users with a more seamless and enriched experience.

3. What did you do right?
The tech stack was a good choice from the start. Time management was on track, and successfully implemented all the features I had planned for the final product. The initial proof of concept programs played an important role and helped during project development for references.

4. What did you do wrong?
What didn't go as planned was testing. Adhering to Test-Driven Development (TDD) would have facilitated a smoother learning curve. Additionally, I struggled with defining the application goals, leading to changes in the goals at each stage of development, despite ultimately achieving a successful end product.

5. What have you learnt?
Throughout this project, I have gained valuable skills and insights. I have learned how to deploy the front-end and back-end of a website and the steps necessary to build a product from start to finish. Additionally, I understand the significance of selecting the right libraries to ease project development and the importance of following a software engineering methodology. Furthermore, I have gained proficiency in using a development server and implementing offline functionality in applications. Lastly, I have recognized the importance of utilizing academic papers and reputable sources for planning and comprehending technical requirements, enhancing the overall quality and reliability of the project.

Conclusion

In conclusion, this report outlines the efforts made, detailing the methodology employed, the steps taken to achieve the end-product and highlighting the significance of the technologies used.

The culminating point of this work is the creation of a Progressive Web Application (PWA), offering users an native app-like experience whilst leveraging the accessibility of the web. Leveraging HTML5 technologies such as IndexedDB, service workers, and caching mechanisms empowers the application to function seamlessly even when offline, bringing the user experience to the next level. OpenMapTiles library plays an important role in structuring OpenStreetMap data into vector tiles, laying the foundation for a lightweight map and seamless user experience. Integration with the OpenLayers library further enhances the user experience by enabling dynamic rendering of vector tiles on the client side.

The application serves as a platform for users to archive their travel experiences, memories, and visited locations directly onto the map, linking memories to their location, fostering a personalized travel log perfect for travel enthusiasts. Furthermore, the capability to share these experiences with friends through PDF downloads ensures data privacy while facilitating sharing.

Looking ahead, the application holds substantial potential for expansion. Scaling the application to cover global map data would broaden its utility, bringing a wider user base. Additionally, exploring the integration of image cloud storage functionality could enrich the user experience by providing a seamless way to store and organise images associated with trips whilst linking them to their location. This envisioned evolution of the application, is inspired with features similar to the popular mapping platform like Google Maps, while offering users enhanced privacy and personalization. Overall, I believe the application stands to be a valuable tool for travel enthusiasts seeking to document, share and relive their travel adventures.

Appendix

User Manual

This section is a user manual, providing information on how to use the application effectively by outlining its features. The application can be accessed locally by following the installation steps detailed in the subsequent section, or it can be accessed online at `https://offline-map-2o39.onrender.com/`.

The application is a Progressive Web App (PWA) allowing to integrate it as a shortcut to the device's home screen, simulating the experience of a traditional application without the conventional installation procedure. Upon accessing the application via the web browser, two prompts are presented by the browser. The first prompt on the left side of Figure 1 asks for permission to access the device's location information, this information is used to display its location on the map. The second prompt, displayed on the right side of Figure 1, enables adding the application to the device's home screen.

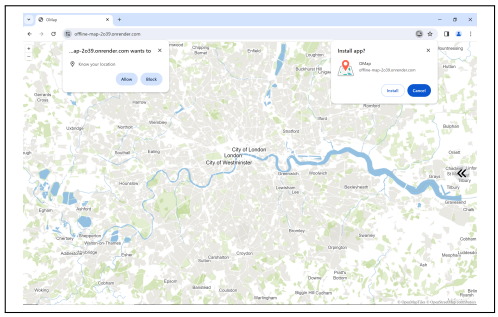

Figure 1: Browser prompts.

The application consists of one page, the index page featuring the map alongside a panel on the right side for desktop devices, or a collapsible panel for mobile devices. The layout is illustrated in the corresponding images from Figure 2. The application showcases the map data of London, and navigation beyond the geographical bounds of London is not supported.

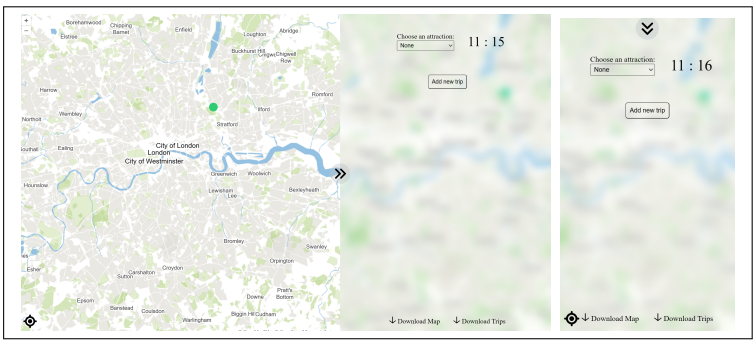

Figure 2: Desktop and mobile design.

