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

April 04, 2024

Table of Contents

Word count(excluding appendix, bibliography and code): 14402

1. Introduction 3

2. Technical considerations 4

3. Literature review 5

3.1 Evolution of HTML5 and Its Impact on Web Applications 5

3.2 The Role of OpenStreetMap in Web Mapping 5

3.3 Advancements in Offline Web Technology 6

3.4 Content moderation in Online Platforms: Challenges and Approaches 7

4. Software Engineering 8

4.1 Version Control and Collaboration: Best Practices 8

4.2 Modelling User Interaction: use case diagrams 8

4.3 Ensuring Quality: Testing Frameworks and Methodologies 9

4.4 Agile Development: Adapting to Changing Requirements 14

5. Basic web page development in HTML5: HTML, JavaScript, CSS 15

5.1 HTML, CSS, and JavaScript: Building the Foundation 15

6. Advanced technologies: jQuery, HTML5 canvas 17

6.1 Leveraging jQuery for Enhanced Interactivity 17

6.2 Utilizing HTML5 Canvas for Dynamic Visuals 17

7. Developing an offline HTML5 application: Web Storage, Indexed DB, Service Workers, and the Cache API 20

7.1 Data Management with IndexedDB and Web Storage 20

7.2 Enabling Offline Capabilities with Service Workers and Cache API 20

8. Open Street Map data representation and vector vs. image tile maps 26

8.1 Understanding Vector vs. Image Tile Maps 26

8.2 Integrating OSM Data with Leaflet.js 26

9 AWS technologies: DynamoDB and Comprehend 29

9.1 DynamoDB for Scalable NoSQL Storage 29

9.2 Content Moderation with AWS Comprehend 30

10. Content moderation 37

10.1 Implementing Automated Keyword Filtering 37

10.2 Advanced Content Moderation 37

11. Ethics and professional issues 39

11.1 Ethical Considerations in User Data Handling and Content Moderation 39

11.2 Navigating GDPR Compliance 40

11.3 The Impact of Offline Functionality 41

12. Conclusion 42

13. Bibliography 43

14. Appendix.. 45

14.1 Diary 45

14.2 Installation Manual 47

14.3 User Manual for interim programs 48

14.4 User Manual 49

1. Introduction

This report outlines the development of the Offline HTML5 Maps Application. The core feature of this application is to allow users to leave location-based messages offline, of which will be available for other users to see once the user connects to the internet. The technologies used are HTML/CSS, Cache, JavaScript, Service Workers, IndexedDB, Django, DynamoDB, AWS Comprehend, 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 as a first line of defence, and 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 (“Look at this apple shaped rock!”), hazard warnings (“Careful, the ground is slippery here!”), 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.

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. Of course, these proofs of concepts are not in the end product.

This project has the following features: the application’s core functionality works offline and allows the user to zoom and move around the map. Furthermore, and as mentioned before, the user is able to leave public messages. These messages are automatically moderated for inappropriate content with keyword lists and can also be reported by other users for human moderators to evaluate. With human moderation comes with a login page and a moderator page allowing moderators to see reports and remove messages that are deemed inappropriate. Additionally, should the message pass the initial filter, a language model trained with AWS Comprehend will evaluate the message. Lastly, we will also discuss the timescale and planning of this project (e.g. goals, difficulties, etc.).

2. Technical considerations

There were some difficulties regarding the implementation of the offline map. Initially, the goal was for the application to be able to display the entirety of United Kingdom and Ireland. This proved to be quite difficult as rendering vector tiles from OSM data using tools such as QGIS would take several hours. Furthermore, the vector tiles’ size was around 2 GB, which would make it unfeasible to cache all at once.

Further testing also revealed that the plugins for Leaflet.js did not support vector tiles as well as previously thought. The area coverage was significantly scaled down due to these reasons, so the end product renders Surrey using raster tiles.

Another difficulty that was encountered was staying within the limits of the AWS Free Tier. This limited the exhaustiveness of the keyword list as well as testing the classifier model from AWS Comprehend because deploying the model has an associated cost. For instance, just deploying at 1 inference unit (IU), i.e. 100 characters per second, would cost approximately $ 1.80.

Estimated hourly cost	Estimated daily cost	Estimated monthly cost
$1.80	$43.20	$1,314.00

Of course, in a real scenario, the model would have to go through more than 100 characters per second. As such, it has been decided that the model will not be deployed outside of testing and demo scenarios. An option worth exploring could have been Tensorflow and PyTorch, with the downside of expending computational resources.

As such, hosting the application with the current technology stack is unfeasible and outside the scope of the project. The main reason being costs associated with DynamoDB and Comprehend. Their cumulative costs, including services such as EC2, and S3 may add up and potentially become more expensive than traditional hosting options. Another reason is due to infrequent or unpredictable traffic patterns. AWS operates on a pay-as-you-go model, i.e. only paying for the resources consumed. This pricing model can be advantageous with consistent or predictable traffic patterns, but unpredictable traffic patterns make it challenging to optimize resource provisioning and cost management.

Another important factor is the risk of spam, which has not been accounted for in the implementation of the application. If the website becomes a target for spammers or bots, it can lead to sudden and significant spikes in traffic and resource usage, triggering AWS's auto-scaling mechanisms, resulting in the provisioning of additional resources and higher costs, not to mention security risks such as distribution of malware and DDoS attacks.

Regardless of these challenges, all proposed features and some extensions were implemented on time following the plan. However, a larger area coverage could have been achieved by optimizing and implementing more appropriate algorithms for caching, and even going as far as setting up a tile server (as OSM imposes download limits on their tile servers).

3. Literature review

3.1 Evolution of HTML5 and Its Impact on Web Applications

The evolution of HTML5 technologies has had a major role in significantly enhancing offline functionality and maps applications. Boulos (2010) and Taraldsvik (2011) both highlight the role of HTML5 in enabling the development of offline-capable, interactive maps, with the latter emphasizing the use of HTML5's geolocation specification and Web Sockets API.

There has been an emergence of the need of multiple native applications due to the worldwide increase of mobile device usage and how HTML5 promises to be a common platform for application development across different devices and operating systems. There is less need for proprietary, plug-in based rich Internet application (RIA) technologies such as the now discontinued Adobe Flash as, for instance, web browsers use HTML5 instead to play videos. Most importantly, the evolution of HTML5 has allowed for the development of geographic information system (GIS) applications that can be used in devices supporting HTML5, improving performance and decreasing latency and bandwidth. As such, the new HTML5 standard allows developers and companies to create GIS applications that can compete with desktop applications.

One of the most impactful features of HTML5 is the canvas element due to its potential applications. An interesting example of the application potential of HTML5 maps that Boulos (2010) presents is Cartagen, developed at MIT Media Lab's Design Ecology, a client-side framework for vector web mapping in native HTML5. Cartagen makes use of the Canvas element to load mapping data from various sources such as OpenStreetMap (OSM). A major feature of this framework is its ability to display maps that change based on live data streams. This means that users can see edits happening in real time with no rendering load on the server.

Furthermore, Cartagen may also enable community participation, even reaching rural areas, through SMS mapping and search. With a basic cell phone, users can produce their own maps thanks to string-based geocoding. The impact of this cannot be understated, as there are over 4 billion smartphone users worldwide [13]. SMS mapping also allows for incident reporting, crisis response, as well as reporting medical emergencies.

3.2 The Role of OpenStreetMap in Web Mapping

Possibly the most well known source of map data is OSM. The OpenStreetMap (OSM) project, established in 2004, is a significant example of Volunteered Geographic Information (VGI) that aims to create free vector geographic databases through contributions from internet users (Haklay et al., 2008). The aim of OSM is to create a set of map data that's free to use, editable, and licensed under new copyright schemes (a similar model adopted by Wikipedia). The project has attracted a large number of contributors, with over half a million members registered by 2011, although only a small percentage actively contribute to the database (Neis et al., 2012). Only 42% of the registered members carried out at least one edit in the OSM database and only 5% actively contributed to the project in a more productive way according to the results. The majority of contributors are located in Europe, with varying activity areas ranging from one soccer field to over 50 km2.

The wide adoption of OSM begs the question of data quality. This has been a key consideration, with studies focusing on elements such as geometric, attribute, semantic, and temporal accuracy, as well as completeness and lineage (Girres et al., 2010). The quality of French OpenStreetMap data is studied in this article. It gives a larger set of spatial data quality element assessments, uses different methods of quality control, and extends the work of Haklay to France. Heterogeneity of processes, scales of production, and compliance to standardized and accepted specifications are some of the questions raised by the study's outcome. Girres et al. raise an issue, that balance needs to be struck between the contributors' freedom and their respect of specifications in order to improve data quality, and the development of appropriate solutions to provide this balance. Furthermore, the rapid increase in volunteered spatial data in OSM has led to the development of real-time routing services using OSM data, with data quality in many areas matching or surpassing commercial map data (Luxen et al., 2011). This paper shows both a server and a hand-held device based implementation using OSM data. Both applications provide real-time and exact shortest path computation on continental sized networks with millions of street segments.

Overall, the OSM project has created a collaborative platform for creating and improving map data, with a growing community of contributors and developers working towards enhancing the quality and usability of the data across various applications and devices. The balance between contributors' freedom and adherence to specifications remains a challenge, highlighting the importance of ongoing research and development efforts in the field of VGI.

On the other hand, Lubbers (2010) and He (2014) further underscore the importance of HTML5 in creating offline web applications, with He specifically focusing on the detailed implementation steps of HTML5's offline function. These studies collectively demonstrate the role of HTML5 in advancing the capabilities of offline functionality and maps applications.

3.3 Advancements in Offline Web Technology

While there is an increasing importance of web applications in the network software system, this would not work if the web were offline. Fortunately, HTML5 proposes several solutions to this problem. The evolution of offline web technology began with simple cookies and progressed to more sophisticated HTML5 features. Initially, cookies were used to cache data such as user sessions, however it had a limited size of 4 KB and lacked security. Web SQL allowed data to be stored in databases, however it was deprecated in favour to IndexedDB for the same reasons. Web Storage allowed for basic key-value storage, and IndexedDB offers complex, database-like capabilities with even greater storage capacity. The Application Cache, though initially promising, faced limitations and was superseded by Service Workers. Service Workers (essentially acting as proxy servers) provide a more flexible and powerful way to handle offline functionality by facilitating the caching of application resources, enabling web apps to offer a smooth offline experience.

Speaking of Service Workers, they are at the core of Progressive Web Apps (PWAs), introduced by Google. These are a new generation of Web application designed to provide native app-like browsing experiences even when a browser is offline. They run in the background, separate from the web page, and allow for features such as push notifications and background sync. Malavolta et. al., (2017) present an empirical study that evaluates the impact of service workers on the energy efficiency of PWAs, when operating in different network conditions on two different generations of mobile devices. It concluded that Service Workers have no impact on the energy consumption of the two devices, regardless of the network conditions.

In short, developers are now able to have explicit control over caching, going as far as having the best of two worlds: HTML5 applications that can run in a browser, update when it is online, and still be usable when offline.

Map applications can greatly benefit from offline web technology that HTML5 provides, as users in rural areas can take advantage of this. Park et al. (2016) discusses the implementation of a mobile web application with geo-based data visualization processing on online and offline mode using HTML5. This study highlights the usefulness of HTML5 in combining geo-based data processing modules and mobile environments, as it does not require the downloading or installation of programs and only requires a web browser. This feature is especially important in areas with limited interned connection. The IndexedDB API within HTML5 provides offline data storage functions on mobile environments, allowing datasets (such as the base map of OSM) to be permanently stored in mobile devices.

There are still many challenges in developing offline HTML5 map applications such as data synchronization, where ensuring up to date map information is crucial and complex. Storage limitations also pose constraints due to browser storage caps. Future directions might explore advanced data compression and efficient synchronization techniques to mitigate these issues, alongside evolving HTML5 specifications to enhance offline capabilities and storage solutions, ensuring more robust and user-friendly offline map applications.

3.4 Content moderation in Online Platforms: Challenges and Approaches

Another important point of discussion is content moderation due to the social nature of this application. Content moderation is an important part of popular social media platforms. Gorwa et al. (2020) discuss the increasing use of algorithmic moderation systems by major platforms such as Facebook, YouTube, and Twitter with automated hash-matching and predictive machine learning tools to handle copyright infringement, terrorism, and toxic speech. While content moderation is capable of limiting this sort of behavior, the authors highlight the political and ethical issues associated with these systems, emphasizing their opacity, lack of accountability, and potential to exacerbate existing problems with content policy.

And on a related note, Binns et al. (2017) explore the inheritance of bias in algorithmic content moderation systems. A case study using an existing dataset of comments labelled for offence is provided in the paper. They discuss the use of machine learning classifiers trained on texts manually annotated for offense and the potential for these systems to be biased towards or against particular norms of offense. These systems need to navigate inherently contestable boundaries and are subject to the quirks of the human raters who provide the training data. The authors provide methods to measure the normative biases of algorithmic content moderation systems and discuss the ethical choices facing the implementers of such systems.

One interesting case study involves Instagram content moderation in pro-eating disorder (pro-ED) communities (a dataset containing 2.5 million posts between 2011 and 2014) presented by Chancellor et al. (2016). Despite Instagram's moderation strategies, pro-ED communities have adopted non-standard lexical variations of moderated tags to circumvent restrictions. The use of these lexical variants has led to increased participation and support of pro-ED, with the associated content expressing more toxic, self-harm, and vulnerable content. The study puts into question the effectiveness of content moderation.

