PROG7312: PORTFOLIO OF EVIDENCE

Fisokuhle Mkhize

Table of Contents

Introduction

In response to the increased demand for effective municipal service administration and community participation, this study describes the design and implementation of three interconnected aspects within a municipal service system. The method intends to improve communication between people and municipal officials while also increasing community participation in local activities. The three essential components - the Report Issue Feature, the Local Events Feature, and the Service Request Status Feature - work together to produce a comprehensive platform that solves critical challenges in municipal service delivery and community participation.

Task 1

The **Report Issue Feature** is designed to allow users to log issues in a structured and organized manner. It enables users to input details about an issue, such as its location, description, priority level, category, and an optional image attachment. The feature also provides real-time progress tracking, guiding users to complete all required fields before submission. By ensuring validation at each step, this feature enhances data integrity while making it simple for users to submit comprehensive reports. Once an issue is submitted, it is saved in the system for future tracking and resolution.

The **Report Issue Feature** was implemented using Windows Forms in C# with a modern Material Design theme provided by the MaterialSkin library. It integrates interactive UI components, real-time progress tracking, and validation to ensure a smooth user experience and accurate data capture. Below is an explanation of its core components and functionalities:

The form initializes with InitializeProgress, which sets up a MaterialProgressBar to track the completion status of the input fields. The progress bar ranges from 0 to 3, corresponding to the number of required fields. Every time a user interacts with the form, the UpdateProgress method evaluates the state of all fields and adjusts the progress bar accordingly. Once all fields are complete, a success message ("All input fields successfully entered") is displayed in green, offering immediate feedback.

The user interface employs MaterialTextBox controls for entering the location, description, and image path. The MaterialComboBox allows users to select the issue priority and category. A MaterialButton is used for uploading an image via a file dialog, enhancing the usability of the image attachment functionality. The modern design and intuitive layout provided by MaterialSkin ensure that the form is both functional and visually appealing.

The feature includes comprehensive validation to ensure the integrity of submitted data:

- **Text Fields**: Validation checks ensure that location, description, and image path fields are not empty.
- **ComboBoxes**: Users are required to select both a priority and category before submission.

- **Image Path**: The materialButton2_Click method validates the file path to ensure it exists and corresponds to an acceptable image format. Error messages are displayed using MessageBox.Show for any invalid input, making it easy for users to identify and resolve issues before submission.

When the user clicks the submit button, the system validates all input fields. If validation passes, the issue details are saved using the tracker.AddIssue method, which interacts with an IssueTracker class to manage logged issues. The total number of logged issues is then retrieved with tracker.GetIssues().Count and displayed in a message box. This provides immediate feedback on the submission's success and the system's growing repository of logged issues.

The image upload feature uses an OpenFileDialog to let users select an image from their local system. The file dialog is configured with filters to allow only specific image formats (e.g., .jpg, .png, .gif). Upon successful selection, the file path is displayed in the corresponding MaterialTextBox, ensuring the image is correctly linked to the issue.

The feature includes a return button (materialButton3_Click) to navigate back to the main form (Form1). The current form is disposed of upon navigation, efficiently managing system resources and ensuring a seamless transition for the user.

Task 2

In this task, a “Local Events” feature was developed to help users in the community discover and explore events set to happen, in addition to this, the feature categorizes events into Sports, Cultural and Educational activities, which as a result, enables users to filter events based on their interests and specific dates.

Upon successfully implementing this feature, users can are able to:
View All Events: They would be able to browse through pre-seeded events with details such as the name, date, category and location of these events.

Filter Events: Users are able to search for events by selecting a category and a specific date.

Get Recommendations: Receiving personalized event suggestions based on frequently searched categories.

Navigate back: Being able to return to the main form after exploring the events.

The **Local Events Feature** was implemented using a combination of backend logic and user interface elements in C#. At its core, the feature leverages a SortedDictionary<DateTime, List<Event>> to efficiently organize and manage event data by date. This structure allows for quick retrieval and filtering of events based on user-selected criteria. To populate the application with sample data, a SeedEvents method was created, which adds a variety of events categorized into **Sports**, **Cultural**, and **Educational** activities. This ensures users have a rich dataset to explore upon accessing the feature.

The user interface is built using Windows Forms, specifically leveraging modern Material Design principles with a MaterialForm base for a clean and professional appearance. Key components include a ListBox for displaying event details, a ComboBox for category selection, and a DateTimePicker for date filtering. Users interact with these controls to filter events dynamically. Buttons are provided for actions like initiating a search or navigating back to the main form, ensuring an intuitive and seamless user experience.

Search and filtering functionalities are at the heart of the Local Events Feature. The SearchButton_Click event handler processes user input to filter events by category and date, using LINQ to query the SortedDictionary. If the user opts to view all categories, the application displays all available events for the selected date. This dynamic filtering ensures users can quickly find relevant events without needing to browse through unrelated options, enhancing usability and efficiency.

