# Designing and Implementing a Full-Stack Web Application to Facilitate Location-Segmented User Uploads

Ricky Lin
*ECE Department*
*University of Victoria*
Victoria, Canada
rickylin@uvic.ca
V00044860

Chris McLaughlin
*ECE Department*
*University of Victoria*
Victoria, Canada
chrism0936@gmail.com
V00912353

Zhang Zhang
*ECE Department*
*University of Victoria*
Victoria, Canada
zhangzhang0720@uvic.ca
V01046193

Xuqiao Jiang
*ECE Department*
*University of Victoria*
Victoria, Canada
xuqiao.jmaijiao@gmail.com
V01040336

Onkar Prakash Kadam
*ECE Department*
*University of Victoria*
Victoria, Canada
onkark071@gmail.com
V01073466

## Introduction, Tech Overview, and Market Analysis

The proliferation of large scale IoT deployments has created a need for information storage setups that operate independently of user-entered information. Specifically, information must be tagged, categorised, and stored appropriately via a concert of frontend and backend services working together independent of any active user input. This project serves as a demonstration of such a system. It allows a user's uploaded file to be automatically tagged with a user's location in the form of a city string, which leads to storage in an independent filesystem directory in the backend, entirely independent of user input except in cases of system failure due to user-imposed permissions limitations, in which case the user is consulted.

### Tech Overview

The project is implemented as a full-stack web application utilising a client-side javascript frontend that captures a webform, appending user location in the form of a city string, acquired through using builtin browser modules to get the user's latitude and longitude coordinates and checking them against the public OpenStreetMap API. It additionally appends a hash of the uploaded file obtained via the builtin Web Crypto API, to enable crosschecking of file integrity by the backend. The form is then submitted to an endpoint defined on the backend, which captures the information provided by the upload request,

crosschecks the hash to verify file integrity, and then stores the file in the server filesystem based on the city name string provided.

As a fallback, the frontend provides for manual user input of a city name string should user-configured permissions interfere with the frontend's ability to capture it automatically.

After each refresh of the DOM in frontend, the backend is queried for a list of uploaded files broken down by city through a specialised endpoint. The server responds to this request with a json formatted response which is then parsed in javascript and output to the user in the form of a hierarchical list.

Backend services are implemented in Python 3 with Flask. Endpoints are defined for each of the needed endpoints and file system and hashing interfaces are provided through default Python libraries.

### Market Research

IoT-specialised backend services have exploded in popularity in the web hosting and Software-as-a-Service (SaaS) spaces as IoT deployments have expanded exponentially[3][4]. This suggests an emergent trend towards IoT-specific infrastructure and backend solutions. We are as such at a transition point where IoT specific hardware and solutions have reached the mass market but IoT web software stacks remain largely based in a traditional web development paradigm with the same languages, libraries, and toolsets utilised in standard active-user web development. We can thus predict a transition to more specialised IoT development stacks inline with the emerging IoT-specialised backend service provision sector as the technology and market continues to evolve and grow.

## System Design and Architecture

The system design is composed of a traditional frontend-backend web application, with the frontend implemented in traditional client-side non-framework javascript and the backend implemented in Python 3 utilising the flask library. Communication between the two is done through standard HTTP requests, while the backend information retention is handled through extremely simple direct filesystem storage, organised by directory with directory name corresponding to the city string provided by the upload process for a given file.

The system's implementation as an IoT-relevant framework is enabled through the automatic location tagging provided by the frontend (absent security policy interference), which directly collects the user's location information from the browser, cross references it with OpenStreetMap to acquire the name of the city the user is operating within, and then passes this data to the backend to allow for the categorisation of the uploaded files.

Due to the likelihood of client side security policy interference with the process of automatic location detection, the system provides for user intervention in the case where the location is not able to be determined through the normal means. In such a scenario, the user receives a popup alerting them to their system's interference with location detection, and is able to manually provide a city name that they would like to have tagged to the upload. This is then passed to the backend as if it had been derived through the

normal process of interfacing with the OpenStreetMap API and acquiring a city string based off user latitude and longitude.