The location feature is activated when online and deactivated when offline. Clicking the button located on the bottom left corner, as shown in Figure 3, zooms the map on the device's location, depicted by the green circle on the map.

Figure 3: Geolocation.

To achieve full offline functionality the map data must be downloaded. This is achieved by clicking the download button located at the bottom of the panel, as highlighted in Figure 4. As the map data is being downloaded, the progress bar shows the download progress. Upon download completion, the application will have full offline functionality.

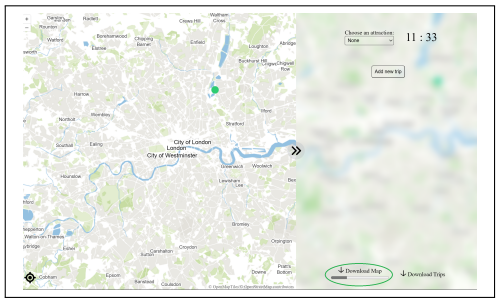

Figure 4: Download map data.

The features presented so far are dependent on an internet connection. When offline, the location feature and download functionality become unavailable. Consequently, the buttons associated with these features are removed. The map can be navigated and offers zoom in/out functionality.

The application offers a feature to display the locations of various attractions. By selecting the "Choose an attraction" button within the panel, four attractions are available: internet cafes, museums, malls and theatres. Map pins representing the selected attraction are displayed on the map at their corresponding locations. An example can be seen in the figure below.

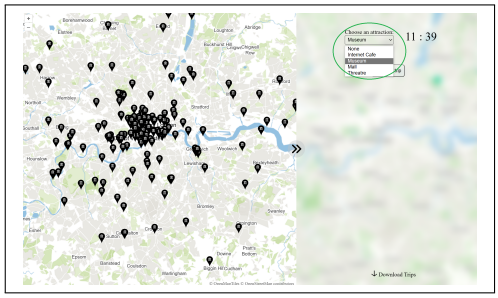

Figure 5: Attractions.

A central feature of the application is adding trips to the map, saving memories linked to specific locations. A trip can be added from the panel by selecting "Add new trip" button highlighted below. Upon clicking the button, a form is presented enabling to add trip information and upload images. An example of a completed form is illustrated in Figure 6.

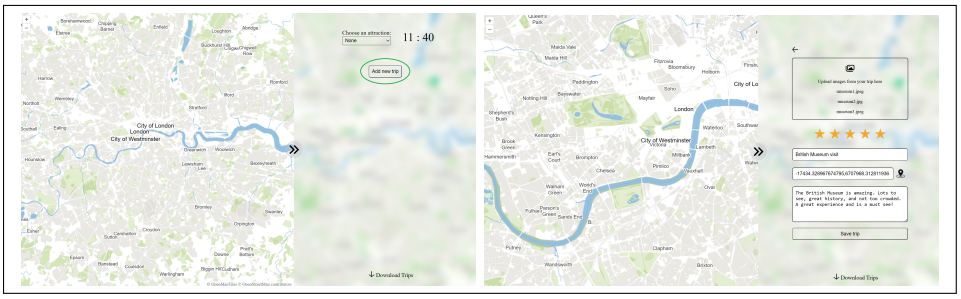

Figure 6: Add trip.

A trip location can be added in two different ways: typing in the location address manually, or by selecting the icon next to the trip location field, as highlighted in Figure 7. If the address is manually typed, the trip is not displayed on the map. However, if the icon is clicked, then the trip location can be selected by directly selecting the location on the map. This feature automatically retrieves the trip location coordinates, allowing to add the trip to the map.
Note: the trip data is saved locally in the browser, thus clearing the browser cache will delete all saved trips

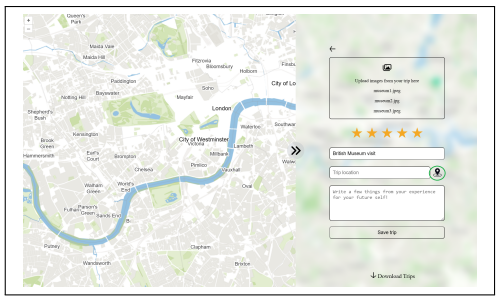

Figure 7: Trip location.

The saved trip is added to the panel, displaying trip images, ratings, and other provided information. If the location has been selected using the select location feature, a pin is placed on the map at the selected location. Upon selecting a trip pin from the map, the associated trip is highlighted in the panel by adding a box shadow around it, as shown in Figure 8. Additionally, the selected trip is repositioned to the top of the panel, as illustrated in the second snapshot.