Conversely, a key component of content moderation is human moderation. Seering et al. (2019) discuss the use of commercial content moderation, including moderation algorithms and professional moderators, to manage online communities. They present findings from interviews across Twitch, Reddit, and Facebook, with volunteer moderators and derive a model categorizing the ways moderators engage with their communities and contribute to community development. The study emphasizes the role of moderators in the development of meaningful communities (both with and without algorithmic support).

The discussed studies highlight the technical, political, ethical, and social implications of content moderation, including the use of algorithmic moderation systems, the effectiveness of moderation strategies, the role of moderators in community development, and the inheritance of bias in algorithmic content moderation systems. These insights help paint a broader picture of the challenges and complexities associated with content moderation in online platforms.

4. Software Engineering

4.1 Version Control and Collaboration: Best Practices

Software engineering involves a systematic approach to the development, operation, and maintenance of software. An integral part of this process is 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.

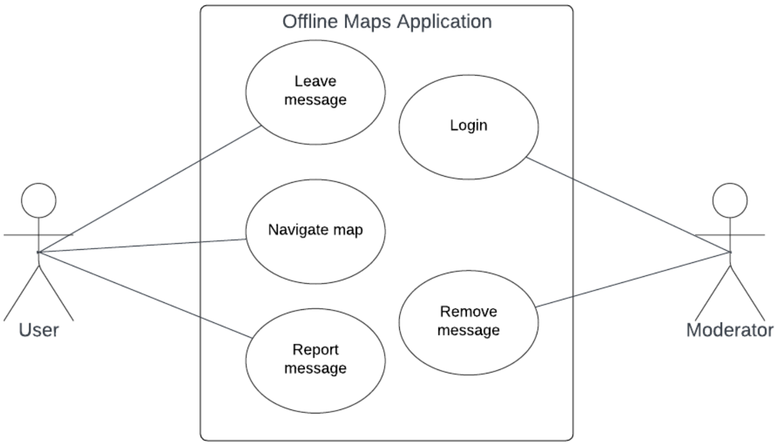
There are several benefits of working with Git. Git helps to track iterations of the application, which includes the implementation of various features like offline functionality, user messaging, and content moderation. Thanks to branches, it allows experimentation with new features or algorithms, such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

To make the most out of Git, several best practices were in use: keeping the main branch as a stable branch representing the most up-to-date working version, a new branch was created for every new feature or experiment (and merged back to the main branch when complete) and making commits frequently with meaningful messages that explain the changes made.

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.

4.2 Modelling User Interaction: use case diagrams

Another part of software engineering is the use of use case diagrams to represent the functional requirements of a system. It depicts the interactions between actors and the system itself.



The use case diagram depicts the actors User and Moderator. The use cases represent the functionality of the application: leave message, navigate map, report message, login and remove message. The User actor is associated with the "Leave message," "Navigate map," and "Report message". The Moderator actor is associated with the "Remove message" use case.

The use cases are enclosed within the boundary of "Offline Maps Application," indicating that these functionalities are part of the system under consideration.

4.3 Ensuring Quality: Testing Frameworks and Methodologies

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. 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.

While Jest is used to unit test JavaScript, Django itself provides built in unit testing for backend components. Django is a free and open-source high-level Python web framework that follows the model–template–views (MTV) architectural pattern, a variation of the Model-View-Controller (MVC) pattern. The MTV pattern has the following components:

Model: the data layer defining the structure of the database. It defines the fields and behaviors of the data you’re storing. Additionally, Django follows the DRY Principle (Don't Repeat Yourself), so it is only required to define the data schema, and it automatically provides the tools to access and manage the database. This reduces code duplication and makes the codebase easier to maintain. Models are used to perform the create, read, update, and delete (CRUD) operations.

Template: the presentation layer controlling how the data is presented to the user. Templates are essentially HTML files which allow Python-like expressions for dynamic content generation. Django's template engine provides a powerful mini language for defining the structure of your web pages, including loops, conditionals, and inheritance. This allows developers to create flexible and reusable templates. Typically, templates are responsible for defining the layout of a web page, displaying data, and presenting forms to the user.

Views: the business logic layer. It processes requests and returns responses. Views in Django are Python functions or classes that take a web request and return a web response. Views access the data through models and delegate formatting to the templates. They can execute business logic, query the database, and pass the retrieved data to templates for presentation. A view can generate HTML pages, redirect to another view, or create JSON responses for APIs.

This pattern is designed to separate data handling, user interface, and application logic, making it easier to manage and scale web applications. However, the Model component of this pattern sees little use in this application, as most of the database functionality resides in DynamoDB. To demonstrate unit testing in Django and moto for mock AWS services, this following is a code snippet testing the DynamoDB interactions.

```
@mock_aws
class KeywordTests(TestCase):

    def setUp(self):
        self.dynamodb = boto3.resource('dynamodb', region_name='eu-
west-2')
        self.table = self.dynamodb.create_table(
            TableName='keyword-list',
             KeySchema=[{'AttributeName': 'keyword-id', 'KeyType':
'HASH'}],
             AttributeDefinitions=[{'AttributeName': 'keyword-id',
'AttributeType': 'S'}],
                   ProvisionedThroughput={'ReadCapacityUnits':  1,
'WriteCapacityUnits': 1}
        )
          self.table.put_item(Item={'keyword-id':  '1',  'keyword':
'test'})
          self.table.put_item(Item={'keyword-id':  '2',  'keyword':
'example'})

             @override_settings(CACHES={'default':      {'BACKEND':
'django.core.cache.backends.locmem.LocMemCache'}})
    def test_fetch_keywords_from_dynamodb(self):
        keywords = fetch_keywords_from_dynamodb()
        self.assertTrue('test' in keywords)
        self.assertTrue('example' in keywords)

             @override_settings(CACHES={'default':      {'BACKEND':
'django.core.cache.backends.locmem.LocMemCache'}})
    def test_load_keywords_to_cache(self):
        load_keywords_to_cache()
        keywords = cache.get('moderation_keywords')
        self.assertTrue('test' in keywords)
        self.assertTrue('example' in keywords)

             @override_settings(CACHES={'default':      {'BACKEND':
'django.core.cache.backends.locmem.LocMemCache'}})
    def test_contains_moderation_keywords(self):
        load_keywords_to_cache()
          self.assertTrue(contains_moderation_keywords("This is a test"))
         self.assertFalse(contains_moderation_keywords("This is a sample text without keywords"))
```

These tests are focused on verifying the behaviour of functions that interact with a DynamoDB table **keyword-list**, which stores keywords that need to be moderated in.

The **KeywordTests** class inherits from **TestCase**, indicating that it contains test methods. It is decorated with **@mock_aws**, which mocks AWS services calls within this class, allowing the tests to run without actually interacting with AWS.

setup sets up the environment for each test method in the class. It's automatically called before each test method. This initializes a mock DynamoDB resource and creates a table named **keyword-list** with a primary key **keyword-id**. It subsequently inserts two items into the mock table, with **keyword-ids** '1' and '2', and corresponding **keywords** 'test' and 'example'.

The test methods **test_fetch_keywords_from_dynamodb, test_load_keywords_to_cache and test_contains_moderation_keywords** test the correct retrieval of keywords, loading of keywords into cache, identification of a given string containing any keyword and ensuring that the cache is populated with keywords.

@override_settings with cache settings ensures that each test method uses a clean, isolated cache instance, preventing tests from affecting each other. The local memory cache backend is used for the scope of the tests to ensure no external caching systems interfere with it.

This approach to testing shows a combination of mocking, Django's testing utilities, and application-specific logic to ensure the reliability and correctness of critical functionalities in this application, interacting with Django's caching mechanism and AWS DynamoDB.

While in theory it is possible to perform E2E testing with unit tests, it is not feasible or practical. Unit tests are not designed to interact with a fully integrated system or to mimic user interactions across the entire application flow. Regarding Jest, mocking the environment would be too complex and possibly require third party libraries. That is why Selenium was used for this task.

Selenium is an open-source automated testing framework for web applications. It enables developers and testers to write scripts that can automate web browser actions, such as clicking buttons, entering text, navigating pages, etc. A key component of Selenium is WebDriver, which provides a programming interface to create and execute test cases. It interacts directly with the browser by sending commands to the browser's driver (ChromeDriver for Chrome, GeckoDriver for Firefox, etc.) and then retrieves the results. This allows tests to be executed in a real browser environment, closely simulating how an end-user interacts with the application. Selenium supports multiple programming languages, but for the purposes of this application, Python was used and tests were ran on Chrome. This is how messaging functionality was tested:

```
class MapInteractionTest(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get("http://127.0.0.1:8000/map/")

    def test_click_outside_circle(self):
        driver = self.driver
            driver.set_network_conditions(offline=True, latency=0, throughput=0)
        WebDriverWait(driver, 15).until(
            EC.presence_of_element_located((By.ID, "map"))
        )
        map_element = driver.find_element(By.ID, "map")
        driver.implicitly_wait(5)

          marker  =  driver.find_element(By.CLASS_NAME,  "clickable-
circle")
        marker.click()

        try:
            alert = driver.switch_to.alert
           self.fail("Prompt appeared on click outside of circle.")
        except NoAlertPresentException:
            pass

    def test_click_within_circle(self):
        driver = self.driver
            driver.set_network_conditions(offline=True,  latency=0,
throughput=0)
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.ID, "map"))
        )

        map_element = driver.find_element(By.ID, "map")
        driver.implicitly_wait(5)
        action = ActionChains(driver)
        action.move_to_element_with_offset(map_element, 200, 250)
        action.click()
        action.perform()

        WebDriverWait(driver, 10).until(EC.alert_is_present())
        try:
            alert = driver.switch_to.alert
            test_message = "This test was written by Selenium!"
            alert.send_keys(test_message)
            alert.accept()
        except NoAlertPresentException:
              self.fail("Prompt  did  not  appear  on  click  inside  of
circle.")

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main()
```

MapInteractionTest inherits from **unittest.TestCase**, making it a test case defined using Python's built-in **unittest** framework.

setUp initializes the test environment; it's executed before each test method. It sets up the **webdriver.Chrome()** instance to interact with the Chrome browser and navigates to the local URL.

The method **test_click_outside_circle** Simulates a scenario where the user clicks outside a designated clickable area (circle) on the map. It sets the network conditions offline to simulate the application's behavior without an internet connection. The test waits until the map element is loaded, attempts to click on an element with the class name "**clickable-circle**", and checks for an alert dialog; if an alert appears (indicating a click inside the circle), the test fails, implying the click was supposed to be outside the clickable area.

test_click_within_circle tests the behavior when a user clicks inside the clickable area (circle) on the map. Like the first test, it sets the browser to offline mode and waits for the map element to be loaded. **ActionChains** moves the mouse to a specific offset within the map element and performs a click, simulating a click inside the circle. It finally waits for an alert dialog to appear and interacts with it by sending keys and accepting the alert. If no alert is present, the test fails.

Lastly, **tearDown** is called after each test method to clean up, closing the browser window and ending the WebDriver session.

This is an example of using Selenium WebDriver for complex web interaction tests, including handling dynamic elements, browser alerts, and simulating user actions like mouse movements and clicks.

Comprehensive unit testing, with tools such as Jest, is integral to the software engineering process, ensuring the application meets its functional requirements. This approach to testing is vital in maintaining software quality, particularly in complex applications that require reliable offline functionality. Additionally, Selenium's ability to simulate real-user interactions on a web browser makes it a powerful tool for end-to-end testing of web applications.

Another important concept related to testing is Test-Driven Development (TDD). It is a software development process that relies on the repetition of the following development cycle: first, the developer writes a failing automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

This process comes with several benefits. The first is better code quality as writing tests first helps ensure that the code is testable and maintainable. It also leads to faster development, as developers can focus on small, incremental steps, which can lead to faster development cycles.

Tests also serve as living documentation of the expected behavior of the code. With a comprehensive suite of tests, developers can confidently refactor code without introducing new bugs.

4.4 Agile Development: Adapting to Changing Requirements

Another related concept is Agile, an iterative approach to project management and software development that emphasizes flexibility, collaboration, and continuous delivery of working software. The four key values that defines the Agile Manifesto are: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan. Agile methodologies on the other hand, such as Scrum, Kanban, Lean, and Extreme Programming (XP), provide frameworks and practices for implementing the Agile principles and values.

Agile has been beneficial due to the changing requirements, especially during the early stages of the project. Agile promotes iterative and incremental development, where software is developed and delivered in small, functional increments. This allowed for frequent feedback and adjustments to the requirements, making it easier to incorporate changes. TDD posed several challenges: when requirements change significantly, existing tests need to be updated or rewritten, which can be time-consuming, especially in growing codebases, making adherence to TDD difficult. There is also a risk of over-engineering to accommodate potential future changes, leading to unnecessary complexity and decreased productivity. While TDD and Agile are not mutually exclusive, it has been more beneficial and appropriate to lean on Agile instead of a specific testing approach such as TDD.

5. 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.

5.1 HTML, CSS, and JavaScript: Building the Foundation

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.