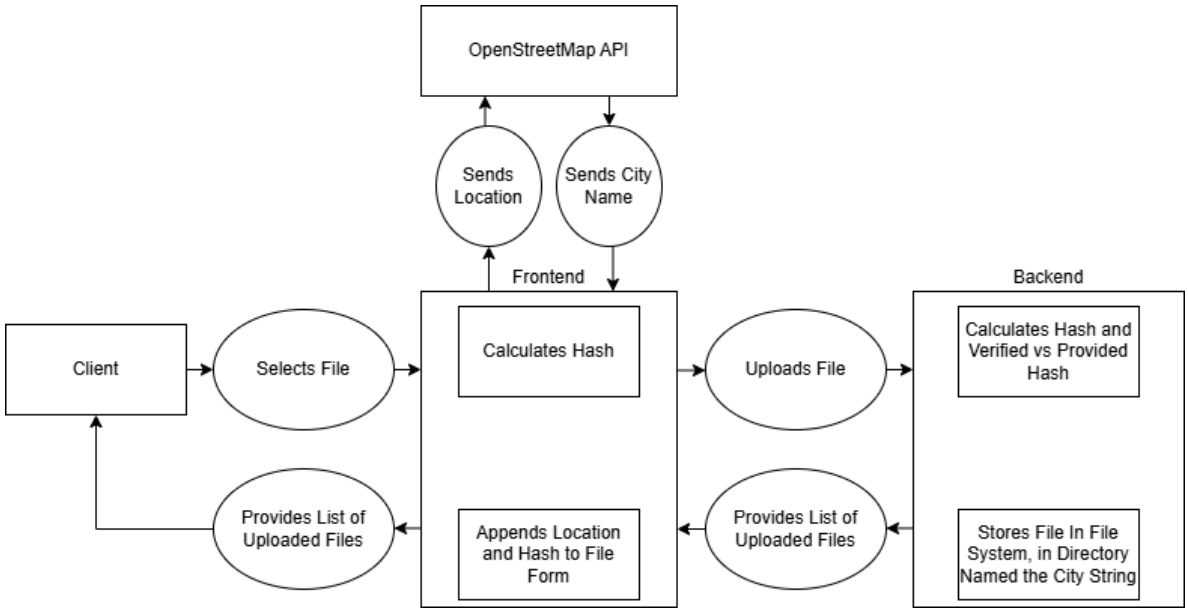


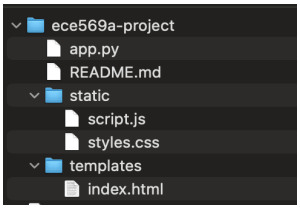

*Figure 1. A Diagram of the System Architecture of the Project*

# System Implementation Approach

Complete code can be see here: [https://github.com/ricl9/ece569a-project](https://github.com/ricl9/ece569a-project)
We use Flask as the server framework, and the corresponding files are placed in the locations specified by the framework's documentation. The complete list of files is as follows:

## Frontend

The frontend code consists of three main files: index.html, which provides the static webpage structure; styles.css, which contains all the required CSS styles; and script.js, which is the JavaScript file responsible for sending requests to fetch data and updating the webpage dynamically. The webpage allows users to both upload new files and see what files have already been uploaded.




## Uploading New Files

The static HTML code provides a file selection button, an "Upload" button, and a hidden text input field (to allow users to manually input the city name as a fallback if the city cannot be automatically determined). The input field is initially hidden. In the JavaScript code, we listen for the "Upload" button's click event to trigger the file upload logic. We have a few steps before sending a request to upload the file.

**File Validation**

First, we check whether the user has selected a file for upload. If no file is selected, an alert is displayed prompting the user to choose a file, and no request is sent to the server.

```
function gethash(in_file) {
    var hashprom = new Promise(resolve => {
        var reader = new FileReader();
        reader.onload = e => {
            var data = e.target.result;
            window.crypto.subtle.digest("SHA-256", data).then(hashBuffer => {
                const hashArray = Array.from(new Uint8Array(hashBuffer));
                const hashHex = hashArray.map(b => b.toString(16).padStart(2, "0")).join('');
                resolve(hashHex);
            })
        }
        reader.readAsArrayBuffer(in_file);
    });
    return hashprom;
```

Using FileReader, the selected file is read, and the SHA-256 hash of the file is computed using the digest() function. This hash value is included as a parameter in the request sent to the server for file integrity verification.

**Retrieving the City Name**

```
if (!cityName) {
    if ("geolocation" in navigator) {
        navigator.geolocation.getCurrentPosition(function(position) {
            fetch(`https://nominatim.openstreetmap.org/reverse?format=json&lat=${position.coords.latitude}&lon=${position.coords.longitude}`)
                .then(response => response.json())
                .then(data => {
                    cityName = data.address.city || data.address.town || data.address.village || data.address.state;
                    if (cityName) {
                        uploadFile(cityName, hash);
                    } else {
                        document.getElementById('cityNameInput').style.display = 'block';
                    }
                })
                .catch(() => {
                    document.getElementById('cityNameInput').style.display = 'block';
                });
        }, function() {
            document.getElementById('cityNameInput').style.display = 'block';
        });
    } else {
        document.getElementById('cityNameInput').style.display = 'block';
    }
} else {
    uploadFile(cityName, hash);
}
```

The city name can either be manually input by the user or automatically determined using geolocation. Since the input field is initially hidden, we first attempt to retrieve the user's location using navigator.geolocation.getCurrentPosition(), which typically tries to get the user's latitude and longitude.

