

Final Year Project Report

Full Unit – Interim Report

Offline HTML5 Maps Application

Noe Chen Jiang

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

December 07, 2023

Table of Contents

Abstract	2
Timescale and planning	3
Software Engineering.....	5
Basic web page development in HTML5: HTML, JavaScript, CSS.....	8
Advanced technologies: jQuery, HTML5 canvas.....	10
Developing an offline HTML5 application: Web Storage, Indexed DB, Service Workers and the Cache API.....	13
Open Street Map data representation and vector vs. image tile maps	18
Content moderation	21
Bibliography.....	23
Ethics.....	22
Appendix: diary and video demonstration	24

Abstract

This interim report outlines the ongoing development of the Offline HTML5 Maps Application. The core feature of this application will allow users to leave location-based messages offline, which will be available for other users to see once the user connects to the internet. So far, the technologies used are HTML, Cache, JavaScript, Service Workers, IndexedDB, Django, DynamoDB, and Leaflet.js. As this project is in its early stages, future developments will include the integration of additional technologies: CSS and jQuery to further enhance user interface and expand functionalities.

Key proofs of concept, demonstrating these technologies are a "hello world" offline HTML5 application using Cache and Service Workers, a "todo list" application using IndexedDB, an application for drawing shapes with HTML5 canvas, and a web page that presents Open Street Map (OSM) data in raw form. This report also includes related reports in areas such as basic and advanced web page development with HTML5, CSS, and jQuery, the development of offline applications with Web Storage, Indexed DB, Service Workers and the Cache API, and the representation of Open Street Map data and vector vs image tile maps.

An important feature that will be discussed further in this report is the development of a content moderation system. This mainly will consist of filtering messages using keyword lists and potentially language models for inappropriate content in order to ensure a safe and respectful user environment. There are many challenges in implementing this system, such as automated moderation and ethical implications of censorship. Ideally, comments should be, for example, an observation of an interesting geographical feature, hazard warnings, casual comments ("There is a gathering of ducks here!"), etc. These are the desirable types of comments, which will guide the implementation of content moderation. Human moderation will also be a topic of discussion.

At the time of writing this report, there are web pages showcasing each proof of concept, as well as a map created with Leaflet.js using OSM data. These pages have little to no styling, as the intent is to demonstrate functionality above all else for proofs of concepts. However, styling with CSS remains an important part of this application. The implementation of these proofs of concepts will be discussed in depth in this report.

Upon completion, this project will have the following features: the application will work offline, allow the user to load, display map data, zoom and move around the map. Furthermore, and as mentioned before, the user will be able to leave public messages. These messages will also be automatically moderated for inappropriate content with keyword lists and can also be reported by other users for human moderation to evaluate. Lastly, possible extensions include features for automate moderation using natural language processing, additional social features such as message reactions and share options, and search and filter options for messages.

Timescale and planning

Fortunately, the goals that have been set so far have been achieved. However, there were a few problems with the initial timeline proposed. First and foremost, the tasks were spread unevenly as a result of overestimating some tasks, which were mainly the development of proof of concept programs. On the other hand, the interim report was underestimated, and an additional two weeks were needed to complete such task. Thankfully, the early completion of proof of concept programs helped minimize the impact of such underestimation.

Another problem with the previous plan was the lack of a mention of a testing framework or methodology. This should have been part of the initial setup but was implemented much later on than it should have been. This was not a big issue as the proofs of concepts are smaller scale programs, but it is important to note that this would have become a major problem on a larger scale development environment. These problems were thankfully minor setbacks and will be taken into account in the future.

While the initial scope of the project was generic, it was expected that a significant part of the functionality of the application would be related to processing and/or filtering (and potentially rendering) OSM data. However, it was later decided that messaging functionality (and additional social features as extensions) would be the main focus of this project. Furthermore, messaging functionality would require content moderation. As such, much of the original proposal regarding OSM data had to be discarded to avoid scope creep. This is reflected in the choice of APIs, such as Leaflet.js. Below is an updated version of the project plan that takes into account the changes and problems discussed (starting from week 10).

September-January: Proof of concepts, reports, and research

Week 1-3 (late September to early October)

- Project plan: define scope, goals, and milestones, as well as research.
- Development environment setup and version control.

Week 3-4

- Use Service Workers and Cache to create a “Hello World” offline HTML5 application.
- Document and report findings and challenges.

Week 5-6

- Develop “to do list” application with IndexedDB/Web Storage.
- Experiment with IndexedDB/Web Storage to find which technology is more suitable.
- Document and report findings and challenges.

Week 7-8

- Build an application to draw shapes using HTML5 canvas.
- Document and report findings and challenges.

Week 9-10

- Create web page that loads and list raw OSM data.
- Document and report findings and challenges.

Week 10-12

- Write a report on basic web page development in HTML5: HTML, JavaScript, CSS.
- Write a report on advanced technologies: jQuery, HTML5 canvas.
- Write a report on developing an offline HTML5 application: Web Storage, Indexed DB, Service Workers and the Cache API
- Write a report on Open Street Map data representation and vector vs. image tile maps.
- Include insights from research and proof of concept programs.
- Prepare interim report and presentation.

Week 13-15

- Begin development of the offline HTML5 Maps application.
- Implementation of basic functionality: offline access, loading and displaying map, zoom and map navigation.
- Document progress, findings, and challenges.
- Implement end-to-end testing.

January-May: Development and extensions**Week 16-18**

- Implement messaging functionality and automated content moderation with keyword lists.
- Implement tests for end-to-end testing.
- Document progress, findings, and challenges.

Week 19-21

- Develop features for improving rendering performance, if needed.
- Implement human moderation.
- Add styling to the application.
- Do performance tests and optimize accordingly, if needed.

Week 22-24

- Finalize application and conduct testing of the entire system to ensure requirements are met.
- Consider implementing extensions (only and only if progress and time allows): automated content moderation using a language model (AWS Comprehend), additional social features such as message reactions, and search and filter options for messages.

Week 25-27

- Compile project documentation: reports and code.
- Prepare for project submission and demo/presentation.

Software Engineering

Software engineering involves a systematic approach to the development, operation, and maintenance of software. An integral part of this process are Version Control Systems (VCS): essential tools in software development, used for tracking and managing changes to the codebase. The main selling points are to allow several developers to work on a project, provide a history of changes, and help to resolve conflicts when merging contributions from different sources. Despite this being a solo project, a VCS such as Git is still an indispensable tool for managing the codebase.

One of the key features of Git is branching and merging. Branches can be created to work on features or debugging separate from the main codebase, which then can be merged onto the main branch when they are finalised. Git also has a unique staging area where changes can be formatted and reviewed before completing a commit (a snapshot of the staged changes). Furthermore, given Git's distributed nature, each working copy of the codebase is effectively a full backup. Therefore, this minimizes the risk of data loss.

There are several benefits of working with Git. Git helps to track iterations of the application, which includes the implementation of various features like offline functionality, user messaging, and content moderation. Thanks to branches, it allows experimentation with new features or algorithms, such as different content moderation implementations, without affecting the main codebase.