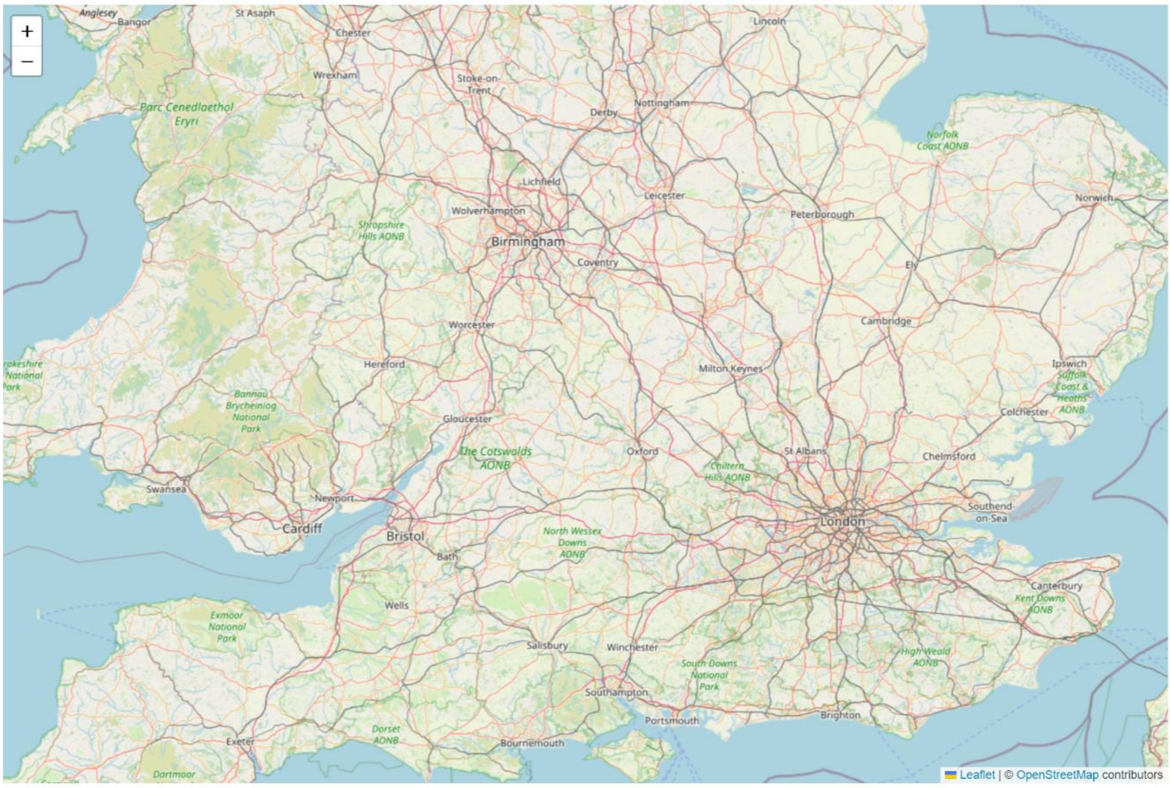
```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet"
href="https://unpkg.com/leaflet/dist/leaflet.css" />
        <script
src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
    </head>
    <body>
        <div id="map" style="width: 1200px; height: 800px;"></div>
        <script>
            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; <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
contributors'
            }).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.



Of course, this basic implementation does not have any offline functionality (this will be discussed later), as it is using OSM servers to retrieve the tiles.

6. 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 integrations.

6.1 Leveraging jQuery for Enhanced Interactivity

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) con be done in a single line of code of jQuery. This therefore makes the code more readable and maintainable.

jQuery can simplify DOM manipulations, manage event listener for better user interactions, handle asynchronous HTTP requests to efficiently load map data, etc.

6.2 Utilizing HTML5 Canvas for Dynamic Visuals

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.

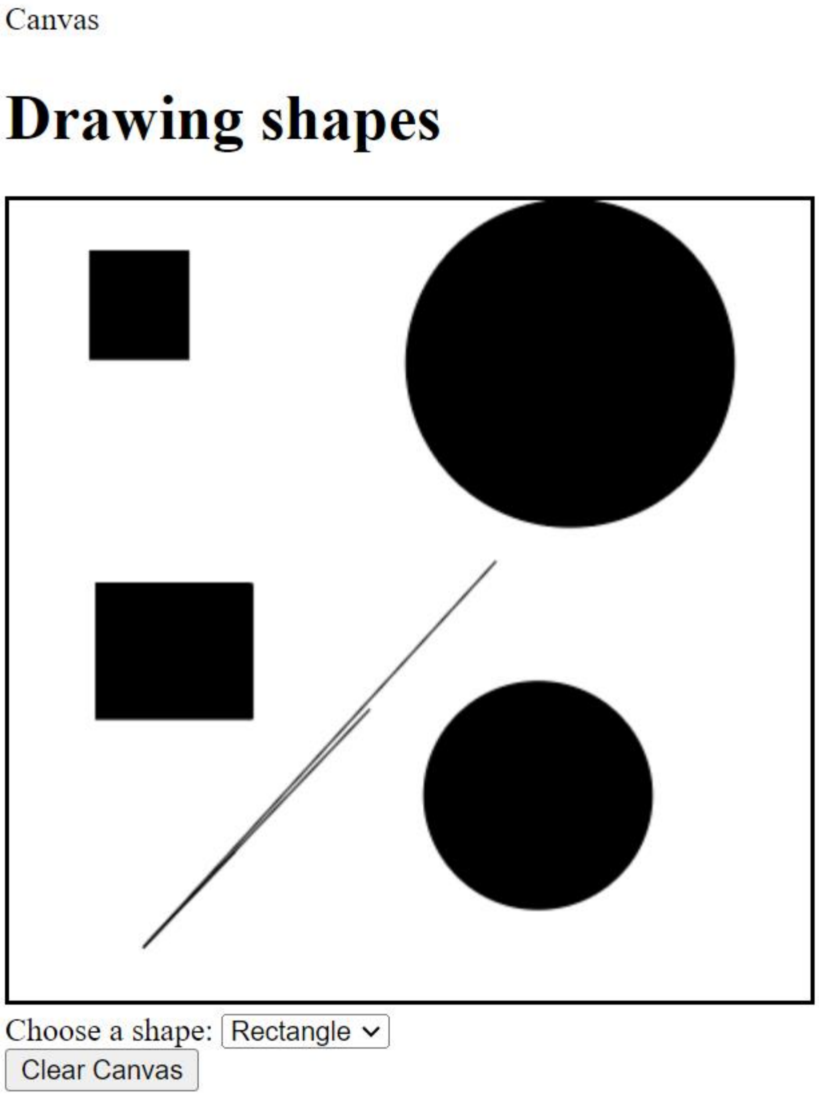
jQuery and HTML5 Canvas have had crucial roles in modern web development. While jQuery simplifies and streamlines writing code in JavaScript, HTML5 canvas creates many possibilities for graphical creations directly in the browser. The following piece of code is for the following proof of concept: an 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.



7. 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.

7.1 Data Management with IndexedDB and Web Storage

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.

7.2 Enabling Offline Capabilities with Service Workers and Cache API

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>
```

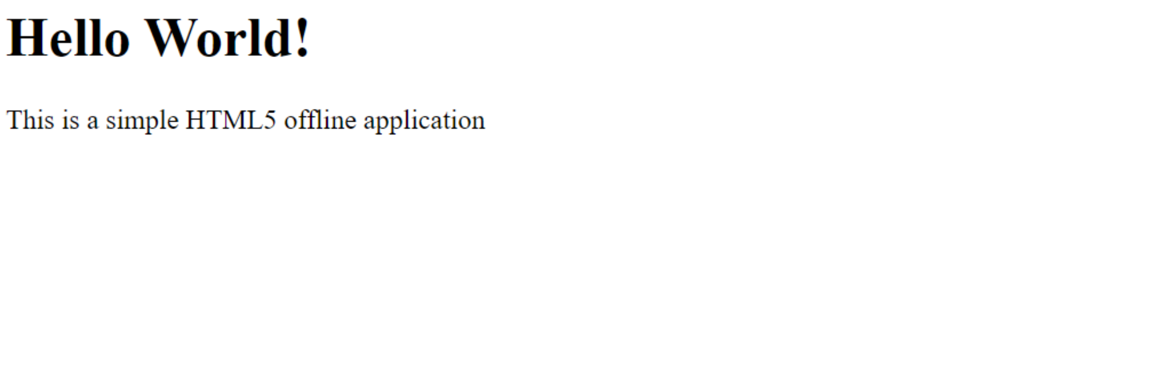
This piece of code demonstrates an important step of enabling offline functionality. Firstly, Service Worker registration is achieved by first checking for Service Worker support in the browser (**'serviceWorker' in navigator**). If supported, it proceeds to register it using **navigator.serviceWorker.register**. 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.

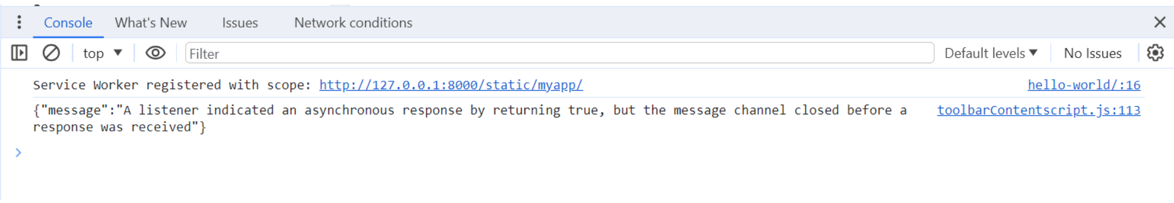
The second event listener listens for the 'fetch' event, triggered every time a resource controlled by the Service Worker is fetched. **event.respondWith(...)** tells the browser to respond to the fetch event with the result of the code inside. **caches.match(request).then(...)** tries to find a match 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.

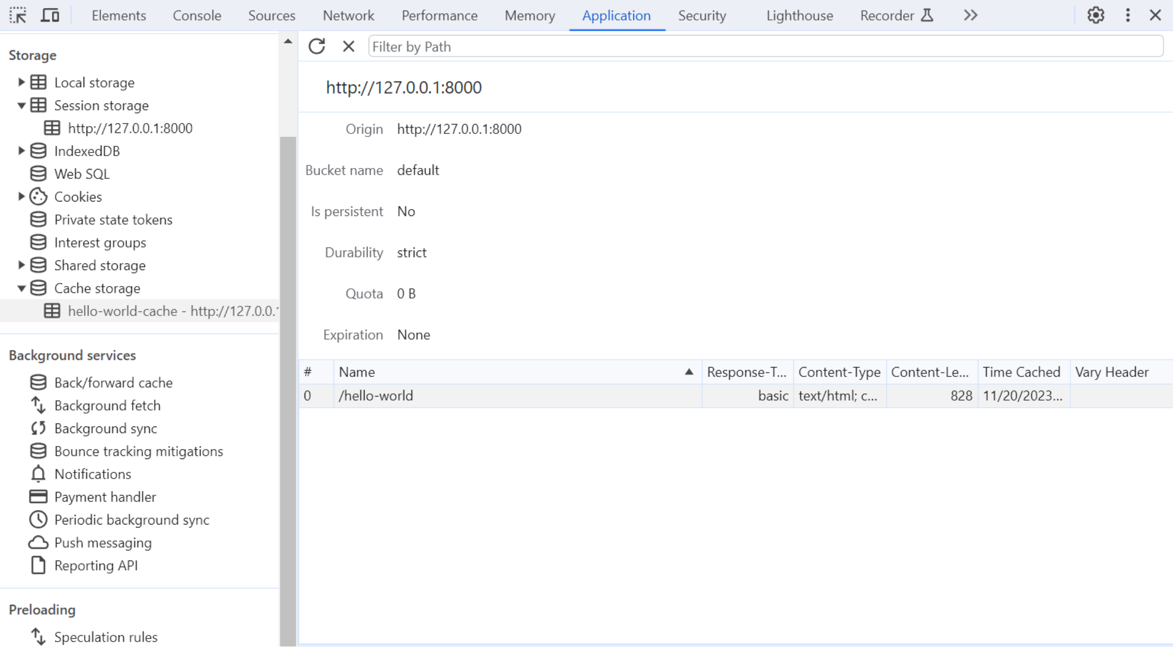


Hello World!

This is a simple HTML5 offline application



```
Service Worker registered with scope: http://127.0.0.1:8000/static/myapp/          hello-world/:16
{"message":"A listener indicated an asynchronous response by returning true, but the message channel closed before a   toolbarContentscript.js:113
response was received"}
```



http://127.0.0.1:8000

Origin http://127.0.0.1:8000

Bucket name default

Is persistent No

Durability strict

Quota 0 B

Expiration None

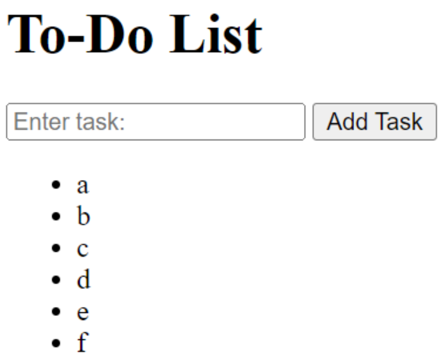
#	Name	Response-T...	Content-Type	Content-Le...	Time Cached	Vary Header
0	/hello-world	basic	text/html; c...	828	11/20/2023...	