With this geolocation data, we call the OpenStreetMap API to fetch the corresponding address. We chose OpenStreetMap because it is free, has extensive global data, and provides geolocation coverage worldwide. The API response includes detailed location information, and we use the city, town, village, or state fields as the city name.

If the browser does not support geolocation, the geolocation request fails, or the response does not include a city name, we execute the fallback logic: the input field is displayed, allowing the user to manually input the city name.

Once the file, city name, and hash value are obtained, a POST request is sent to the backend to upload the file along with the parameters

```
function uploadFile(city, hash) {
    console.log(hash);
    const formData = new FormData();
    formData.append('file', fileInput);
    formData.append('city', city);
    formData.append('hash', hash);

    fetch('/upload', {
        method: 'POST',
        body: formData
    })
    .then(response => response.json())
    .then(data => {
        if (data.code == 200) {
            alert('File uploaded successfully!');
            document.getElementById('cityNameInput').style.display = 'none';
            listFiles();
        } else {
            alert(data.message || 'File upload failed!');
        }
    })
    .catch(error => console.error('Error:', error));
}
```

The server responds with a status code and a message:

- If the code is 200, it indicates that the file upload and saving were successful. A notification is displayed to inform the user that the file was uploaded successfully.
- If the code is not 200, it indicates an error occurred during some part of the process. Different error codes are defined to identify specific issues. Based on the returned message, an alert is displayed to inform the user of the error's cause.

## Viewing Uploaded Files

```
function listFiles() {
    fetch('/files')
        .then(response => response.json())
        .then(responsejson => responsejson.message)
        .then(citiesobj => {
            //console.log(citiesobj);
            //console.log(typeof citiesobj);

            const fileList = document.getElementById('fileList');
            fileList.innerHTML = '';

            Object.keys(citiesobj).forEach(city => {
                const newheading = document.createElement('h4');
                newheading.textContent = city;
                fileList.appendChild(newheading);
                //console.log(citiesobj[city])
                citiesobj[city].forEach(file => {
                    const newlistentry = document.createElement('li');
                    newlistentry.textContent = file;
                    fileList.appendChild(newlistentry);
                })
            })
        });
};
```

In the JavaScript code, we listen for the "DOMContentLoaded" event (indicating that the HTML has fully loaded) to trigger the listFiles function. Once the event is triggered, the listFiles function is called to send a request to the server and retrieve the list of uploaded files. The server returns a JSON dictionary, where each key represents a city name, and the corresponding value is an array containing all the files uploaded under that city.

The function then iterates through the dictionary, dynamically creating an HTML heading(<h6>) for each city using the city name as the heading text. For each file listed in the corresponding array, a list item (<li>) is created. These dynamically generated HTML elements are then appended under the filesList block in the HTML document. So the webpage shows all uploaded files.

## Backend

Given the limited scope of tasks and the low frequency of access required, we needed a backend framework that is simple, quick to set up, and easy to implement. Flask emerged as the ideal choice for several reasons. First of all, its simplicity and lightweight nature make it straightforward to use, requiring no complex configurations to start running, allowing developers to focus entirely on building features. Second, Flask is Python-based, ensuring that everyone in our group, from developers to testers, can easily contribute to the project. Third, it comes with rich built-in features that simplify data processing and API handling, such as request for retrieving parameters and jsonify for converting dictionaries to JSON responses. Lastly, Flask is well-maintained and highly extensible, supported by a robust ecosystem. While its core remains minimal, it can be easily expanded with libraries to meet additional needs, such as using Flask-Security and Flask-Login to add authentication and enhance security. Given these advantages, Flask was the ideal choice for implementing the backend server, enabling us to successfully complete all core tasks efficiently.

## Return the Upload Page

```
app = Flask(__name__)

@app.route('/')
def main_page():
        #print("MAIN PAGE CALLED", flush=True)
        return render_template("index.html")
```

The primary function of the server is to provide users with a web interface. Using Flask's Template feature, we rendered the upload page and sent it back to the browser with a single line of code. Since templates are files that contain static data as well as placeholders for dynamic data, all JavaScript file references are included directly in the HTML. This approach keeps the backend code clean and simple.

## Upload API for Receiving Files and Related Information

The upload API accepts POST requests, where the client must send three parameters via a form:

| Parameter | Type | | |
|---|---|---|---|

| | | | |
|---|---|---|---|
| file | String | Required | The uploaded file |
| city | String | Required | City name, used as the folder name to save the file. |
| hash | String | Required | SHA256 hash value for verifying the integrity of the file |

Then we finish the following steps in the upload function:

**File Handling**

For the file parameter, the backend checks whether the file has a non-zero length. Even an empty-content file should have a file length; otherwise, it serves no purpose. If the file length is zero, the backend responds with an error code and message for the frontend to display.