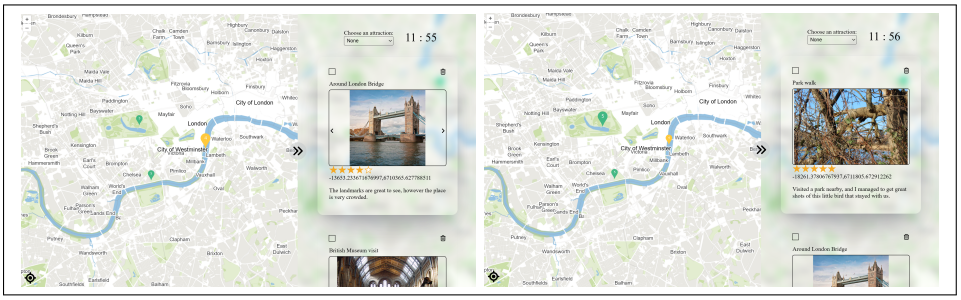

Figure 8: Highlight trip.

Every trip can be represented by six distinct pins, determined by its rating. Each trip pin is visualised by a different colour including black, red, orange, light orange, yellow and green. Additionally, each pin displays the corresponding numerical rating. The various trip pins are shown in Figure 9.

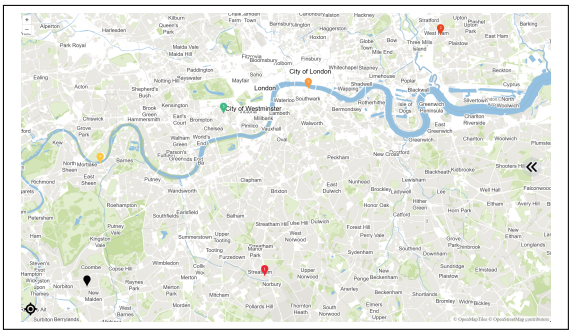

Figure 9: Trip ratings.

A trip can be deleted by selecting the bin icon, as highlighted in Figure 10. This functionality deletes the trip from the panel and the corresponding map pin associated with the trip. This behaviour is shown in the images presented in Figure 10.

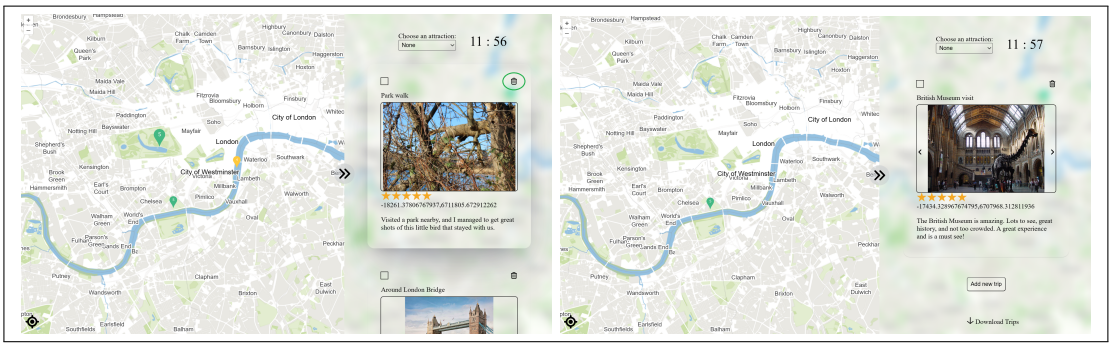

Figure 10: Delete trip.

Trips can be downloaded and shared with others. By clicking on the box from the trip container a trip is selected for download, as highlighted in the first snapshot of Figure 11. Then, clicking on the "Download Trips" button, as highlighted in the second snapshot of Figure 11, will generate a PDF document containing the information of the selected trips, resulting in a document as seen in the third snapshot in Figure 11.

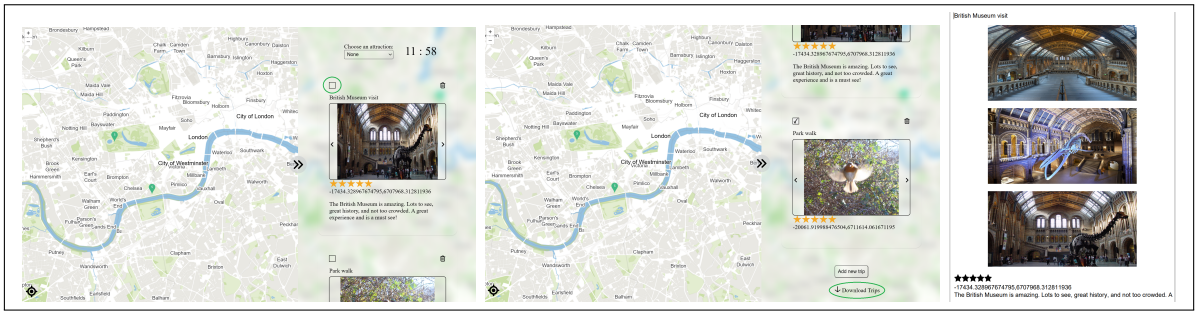

Figure 11: Download Trips.