In software engineering, Git stands out as an essential tool because of its robust version control capabilities, combined with features like branching, merging, and distributed development, making it indispensable for effective project management. Git not only streamlines the development process but also ensures a high level of code integrity.

For software engineering to be effective, testing needs to be approached meticulously. The first approach taken was unit testing with Jest. Jest is a JavaScript testing framework and, as it suggests, is a framework used to ensure the correctness of the JavaScript codebase. The purpose of unit testing is to test individual components of software.

Each function/component, like for instance service worker registration, map data processing, or user message handling, is tested in isolation to ensure that it behaves as expected. Jest provides support for mocking dependencies and testing asynchronous code, which is essential for simulating network responses and user interactions. One example of this would be testing for response handling. As development continues, integration testing is necessary to test entire software modules as features such as human moderation are finalized.

Another essential type of testing is end-to-end (E2E) testing. E2E testing simply means to test the entire application. This is done by replicating user behavior, which is possible via tools such as Selenium or Cypress. For example, automating a typical workflow of opening the app, loading the map, adding a message, and viewing it later.

At the present moment, there is not enough functionality to perform integration and E2E testing, however some unit testing was done with Jest. The following piece of code snippet demonstrates testing specific functions related to processing and displaying OSM data with Jest.

```

const { readOSMFile, parseAndDisplayData, displayOSMElements } =
require("../static/myapp/raw-data-view.js")
beforeEach(() => {
  document.body.innerHTML = `
    <input type="file" id="fileInput">
    <ul id="osmDataList"></ul>
  `;
});
global.FileReader = jest.fn(() => ({
  readAsText: jest.fn(),
  onload: jest.fn(),
  result:
'<osm><node></node><way></way><relation></relation></osm>'
}));

describe('readOSMFile', () => {
  it('reads file and processes data', () => {
    const mockFile = new Blob(['mock file data'], { type:
'text/plain' });
    mockFile.slice = jest.fn().mockImplementation(() =>
mockFile);

    readOSMFile(mockFile);

    expect(FileReader).toHaveBeenCalled();
    expect(mockFile.slice).toHaveBeenCalledWith(0, 1024);

  });
});

describe('parseAndDisplayData', () => {
  it('parses data and adds elements to the DOM', () => {
    const mockData =
'<osm><node></node><way></way><relation></relation></osm>';
    parseAndDisplayData(mockData);

    const listItems = document.querySelectorAll('#osmDataList
li');
    expect(listItems.length).toBe(3);

  });
});

describe('displayOSMElements', () => {
  it('creates and appends list items for each element', () => {
    const mockElements = [document.createElement('node'),
document.createElement('way')];
    displayOSMElements(mockElements);

    const listItems = document.querySelectorAll('#osmDataList
li');
    expect(listItems.length).toBe(2);

  });
});

```

The tests use Jest to mock the DOM environment and the **FileReader** object. This allows testing interactions with web page elements and file inputs handling without relying on a browser environment.

This test for **readOSMFile** checks whether it correctly initiates reading a file using **FileReader**. It ensures that file reading is started with the expected parameters.

The next test verifies that the `parseAndDisplayData` function correctly parses provided data and updates the DOM. It also checks if the correct number of elements is added to the DOM based on the input data.

Lastly, the test for `displayOSMElements` ensures that it creates and appends the correct number of list items to the DOM for each OSM element.

This code demonstrates the validation of key functionalities such as handling OSM data. Comprehensive unit testing is integral to the software engineering process, ensuring the application meets its functional requirements. This approach to testing is vital in maintaining a high standard of software quality, particularly in complex applications that require reliable offline functionality.

Basic web page development in HTML5: HTML, JavaScript, CSS

This report will focus on the fundamental role played by HTML, JavaScript, and CSS in the development of web pages, in the context of this project and how the integration of these technologies can be used to create an interactive mapping application.

HTML (HyperText Markup Language) is used to form the structural foundation of web pages on the World Wide Web. HTML5 refers to a new version of HTML with additional new functionalities such as Web Storage, canvas and support for audio and video.

CSS (Cascading Style Sheets), as suggested by the name, are used to style the HTML components of the application. CSS plays an important role in creating an intuitive user interface, as well as making it visually appealing. As mentioned before, while currently there is a minor use of CSS, more extensive use of it will be added to the application in the future, as the current focus is to implement and showcase functionality.

Javascript is responsible for handling user interactions, which would be for example clicking a location to leave a message or retrieving user location data. Javascript is essential for enabling the offline functionality of the application, in addition with Service Workers and IndexedDB (this will be discussed later).

All these components comprise the frontend aspect of web development, which is essentially the development of the user interface and user experience. HTML provides structure, CSS adds style and layout, and JavaScript introduces interactive and dynamic elements. Below is a snippet of code that shows a basic map page using Leaflet.js, which will be used as a foundation for this project.

```
<!DOCTYPE html>
<html>
  <head>
    <link                                rel="stylesheet"
href="https://unpkg.com/leaflet/dist/leaflet.css" />
    <script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
  </head>
  <body>
    <div id="map" style="width: 1200px; height: 800px;"></div>
    <script>
      var map = L.map('map').setView([-90, 0], 3); // Latitude -
90, Longitude 0, Zoom level 3

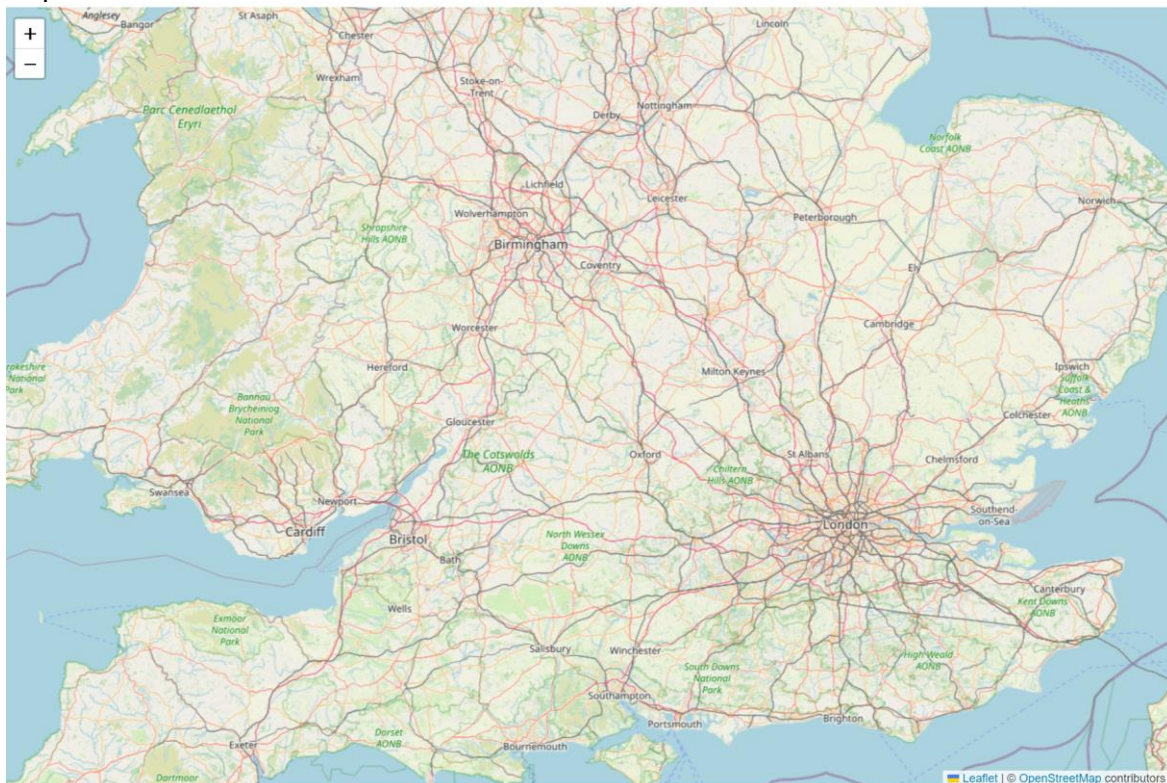
      L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution:      '&copy;
                                contributors'
                                <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
      }).addTo(map);
    </script>
  </body>
</html>
```