**City Name Validation**

```
        #check city name
        city_name = request.form["city"]#request.args.get("city")
        filename = str(file.filename)
        print(city_name)
        if not is_city_name_valid(city_name):
                return build_return_value(402, "City name illegal")

        city_name = city_name.capitalize()


def is_city_name_valid(city_name):
        if not city_name or len(str(city_name)) > 255 or not is_valid_dir(city_name):
                return False
        return True


    def is_valid_dir(city):
            return re.match(r'^[a-zA-Z0-9_\-\. ]+$', city) is not None
```

For the city parameter, the backend does not verify whether it corresponds to a real city. However, to ensure the city name can be used to create a valid folder, we use regular expressions to check the legality of the characters in the string. Additionally, the city name is processed with *capitalize()* to ensure consistent formatting with the first letter capitalized, improving readability and consistency for identical city names.

**File Integrity Check**

```
        #verify_hash
        frontend_hash = request.form["hash"]
        if not verify_hash(file, frontend_hash):
                return build_return_value(409, "Hash Verification Failed: Hashes do not match")
```

```python
def verify_hash(file, hash_from_frontend):
        hasher = hashlib.new("sha256")
        data = file.read()
        file.seek(0)
        hasher.update(data)
        digest = hasher.hexdigest()
        verified = hash_from_frontend==digest
        return verified
```

To ensure the file's integrity, the server calculates the SHA-256 hash of the received file and compares it with the hash parameter sent by the client. If the values match, the file is deemed intact and is ready to be saved.

**File Conflict Handling**

```python
if os.path.exists(filepath):
        with open(filepath, 'rb') as exist_file:
                if verify_hash(exist_file, frontend_hash): # same hash
                        return build_return_value(410, "Same file already exists")
                else:
                        return build_return_value(411, f"A {filename} file already exists in {city_name}. Please rename and upload again")
```

We also considered the possibility that users from the same city might upload files with identical names. This could lead to the new file overwriting the old one without the user being aware, which is clearly unacceptable. To address this, we introduced a file conflict handling step before saving files. This step is triggered whenever a file with the same name already exists in the directory for the specified city.

- If the hash values of the two files are identical, it indicates that the file contents are the same. In this case, the server will return an error, and a popup message will be displayed on the frontend to inform the user that the file already exists.
- If the hash values differ, it indicates a file name conflict. The server will prompt the user to rename the file and try uploading again.

This ensures that existing files are not unintentionally overwritten while providing clear feedback to the user in cases of conflict.

**File Saving**

The server uses the city name as the folder name to create the corresponding directory structure and saves the uploaded file within that path. Once the operation is completed, the server responds to the API caller with a success code.

## Files API for Viewing Uploaded Files

The files API does not require any parameters to be passed during the call. Upon invocation, the server will iterate through the directory where files are stored and return a dictionary. In this dictionary, the keys represent the folder names (city names), and the values are arrays containing the names of the files in each respective folder.

# Data Safety and Security Analysis

In a full deployment of this system, several data safety and security considerations could be addressed to improve user protection and system integrity. These considerations include:

- Transmit Encryption

Data transmitted between the user's browser and the website is vulnerable to interception by malicious actors. Sensitive information can be easily captured. Data integrity is also an issue. There is no guarantee that the data sent or received has not been tampered with during transmission, which can result in the delivery of incorrect or harmful information. To protect all data transferred between user and server, we can set up a TLS/SSL encryption. By configuring the HTTPS web, we can make sure the user is accessing the correct website and all data is encrypted during transfer.

- Access Control

Without access control, once the website is published, anyone on the internet can access the application, potentially leading to abusive and destructive behaviors. A simple yet effective solution is to implement HTTP Basic Access Authentication[1] to prevent unauthorized access to our application. This authentication method will require users to enter their username and password each time they attempt to access the website, but it does so without the need for complex backend and database support.

- File Hash Verification

The uploaded file may have been damaged during transmission. We have implemented a file checksum verification function that allows the front end to send the sha-256 checksum of the file to the server. This enables the server to verify whether the received file is complete and correct.

- Data Backup and Recovery

Data and files stored in the backend can be lost or damaged due to various factors, such as cybersecurity incidents, equipment damage, accidental actions, and more. The key to securing data is to have sufficient backups. We can implement regular backups of files both locally and in the cloud to prevent data loss and ensure that recovery processes are in place and regularly tested. We can follow the 3-2-1 backup strategy to ensure our data security[2]. This approach involves keeping three total copies of our data, storing two copies on different types of storage media, and maintaining one copy offsite. This method helps protect against data loss and enhances our overall data security.