View the app running at: `https://www.youtube.com/watch?v=0j3teGtO1kM`

Installation Manual

To run the application locally follow the next steps:

1. Open the folder *AndreeaGardelean.final* in an editor of your choice.

2. Initiate two terminals
   - Terminal 1: navigate to the root directory of the project by executing *cd PROJECT*
   - Terminal 2: access the scripts directory within the project by executing *cd PROJECT/src/scripts*

3. Install dependencies
   - Within Terminal 1 install essential project dependencies by running the command:

   ```
   npm install
   ```

   Note: If using a different package manager, such as yarn, substitute *npm* with *yarn*

4. Launch tile server
   - After installing dependencies, in Terminal 2 run the command:

   ```
   node tileserver.js
   ```

   This command initiates the tile server for serving map tiles.

5. Run the application
   - Return to Terminal 1 and trigger the application by running the command:

   ```
   npm run preview
   ```

6. Access the application
   - The application will be available at `http://localhost:4173/`

For visual demonstration of the above steps, refer to the following demonstration video. Alternatively, the application can be accessed over the web at the following URL: `https://offline-map-2o39.onrender.com/`.

Diary

08/10/2023
Studied service workers and cache, implemented service workers and cache by completing proof of concept program 'Hello World' which stored an HTML document in the cache and worked while offline.

10/10/2023
Studied IndexedDB, completed 'To-do list' proof of concept which opens a database in the browser has the following functionalities:
- input field to add item
- add task to IndexedDB when user clicks 'Add' button
- display newly added task on the page
- retrieve all tasks from the database and display on screen when page loads
- remove task from database when the radius button is selected
- remove task from screen when radius button is selected

11/10/2023
Finalised To-do proof of concept:
- added CSS styling
- when task is selected allow user to edit task
- when user clicks outside task text after it was focused the task will update in database and on the page
This program followed PWA methodology and provides offline capabilities by using service workers and cache, and has a manifest which allows to add the application to the home screen.

17/10/2023
Studied HTML5 canvas and its functionalities.
Created a clock using HTML5 canvas which displays the current time and moves the seconds, minutes and hour pointers in real time.

Completed "Display OSM data in raw format" proof of concept, which retrieves the UK city names using Overpass API for retrieving the data over the web and Overpass QL (Query Language) to create a query for retrieving the data. The data is the displayed on the HTML document.

Set up project file structure - divided the project into client-side and server-side files.
Set-up the retrieval of OSM data via Overpass API functionality in server side with Node.js, created a Node.js server to run the application.

23-29/10/2023
Studied how to create vector tiles, I have used OpenMapTiles to generate the vector tiles for London and England. Studied how to host vector tiles, set up tilserver-gl to host the tiles using a docker container.
Displayed the vector tile on the HTML page using leaflet with VectorGrid plugin, the map was displayed without styling, the data was displayed with blue lines.
Tried displaying the map using other plugins but unsuccessful because most of them required to use MapTiler and Mapbox and did not align with my prerequisites.

1/11/2023
- added map styling by hand for testing
- added styling created by OpenMapTiles in styles.json file

Tried to add map styling from style.json file in leaflet but was unsuccessful and after some research I have found that leaflet does not provide such functionality and the styling had to be manually defined.
I have done some research on other libraries and I have found OpenLayers and decided to work with this library instead. This has caused some changes to the project, the Node.js server I have created was no longer practical in this case because I had issues whilst importing modules in JavaScript which were necessary for running the application. I have decided to use Vite as a server and run the server using npm. I have also decided to re-structure the file system and ordered the files based on their extension rather than functionality. This has helped me in identifying were the files are in the project.

5/11/2023
Added manifest.json to allow app download, created and registered service worker for offline functionality and added assets to cache.
- enabled geolocation to locate user
- a red circle is drawn where the user's location is
- updated worker to fetch assets from cache only when server is offline

9/11/2023
- calculated the location dot size dynamically based on zoom level
- added map layers dinamically
- using watchPosition() for location user, this way the map can get updates as the user's location updates
- when the user location is identified the map will be focused on the user's location
- changed pointer color to green
- added clock on the right side panel
- created a basic compass using HTML canvas

10/11/2023
- opened IndexedDB database for storing map data
- added credits to the map for using OpenStreetMap and OpenMapTiles as specified by the tools
- added service worker tests using mocha

14/11/2023
Updated "Display raw OSM data" proof of concept to retrieve data for museums from London, this provided a better understanding of the data OSM provides.

16/11/2023
Updated HTML5 canvas proof of concept to allow the user to draw on the canvas. I have miss-understood the ask and created a static application which displays a clock, which was drawn on canvas.

17/11/2023
Updated compass on the right side panel. Added a window event listener for device orientation, which will rotate the pointer of the compass as the user's device orientation changes. This feature is only supported by devices which have an orientation sensor.