To further improve user engagement, the feature incorporates a recommendation system using an instance of the EventsRecommendation class. This system tracks user interactions, such as frequently searched categories, and suggests relevant events based on this data. Recommendations are dynamically updated and displayed in a dedicated ListBox, providing users with personalized suggestions for upcoming events in their areas of interest.

Navigation is designed to be straightforward, with a dedicated ReturnBtn allowing users to return to the main form (Form1) effortlessly. This maintains continuity and ensures users can explore the Local Events Feature without feeling constrained or overwhelmed. By combining robust data management, an interactive interface, and a dynamic recommendation system, the Local Events Feature offers a comprehensive tool for users to discover and explore activities in their community.

Task 3

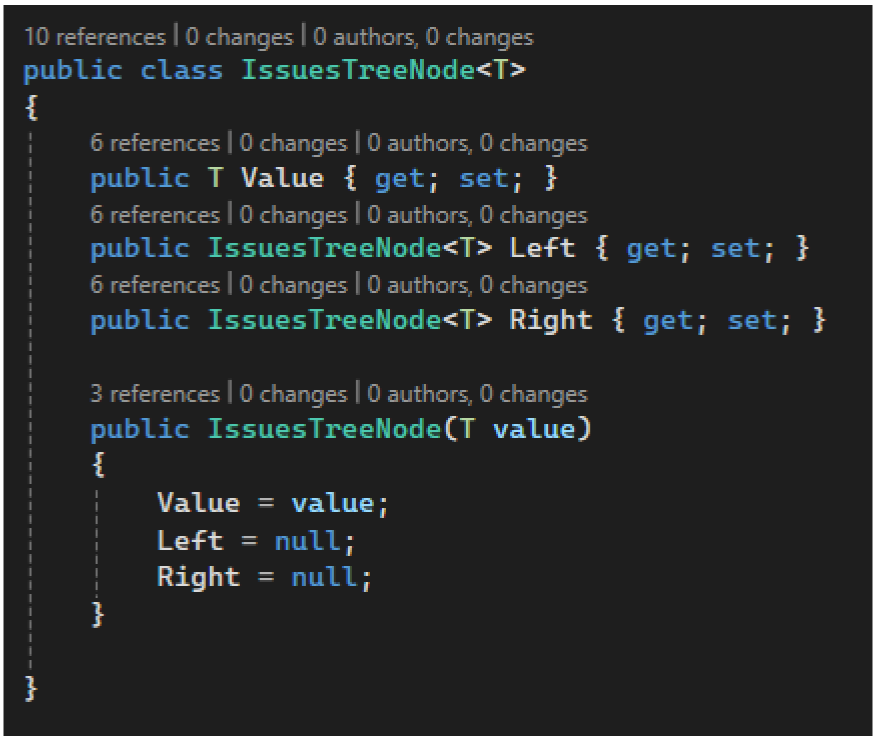
A **Binary Tree Data Structure** is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal (GeeksForGeeks, 2024).

Data Structures Analysis: Service Request Status Feature

1. Binary Search Tree (BST) Implementation

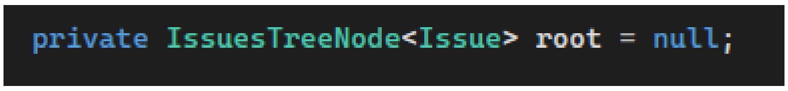
Core Structure

The system uses a custom Binary Search Tree implementation through the IssuesTreeNode<T> class, which serves as the primary data structure for storing and managing service requests.



```
10 references | 0 changes | 0 authors, 0 changes
public class IssuesTreeNode<T>
{
    6 references | 0 changes | 0 authors, 0 changes
    public T Value { get; set; }
    6 references | 0 changes | 0 authors, 0 changes
    public IssuesTreeNode<T> Left { get; set; }
    6 references | 0 changes | 0 authors, 0 changes
    public IssuesTreeNode<T> Right { get; set; }

    3 references | 0 changes | 0 authors, 0 changes
    public IssuesTreeNode(T value)
    {
        Value = value;
        Left = null;
        Right = null;
    }

}
```

Implementation Details



```
private IssuesTreeNode<Issue> root = null;
```

The Binary Search Tree (BST) implementation centres around a root node that maintains references to its left and right child nodes. Each node within the structure contains an Issue object and adheres to the fundamental BST property: the left subtree contains nodes with IDs less than the parent node's ID, while the right subtree contains nodes with IDs greater than the parent node's ID.

(GeeksForGeeks, 2022)into smaller BSTs at each level. It ensures that inorder traversal of the tree produces the keys in sorted order, making it a highly efficient structure for ordered data operations like search, insertion, and deletion (GeeksForGeeks, 2024).