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);
            });
        }
    }
```

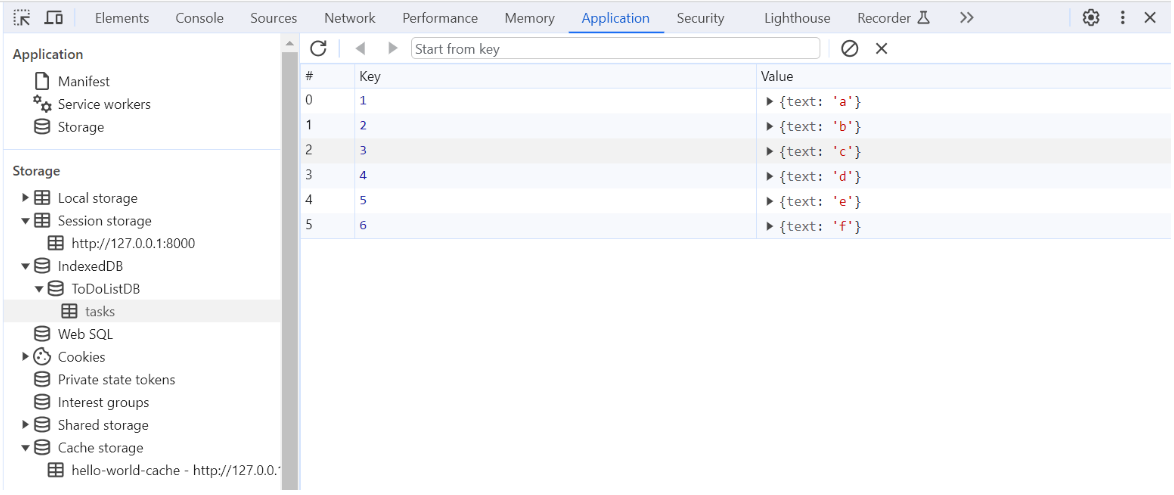
```
            displayTasks(db.transaction(["tasks"]).objectStore("tasks"),
taskList);
};
```

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



Application
- Manifest
- Service workers
- Storage

Storage
- Local storage
- Session storage
  - http://127.0.0.1:8000
- IndexedDB
  - ToDoListDB
    - tasks
- Web SQL
- Cookies
- Private state tokens
- Interest groups
- Shared storage
- Cache storage
  - hello-world-cache - http://127.0.0.

#	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.

The end product takes several approaches to ensure offline functionality. The first step is to ensure that map tiles are cached. While it is possible to download tiles directly from OSM, this approach is severely limited due to the limits imposed (as mentioned before). Therefore, the best approach would be to make use of tools such as Mapiterative and TileMill to generate tiles. The tool that was used is QGIS, an open source GIS software used for viewing, editing, printing, and analysis of geospatial data. The chosen area was Egham, Staines, and Virginia Water for zoom levels of 13 to 16. QGIS can then generate raster tiles into an XYZ directory. XYZ simply refers to the way of structuring and naming tiles in a hierarchical grid system based on x, y coordinates and zoom levels.

These files can then be put in static files which Django can serve. In Django, static files refer to the collective files (e.g. CSS, JavaScript, images) that are needed to render the web pages in a web application.

```
self.addEventListener('fetch', function(event) {
    if (event.request.url.endsWith('.png')) {
        event.respondWith(
            caches.match(event.request).then(function(response) {
                caches.match(event.request).then(function(response) {
                    return response || fetch(event.request);
                })
               return fetch(event.request).then(function(networkResponse)
{
                    caches.open('map-cache').then(function(cache) {
                       cache.put(event.request, networkResponse.clone());
                    });
                    return networkResponse;
                });
            })
        );
    } else {
        event.respondWith(
            caches.match(event.request).then(function(response) {
                return response || fetch(event.request);
            })
        );
    }
});
```

The code above shows how a Service Worker checks if the request URL ends with '.png', indicating an image resource (i.e. a tile). For these requests, it follows a specific caching strategy.

It first attempts to match the request in the cache with **caches.match(event.request)**. If there's a cached version (**response**), it is used immediately. If not, the code fetches the resource from the network using **fetch(event.request)**.

Once the network response is received, it opens 'map-cache' and stores the fetched resource with **cache.put(event.request, networkResponse.clone())**. The response is cloned because a response is a stream and can only be consumed once. The cloned response is stored in the cache, and the original response is returned to the browser to be used by the web page.

Another aspect of offline functionality is the messages using IndexedDB. Firstly, when loading the application, it will attempt to load the messages (latitude, longitude and text) from the local database and then from the remote database (which will be stored in the local database).

When a new message is posted, the application will first attempt to store it in the remote database, before storing it in the IndexedDB object store. There are some more steps in between involving content moderation, which will be discussed later.

8. 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.).

8.1 Understanding Vector vs. Image Tile Maps

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 in 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.

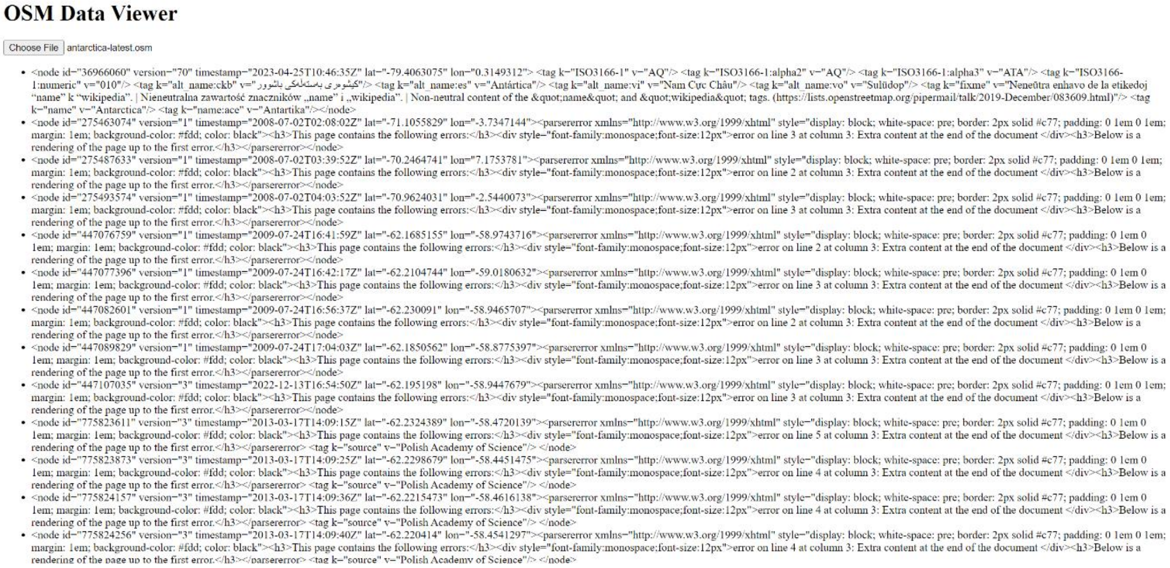
On the other hand, image tiles are pre-rendered bitmap images (i.e., PNG or JPEG) of map data, and is an older technology still widely used. Each tile is a static image representing a specific area at a certain zoom level. These tiles are pre-rendered and as such the browser simply displays them. Unlike 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.

8.2 Integrating OSM Data with Leaflet.js

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 **<ul>** 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 Antartica.

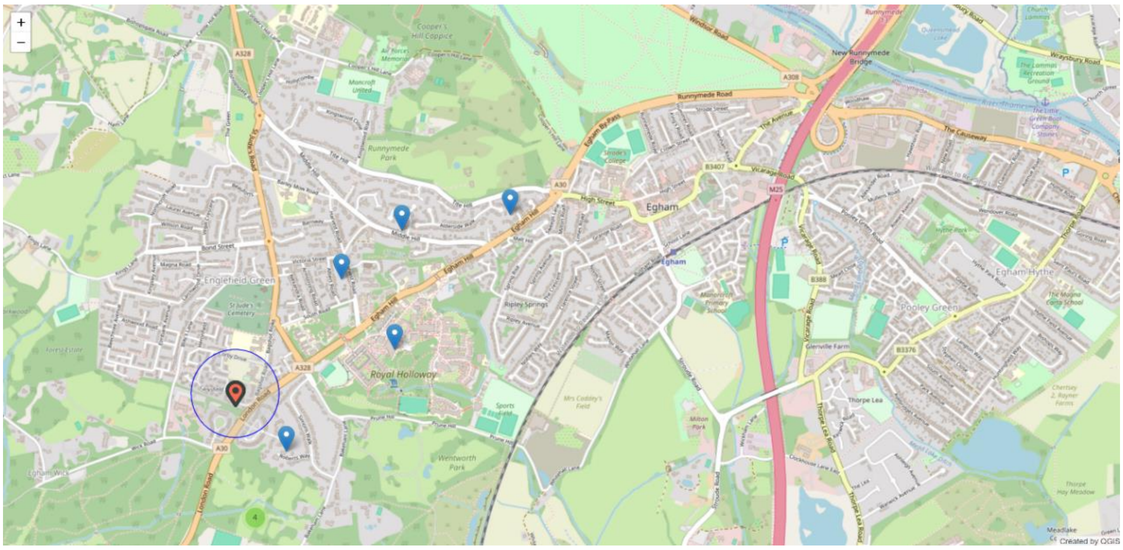


OSM Data Viewer

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="کیشوەری باستەلەکی باشوور"/> <tag k="alt_name:es" v="Antártica"/> <tag k="alt_name:vi" v="Nam Cực Châu"/> <tag k="alt_name:vo" v="Sulüdop"/> <tag k="fixme" v="Neneŭtra enhavo de la etikedoj "name" k "wikipedia". | Nieneutralna zawartość znaczników „name" i „wikipedia". | Non-neutral content of the &quot;name&quot; and &quot;wikipedia&quot; tags. (https://lists.openstreetmap.org/pipermail/talk/2019-December/083609.html)"/> <tag k="name" v="Antarctica"/> <tag k="name:ace" v="Antartika"/></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: #fdd; color: black"><h3>This page contains the following errors:</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: #fdd; color: black"><h3>This page contains the following errors:</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="275493574" 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: #fdd; color: black"><h3>This page contains the following errors:</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: #fdd; color: black"><h3>This page contains the following errors:</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="-59.0180632"><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: #fdd; color: black"><h3>This page contains the following errors:</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: #fdd; color: black"><h3>This page contains the following errors:</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="447089829" 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: #fdd; color: black"><h3>This page contains the following errors:</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="3" 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: #fdd; color: black"><h3>This page contains the following errors:</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: #fdd; color: black"><h3>This page contains the following errors:</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="-62.2298679" 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: #fdd; color: black"><h3>This page contains the following errors:</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="775824157" version="3" timestamp="2013-03-17T14:09:36Z" lat="-62.2215473" lon="-58.4616138"><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: #fdd; color: black"><h3>This page contains the following errors:</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: #fdd; color: black"><h3>This page contains the following errors:</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. The end product (image seen below) makes use of image tiles for reasons discussed before (mainly due to lack of support of vector tiles), but vector tiles are by far a more scalable solution.



9 AWS technologies: DynamoDB and Comprehend

AWS (Amazon Web Services) is a comprehensive cloud computing platform provided by, as the name suggests, Amazon. It offers a broad set of global cloud-based products and services including compute power, database storage, content delivery, and other functionality to help businesses scale and grow. AWS provides its users a scalable, reliable, and low-cost way to deploy applications and store data in the cloud without having to invest in their own on-premises infrastructure. Some of these AWS services include S3 (Simple Storage Service), IAM (Identity and Access Management), EC2 (Elastic Compute Cloud), etc.

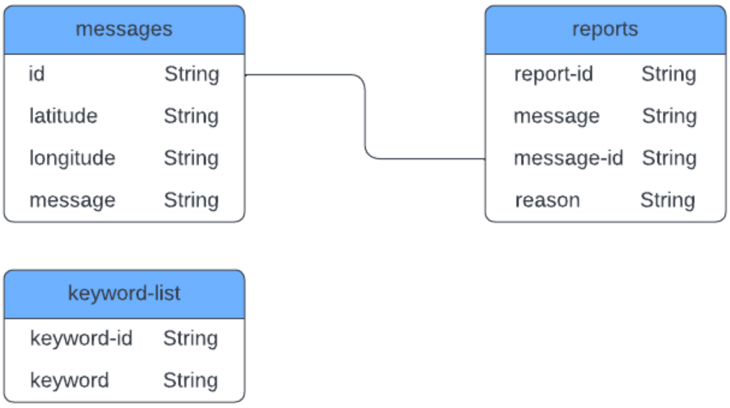
Out of these services, DynamoDB was used as the remote database, keyword list and user reports, and AWS Comprehend for content moderation using a language model.

9.1 DynamoDB for Scalable NoSQL Storage

DynamoDB is a fully managed NoSQL database service. It is a key-value and document database that delivers a high performance at any scale. It is useful for maps applications in several ways.

Thanks to its low latency queries, DynamoDB can efficiently handle the high volume of queries that a maps application would need to serve, such as fetching nearby locations or routing information. Furthermore, as the number of users and data grows, DynamoDB can scale up or down automatically to meet the demand without any sort of intervention or downtime. DynamoDB is designed for horizontal scaling, automatically distributing data and traffic across multiple partitions as needed, as such, it can handle massive amounts of data and high throughput workloads, while traditional SQL databases typically scale vertically by adding more resources (CPU, RAM, storage) to a single server.

On the other hand, noSQL databases have more limited querying capabilities, especially when working with complex data structures or relationships. Complex querying is not a necessity for this application fortunately, as shown by the following diagram. Moreover, managing usage is still very relevant to minimize unnecessary costs.



messages	
id	String
latitude	String
longitude	String
message	String

reports	
report-id	String
message	String
message-id	String
reason	String

keyword-list	
keyword-id	String
keyword	String

The remote database consists of three tables: messages, reports and keyword-list. The id of a message is its geohash, also used as a foreign key to the reports. The keyword list table does not need to interact with the other tables.