21/11/2023
Documented how to store vector tiles in IndexedDB, and decided to convert the map data in GeoJSON format which can then be stored in the browser. This functionality requires to create my own SQLite vector tile server. The user will have the ability to download a chosen

area using a download button.

23/11/2023
Created a download button to download map data.
Researched how database indexing works
Added a search bar to allow user to search for location or other data
Added event listener to the search bar for when the user clicks on the search icon or presses the key 'Enter'
Converted OSM data (.pbf file) format to GeoJSON and tried to render the GeoJSON data with openLayers but did not work.
I am exploring ways on how I can save the vector map data to IndexedDB, and since JSON is key-value ordered it would make it the appropriate format for storing it in IndexedDB - but unable to render it using OpenLayers

24/11/2023
Added an event listener which listens to the event of loading the vector tile. I have implemented this event in order to get the x, y, z coordinates of the requested tile. I believe this will be helpful for the offline functionality because these coordinates will be needed to know which tile data to fetch from the database.

01/12/2023
- updated the side panel to be collapsible using a button - this was achieved using jQuery
- created a 'proof-of-concept' for rendering GeoJSON format of the map using OpenLayers
- resolved typo in 'deviceorientation' event listener
- when the compass pointer is redrawn when angle changes the image is first cleared before drawing again
- created object stores in IndexedDB for each zoom level - each object store will store the tiles at the respective zoom level

04/01/2024
- set up a tile server using NodeJS to read tiles from the tiles.mbtiles file such that the project no longer relies on the third party tileserver-gl for serving tiles
- updated vector tile source URL in OpenLayers to connect to the NodeJS tile server for reading the data
- implemented an endpoint for reading the tiles from tiles.mbtiles using mbtiles library for online use

08/01/2024
- added an event listener for the 'Download' button, allowing users to click it to download map tiles and store them in the database
- created endpoint for reading tiles for offline use
- uncompressed the retrieved map tile data and converted it to a Vector tile using the mapbox plugin (for offline use)
- added vector tiles to IndexedDB for the first 11 zoom levels

09/01/2024
- bug fix: the tiles are added to the database before the database is open. This was solved by wrapping the *openDatabase* function in a promise, and the tiles will be added only when the function returned the promise
- created a function for retrieving map tiles from IndexedDB based on requested coordinates

12/01/2024
- created a custom function to serve as a layer source URL in OpenLayers, to intercept tile requests, enabling retrieval of the requested tile coordinates. The returned URL varies

depending on whether the user is online or offline (if online return the server URL, if offline returns the function for retrieving tile from IndexedDB)
- the tiles would not get rendered because of the error: 'Pbf unimplemented type 4' – this occurred because the tiles were not stored in the correct format in IndexedDB. Converted the tiles data to an array buffer to check then but still no success
- updated the range of requested tiles for adding to IndexedDB
- the tiles are no longer unzipped when retrieved form the database for performance issues and it is not necessary

18/01/2024
- implemented custom *tileLoadFunction* allowing to customise how the vector tiles are loaded on the map – allowing to decide the source of the vector tiles based on the internet connectivity
- updated documentation
- modified the server endpoint for offline tiles to send to the client the existing tile coordinates from tiles.mbtiles for the specified zoom level
- changed the approach for requesting tiles for offline access - when the download button is clicked the method *fetchAndAddTiles* is triggered. Then contacts the offline endpoint of the server returning the coordinates of the existing tiles for the specified zoom level. The client then iterates over the coordinates and requests, from the server, the tiles with the given coordinates. This process will be executed for zoom level 0 to 14. Each returned tile is then added to IndexedDB
- refactored the method of retrieving the tiles from tiles.mbtiles file, no longer uses the 'mbtiles' library, is manually implemented using sqlite3 library, providing more flexibility and control over how the tile is retrieved

24/01/2024
- updated application design with a wider side panel and new colour scheme
- implemented responsive design to ensure scalability across various devices
Re-imagined the application's purpose to enable users to save visited locations on the map. Users can now add photos, descriptions of their experiences, and ratings for each location. Additionally, the application facilitates selecting a location directly by selecting on the map.