- Monitoring and Security Audits

In security design, monitoring user and system activity is essential. A logging system can be implemented to maintain records of user activities, such as file uploads and location-based folder creations, allowing for the prompt detection and response to suspicious activities. By utilizing these logs, we can conduct periodic security audits and vulnerability assessments to identify and address potential security weaknesses in the system architecture.

# Testing Plan Design, Test Cases, and Results

The testing plan ensures the functionality and robustness of the file upload system, regardless of whether the location is successfully fetched in the browser. The focus is on verifying the system's behavior under various user scenarios, including different operating systems and user inputs. The following test cases were designed:

**File Upload Tests**:

- Verify successful file uploads when the target city location is successfully identified.
- Verify successful file uploads when the target city location is not properly identified, allowing users to manually input the city name.

**User Input Validation**:

- Verify the system's ability to handle various user inputs, such as city names with or without spaces, uppercase and lowercase, etc. *(Note: This version does not include checks for real city names)*

**File List Presentation:**

- Verify that uploaded files are correctly displayed in the user interface.

**Cross-Platform Compatibility**:

- Verify the system's functionality across multiple operating systems, such as Ubuntu and Windows.

**Edge Test:**

- Upload without a file

**Test Cases and Results**

**Test Date: Nov. 20, 2024**

| Functional Tests | Test content | Test Case | Expected Result | Result |
|---|---|---|---|---|
| File Upload | Test uploading files when the location of the device is successfully fetched on browser. Test with different file types (e.g., .txt, .jpg, .pdf) and sizes. | Browser: Chrome<br>Device OS: Ubuntu<br>Test file type: .txt/.jpg/.png/.mp4<br>File size: 13bytes / 46KB / 519KB / 27MB<br>Location for the device: ON<br>Location access in browser: Allowed ("Victoria") | File is saved under ./files/Victoria/. | PASS |
| | Test uploading files when location is not successfully fetched on browser. Test with different file types (e.g., .txt, .jpg, .pdf) and sizes. | Browser: Chrome<br>Device OS: Windows<br>Test file type: .txt/.jpg/.png/.mp4<br>File size: 13bytes / 46KB / 519KB / 27MB<br>Location for the device: ON<br>Location access in browser: Blocked | Show a manual input option: "Enter city name if location fails" | PASS |

| | | | | |
|---|---|---|---|---|
| | | Browser: Chrome<br>Device OS: Ubuntu<br>Test file type: .txt/.jpg/.png/.mp4<br>File size: 13bytes / 46KB / 519KB / 27MB<br>Location for the device: OFF<br>Location access in browser: Allowed | Show a manual input option: "Enter city name if location fails" | FAILED:Pop alert: City name illegal |
| | | Browser: Chrome<br>Device OS: Windows<br>Test file type: .jpg<br>File size: 73KB<br>Location for the device: OFF<br>Location access in browser: Allowed | File is saved under ./files/{city}/. | FAILED:The file name is not the city name (Victoria for this case), it shows "Oka Bay" |
| User input check | Input an illegal name (not a city name) when manually input the city name | Input: New Yourk | This project doesn't include real city name verification | PASS |
| | Input a legal name (city name) with space when manually input the city name | Input: New York | File is saved under ./files/New York/. | FAILED:Pop alert: City name illegal |
| | Input a legal name (city name) without space when manually input the city name | Input: Vancouver | File is saved under ./files/Vancouver/. | PASS |
| | Submit a file with the same name | With a file named text.txt under folder "Victoria", upload a new file named text.txt<br>The original text.txt has content "hello"<br>The second text.txt has content "hello again" | TBD | Text.txt is updated with the new content "hello again" |
| File List | Folder name correctly listed upon upload successfully | Browser: Chrome<br>Device OS: Ubuntu<br>Test file type: .txt/.jpg/.png/.mp4<br>File size: 13bytes / 46KB / 519KB / 27MB<br>Location for the device: ON<br>Location access in browser: Allowed ("Victoria") | UI shows Folder "Victoria" | PASS |
| | Files listed correctly under City Folder upon upload successfully | Browser: Chrome<br>Device OS: Ubuntu<br>Test file type: .txt/.jpg/.png/.mp4<br>File size: 13bytes / 46KB / 519KB / 27MB<br>Location for the device: ON<br>Location access in browser: Allowed ("Victoria") | All uploaded files are shown under folder "Victoria" | PASS |
| Edge Test | When no file is added, click upload | Click 'upload' without a file selected | Show warning: please select a file | PASS |