```
dynamodb = boto3.resource(
    'dynamodb',
    region_name=settings.AWS_REGION_NAME,
)

table = dynamodb.Table('keyword-list')

def fetch_keywords_from_dynamodb():
    response = table.scan(AttributesToGet=['keyword'])
    keywords = set()
    for item in response['Items']:
        if 'keyword' in item:
            keywords.add(item['keyword'])
    return keywords

def load_keywords_to_cache():
    keywords = fetch_keywords_from_dynamodb()
    cache.set('moderation_keywords', keywords, timeout=None)
```

This demonstrates how to use the AWS SDK for Python (Boto3) to interact with the table for the keyword list. **boto3.resource** initializes a DynamoDB resource instance, specifying the service name ('dynamodb') and the AWS region where the table is hosted (**region_name=settings.AWS_REGION_NAME**). **settings.AWS_REGION_NAME** corresponds to one of the region codes, like 'us-west-2' or 'eu-central-1'. **dynamodb.Table('keyword-list')** simply sets up a local object that knows how to interact with the specified keyword list table.

fetch_keywords_from_dynamodb retrieves all entries from the 'keyword-list' table and extracts a unique set of keywords. **table.scan(AttributesToGet=['keyword'])** then scans the entire table, returning only the 'keyword' attribute of each item, minimizing the amount of data transferred over the network.

Each keyword is added to a **set** to ensure that each keyword is unique. It is worth noting that the **scan** operation reads every item in the table, which can be time-consuming and costly for large tables. Scans can consume a lot of read capacity, potentially impacting performance and incurring higher costs. As such, to minimize usage, load_keywords_to_cache() sets the cache so that the keywords can be accessed without having to retrieve them.

Due to the limitations of noSQL, Geohash has had a major role as the primary key (message id) to query messages and reports. A Geohash is a geocoding system that encodes geographic coordinates (latitude and longitude) into a short string of letters and digits. It is designed to represent locations on Earth with varying levels of precision, depending on the length of the geohash string. The geohash algorithm divides the world into a grid of cells and assigns each cell a unique string of characters. The string is constructed through a hierarchical spatial data structure called a quadtree, where the world is recursively divided into four quadrants/cells until the desired level of precision is achieved. Using Geohash has simplified much of the database logic.

9.2 Content Moderation with AWS Comprehend

AWS Comprehend is a natural language processing (NLP) service. It uses machine learning to extract insights and relationships from unstructured text data. Some of its capabilities are language detection, sentiment analysis, document classification, etc. As such, it is suitable for content moderation. Comprehend simplifies the process of building natural language processing capabilities into applications by providing pre-trained models and APIs.

As of how it works under the hood, unfortunately Amazon does not publicly disclose the specific model architectures or algorithms used by Amazon Comprehend in detail. This is not uncommon for most commercial AI services where the underlying models and techniques are considered proprietary information.

However, its documentation does provide some hints on Comprehend's inner workings [28]. For example, Comprehend's topic modelling makes use of latent Dirichlet allocation (LDA) [27] to determine topics in a set of documents. Topic modeling is a technique in natural language processing (NLP) and machine learning used to discover abstract "topics" that occur in a collection of documents. It involves algorithms that identify word and phrase patterns within texts, and cluster those patterns into meaningful "topics". LDA is an unsupervised machine learning technique and a generative probabilistic model that automatically discovers topics in a corpus of text documents. When it comes to topic modelling, LDA represents documents as mixtures of topics, where each topic is a probability distribution over words. The assumption is that each document consists of a mixture of topics, and each word in a document is drawn from one of those topics.

The algorithm then attempts to infer the topic distribution for each document in the corpus, as well as the word distribution for each topic, by analyzing patterns of word co-occurrences. Hence, LDA can identify the topics present, their relative prevalence, and how each document exhibits those topics.

The output is clusters of words that co-occur frequently together across documents, representing semantic topics (e.g. the word "apple" in an article that talks about fruits can be assigned to the topic "fruits"). Topic modeling via LDA enables the derivation of high-level insights about a text corpus: finding dominant subjects or themes, detecting changing trends over time, and understanding relationships between topics.

Comprehend can detect offensive language, insults, profanity, and abusive speech in text data. This can help automatically filter out or flag inappropriate content. It can also identify and redact personal information like names, addresses and phone numbers from text. Sentiment analysis can also identify potentially toxic or hateful content that expresses strong negative sentiment towards individuals or groups. Like DynamoDB, its scalability and accuracy make it suitable for moderating high volumes of user-generated content.

While Comprehend already comes with trained models, a custom classification model was trained for the purposes of content moderation.

To train the model, we first need to do data preprocessing. This is done to convert raw data into a clean, organized format that a model can effectively learn from. The first step is gathering the data that will be used for training the model. The data used to train the model was taken from Hugging Face (a community of data scientists and machine learning engineers) and contains a collection of annotated tweets from X (formerly Twitter) for detecting hate speech and offensive language [30]. We then need to clean the data (e.g. removing duplicates, eliminating errors, etc.). Fortunately, the dataset is already curated, but some data cleaning and formatting still needs to be done.

Python has several useful libraries for data preprocessing: **pandas** for data manipulation and reading CSV files, **re** for regular expression operations to clean the text, and **nltk** (Natural Language Toolkit) for natural language processing tasks such as tokenization. As the script shows below, it defines several functions to clean individual tweets.

```
stop_words = set(stopwords.words('english'))
stop_words.add("rt")

def remove_entity(raw_text):
    entity_regex = r"&[^\s;]+;"
    text = re.sub(entity_regex, "", raw_text)
    return text

def change_user(raw_text):
    regex = r"@([^ ]+)"
    text = re.sub(regex, "user", raw_text)
    return text

def remove_url(raw_text):
      url_regex  =  r"(?i)\b((?:https?://|www\d{0,3}[.]|[a-z0-9.\-
]+[.][a-
z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))*\))+(?:\(([
^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]{};:'\".,<>?«»“”‘’]))"
    text = re.sub(url_regex, '', raw_text)
    return text

def remove_noise_symbols(raw_text):
    text = raw_text.replace('"', '')
    text = text.replace("'", '')
    text = text.replace("!", '')
    text = text.replace("`", '')
    text = text.replace("..", '')
    return text

def remove_stopwords(raw_text):
    tokenize = nltk.word_tokenize(raw_text)
     text  =  [word  for  word  in  tokenize  if  not  word.lower()  in
stop_words]
    text = " ".join(text)
    return text
```

remove_entity removes HTML entities (e.g., &amp;, &gt;) from the text using a regular expression.

change_user replaces Twitter usernames (e.g., **@username**) with a generic placeholder (**user**) to anonymize the data and reduce noise. **remove_url** eliminates URLs from the text, as they often do not contribute useful information for text analysis tasks.

remove_noise_symbols cleans up various punctuation and symbols that are not useful for analysis, such as quotes, exclamation marks, and backticks.

remove_stopwords filters out common English stopwords (frequently occurring words such as "the", "is", "in", which do not contribute much meaning) using NLTK's predefined list. It also removes "rt" to filter out retweets. This step involves tokenizing the text into individual words, filtering out stopwords, and then joining the words back into a processed string.

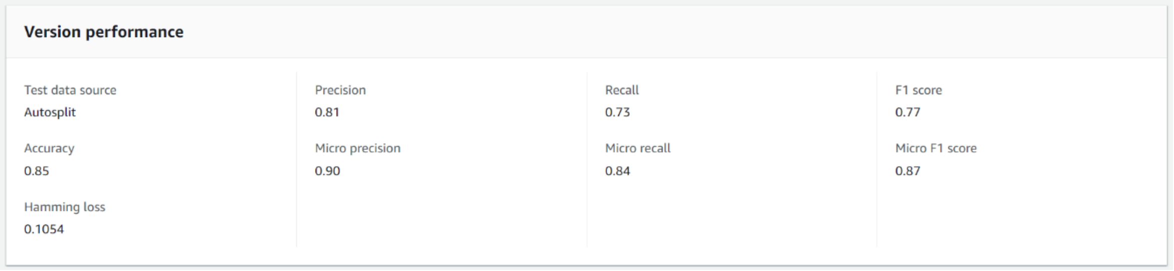
These functions can turn a tweet such as **“!!! RT @mayasolovely: As a woman you shouldn't […]”** into **"user woman shouldnt complain cleaning house […]”**. The dataset is then split into two subsets: the training set and the test set. The training set is used to train the model, while the test set is used to evaluate its performance. The split ratio used was 10% testing, and 90% training.

Using the model in the application is quite simple as seen in the snippet below.

```
comprehend         =         boto3.client(service_name='comprehend',
region_name='eu-west-2')

classification_response = comprehend.classify_document(
       Text=message,
       EndpointArn='foo’,
        )
        labels = classification_response['Labels']
        for label in labels:
        if label['Name'] in ['OFFENSIVE_LANGUAGE', 'HATE_SPEECH']:
              return JsonResponse({'Status' : 'Inappropiate message'},
              status=422)
```

boto3.client initializes an AWS Comprehend client. **classify_document** is called on the client, passing the text to be classified (**Text=message**) and the endpoint ARN (Amazon Resource Name) for the classification model. The response from **classify_document** contains classification results under **'Labels'**. We check if any of them correspond to **'OFFENSIVE_LANGUAGE'** or **'HATE_SPEECH'**. If so, it returns a JSON response indicating an inappropriate message.

While testing has been limited to minimize costs, Comprehend tells us quite a lot about how the model performs.



Version performance

Test data source
Autosplit

Accuracy
0.85

Hamming loss
0.1054

Precision
0.81

Micro precision
0.90

Recall
0.73

Micro recall
0.84

F1 score
0.77

Micro F1 score
0.87

Accuracy is the proportion of true results (both true positives and true negatives) among the total number of cases examined. Here, it's 0.85, meaning the model correctly predicts 85% of the time.

Hamming Loss is the fraction of labels that are incorrectly predicted, i.e., the wrong labels are predicted, or the right labels are not predicted. A value of 0.1054 means that approximately 10.54% of the model's predictions were incorrect on a per-label basis.

Precision is the fraction of relevant instances among the retrieved instances. Here, it's 0.81, indicating that when the model predicts a label, it is correct 81% of the time.

Recall is the fraction of the total amount of relevant instances that were actually retrieved. A recall of 0.73 means that the model correctly identifies 73% of the actual positive instances.

F1 Score is the harmonic mean of precision and recall, where it essentially gives more weight to smaller values in the dataset (in this context, the smaller value between precision and recall). An F1 score reaches its best value at 1 (i.perfect precision and recall) and worst at 0. The F1 score here is 0.77, which is quite robust and suggests a good balance between precision and recall.

Micro Precision aggregates the contributions of all classes to compute the average precision. With a micro precision of 0.90, the model has a high precision rate when considering all classes collectively.

Micro Recall, like micro precision, micro recall aggregates the contributions of all classes to compute the average recall. A micro recall of 0.84 suggests the model is fairly good at capturing the positive instances across all classes.

Micro F1 Score is the F1 Score calculated using micro precision and micro recall. It's a measure of overall performance across all classes. The value of 0.87 indicates good overall performance.

Comprehend also outputs a confusion matrix in a json file, a table used to describe the performance of a classification model on a set of test data for which the true values are known. This is given as a list of lists where each inner list represents the matrix for one class. The matrix is structured with true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP) in that order from left to right, top to bottom. This is for multi-label classification: each instance in the dataset could be labeled with more than one class.