25/01/2024
- implemented 'Locate me' feature – when the button is clicked the map will zoom in on the user's location. The functionality is not available when offline or geolocation service is unavailable.
- default focus of the map is London, users are restricted from zooming out beyond the London area
- bug fix: users cannot zoom the map over zoom level 14 – if is over 14 the map becomes blank
- bug fix: when the location is updated the previous location dot on the map is deleted to prevent clutter
- created a form for users to enter trip information, including the ability to upload images from their device and provide a rating
- the form is only visible when the 'Add new trip' button is clicked, upon submission the form fields data is retrieved, the form fields are reset, and the form is hidden

31/01/2024
- created trip container within the panel to display each trip
- completed front-end design for trip container
- when an image is uploaded the image name will be displayed in the image picker providing, users the ability to preview their uploads effortlessly

01/02/2024
- upon submission of the trip form, a trip container with the submitted data is created and appended to the panel
- opened a new IndexedDB database dedicated to storing user trips
- upon form submission the trip data is added to IndexedDB, when the app loads the trips are retrieved from IndexedDB and displayed in the panel

07/02/2024
- added tests for indexedDB.js, tripIndexedDB.js and tileserver.js
- refactored functions handling indexedDB to return promises upon completion
- bug fixe: - when offline the service worker did not render the css – the issue was fixed by adding an import statement to the index.html file
- added a progress bar to indicate the progress of the map download

09/02/2024
- bug fix: at level 14 when server requests failed during map download a retry mechanism has been implemented. The failed tile request is retried one more time.
- If an error occurred when fetching the tile, the service worker would intercept this. Issue fixed by instructing the service worker to ignore requests from the URL containing substring */data/tiles/*

12/02/2024
- bug fix: when the location is updated the previous position circle would not get deleted – this was fixed by separating the position layer from the position update layer
- implemented trip location pins and integrated them into the map, corresponding to the trip location, this was achieved by: - creating a new vector layer to contain the pins
- assigned each pin a unique id, which corresponds to the trip's id generated when the item is added to the database
- when a new trip is created a pin is added to the map – at the moment the location is hard typed however in the final product the pin will be added to the selected location of the trip
- upon pin selection, the pin is highlighted on the map by increasing its size and the trip linked to the pin is also highlighted on the panel by adding a box shadow to the trip container making it stand out
- to ensure only the selected pin and its corresponding trip are highlighted, all the previously highlighted trips are un-highlighted
- updated functions documentation
- refactored functions to remove long method smell
- added a mechanism for selecting trip locations manually by allowing user to click on the map. When the map is clicked, coordinates are captured and a pin is added to that location upon form submission.

13/02/2024
- enhanced the trip pins with colour coding for easy identification: - trips with no rating: black pin
- 1 star rated trips: red pin with number one in the center
- 2 star rated trips: red-orange pin with number two in the center
- 3 star rated trips: orange pin with number three in the center
- 4 star rated trips: yellow pin with number four in the center
- 5 star rated trips: green pins with number 5 in the center
- Created a back button on the form allowing the user to go back to the trip panel after entering the trip form
- added a trash can icon to each trip container, allowing the user to delete the trip from the database and panel

21/02/2024

- enhanced user interface for mobile mode with the following updates:
- implemented touch-based interaction, enabling users to open and close the panel via swipe gestures
- upon selecting the select location button from the form, the panel will close 70%, allowing the user to interact with the map
- refactored and extracted code in other functions for reusability
- upon selecting the locate me button the panel will be close if open
- added a window event listener to enable touch events when the window width is under 700px and removes the event when exceeds 700px such that the panel cannot be dragged up and down when on larger screens
- the data is added to the database by using *put()* method, allowing to overwrite existing data in the database
- if fetching a tile does not go through because of a TypeError the method will retry fetching three more times to prevent an infinite loop and ensure all tiles are available

27/02/2024
- created vector layers for adding attractions to the map - created a new database for storing attractions - created 2 layers which hold the attractions museums and internet cafes

28/02/2024 - added a button in the panel enabling users to select attractions to see on the map
- by default, the attractions layers are hidden and when an option is selected the layer corresponding to that option is displayed
- upon selecting an attraction for display, all other layers are hidden
- added two more attraction layers – mall and theaters
- updated the application from online-first to offline-first behavior
- regenerated map tiles
- if the resource requested is not in the cache it will fallback to the network and the missing resource is added to the cache
- if it cannot be found in the network it will return nothing
- download button is hidden when no internet connection is available
- when a tile does not exist in the IndexedDB is fetched from the server, rendered from the server and added to the IndexedDB

02/03/2024
- made small CSS updates
- checked and updated documentation
- generated JSDoc
- refactored code
- updated folder structure
- released the application to production using Render
- updated the panel touch move event

05/03/2024
- updated service worker to have an update() event which updates the service worker when an internet connection is available
- map size is updated to cover 70% of screen size when panel is open

06/03/2024
- bug fix: when offline the OpenLayers library is not available and needs to be cached. This was achieved by using PWA Vite plugin which allows to pre-cache all the bundled resources when the service worker is registered
- updated README.md
- CSS updates to the form and map

11/03/2024
- updated service worker tests
- code format updates
- the trip is deleted from the panel only after a promise has been returned from IndexedDB that the transaction is successful

18/03/2024
- added download trips button for downloading selected trips in pdf format
- added an icon for each trip, allowing to select the trip for pdf download

25/03/2024
- when the download button is clicked the information for the trips is retrieved and converted to a pdf