Firstly, **<!DOCTYPE html>** is a declaration that defines the document type and version of HTML. Here, it specifies HTML5. **<html>** is the root element of an HTML page, **<head>** contains meta-data about the document. In this case, it's linking to external Leaflet stylesheet and JavaScript files. The stylesheet is used to apply the default styles of the map and the JavaScript files provides mapping functionalities.

`<div id="map" style="width: 1200px; height: 800px;"></div>` is a **div** element, which acts as a container for the map. It contains the attribute **id** and is used by Leaflet.js to render the map. The inline CSS **width** and **height** determine its size displayed. The `<script>` tag here is used to embed JavaScript code. Essentially the code initializes a new map in the div with the id “map” and sets its geographical centre. Additionally, a tile layer is added to the map using an URL template for the tile images. Obviously, we are using OSM. Lastly, attribution specifies the attribution text to be shown on the map, which is legally required when using OSM.

To test the map, the **map.html** file was loaded on the local server and the map’s rendering and responsiveness was evaluated. Basic user interactions such as zooming and panning were tested to ensure smooth and responsive behavior. Browser developer tools were used to monitor any potential errors or warnings that could affect performance or user experience.

As shown by the sample code, the synergy of HTML, CSS, and JavaScript is evident in web development. HTML provides the structure, CSS adds style and layout, and JavaScript introduces interactive and dynamic elements. This basic web page shows the capabilities of Leaflet.js in conjunction with HTML, CSS, and JavaScript. Below is an image of the resulting map.



Advanced technologies: jQuery, HTML5 canvas

This report aims to explore the functionalities and applications of these advanced technologies, their importance in modern web development, using examples from the project as a case study and potential integrations.

jQuery, which was introduced in 2009, is a JavaScript library with a reputation for being fast and concise (hence the expression “write less, do more”). jQuery simplifies many aspects of JavaScript such as DOM traversal and Manipulation, Event Handling, Asynchronous JavaScript and XML(Ajax), etc. For example, a complex JavaScript function to manipulate Document Object Model (DOM) can be done in a single line of code of jQuery. This therefore makes the code more readable and maintainable.

While at the time of writing the project contains no jQuery, there are many ways in which jQuery can be incorporated into the project. Most importantly, it can simplify DOM manipulations, manage event listener for better user interactions, handle asynchronous HTTP requests to efficiently load map data, etc.

The HTML5 canvas element is a tool used to draw graphics via scripting (e.g., JavaScript). It can draw anything from simple graphics to complex interactive visuals. Canvas can draw shapes, text, images and even animations. Hence, it has numerous uses, notably in game development, interactive graphs, etc. Another HTML5 technology that should be mentioned is Scalable Vector Graphics (SVG). Unlike canvas, which draws in raster-based image formats (i.e. pixel bitmap), SVG is vector-based. The differences between these two technologies, concerning vector and image-based image formats, and whether SVG or canvas should be used for mapping will be discussed further in the following report: Open Street Map data representation and vector vs. image tile maps.

Canvas can be used to render custom map overlays, draw interactive elements on the map, or even create dynamic user data visualizations. Its capacity to handle complex graphics makes it an ideal choice for adding sophisticated visual elements to the map. When using API to render maps, the choice between SVG or canvas is up to the renderer. In the case of Leaflet.js, this choice is dependent on browser support.

jQuery and HTML5 Canvas have had crucial roles in modern web development. While jQuery simplifies and streamlines writing code in JavaScript, HTML5 canvas creates many possibilities for graphical creations directly in the browser. Their integration into projects such as Offline HTML5 Maps Application is evident. The following piece of code is for the following proof of concept: an application for drawing shapes with HTML5 canvas.

```
<!DOCTYPE html>
{% load static %}
<html>
<head>Canvas</head>
<body>
  <h1>Drawing shapes</h1>
  <canvas id="drawingCanvas" width="400" height="400" style="border:
2px solid black"></canvas>
  <div>
    <label for="shape">Choose a shape:</label>
    <select id="shape">
      <option value="rectangle">Rectangle</option>
      <option value="circle">Circle</option>
      <option value="line">Line</option>
    </select>
  </div>
```

```

        <button id="clearCanvas">Clear Canvas</button>
        <script src="{% static 'myapp/canvas.js' %}"></script>
    </body>
</html>
const canvas = document.getElementById("drawingCanvas");
const context = canvas.getContext("2d");
const shapeSelect = document.getElementById("shape");
const clearButton = document.getElementById("clearCanvas");
let isDrawing = false;
let startX, startY;

canvas.addEventListener("mousedown", function (e) {
    isDrawing = true;
    startX = e.clientX - canvas.getBoundingClientRect().left;
    startY = e.clientY - canvas.getBoundingClientRect().top;
});

canvas.addEventListener("mouseup", function (e) {
    isDrawing = false;
    startX = null;
    startY = null;
});

canvas.addEventListener("mousemove", function (e) {
    if (!isDrawing) return;
    endX = e.clientX - canvas.getBoundingClientRect().left;
    endY = e.clientY - canvas.getBoundingClientRect().top;
    const selectedShape = shapeSelect.value;
    switch(selectedShape) {
        case "rectangle":
            context.fillRect(startX, startY, endX - startX, endY
- startY);
            break;
        case "circle":
            const radius = Math.sqrt(Math.pow(endX - startX, 2) +
Math.pow(endY - startY, 2));
            context.beginPath();
            context.arc(startX, startY, radius, 0, 2*Math.PI);
            context.fill();
            break;
        case "line":
            context.beginPath();
            context.moveTo(startX, startY);
            context.lineTo(endX, endY);
            context.stroke();
            break;
    }
});

clearButton.addEventListener("click", function () {
    context.clearRect(0, 0, canvas.width, canvas.height);
});