```
{"confusion_matrix":
[[[1797, 183], [234, 266]],
[[1835, 55], [162, 428]],
[[294, 54], [96, 2036]]],
"labels": ["HATE_SPEECH", "NEITHER", "OFFENSIVE_LANGUAGE"],
"type": "multi_label",
"all_labels": ["HATE_SPEECH", "NEITHER", "OFFENSIVE_LANGUAGE"]}
```

For **HATE_SPEECH**:

- TN: 1797 - Number of times the model correctly predicted 'not hate speech'.
- FP: 183 - The model incorrectly predicted 'hate speech' when it wasn't.
- FN: 234 - Missed 'hate speech' occurrences.
- TP: 266 - Correctly identified 'hate speech'.

For **NEITHER**:

- TN: 1835 - The model correctly predicted 'not neither.'
- FP: 55 - The model incorrectly predicted 'neither' when it was a different category.
- FN: 162 - The model missed 'neither' occurrences.
- TP: 428 - The model correctly identified 'neither.'

For **OFFENSIVE_LANGUAGE**:

- TN: 294 - The model correctly predicted 'not offensive language'.
- FP: 54 - The model incorrectly predicted 'offensive language' when it wasn't.
- FN: 96 - The model missed 'offensive language' occurrences.'
- TP: 2036 - The model correctly identified 'offensive language'

From the confusion matrix we can see that the model is relatively better at identifying "OFFENSIVE_LANGUAGE" compared to "HATE_SPEECH" and "NEITHER". This is evident from the higher TP rate for "OFFENSIVE_LANGUAGE" (2036 correct predictions) than the other labels.

It struggles with correctly identifying "HATE_SPEECH", as indicated by a lower TP count (266) and a higher FN count (234).

For the label "NEITHER", the model has a low FP rate (55), suggesting it's cautious about wrongly labeling something as "NEITHER", but there's still a considerable number of FN (162), indicating missed "NEITHER" classifications. It has a high number of TN, suggesting the model is reliable in identifying content that does not belong to this class.

The relatively low number of FP for "NEITHER" and "OFFENSIVE_LANGUAGE" (55 and 54, respectively) compared to TN suggests that the dataset might be imbalanced, with fewer instances of these labels compared to "HATE_SPEECH".

The higher TN counts for "HATE_SPEECH" and "NEITHER" suggest that the dataset might have more instances that are not labeled as such, or that the model is conservative in predicting these classes.

The model's higher FN count for "HATE_SPEECH" and "NEITHER" indicates a tendency to miss labeling these classes when they are present.

Conversely, "OFFENSIVE_LANGUAGE" has a lower FN rate, indicating the model is more adept at catching this type of content.

The model may need improvements in sensitivity, especially for the "HATE_SPEECH" category, as it's missing many actual cases.

There may be a need for more representative training data for "HATE_SPEECH" and "NEITHER" categories, or a reevaluation of feature selection and model parameters to better capture the nuances of these labels.

Overall, this model seems to perform well, with high values for accuracy and micro-averaged metrics, indicating that it may be quite effective for practical use. It also seems to be quite good at identifying "OFFENSIVE_LANGUAGE" but may have benefited from further tuning or additional training data to improve its performance on "HATE_SPEECH" and "NEITHER".

A good way of visualizing these metrics is to plot a Receiver Operating Characteristic (ROC) curve. It is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold (the probability or score at which the positive class is chosen over the negative class, by default 0.5) is varied and is used to visualize and evaluate the performance of classification models, especially when there is an imbalance between the classes. The ROC curve plots the true positive rate (TPR) or sensitivity against the false positive rate (FPR) at various threshold settings. To calculate TPR and FPR:

$$FPR = FP/(FP + TN)$$

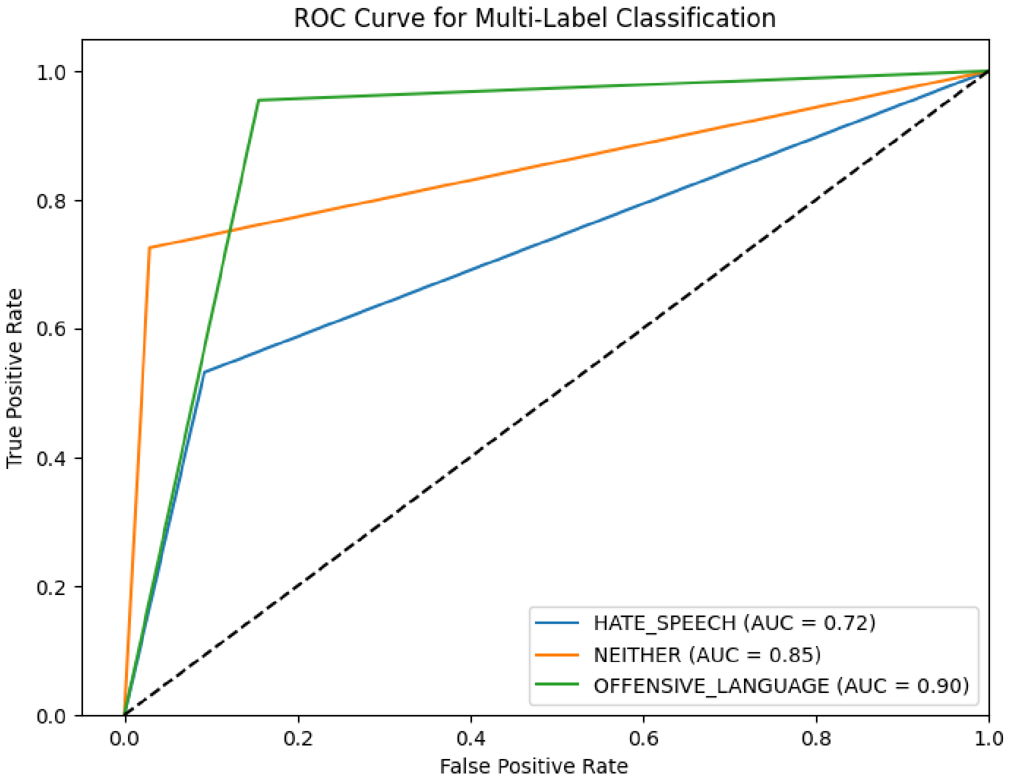
$$TPR = TP/(TP + FN)$$

The ROC curve is created by plotting TPR against FPR at different classification thresholds.

A perfect classifier would have a TPR of 1 for all thresholds (i.e. 100% sensitivity) with an FPR of 0 (i.e. 100% specificity), which would give a ROC curve passing through the top-left corner of the plot. The more the curve bulges towards the top-left corner, the better the classification performance.

The area under the ROC curve (AUROC or AUC) is a single metric to evaluate the overall diagnostic ability of a classifier. A perfect classifier has AUC=1, while a random classifier has AUC around 0.5.

ROC curves and AUC are useful for evaluating and comparing machine learning classification models, especially for imbalanced datasets. To plot a ROC curve from a multi-label confusion matrix in Python, we can use the scikit-learn library. The approach taken computes the ROC curve and AUC for each class separately, treating it as a binary classification problem. The resulting plot shows the ROC curves and AUC values for each label in the multi-label confusion matrix.



We can draw similar conclusions. Based on the AUC values, the model performs best at classifying "OFFENSIVE_LANGUAGE" with an AUC of 0.90, followed by "NEITHER" with an AUC of 0.85, and then "HATE_SPEECH" with an AUC of 0.72. The model performs reasonably well overall, with all three classes having AUC values above 0.7. However, there is still room for improvement, especially for the "HATE_SPEECH" class (AUC = 0.72). The ROC curves for the three labels have distinctively different shapes and AUC values, suggesting class imbalance.

The shape of the ROC curves indicates the potential tradeoff between precision and recall for each class. For example, the "OFFENSIVE_LANGUAGE" curve has a steeper initial rise, suggesting that the model can achieve high recall (sensitivity) for this class at the cost of lower precision (more false positives). In contrast, the "HATE_SPEECH" curve rises more gradually, indicating a potential bias towards higher precision but lower recall.

10. 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.

10.1 Implementing Automated Keyword Filtering

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.

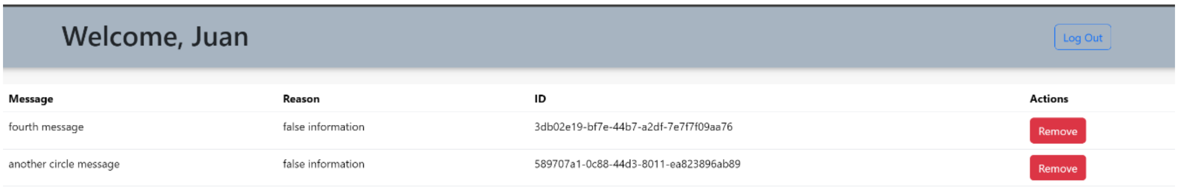
To accommodate for this, it would require extremely large keyword lists. As discussed before, scan operations can be very time consuming and expensive, especially for the size required. While this implementation could be considered a first line of defence, it would not be appropriate to fully rely on it as the backbone of content moderation.

10.2 Advanced Content Moderation

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. As shown before, the model created performs well overall, although it struggles with some labels, likely due to possible imbalances in the dataset.

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

A warning is displayed to the user if a message is deemed to be inappropriate, and if the toxic message makes it through automated moderation, users can click a marker's popup (the message) to report it. The moderator is then able to see the reports and remove the messages accordingly, as shown here:



Welcome, Juan

Message	Reason	ID	Actions
fourth message	false information	3db02e19-bf7e-44b7-a2df-7e7f7f09aa76	Remove
another circle message	false information	589707a1-0c88-44d3-8011-ea823896ab89	Remove

The implementation was similar to what was done for messaging functionality. There is a table for user reports in the remote database, which is updated each time a user submits a report through the modal. The nature of this implementation makes it so that it is online only. However, messages in the IndexedDB object store that were not initially present in the remote database (searching by Geohash) will be automatically moderated once the network conditions allow it.

The goal is to filter out harmful content: hate speech, misinformation, harassment, and explicit material. The only supported language is English, and while Comprehend does support multiple languages, 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.

11. Ethics and professional issues

There are various ethical and professional issues to consider regarding the development of the maps application. 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 and professional aspects associated with this project.

11.1 Ethical Considerations in User Data Handling and Content Moderation

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. For example, for user registration, obtaining user consent for data collection, and informing about its usage is ethically necessary.

On the topic of content moderation, it is important to highlight several issues at hand. Firstly, there are scalability challenges. As online platforms grow rapidly, finding and retaining enough skilled human moderators, especially for non-English languages, is an ongoing staffing challenge. Human moderators can also exhibit conscious or unconscious biases that lead to inconsistent or unfair content moderation decisions based on such biases. As such, they need thorough training on policies, cultural context, and ethical decision-making to consistently apply moderation guidelines accurately across different scenarios in order to mitigate potential biases and ensuring consistent policy interpretations, in addition to understanding cultural context and nuances across different languages, regions, and communities. This is critical for moderators to avoid inappropriate removals.

Online platforms such as Twitter (now named X) contain questionable content that is being regularly uploaded. One such example that has been discussed extensively is pro-eating disorder communities on Instagram [22], exhibiting lexical variations that human moderators are well equipped to combat. Moderators are exposed to potentially disturbing, violent, or traumatic content on a regular basis. While content moderators are needed to prevent these sorts of content from being spread, moderating comes at the expense of mental health. The use of language models hopes to mitigate this problem, serving as an additional filter before human moderation comes into play.

Of course, automated content moderation, especially involving language models, raises several important professional issues. Like human moderators, language models can exhibit biases present in the data they were trained on, leading to unfair or discriminatory content moderation decisions. A key challenge is ensuring fairness and mitigating societal biases.

Additionally, many language models are "black boxes", making it difficult to understand how they arrive at particular moderation decisions. These are valid concerns around transparency and the ability to audit/explain these automated systems. While Comprehend itself is not a complete black box, the inner workings, training algorithms, and model architectures used by Comprehend are not fully open-sourced or disclosed, which is not unheard of for commercial AI services. The approach taken in this application involves using a custom classification model trained with a custom dataset to mitigate this issue.

Content moderation systems may also process large volumes of personal data and user-generated content, raising privacy concerns around data collection, storage, and use. Fortunately, Comprehend already comes with Personally Identifiable Information (PII) detection, but the question still remains on how the PII detection feature was developed and the data used for development.

The custom classification model that was trained is, as shown in the confusion matrix before, fallible. When automated systems make mistakes or controversial decisions (this can happen with topics such as politics), questions arise about who is liable - the model developers, deployers or end-users. The are many key considerations with accountability and liability. When an automated system makes incorrect content moderation decisions (e.g. removing non-toxic content or failing to flag harmful content), it raises the question of who is liable - the developers who created the model or the company or platform deploying it? Determining culpability is a complex endeavour given the distributed nature of AI development and deployment.

For instance, there are open questions around what constitutes "reasonable care" by developers and deployers of these AI systems in terms of sufficient testing, human oversight, bias mitigation efforts, etc. There are also challenges in interpreting and explaining the decisions from large language models making it more difficult to attribute fault and defensibility for errors. Stronger capabilities for AI interpretability would aid accountability.

11.2 Navigating GDPR Compliance

On the topic of PII, General Data Protection Regulation (GDPR) also comes into discussion. It is a comprehensive data privacy and security law that went into effect in the European Union in 2018 [29] and introduces several professional issues and compliance requirements that organizations must address. As such, adherence to the GDPR represents a critical professional and ethical obligation.

Under GDPR, the principles of data minimization and purpose limitation are paramount. This means that the application should only collect data that is directly relevant and necessary for the intended purpose of leaving location-based messages. Additionally, the collected data should not be repurposed without obtaining further consent from the users.

Obtaining explicit consent from users before collecting and processing their data is another aspect of GDPR. This means providing clear and comprehensive information about what data is collected, for what purpose, and how it will be used, ensuring transparency. This information should be conveyed through an easily accessible and understandable privacy policy. Users should have the option to opt-in to data collection practices, rather than being automatically enrolled, and they should be able to withdraw their consent at any time.

The application must also implement robust security measures to protect user data from unauthorized access, data breaches, and other security threats. This includes employing encryption for data transmission, securing databases, and regularly updating security protocols. Furthermore, anonymizing user data wherever possible enhanced privacy. For example, when displaying location-based messages to other users, the application avoids directly associating messages with specific user identities.

Additionally, GDPR grants individuals the right to access their personal data held by an organization and the right to have their data erased under certain conditions, also known as the "right to be forgotten." In implementation, this involves providing mechanisms for users to request a copy of their data and to delete their account along with all associated data. This ensures that users have full control over their information and can exercise their rights under GDPR.

As such, GDPR compliance is a multifaceted issue that encompasses data minimization, user consent, data security, and user rights.

11.3 The Impact of Offline Functionality

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

When it comes to relevant professional issues, it is important to set clear expectations for users about the limitations and capabilities of offline functionality. While messaging can be done offline, much of the content moderation is server side, so the user would not see their effects unless they are online.

The ability to leave location-based messages can influence local communities and businesses. These features can be misused to negatively impact them. One particular example of this would be to maliciously leave bad review messages near the business. Regarding user generated content reporting systems, it raises several professional issues.

One of these is preventing malicious or coordinated abuse of reporting tools to silence voices or censor legitimate content. Therefore, there should be measures to detect and mitigate false or frivolous reports. This, in turn, involves achieving a balance of avoiding over censorship while still addressing harmful content effectively, and deciding appropriate levels of automation versus human review for different report categories. Human moderators are especially suited to deal with this, more so with robust training programs and guidelines, as discussed before.

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, as evidenced by the good performance of the trained model. Human moderation is possibly the most effective option to combat this issue.

Ethical and professional 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 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 bases.

12. Conclusion

HTML5 technologies play a major role in enhancing offline functionality and maps applications as we have accomplished: an offline maps application that allows its users to publicly leave location-based messages, with these messages becoming accessible upon re-establishing internet connectivity. This project sought to leverage advanced web technologies to enhance user experience and functionality in offline environments.

This project successfully demonstrated the integration of various technologies: HTML5, CSS, JavaScript, Django, IndexedDB, DynamoDB, AWS Comprehend, and Leaflet.js, to achieve its offline capabilities and interactive map features. Key achievements that laid the groundwork include the development of a "hello world" offline application using Cache and Service Workers, a "to-do list" leveraging IndexedDB, and a shape drawing feature utilizing HTML5 canvas. Moreover, the application effectively utilized Open Street Map data and explored the dynamics between vector and image tile maps, setting a precedent for future applications in similar domains.

The use of OSM data was a pivotal aspect of the project, with an in-depth exploration of vector versus image tile maps providing valuable insights into the optimal representation of map data for web applications, thus highlighting the nuanced considerations involved in choosing between vector and image tiles in relation to offline access, user interaction, and application performance.

This application holds significant implications for the development of web applications with robust offline functionalities. By addressing the challenge of maintaining functionality in the absence of internet connectivity, the project contributes valuable insights and methodologies to the field of web development. Furthermore, the integration of content moderation systems using keyword lists and language models underscores the commitment to creating safe and respectful user environments, with the potential of guiding future applications in content moderation practices.

Despite its successes, there are several improvements the can be made, especially involving AWS usage limits and the testing framework. Another aspect was the choice of libraries for rendering, leading to the application's reliance on image tiles due to limitations in vector tile support by Leaflet.js, suggesting an area for future optimization.

Future work could explore the integration of vector tiles for enhanced scalability and user interaction, optimization of caching algorithms, as well as the expansion of social features such as likes and dislikes. user profiles, friend networks, and shared itineraries, and the application's content moderation capabilities to address a wider array of user-generated content.

The Offline HTML5 Maps Application represents a significant step towards realizing the potential of web applications in offline environments. It underscores the viability of integrating complex web technologies to enhance user experience and address practical challenges in connectivity. Future enhancements and research in this domain can build upon the foundations laid by this project, pushing the boundaries of what is achievable in web application development and content moderation.

13. Bibliography

[1] Wang, J. (1999). A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, *29*(5), 36-46.

[2] 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.

[3] Bibeault, B., De Rosa, A., & Katz, Y. (2015). *jQuery in Action*. Simon and Schuster.

[4] Haklay, M., & Weber, P. (2008). Openstreetmap: User-generated street maps. *IEEE Pervasive computing*, *7*(4), 12-18.

[5] 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

[6] 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

[7] Gillespie, T. (2020). Content moderation, AI, and the question of scale. *Big Data & Society*, *7*(2), 205395172094323. https://doi.org/10.1177/2053951720943234

[8] Lawson, B., & Sharp, R. (2012). *Introducing HTML5*. Peachpit Press.

[9] Crickard III, P. (2014). *Leaflet.js essentials*. Packt Publishing Ltd.

[10] Arsht, A., & Etcovitch, D. (2018). The human cost of online content moderation. Harvard Journal of Law and Technology, 2.

[11] Aggarwal, K. K. (2005). Software engineering. New Age International.

[12] Tabarés, R. (2021). HTML5 and the evolution of HTML; tracing the origins of digital platforms. *Technology in Society*, *65*, 101529.

[13] Degenhard, J. (2024, February 28). Global: Number of smartphone users 2014-2029. Statista. https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world

[14] Boulos, M. N. K., Warren, J., Gong, J., & Yue, P. (2010). Web GIS in practice VIII: HTML5 and the canvas element for interactive online mapping. *International journal of health geographics*, *9*, 1-13.

[15] Taraldsvik, M. (2011). Exploring the Future: is HTML5 the solution for GIS Applications on the World Wide Web?

[16] Lubbers, P., Albers, B., Salim, F., Lubbers, P., Albers, B., & Salim, F. (2010). Creating HTML5 offline web applications. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*, 243-257.

[17] He, Y., & Wang, J. J. (2014). The Offline Storage Technology Based on HTML5. Applied Mechanics and Materials, 687–691, 2104–2107. https://doi.org/10.4028/www.scientific.net/amm.687-691.2104

[18] Park, H., Kim, K., & Lee, K. (2016, July). Geo-data visualization on online and offline mode of mobile web using HTML5. In *2016 4th International Workshop on Earth Observation and Remote Sensing Applications (EORSA)* (pp. 237-240). IEEE.

[19] Malavolta, I., Procaccianti, G., Noorland, P., & Vukmirovic, P. (2017, May). Assessing the impact of service workers on the energy efficiency of progressive web apps.

In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (pp. 35-45). IEEE.

[20] Gorwa, R., Binns, R., & Katzenbach, C. (2020). Algorithmic content moderation: Technical and political challenges in the automation of platform governance. *Big Data & Society*, *7*(1), 2053951719897945.

[21] Binns, R., Veale, M., Van Kleek, M., & Shadbolt, N. (2017). Like trainer, like bot? Inheritance of bias in algorithmic content moderation. In *Social Informatics: 9th International Conference, SocInfo 2017, Oxford, UK, September 13-15, 2017, Proceedings, Part II 9* (pp. 405-415). Springer International Publishing.

[22] Chancellor, S., Pater, J. A., Clear, T., Gilbert, E., & De Choudhury, M. (2016, February). # thyghgapp: Instagram content moderation and lexical variation in pro-eating disorder communities. In *Proceedings of the 19th ACM conference on computer-supported cooperative work & social computing* (pp. 1201-1213).

[23] Seering, J., Wang, T., Yoon, J., & Kaufman, G. (2019). Moderator engagement and community development in the age of algorithms. *New Media & Society*, *21*(7), 1417-1443.

[24] Girres, J. F., & Touya, G. (2010). Quality assessment of the French OpenStreetMap dataset. *Transactions in GIS*, *14*(4), 435-459.

[25] Neis, P., & Zipf, A. (2012). Analyzing the contributor activity of a volunteered geographic information project—The case of OpenStreetMap. *ISPRS International Journal of Geo-Information*, *1*(2), 146-165.

[26] Luxen, D., & Vetter, C. (2011, November). Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems* (pp. 513-516).

[27] Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, *3*(Jan), 993-1022.

[28] Amazon. (2024). Topic modeling. Topic modeling - Amazon Comprehend. https://docs.aws.amazon.com/comprehend/latest/dg/topic-modeling.html

[29] General Data Protection Regulation (GDPR). General Data Protection Regulation (GDPR) - Official Legal Text. (2022, September 27). https://gdpr-info.eu/

[30] Davidson, T. (2017). *Tdavidson/hate_speech_offensive · datasets at hugging face*. tdavidson/hate_speech_offensive · Datasets at Hugging Face. https://huggingface.co/datasets/tdavidson/hate_speech_offensive

14. Appendix

14.1 Diary

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 appropiate 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 Antartica. 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.

17/12/2023

After experimenting with many different ways of generating tiles (QGIS, Tilemaker, etc.), I have decided to use a small area of the UK with raster tiles. The biggest limiting factor is the limits on cache size. Just vector tiles of the UK would (generally) be 2GB big. I will be focusing on other features for the moment.

25/12/2023

The main messaging functionality with IndexedDB and DynamoDB is complete. There is still some improvements that can be made with styling, as well as expanding the map. The next step is implementing moderation: first simple automated moderation with keyword lists, then human moderation and potentially using an LLM with AWS Comprehend.

31/12/2023

The basic implememtation of automated moderation involving keyword lists is done. This implememtation ensures that inappropiate messages are not added to the remote database records. A limitation is that it does not work offline, one reason why is that keyword lists can be extemely big (due to the many possibilites of obfuscation) and it will not scale well in the client-side, not to mention that client-side validation can be bypassed easily. I have also replaced uuid with geohash as the primary key for messages to prevent attribute duplicates in the remote database. Next up, I will then explore human moderation.

23/01/2023

The implememtation for human moderation is complete. There are some improvements needed for styling, some bugs need fixing and further tests must be done. The next consideration will be content moderation with a LLM. This can be done with AWS Comprehend. My expectations with this experiment are not very high, considering that the free tier limits will be a bottleneck and investing in this possibility is off the table.

30/01/2023

User is now only able to add messages within their surrounding areas, this concludes the main functionality that was proposed. The current focus is now on testing the application with unit tests and Selenium.

10/02/2023

The unit tests on the application cover the backend functionality (dynamoDB and views) and some of the JavaScript functionality. I have found to be quite difficult to test for the frontend functionality. The issue mainly lies with properly mocking the complex environment. As such, a more exhaustive way to test would be end-to-end testing using Selenium to verify the functionality and performance of the entire application by simulating real-world user scenarios.

18/02/2023

Selenium testing has made achieving a higher code coverage less difficult. So far, such tests cover both messaging and reporting messages. I will be focusing on polishing the application and looking into an LLM based automated moderation.

02/03/2023

I have trained a model using AWS Comprehend with a dataset of X (formerly Twitter) data, which labelled text with three labels: offensive language, hate speech and neither. Overall I am satisfied with the performance of the model (with an accuracy of 0.85). However, testing has been limited as deploying the model would incur additional costs.

Final video demonstration: https://www.youtube.com/watch?v=Czbw1-bSpYQ

14.2 Installation Manual

First, install the needed packages from the **requirements.txt** file.