Bibliography

[1] T. Ater. (2017, Sep). "Introducing Progressive Web Apps" in *Building Progressive Web Apps: Bringing the Power of Native to the Browser*, 1st ed [Online], A. MacDonald, J. Bleiel, Sebastopol, CA: O'Reilly Media, 2017 [Online]. Available:`https://learning.oreilly.com/library/view/building-progressive-web`

[2] MDN Web Docs (2023, Jun. 08). *HTML5* [Online]. Available:`https://developer.mozilla.org/en-US/docs/Glossary/HTML5`

[3] Geeks for Geeks (2021, Mar. 15). *Top 10 new features of HTML5* [Online]. Available:`https://www.geeksforgeeks.org/top-10-new-features-of-html5/`

[4] E. A. Meyer, "Chapter 1: CSS Fundamentals" in *CSS the definitive guide* (Safari Books Online.), eng, 3rd ed. Sebastopol, Calif: O'Reilly, 2006, isbn: 0596527330. [Online]. Available:`https://learning.oreilly.com/library/view/css-the-definitive/9781098117603/ch01.html#a_brief_history_of_open_parenthesis_web`

[5] MDN Web Docs. (2023, Oct. 10). *What is JavaScript?* [Online]. Available:`https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript`

[6] MDN Web Docs. (2023, May. 21). *Introduction to the DOM* [Online]. Available:`https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`

[7] S. Fulton and J. Fulton, *HTML5 canvas*, eng, 2nd ed. Sebastopol, Calif: O'Reilly, 2013, isbn: 9781449334987. [Online]. Available: `https://learning.oreilly.com/library/view/html5-canvas-2nd/9781449335847/ch01.html.40`

[8] S. Holzner, *jQuery (Visual quickstart guide)*, eng. Berkeley, Calif: Peachpit Press, 2009, isbn: 9780321679673. [Online]. Available: `https://learning.oreilly.com/library/view/jquery-visual-quickstart/9780321679673/fm.html.`

[9] T. Ater. (2017, Sep). *Building Progressive Web Apps: Bringing the Power of Native to the Browser*, 1st ed. Sebastopol, CA: O'Reilly Media, 2017 [Online]. Available:`https://learning.oreilly.com/library/view/building-progressive-web`

[10] MDN Web Docs. (07/07/2023). *Using the Web Storage API* [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API`

[11] W3 Schools. *HTML Web Storage API* [Online]. Available:`https://www.w3schools.com/html/html5_webstorage.asp`

[12] MDN Web Docs. (26/05/2023). *Web Storage API* [Online]. Available:`https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API`

[13] LearnOSM. *File Formats* [Online]. Available:`https://learnosm.org/en/osm-data/file-formats/`

[14] OpenStreetMap Wiki (03/10/2023). *Elements* [Online]. Available:`https://wiki.openstreetmap.org/wiki/Elements`

[15] R. García, E. Verdú, L. M. Regueras, J. P. de Castro, and M. J. Verdú, "A neural network based intelligent system for tile prefetching in web map services," *Expert Systems with Applications*, vol. 40, no. 10, pp. 4096–4105, 2013. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S095741741300050X`

[16] T. Dubrava (Sep. 2017). *Design of a Multi-Scale Base Map for a Tiled Web Map Service* [Online]. Available: https://cartographymaster.eu/wp-content/theses/2017_DUBRAVA_Thesis.pdf

[17] R. Netek, J. Masopust, F. Pavlicek, and V. Pechanec, "Performance Testing on Vector vs. Raster Map Tiles—Comparative Study on Load Metrics," *ISPRS International Journal of Geo-Information*, vol. 9, no. 2, p. 101, Feb. 2020, doi: 10.3390/ijgi9020101. Available: https://www.mdpi.com/2220-9964/9/2/101

[18] OpenMapTiles. *OpenMapTiles* [Online]. Available:https://wiki.openstreetmap.org/wiki/OpenMapTiles

[19] GitHub. *MBTiles Specification* [Online]. Available:https://github.com/mapbox/mbtiles-spec

[20] L. Matinelly, M. Roth (2016). *Updatable Vector Tiles from OpenStreetMap* [Online]. Available:https://cdn.jsdelivr.net/gh/osm2vectortiles/bachelor-thesis@882a46977f7b984fc59be188335127d62921c44c/thesis.pdf

[21] J. Gaffuri, "Toward web mapping with vector data," in *Geographic Information Science*, N. Xiao, M.-P. Kwan, M. F. Goodchild, and S. Shekhar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 87–101 [Online]. Available:https://link-springer-com.ezproxy01.rhul.ac.uk/chapter/10.1007/978-3-642-33024-7_7

[22] G. Farkas, *Mastering OpenLayers 3: create powerful applications with the most robust open source web mapping library using this advanced guide*, 1st ed., ser. Community experience distilled. Birmingham: Packt Publishing, 2016 [Online]. Available: https://learning.oreilly.com/library/view/mastering-openlayers-3/9781785281006/