```

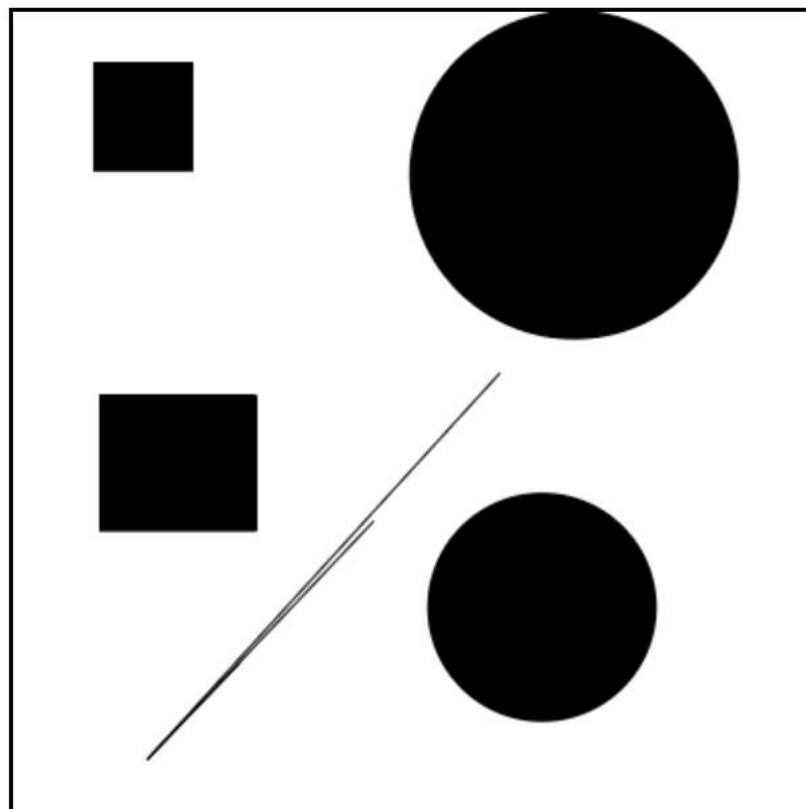
This is a basic application that leverages canvas to draw rectangles, circles, and lines. The **<canvas>** tag creates an area in the page in which drawings can be done. The **id** attribute is used for referencing the canvas in JavaScript, and the inline CSS styles define the appearance of the canvas border area. A **<select>** dropdown is provided for the user to choose the shape to draw. This interaction alters the shape that is drawn on the canvas. This element is in a **<div>**, which is a container for HTML elements (which can also be styled with CSS or manipulated with JavaScript). The last element of the user interface is a **<button>** to clear the canvas.

Moving onto the JavaScript code, the canvas element is accessed by using the HTML DOM method `getElementById()`. We first get the 2D rendering context and drawing functions and then we add event listeners (essentially a JavaScript procedure that waits for an event to occur) for **mousedown**, **mouseup**, and **mousemove**. **mousedown** initiates the drawing process, capturing the starting coordinates, **mouseup** marks the end of the drawing action, and **mousemove** is where the actual drawing happens, according to the selected shape and current mouse position.

The program logic involves taking the starting and ending coordinates of the drawing. The rectangle is filled with the **fillRect** method, requiring start coordinates, width, and height. For the circle, the **arc** method is used to draw a circle, with center coordinates, radius, start angle, and end angle. Lastly, to draw a line, we use the **moveTo** and **lineTo** methods within a path (**beginPath** and **stroke**) from the start to the end point. Finally, the **clearRect** method clears the entire canvas when the clear button is clicked, effectively resetting the drawing area. The following image shows several shapes drawn on the canvas.

Canvas

Drawing shapes



Developing an offline HTML5 application: Web Storage, Indexed DB, Service Workers and the Cache API

Most web applications assume online connectivity. However, offline functionality is required for some purposes (e.g. reading books, listening to music on an airplane). It can also be used to enhance user experience and is a feature for many popular applications such as Google Drive and Amazon Kindle. HTML5 introduces technologies that make offline functionality possible: Web Storage, IndexedDB, Service Workers, and the Cache API.

Web Storage allows for larger storage capabilities when compared to traditional cookies. There are two main mechanisms that developers can leverage: local storage and session storage. Local storage is designed for long-term data storage, without expiration date. Data stored in local storage persists across browser sessions, meaning it remains available even after the browser is closed and reopened. Session storage is limited to a single session. The data stored is cleared when the tab or the browser is closed, hence making it ideal for storing temporary data. Some uses of Web Storage are saving user preferences, remembering cart contents, etc.

IndexedDB is also used for offline data storage and a NoSQL storage system. NoSQL offers more flexibility in data structures, which will be beneficial for this project. Compared to Web Storage, it can also handle complex data structures as well as even larger data sets and complex queries. For these reasons, IndexedDB was chosen over Web Storage due to the complexity of data that will be locally stored in the project.

Service Workers have a major role in enabling the offline functionality of web applications. These JavaScript workers act as proxy between the web browser and the web server, capable of intercepting and handling network requests, caching or retrieving resources from the cache in a granular way (giving control over how the application behaves when there is no connection). For instance, on a first visit, the service worker can cache the necessary assets (HTML, CSS, Javascript files) so that the application is still usable without an internet connection.

The Cache API provides a programmable cache of Request/Response object pairs. It is specifically used to store network requests and their corresponding responses. It is commonly used alongside Service Workers (although not necessarily required) to cache important assets. During the service worker's install event, these assets can be cached, and during subsequent access, the service worker can check the cache first for these resources, falling back to the network in the case that resources are not in the cache. This is essential in order to achieve robust offline functionality.

The two following pieces of code demonstrate these technologies: a “Hello World” offline HTML5 application and a “to do list” application with IndexedDB.

```
<!DOCTYPE html>
{% load static %}
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This is a simple HTML5 offline application</p>
    <script>
```

```

        if ('serviceWorker' in navigator) {
            navigator.serviceWorker.register('{% static
"myapp/service-worker.js" %}', {scope: '/static/myapp'})
            .then(function(registration) {
                console.log('Service Worker registered with
scope:', registration.scope);
            })
            .catch(function(error) {
                console.error('Service Worker registration
failed:', error);
            });
        }
    </script>
</body>
</html>