```
pip install -r requirements.txt
```

Create the database migration files based on the model definitions.

```
python manage.py makemigrations
```

We then need to create the database using the migration files..

```
python manage.py migrate
```

To create a moderator user:

```
python manage.py createsuperuser
```

To access the AWS services that the application uses, use the AWS CLI

```
aws configure
```

You will be prompted to enter the AWS access keys. The following keys will be valid temporarily:

Access key: AKIA6Q5DIMO6VNWOHF64

AWS Secret key: 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

Additionally, set the default region name and output format to **eu-west-2** and **json.**

To run the application:

```
python manage.py runserver
```

14.3 User Manual for interim programs

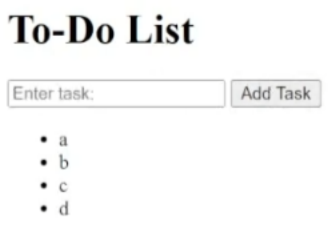
Accessing "hello world" offline application: http://127.0.0.1:8000/hello-world

Hello World!

This is a simple HTML5 offline application

Accessing "todo list" application: http://127.0.0.1:8000/

Enter the task and click the “add task” button.

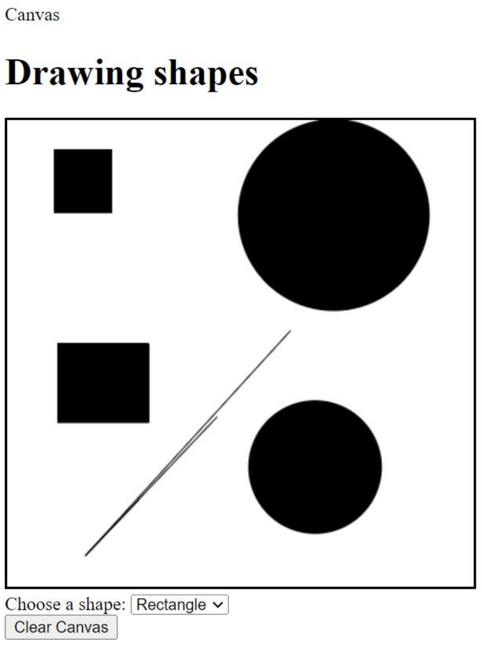


To-Do List

- a
- b
- c
- d

Accessing an application for drawing shapes with HTML5 canvas: http://127.0.0.1:8000/canvas

Select a shape from the drop-down menu (circle, rectangle, line) and draw by dragging the cursor in a direction.



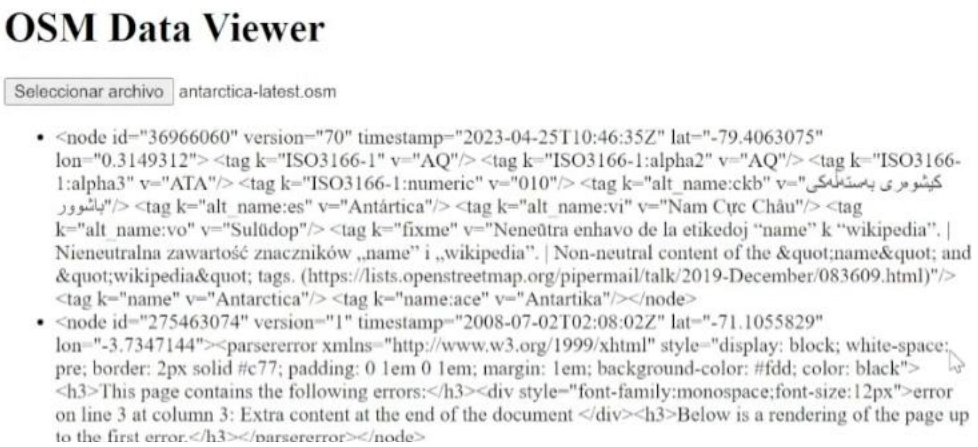
Canvas

Drawing shapes

Choose a shape: Rectangle

Accessing raw OSM data viewer: http://127.0.0.1:8000/raw-data-view

To display raw OSM data, simply upload an .osm file.



OSM Data Viewer

Seleccionar archivo 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="کیشوەری بەستەڵەکی
  باشوور"/> <tag k="alt_name:es" v="Antártica"/> <tag k="alt_name:vi" v="Nam Cực Châu"/> <tag
  k="alt_name:vo" v="Sulüdop"/> <tag k="fixme" v="Neneŭtra enhavo de la etikedoj "name" k "wikipedia". |
  Nieneutralna zawartość znaczników „name" i „wikipedia". | Non-neutral content of the &quot;name&quot; and
  &quot;wikipedia&quot; tags. (https://lists.openstreetmap.org/pipermail/talk/2019-December/083609.html)"/>
  <tag k="name" v="Antarctica"/> <tag k="name:ace" v="Antartika"/></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: #fdd; color: black">
  <h3>This page contains the following errors:</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>
```