[23] OpenLayers Home page. Available: https://openlayers.org/

[24] OpenLayers, *ol/source/VectorTile* [Online] . Available: https://openlayers.org/en/latest/apidoc/module-ol_source_VectorTile.html

[25] OpenLayers API documentation [Online]. Available: https://openlayers.org/en/latest/apidoc/

[26] D. Parker, *JavaScript with promises*, first edition. ed. Sebastopol, CA: O'Reilly Media, 2015 [Online]. Available: https://learning.oreilly.com/library/view/javascript-with-promises/9781491930779/

[27] C. S. Horstmann. *Modern JavaScript for the Impatient*. Pearson Education, 2020. [Online]. Available: https://learning.oreilly.com/library/view/modern-javascript-for/9780136502166/

[28] C. Hudson and T. Leadbetter, "10. Location Awarness with the Geolocation API" in *HTML5 developer's cookbook*, ser. Developer's library. S.l: Addision-Wesley Professional, 2012 [Online]. Available: https://learning.oreilly.com/library/view/html5-developers-cookbook/9780132697361/?ar=

[29] A. T. Holdener, *HTML5 Geolocation*. Sebastopol, CA: O'Reilly, 2011 [Online]. Available: https://learning.oreilly.com/library/view/html5-geolocation/9781449308049/ch03.html#current_api_support

[30] MDN Web Docs (01/12/2023). *Geolocation API* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API

[31] JSDoc. *Getting started with JSDoc* [Online]. Available: https://jsdoc.app/about-getting-started

[32] James Long (10/01/2017). *A Prettier JavaScript Formatter* [Online]. Available https://archive.jlongster.com/A-Prettier-Formatter

[33] A. Stellman and J. Greene, *Learning Agile : understanding Scrum, XP, Lean, and Kanban*, first edition ed. Sebastopol, CA: O'Reilly Media, 2014 - 2015. Available: https://learning.oreilly.com/library/view/learning-agile/9781449363819/ch01.html

[34] GitHub. *node-mbtiles* [Online]. Available: https://github.com/mapbox/node-mbtiles/blob/master/lib/mbtiles.js

[35] Vite PWA (2023, Nov. 18). *Getting Started* [Online]. Available: https://vite-pwa-org.netlify.app/guide/

[36] Vite PWA (2023, Nov. 18). *Service Worker Precache* [Online]. Available: https://vite-pwa-org.netlify.app/guide/service-worker-precache.html

[37] C. Peterson, *Learning responsive web design a beginner's guide*, 1st ed. Sebastopol, CA: O'Reilly Media, 2014 [Online]. Available: https://learning.oreilly.com/library/view/learning-responsive-web/9781449363659/

[38] Cambridge Dictionary [Online]. Available: https://dictionary.cambridge.org/dictionary/english/privacy

[39] S. Garfinkel and G. Spafford, *Web security, privacy and commerce*, 2nd ed. Cambridge, Mass: O'Reilly, 2001 [Online]. Available: https://learning.oreilly.com/library/view/web-security-privacy/0596000456/

[40] F. Bott, *Professional Issues in Information Technology*. 2nd ed. Swindon, UK: BCS, 2014 [Online]. Available: https://learning.oreilly.com/library/view/professional-issues-in/9781780171807/

[41] GOV.UK. *The Data Protection Act* [Online]. Available: https://www.gov.uk/data-protection

[42] M. Hu, "Cambridge analytica's black box" *Big Data and Society*, vol. 7, no. 2, p. 2053951720938091, 2020. [Online]. Available: https://journals.sagepub.com/doi/epub/10.1177/2053951720938091

[43] A. J. Brown, ""should i stay or should i leave?": Exploring (dis)continued facebook use after the cambridge analytica scandal," *Social media + society*, vol. 6, no. 1, pp. 205 630 512 091 388–, 2020 [Online]. Available: https://journals.sagepub.com/doi/pdf/10.1177/2056305120913884

[44] D. Masruroh and R. Satria, "13. the effect of cambridge analytica case in cyberspace politics," in *Proceedings of the 5th International Conference on Social and Political Sciences (IcoSaPS 2018)*. Atlantis Press, 2018/08, pp. 60–63. [Online]. Available: https://www.atlantis-press.com/proceedings/icosaps-18/25904097

[45] B. Tarran, "What can we learn from the Facebook—Cambridge Analytica Scandal?" *Significance*, vol. 15, no. 3, pp. 4–5, 05 2018. [Online]. Available: https://academic.oup.com/jrssig/article/15/3/4/7029334?login=true

[46] Moodle (02/05/2022). *Professional Issues in your final project report* Available: https://moodle.royalholloway.ac.uk/mod/page/view.php?id=744455