```

This piece of code demonstrates an important step of enabling offline functionality. Firstly, Service Worker registration is achieved by first checking for Service Worker support in the browser (**'serviceWorker' in navigator**). If supported, it proceeds to register it using **navigator.serviceWorker.register**. The **{% static "myapp/service-worker.js" %}** is a template language (specific to Django) that dynamically sets the path to the static service worker file. This ensures the Service Worker is correctly located and loaded. The **scope: '/static/myapp'** parameter in the **register** method defines the scope of the Service Worker. It is set to the root of the domain, meaning the Service Worker will have control over the entire scope of the application. Lastly, the **then** and **catch** callbacks handles the result of the registration process. A successful registration logs the scope of the Service Worker, while a failure logs an error. We will then explore the code for the Service Worker, which demonstrates caching assets and handling network requests to enable offline access.

```

self.addEventListener('install', function(event) {
    event.waitUntil(
        caches.open('hello-world-cache').then(function(cache) {
            return cache.addAll([
                '/',
                '/hello-world'
            ]);
        })
    );
});

self.addEventListener('fetch', function(event) {
    event.respondWith(
        caches.match(request).then(function(response) {
            return response || fetch(event.request);
        })
    );
});

```

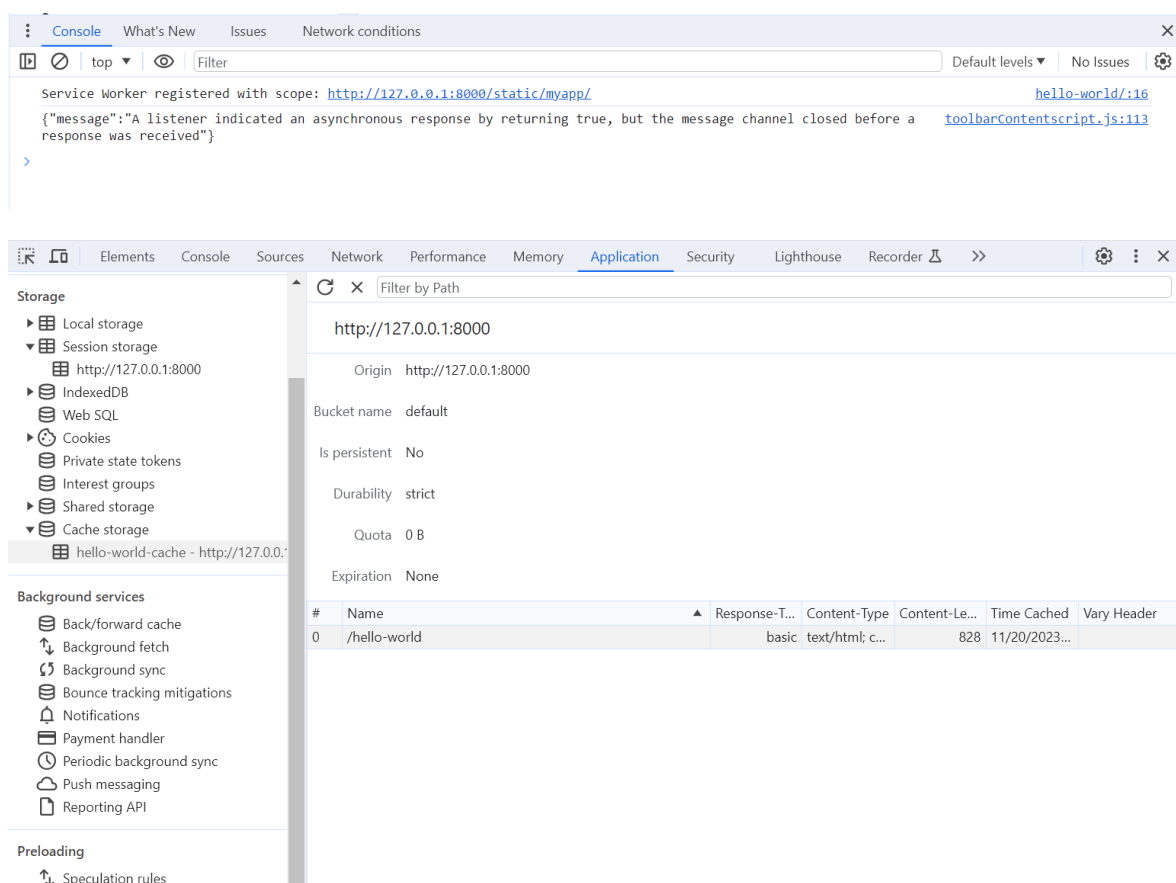
The first event listener listens for the 'install' event, which is fired when the Service Worker is being installed. **event.waitUntil(...)** makes sure that the Service Worker doesn't move on from the installing phase until the code inside is executed. Then a cache named 'hello-world-cache' is opened. If it doesn't exist, it's created. Lastly, **cache.addAll([...])** adds an array of URLs to the cache. Here, the root ('/') and '/hello-world' are cached when the Service Worker is installed, making them available for offline use.

The second event listener listens for the 'fetch' event, triggered every time a resource controlled by the Service Worker is fetched. **event.respondWith(...)** tells the browser to respond to the fetch event with the result of the code inside. **caches.match(request).then(...)** tries to find a match for the request in the cache. If a cached response is found, it's returned; otherwise, a network fetch is performed. This allows the application to serve cached resources when there is no internet connection.

While this proof-of-concept program only caches the root and helloWorld.html, it can be extended in cache CSS, JavaScript files and even map tiles. While the web page appears unremarkable, the next images will show the use of developer tools to demonstrate offline functionality.

Hello World!

This is a simple HTML5 offline application



As seen above, the Service Worker was successfully registered and the ‘/hello-world’ page cached so that it can be accessed offline. The next piece of code shows a “to do” list application using IndexedDB.

```
<!DOCTYPE html>
{% load static %}
<html>
<head>
    <title>To Do List</title>
</head>
<body>
    <h1>To-Do List</h1>
    <input type="text" id="task" placeholder="Enter task:">
    <button id="addTask">Add Task</button>
```



```

    <ul id="taskList">
      <li></li>
    </ul>
    <script src="{% static 'myapp/to-do-list.js' %}"></script>
  </body>
</html>

const request = indexedDB.open("ToDoListDB", 1);
request.onupgradeneeded = function (event) {
  const db = event.target.result;
  db.createObjectStore("tasks", {keypath: "id", autoIncrement: true});
};

request.onsuccess = function (event) {
  const db = event.target.result;
  const addTaskButton = document.getElementById("addTask");
  const taskInput = document.getElementById("task");
  const taskList = document.getElementById("taskList");

  addTaskButton.addEventListener("click", function() {
    const taskText = taskInput.value.trim();
    if (taskText) {
      const transaction = db.transaction(["tasks"], "readwrite");
      const store = transaction.objectStore("tasks");
      const task = {text: taskText}
      const addRequest = store.add(task);
      addRequest.onsuccess = function (event) {
        taskInput.value = "";
        displayTasks(store, taskList);
      };
      addRequest.onerror = function (event) {
        console.error("Error adding task:", event.target.error);
      };
    } else {
      console.log("Task text is empty or invalid.");
    }
  });

  function displayTasks(store, list) {
    const request = store.getAll();
    request.onsuccess = function (event) {
      list.innerHTML = "";
      const tasks = event.target.result;
      tasks.forEach(function (task) {
        const li = document.createElement("li");
        li.textContent = task.text;
        list.appendChild(li);
      });
    }
  }

  displayTasks(db.transaction(["tasks"]).objectStore("tasks"),
taskList);
};

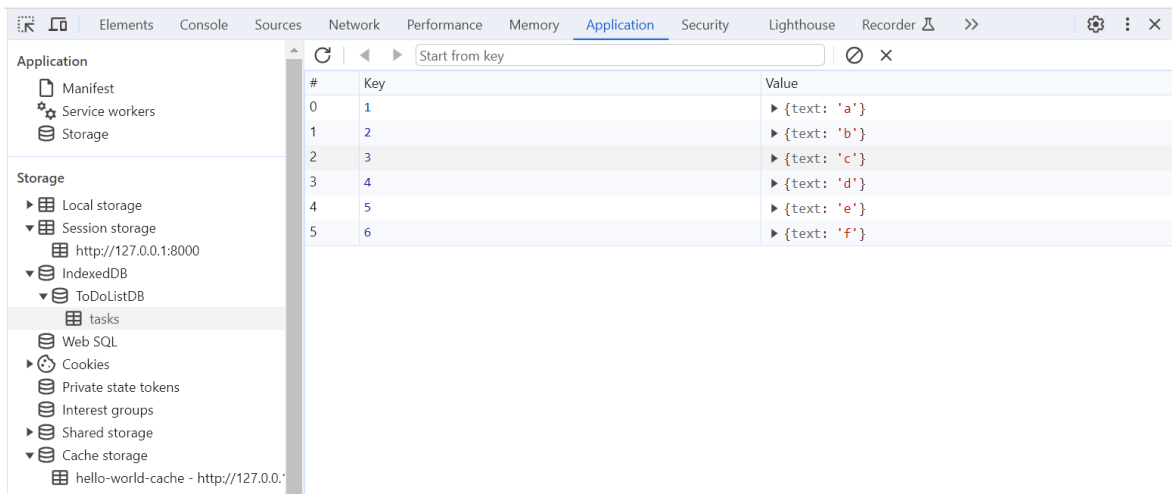
```