## Test Date: Nov. 26, 2024

| | | | | |
|---|---|---|---|---|
| User input check | Input a legal name (city name) with space when manually input the city name | Input: New York | File is saved under ./files/New York/. | PASS |
| | Submit a file with the same name | With a file named Rocky.jpg under folder "Victoria", upload a new file named Rocky.jpg | Alert: File exists, rename and upload again | PASS |

# Timeline, Resource Allocation, and Project Planning

Resource commitments in the form of time, effort, and output were split between group members of the project team with the aim of being approximately equal between all five members. The project took a total time to complete of approximately three months, with work on the code and app itself concentrated in the middle of that timeframe and work on the report and presentation slidedeck concentrated at the end. A breakdown of project responsibilities and output by group members follows:

## *Ricky Lin*

Application Development:

- Responsible for Frontend Development with regards to core file upload functionality, user interface and experience, and OpenStreetMap API integration.

Project Report, Slides, and Presentation:

- Wrote analysis on data safety and security. Responsible for formatting and style of presentation.

## *Chris McLaughlin*

Application Development:

- Responsible for implementation of all functionality related to file hash checking, as well as non-visual-design elements of listing uploaded files, on both frontend and backend.

Project Report, Slides, and Presentation:

- Wrote the introduction and tech and market overview section, as well as system design and architecture for both the report and presentation. Created system architecture flowchart. Did report formatting and styling and wrote the resource allocation section of the report.

## *Onkar Prakash Kadam*

Application Development:

- Responsible for initial exploratory investigation of file hash checking, attempted md5-based implementation which was later scrapped in favour of sha-256 checking due to depreciation by default javascript modules.

Project Report, Slides, and Presentation:

- Storyboarded and produced screenshots of code and functionality for the demonstration section of the report..

## *Xuqiao Jiang*

Application Development:

- Designed and implemented all application testing protocols, test cases, and manual functionality testing across all areas of the application.

Project Report, Slides, and Presentation:

- Wrote the report and presentation sections on testing plan, test case design, and testing results.

## *Zhang Zhang*

Application Development:

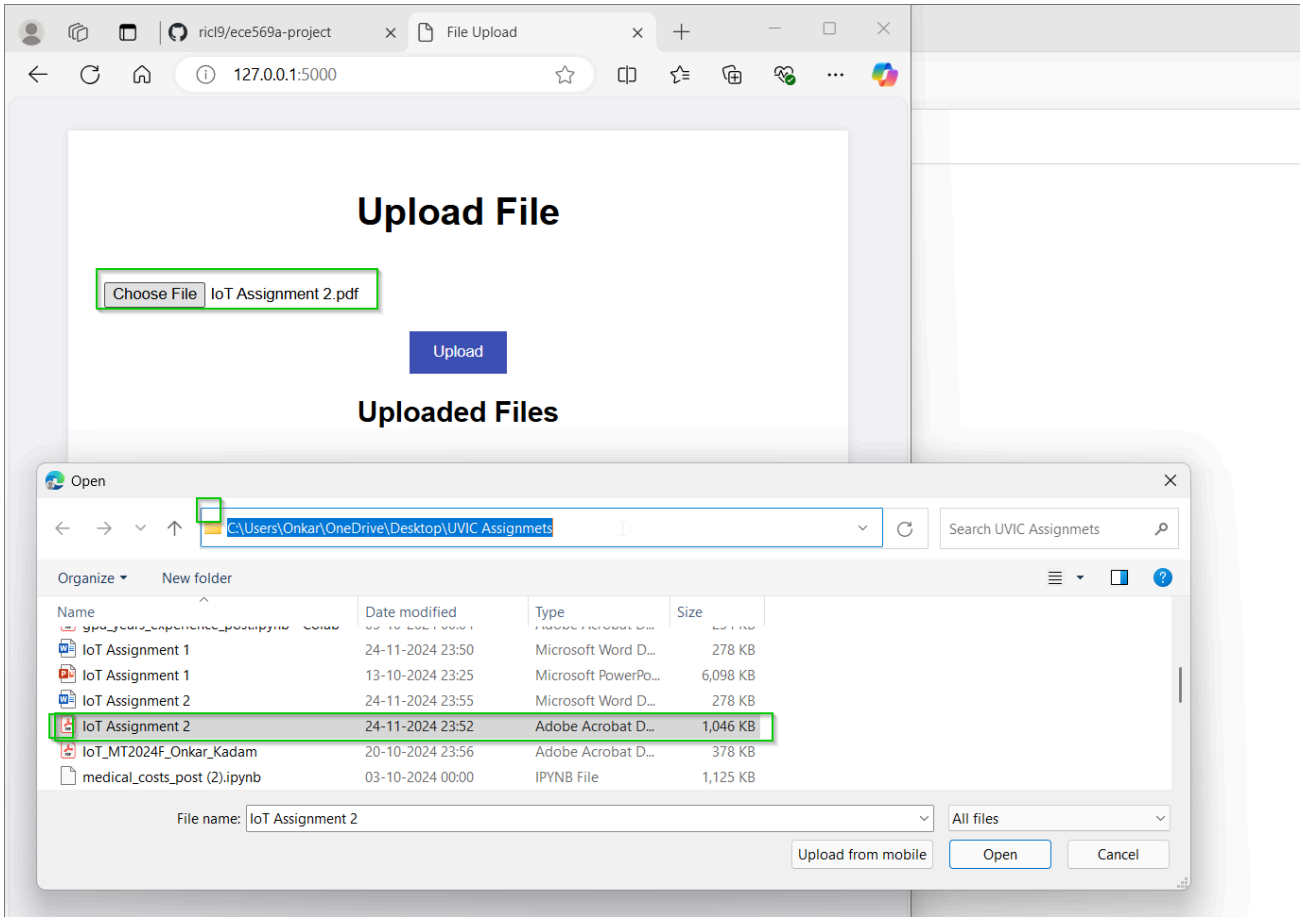
- Responsible for Backend Development with regards to server environment setup and deployment, core file upload and functionality, file system interfacing, file conflict handling, and sanity checking of city name strings.