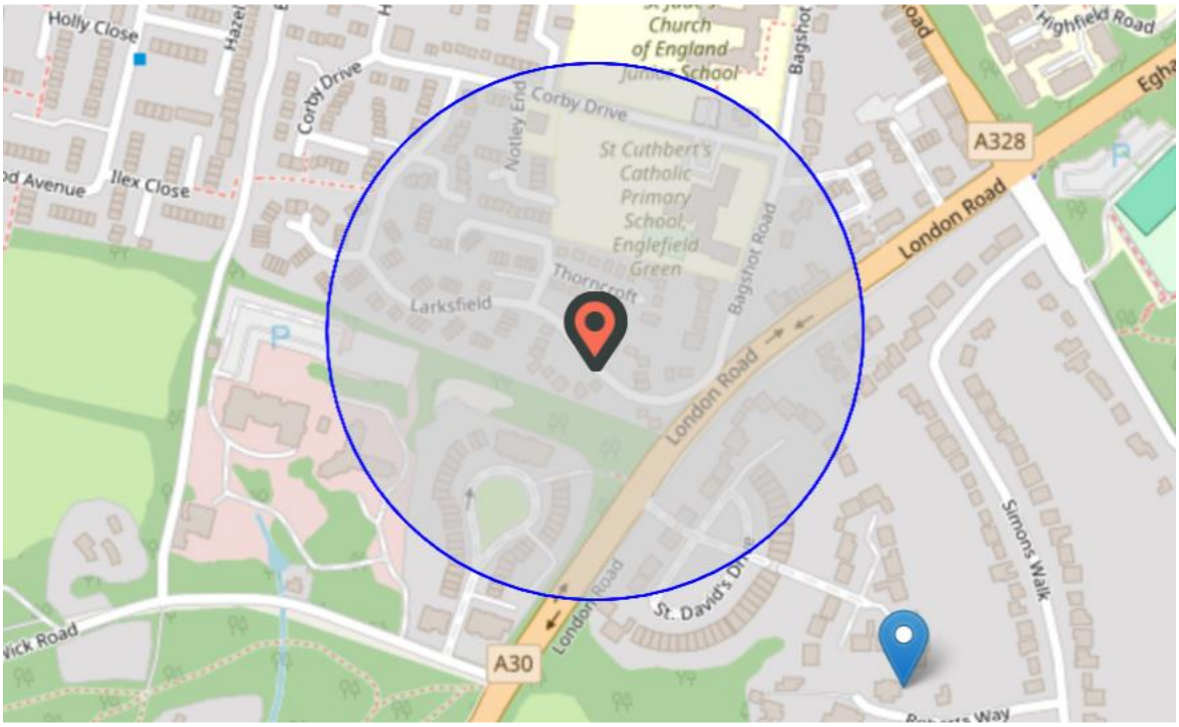
14.4 User Manual

Accessing the map: http://127.0.0.1:8000/map

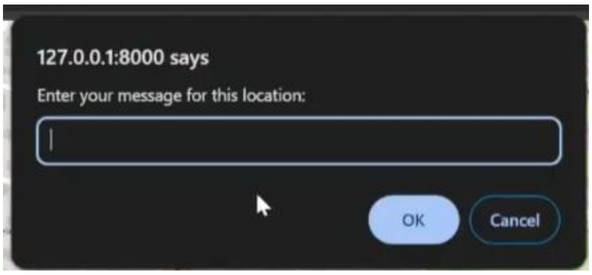
Navigating the Map: Drag the mouse across the screen to pan the map. Use the +/- buttons at the top left or use the mouse wheel to zoom in or out.

Leaving Messages

1. **Select a location**: Click on the map where you want to leave a message inside the highlighted radius around your location.



2. **Write the message**: a dialogue box will prompt you to type your message. Keep it concise and relevant to the location.

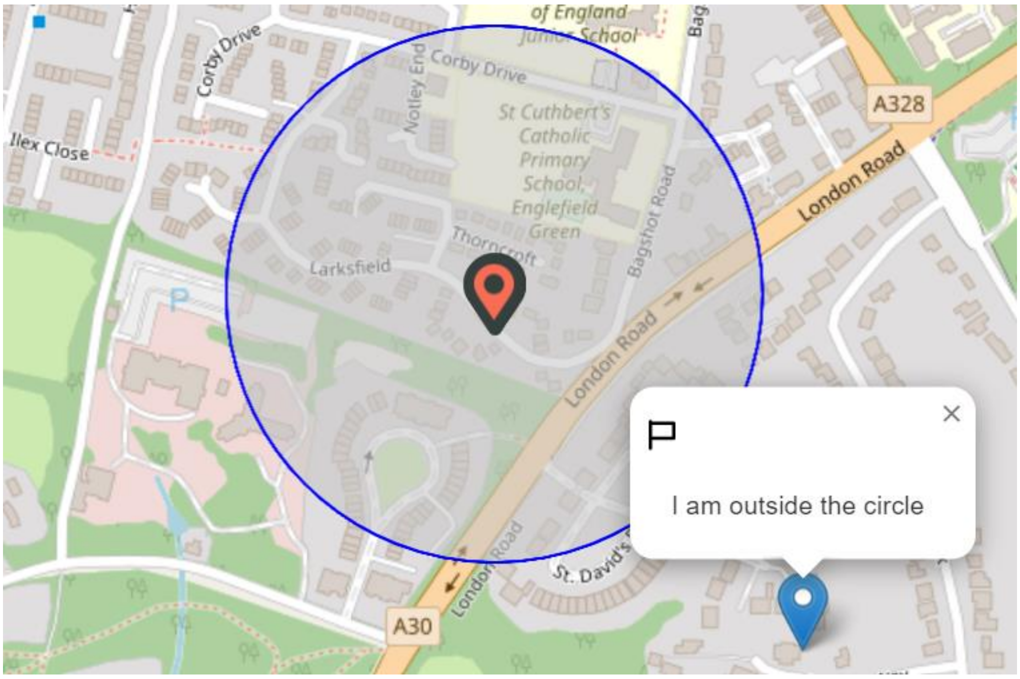


127.0.0.1:8000 says

Enter your message for this location:

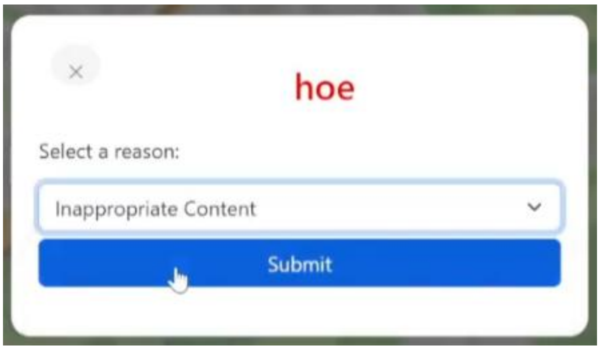
Post: once satisfied, tap the "OK" button. The message will be saved and a marker will appear.

Viewing Messages: click on any visible markers to read messages left by others.



I am outside the circle

Reporting a message: click on the flag symbol on the popup, this will display a modal. Select the reason and click the "Submit" button.



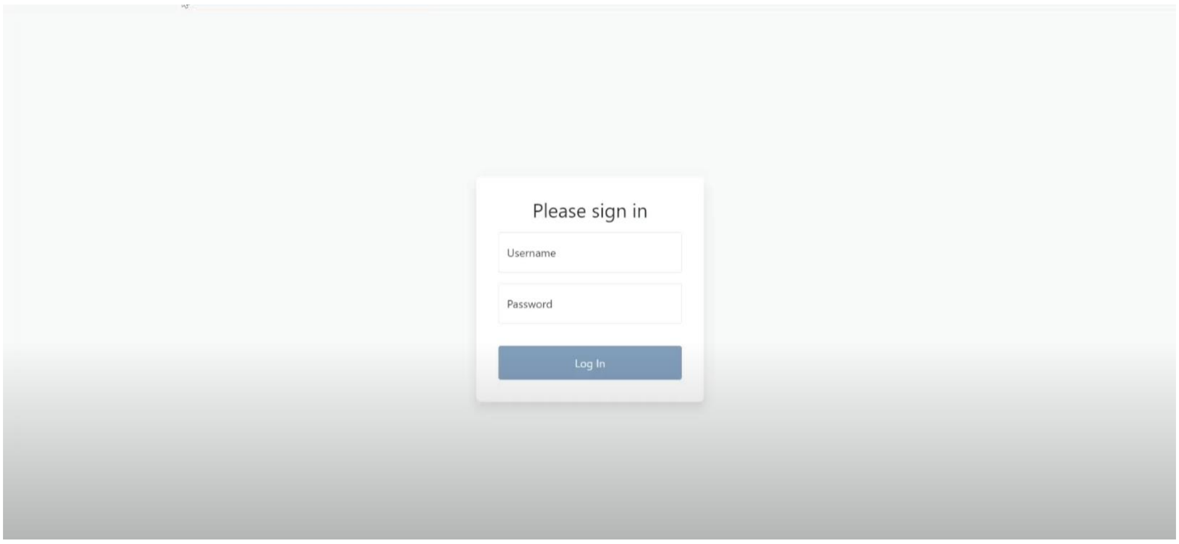
hoe

Select a reason: Inappropriate Content

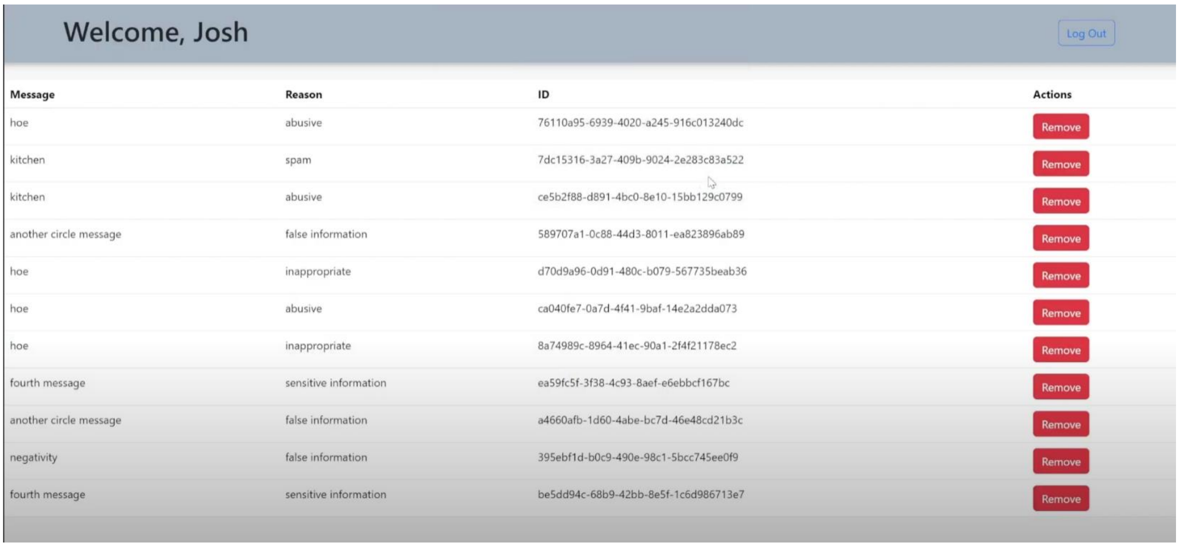
Offline Functionality: the app automatically stores messages and caches the map for when no internet connection is detected.

Accessing moderation page: http://127.0.0.1:8000/moderation

Use the login details of the superuser created to log in.



Removing messages: a list of messages and the reason of which they were being reported will be displayed. Messages can be removed with the “Delete” button.



Welcome, Josh

Message	Reason	ID	Actions
hoe	abusive	76110a95-6939-4020-a245-916c013240dc	Remove
kitchen	spam	7dc15316-3a27-409b-9024-2e283c83a522	Remove
kitchen	abusive	ce5b2f88-d891-4bc0-8e10-15bb129c0799	Remove
another circle message	false information	589707a1-0c88-44d3-8011-ea823896ab89	Remove
hoe	inappropriate	d70d9a96-0d91-480c-b079-567735beab36	Remove
hoe	abusive	ca040fe7-0a7d-4f41-9baf-14e2a2dda073	Remove
hoe	inappropriate	8a74989c-8964-41ec-90a1-2f4f21178ec2	Remove
fourth message	sensitive information	ea59fc5f-3f38-4c93-8aef-e6ebbcf167bc	Remove
another circle message	false information	a4660afb-1d60-4abe-bc7d-46e48cd21b3c	Remove
negativity	false information	395ebf1d-b0c9-490e-98c1-5bcc745ee0f9	Remove
fourth message	sensitive information	be5dd94c-68b9-42bb-8e5f-1c6d986713e7	Remove