const request = indexedDB.open("ToDoListDB", 1) requests to open a database named "ToDoListDB". If it doesn't exist, it's created. **request.onupgradeneeded = function (event) { ... }** is triggered if the database is newly created or a database with a bigger version number is loaded. It's used to set up the database schema. **db.createObjectStore("tasks", {keypath: "id", autoIncrement: true});** creates an object store, which is similar to a table in relational databases, named "tasks" with an auto-incrementing "id" key. The event listener is added on the "Add Task" button to respond to clicks. The task text is retrieved, and a database transaction is initiated to add

the task to the "tasks" store. If successful, **displayTasks** is called to update the UI. This is done by fetching all tasks from the IndexedDB and updating the **taskList** element. As seen here, the tasks are successfully added to the list and stored in the database.

To-Do List

- a
- b
- c
- d
- e
- f



The screenshot shows the Chrome DevTools Application tab. On the left, the 'Storage' section is expanded, showing 'IndexedDB' > 'ToDoListDB' > 'tasks'. The main pane displays a table of data stored in the database.

#	Key	Value
0	1	{text: 'a'}
1	2	{text: 'b'}
2	3	{text: 'c'}
3	4	{text: 'd'}
4	5	{text: 'e'}
5	6	{text: 'f'}

Web Storage and IndexedDB provide robust options for data storage, while Service Workers and the Cache API are important for managing and caching resources for offline use. This has been demonstrated by these proofs of concepts. There is much potential for these technologies in this project, as Service Workers can enable offline access to maps and related data, while IndexedDB can be used to store user-generated content such as marked locations or messages.

Open Street Map data representation and vector vs. image tile maps

OpenStreetMap (OSM), is a collaborative project to create an open and free editable database of geographic data, and is widely used in applications such as Pokémon Go, Amazon, Facebook, Snapchat, etc. This report will discuss two methods of representing such data in web applications: vector tile maps and image (raster) tile maps. We will explore technical aspects, advantages, and limitations of each method to understand their applications in modern mapping applications and Leaflet.js in for this project.

OSM is represented as topological data structures, which consists of nodes, ways, relations, and tags. Nodes are essentially coordinates of latitude and longitude (can be used to represent points of interest such as the Eiffel Tower). Ways are connected ordered lists of nodes and can represent lines (e.g., roads, rivers) and areas (e.g., buildings, lakes). Relations are ordered lists of nodes, ways and relations. As the name suggests, relations specify relationships between existing nodes and ways such as specifying turn restrictions at intersections. Lastly, tags are key-value pairs that store metadata attached to map objects (names, type of building, material, etc.).

Vector tiles contain geometric shapes that are represented as vectors (coordinates and paths). They encode map data like points (for locations or points of interests), lines (for streets), and polygons (for areas). When it comes to rendering, it occurs on the client-side; the browser interprets the vector data and renders it into a map. Because this is done in real-time, users can change styles, filter data, and interact with individual map elements without having to send new requests to the server. Some of the advantages of this approach are being able to maintain high detail at various zoom levels without an increase in file size, making them more efficient. As mentioned before, there are vast possibilities for customization on the client-side, allowing for interactive features. However, vector tiles require more processing power to render, which is especially a problem for less powerful devices.

On the other hand, image tiles are pre-rendered bitmap images (i.e., PNG or JPEG) of map data, and is an older technology still widely used. Each tile is a static image representing a specific area at a certain zoom level. These tiles are pre-rendered and as such the browser simply displays them. Unlike vector tiles, there is no dynamic rendering/styling on the client-side. What is advantageous about image tiles is its simplicity and that less processing power is required and will be compatible with even older browsers. However, the style is fixed at the time of tile generation and higher zoom levels require loading numerous tiles, which can impact performance.

Leaflet.js is a JavaScript library. By default, Leaflet does not support vector tiles. However, there are many available plugins that can be used to display vector tiles. One of the main selling points of this library is that it requires minimal code to set up. Implementing image tiles in Leaflet.js is quite simple, involving some basic setup for map initialization and tile source specification. Using plugins such as Leaflet.VectorGrid can be crucial for the project, because of the support of dynamic aspects of the application, such as real-time updates to user messages and other interactive map elements. As a proof of concept, the following code is for a web interface allowing users to upload and parse .osm files.

```
<!DOCTYPE html>
{% load static %}
<html>
<head>
    <title>OSM Data Viewer</title>
</head>
<body>
    <h1>OSM Data Viewer</h1>
```

```

    <input type="file" id="fileInput" accept=".osm">
    <ul id="osmDataList"></ul>
    <script src = "{% static 'myapp/raw-data-view.js' %}"></script>
</body>
</html>

document.addEventListener('DOMContentLoaded', (event) => {
  const fileInput = document.getElementById("fileInput");
  const osmDataList = document.getElementById("osmDataList");

  fileInput.addEventListener("change", function (event) {
    const file = event.target.files[0];
    readOSMFile(file);
  });
});

function readOSMFile(file) {
  const reader = new FileReader();
  const chunkSize = 1024;
  let offset = 0;

  reader.onload = function (e) {
    const data = e.target.result;
    parseAndDisplayData(data);

    offset += chunkSize;
    if (offset < file.size) {
      const nextChunk = file.slice(offset, offset + chunkSize);
      reader.readAsText(new Blob([nextChunk]));
    }
  };

  const initialChunk = file.slice(0, chunkSize);
  reader.readAsText(new Blob([initialChunk]));
}

function parseAndDisplayData(data) {
  const parser = new DOMParser();
  const xmlDoc = parser.parseFromString(data, "text/xml");
  const osmElements = xmlDoc.querySelectorAll("node, way, relation");
  displayOSMElements(osmElements);
}

function displayOSMElements(elements) {
  elements.forEach(element => {
    const li = document.createElement("li");
    const text = document.createTextNode(element.outerHTML);
    li.appendChild(text);
    osmDataList.appendChild(li);
  });
}

module.exports = {
  readOSMFile,
  parseAndDisplayData,
  displayOSMElements
};