Project Report, Slides, and Presentation:

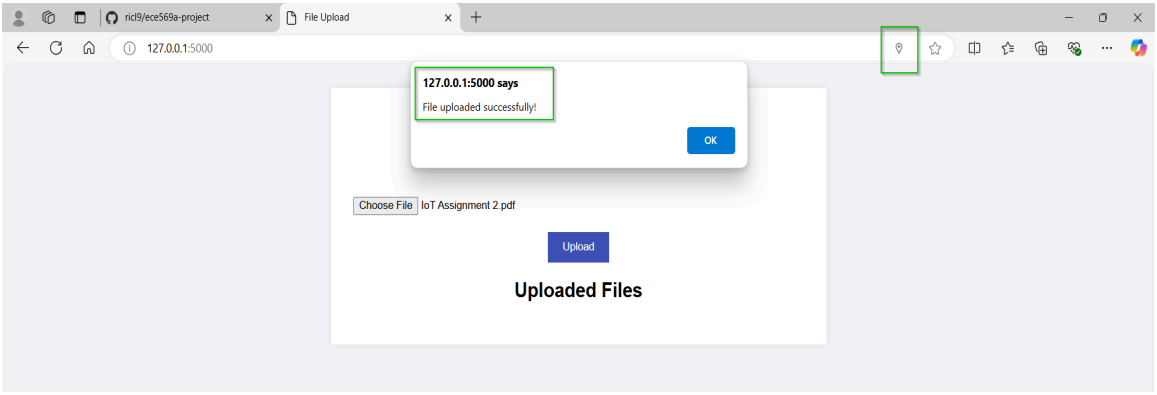
- Wrote the report and presentation sections on the system implementation approach.

# Demonstration

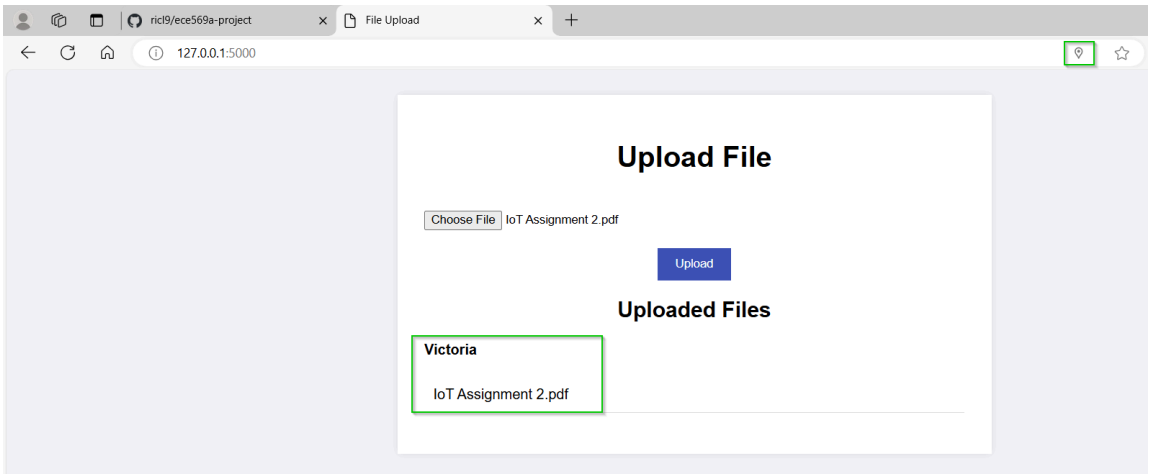
The following is a demonstration of the application working under normal operation, where the location is not blocked by any user-configured security policy:



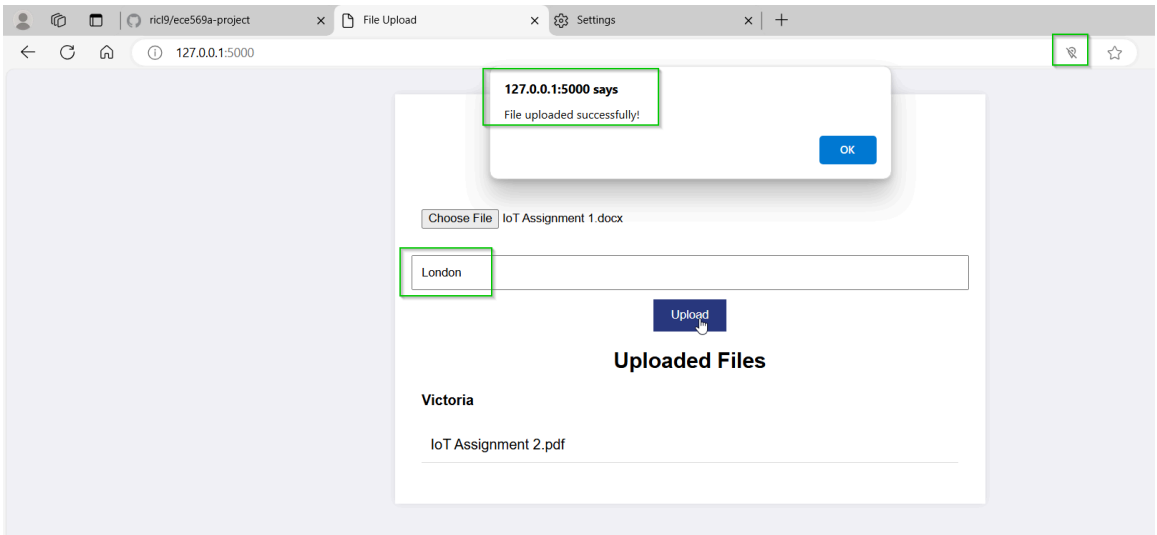

*The user selects a file they would like to upload and clicks the "Upload" button*




*If upload is successful, the user receives an alert informing them*




*When the DOM refreshes, a query is sent to the backend for all uploaded files, which are then displayed grouped by city on the frontend*




*If the user's location cannot be acquired automatically due to user security policy, we fall back to manual input of a city name, here we provide London*

# Summary and Insights

This project is a full-stack web application designed to enable users to upload files, which are organized and stored based on their geographical location. The application employs a Flask backend and client-side JavaScript frontend, and implements file integrity verification via SHA-256 hashing, as well as system status feedback in the form of a dynamically updated list of currently uploaded files.

The system demonstrates an emerging role in IoT-focused development aimed at a streamlined and minimalistic user interface, collecting in all cases the bare minimum manually-input information required from a user in order to perform its task.

Similar real-life use cases in IoT deployments might come in the form of geotagging enabled metadata on wearable devices or social media applications, which facilitate streamlined user uploading of desired information by minimising the amount of manual data input required in favour of dynamically aggregating information left out by the user.

While this project represents a demonstration of emergent IoT technology, it is in theory fully serviceable and expandable to real world IoT infrastructure needs. However, it is likely that we will continue to see dedicated IoT frameworks and development stacks emerge as the technology space separates from the existing embedded and web development spaces as IoT technology continues to evolve and deployments continue at ever expanding degrees of scale.

Additionally notable are hardware limitations. The project was limited due to its reliance on traditional hardware and subsequent externalities outside the scope and control of the system. On dedicated hardware in a real-life IoT deployment, additional development resources could be saved through the elimination of fallback paths required on non-standard setups, such as was encountered on this project, where the possibility of user-defined security policy interference necessitated a manual city input fallback.

# References

[1] “HTTP authentication - HTTP | MDN.” Accessed: Nov. 21, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

[2] Yev, “Backup Strategies: Why the 3-2-1 Backup Strategy is the Best,” Backblaze Blog | Cloud Storage & Cloud Backup. Accessed: Nov. 23, 2024. [Online]. Available: https://www.backblaze.com/blog/the-3-2-1-backup-strategy/

[3] “IoT Backend as a Service,” Back4App Blog, Mar. 28, 2021. https://blog.back4app.com/iot-backend-as-a-service/

[4] “Best practices for running an IoT backend on Google Cloud | Connected device architectures on Google Cloud,” Google Cloud. https://cloud.google.com/architecture/connected-devices/bps-running-iot-backend-securely