```

<input type="file"> enables users to upload **.osm** files and the **accept** attribute restricts file selection to OSM format. The **** element serves as a container to display the parsed OSM data. Then, we get references to the HTML elements for handling file input and data display. An event listener is attached to the file input so when a file is selected, it triggers the file reading process.

The file is read in chunks of 1024 bytes to handle large files and ensure that the browser remains responsive. As each file chunk is loaded, the **onload** event of the **FileReader** object is triggered, the **DOMParser** object parses the chunk of OSM data (in XML format), extracting OSM elements. File chunks are continuously read until it is entirely processed. The following image shows OSM elements extracted from a sample of OSM data of Antarctica.

OSM Data Viewer

Choose File: antarctica-latest.osm

```
<node id="36966060" version="70" timestamp="2023-04-25T10:46:35Z" lat="-79.4063075" lon="0.3149312"><tag k="ISO3166-1" v="AQ"/><tag k="ISO3166-1:alpha2" v="AQ"/><tag k="ISO3166-1:alpha3" v="ATA"/><tag k="ISO3166-1:numeric" v="010"/><tag k="alt_name:ckb" v="کۆنۆی ی بێمەتەنگی باشۆر"/><tag k="alt_names" v="Antarctica"/><tag k="alt_name:vi" v="Nam Cúc Cháu"/><tag k="alt_name:vo" v="Sulödop"/><tag k="fixme" v="Nenetta ehavo de la etikedo j" name" k="wikipedia"/><tag k="name:ace" v="Antartika"/><tag k="name" v="Antarctica"/><tag k="name:ast" v="Antártica"/></node>
<node id="275463074" version="1" timestamp="2008-07-02T02:08:02Z" lat="-71.1055829" lon="-3.7347144"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 3 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="275487633" version="1" timestamp="2008-07-02T03:39:52Z" lat="-70.2464741" lon="-7.1753781"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 2 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="225493574" version="1" timestamp="2008-07-02T04:03:52Z" lat="-70.9624031" lon="-2.5440073"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 3 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="447076759" version="1" timestamp="2009-07-24T16:41:59Z" lat="-62.1685155" lon="-58.9743716"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 2 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="447077396" version="1" timestamp="2009-07-24T16:42:17Z" lat="-62.2104744" lon="-59.0180623"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 3 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="447082601" version="1" timestamp="2009-07-24T16:56:37Z" lat="-62.230091" lon="-58.9465707"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 2 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="447098239" version="1" timestamp="2009-07-24T17:04:03Z" lat="-62.1850562" lon="-58.8775397"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 3 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="447107035" version="1" timestamp="2022-12-13T16:54:50Z" lat="-62.195198" lon="-58.9447679"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 3 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror></node>
<node id="775823611" version="3" timestamp="2013-03-17T14:09:15Z" lat="-62.2324389" lon="-58.4720139"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 5 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror><tag k="source" v="Polish Academy of Science"/></node>
<node id="775823873" version="3" timestamp="2013-03-17T14:09:25Z" lat="-62.229679" lon="-58.4451475"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 4 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror><tag k="source" v="Polish Academy of Science"/></node>
<node id="775824256" version="3" timestamp="2013-03-17T14:09:40Z" lat="-62.220414" lon="-58.4541297"><parsererror xmlns="http://www.w3.org/1999/xhtml" style="display: block; white-space: pre; border: 2px solid #c77; padding: 0 1em 0 1em; margin: 1em; background-color: #fff; color: black"><h3><div style="font-family:monospace;font-size:12px">error on line 4 at column 3: Extra content at the end of the document </div><h3>Below is a rendering of the page up to the first error:</h3></parsererror><tag k="source" v="Polish Academy of Science"/></node>
```

There are many factors to consider when deciding whether to use image or vector tiles. Image tiles can be more straightforward to cache and use offline. However, vector tiles, due to their smaller size and scalability, might potentially offer a more efficient solution for offline storage, especially given varying levels of zoom. The functionality of interacting with location-based messages aligns more closely with vector tiles thanks to dynamic rendering. And, on the topic of performance, Image tiles offer broader compatibility, but vector tiles are generally more demanding but provide a richer user experience.

Content moderation

Content moderation is the process of monitoring, evaluating, and managing user-generated content to ensure that it adheres to certain standards or policies. The primary aim is to prevent harmful content, such as hate speech, misinformation, harassment. Like any digital platform with social features, content moderation is a crucial part of this application. The main challenge is to achieve a balance between free expression and protecting users of harmful content. There are many different approaches to content moderation, we will mainly discuss automated moderation, but we will also consider human moderation.

The simplest implementation would be keyword lists. This is essentially checking content for a list of banned keywords. This approach is easy to set up and to work with, and such lists can be updated easily (this is as simple as adding/removing words). However, keyword filtering has several limitations. It is not easy to find exhaustive lists, especially in languages other than English. Moreover, language evolves constantly, and lists must constantly be updated to keep up. Another factor that lists cannot account for is context; bad words have several meanings and might be considered acceptable in certain situations, not to mention that words have varying degrees of severity and type. Furthermore, it is entirely possible for toxic content to not contain any bad words and additionally, end-users can easily circumvent keyword filtering implementations by obfuscating. Obfuscation can be as simple as replacing characters (changing “i” to “1”) so that words can still be recognizable to more creative ways such as emojis or special characters. The possibilities are nearly endless.

Another more modern implementation involves language models trained on large, labelled datasets to understand and interpret human language. The main advantage of this approach is that it can take into account context and cultural nuances, as well as scalability. There are several tools that can be integrated into the project to achieve this, such as AWS Comprehend. Still, adapting to evolving language can be a challenge. What data is used to train the language model is relevant. If the data is biased, the language model can perpetuate such biases. As we’re using DynamoDB (which is part of the AWS portfolio), it is only natural to also use AWS Comprehend in conjunction. The intent is to first implement content moderation with keyword lists and, if progress in development allows, use AWS Comprehend.

Lastly, human moderation, as the name suggests, relies on human moderation to review, and make decisions on flagged content. An implementation would be to allow users to report messages (that have passed the automated moderation), and such messages would be available for moderators to review. Moderators would be assigned to regions to moderate messages from those regions. The most obvious advantage of human moderation is, similar to language models, the ability to understand context, sarcasm, and cultural nuances. Human moderators (that are local to the area) are also able to combat misinformation. However, like language models, human moderators can be biased, as well as being costly and time-consuming.

Currently, the goal is to filter out harmful content: hate speech, misinformation, harassment, and explicit material. The only supported language is English, as moderating for more languages can quickly lead to scope creep. Ideally, a hybrid approach should be taken, where automated moderation filters out the more obvious violations, while human moderators handle edge or more complex cases. A message would have to pass a keyword list and scrutiny from a language model, before being evaluated by a human moderator if such message is reported.

Ethics

There are various ethical issues to consider as development for the maps application progresses. These not only affect the design and functionality of this application, but also impact users and society at large. Here, we will explore the ethical aspects associated with this project.

Firstly, this application involves collecting and storing user-generated content, which may include sensitive data. Messages would be posted anonymously and are public. Of course, if such messages contain sensitive information, it is essential that users would be able to report them for containing sensitive information so that they can be removed. If user registration is to be implemented in the future, obtaining user consent for data collection, and informing about its usage is ethically necessary.

On the other hand, content moderation, while crucial for combating abuse and harmful content, can potentially infringe on freedom of expression. It is hence important to make sure that biases are not perpetuated in automated moderation through the careful choice of training data, especially when it comes to language models.

On the topic of content moderation, it is also important to highlight the human cost. Online platforms such as Twitter (now named X) contain questionable content that is being regularly uploaded. While content moderators are needed to prevent these sorts of content from being spread, moderating comes at the expense of mental health. The (possible) use of language models hopes to mitigate this problem, serving as an additional filter before human moderation comes into play.

This app bridges the digital divide by offering offline functionality, providing accessibility in low-connectivity areas. To ensure robust offline functionality, testing will be a crucial part of development.

The ability to leave location-based messages can influence local communities and businesses. These features can be misused to negatively impact them. Hence, a reporting system can help to mitigate this issue. While keyword lists are ineffective, language models can certainly be trained to recognize such messages. Human moderation is also a valid option to combat this issue.

Ethical considerations in the development of this application are crucial in ensuring that it serves users responsibly. As discussed before, there are several measures taken to address these ethical issues. By engaging with these ethical challenges, this project will be able to enhance its societal value and also foster trust and reliability among its user base.

Bibliography

- [1] Wang, J. (1999). A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5), 36-46.
- [2] MDN Web Docs. (n.d.). *Service worker API*. Web APIs | MDN.
https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
- [3] WHATWG. (2023, September). *HTML Living Standard*.
<https://html.spec.whatwg.org/multipage/>
- [4] W3C. (2023, August 8). *Indexed database API 3.0*. <https://www.w3.org/TR/IndexedDB/>
- [5] Kimak, S., & Ellman, J. (2015, December). The role of HTML5 IndexedDB, the past, present and future. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 379-383). IEEE.
- [6] Bibeault, B., De Rosa, A., & Katz, Y. (2015). *jQuery in Action*. Simon and Schuster.
- [7] Bennett, J. (2010). *OpenStreetMap*. Packt Publishing Ltd.
- [8] Netek, R., Masopust, J., Pavlicek, F., & Pechanec, V. (2020). Performance testing on vector vs. raster map tiles—comparative study on Load Metrics. *ISPRS International Journal of Geo-Information*, 9(2), 101. <https://doi.org/10.3390/ijgi9020101>
- [9] Foody, G., See, L., Fritz, S., Mooney, P., Olteanu-Raimond, A.-M., Fonte, C. C., & Antoniou, V. (Eds.). (2017). *Mapping and the Citizen Sensor*. Ubiquity Press.
<http://www.jstor.org/stable/j.ctv3t5qzc>
- [10] Gillespie, T. (2020). Content moderation, AI, and the question of scale. *Big Data & Society*, 7(2), 205395172094323. <https://doi.org/10.1177/2053951720943234>
- [11] Lawson, B., & Sharp, R. (2012). *Introducing HTML5*. Peachpit Press.
- [12] Crickard III, P. (2014). *Leaflet.js essentials*. Packt Publishing Ltd.
- [13] Arsht, A., & Etcovitch, D. (2018). The human cost of online content moderation. *Harvard Journal of Law and Technology*, 2.
- [14] Aggarwal, K. K. (2005). *Software engineering*. New Age International.

Appendix: diary and video demonstration

07/10/2023

I have written a small page to demonstrate service workers and cache. The intent is to cache the 'helloWorld.html' file to demonstrate offline functionality. Further testing is needed.

09/10/2023

I have set up an Django application to handle the remote database in the future, I am considering using DynamoDB to do this. I will further explore this once I learn more about OSM data and how to store it.

11/10/2023

I have begun experimenting and for now believe that using IndexedDB is more suitable to Web Storage as it offers a better (and much needed) scalability. Further testing is needed to reach a conclusion on which technology to use. Lastly, a bug in which service workers were not being registered has been fixed.

14/10/2023

While IndexedDB involved a more significant learning curve, I have decided to go forward with IndexedDB as scalability is necessary. I have also made a basic page that allows users to draw shapes using HTML5 canvas. I am considering using Mapbox for processing and displaying OSM data, further experimentation is needed to decide which API is appropriate for the project's requirements.

17/10/2023

The canvas page works as intended. Next up I will make a simple page to display OSM data and explore possible forms of remote database storage, processing, and local storage of such data.

18/10/2023

This page allows an .osm file to be uploaded and its contents displayed as elements (nodes) in a list. The sample used was OSM data of Antarctica. The next challenge will be to turn such data into a map, how to store this data in a remote database, and how to make it lightweight for the client to render. I will be experimenting with Mapbox to achieve this.

23/10/2023

Now that the page to display the contents of an .osm file works correctly. I will continue investigating Mapbox and possible integration with a remote and local database. So far the most notable candidates are MongoDB and DynamoDB for remote database and noSQL (as part of IndexedDB) for local database.

04/11/2023

I have a better idea of the scope of my project now. The main idea is that the application allows users to leave messages behind (based on their location). This will be the main focus of the application, and for mapping I will use high level APIs such as Leaflet.js. I will further explore how this will affect the choice of other tools used.

18/11/2023

I now have a more clear idea on how to conduct testing. While so far testing has been done manually and using developer tools, I will implement unit tests with Jest for JavaScript. Django also has built in testing, however there is no code in the backend that requires testing at the moment. In the future end-to-end and integration testing will be implemented with APIs such as Selenium.

03/11/2023

I have done some basic testing using Jest on my proofs of concepts. I now have a better picture of how to handle content moderation, which will be, first and foremost, using keyword lists. This approach has many limitations, so if development allows I intend to train a language model and implement human moderation.

Video demonstration: <https://youtu.be/MX8nTSGTuZl>