Efficiency Contributions

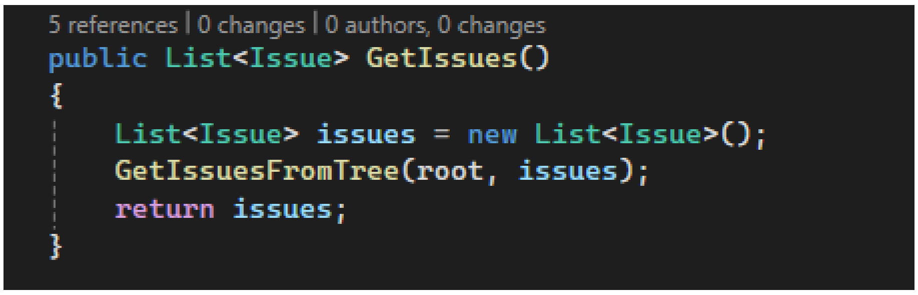
Insertion operations maintain an average case complexity of O(log n), where new issues are inserted based on their ID values. The recursive AddToIssuesTree method ensures proper placement within the structure while maintaining a balanced structure for efficient access. Similarly, search operations also achieve O(log n) average case complexity, with the FindIssue method implementing efficient binary search techniques. This approach is particularly effective for large datasets as it reduces the search space by half at each step of the process (GeeksForGeeks, 2024).

Insertion operations in a Binary Search Tree (BST) maintain an average case time complexity of O(log⁡n)O(\log n)O(logn), assuming a balanced tree structure. This efficiency stems from the BST property, which systematically reduces the search space for determining the correct placement of new nodes. When inserting a new issue, the **AddToIssuesTree** method follows these principles:

1. **Binary Decision Path**:
   During insertion, the algorithm checks the new issue's ID to the current node's ID. Depending on whether the new ID is smaller or larger, the algorithm moves to the left or right subtree. This binary decision cuts the search space in half at each level, which contributes to logarithmic efficiency in the average scenario. (GeeksForGeeks, 2024)
2. **Localized Changes**:
   Insertions only change the tree structure at one branch, leaving the rest of the tree unchanged. This localised operation minimises interruption while preserving the BST's efficiency for future searches and insertions (GeeksForGeeks, 2022).
3. **Efficient Memory Utilization**:
   Unlike array-based solutions, which may necessitate expensive resizing and shifting operations, a BST dynamically allocates memory only for the nodes being inserted. This characteristic makes it ideal for systems dealing with huge, dynamic datasets where scalability is critical (GeeksForGeeks, 2022).
4. **Scalability with Large Datasets**:
   For large datasets, the logarithmic insertion process significantly outperforms linear alternatives like unsorted lists or arrays. The structured approach reduces computational overhead, making BSTs ideal for programs where frequent insertions are coupled with real-time search requirements (GeeksforGeeks, 2024).

2. List Data Structure

Implementation



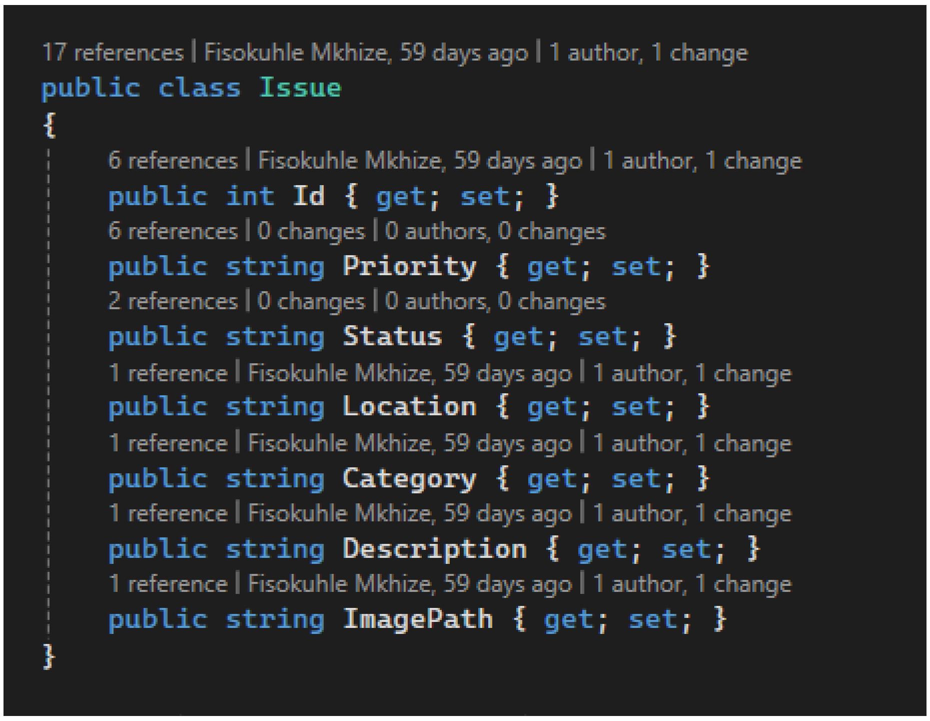
```
5 references | 0 changes | 0 authors, 0 changes
public List<Issue> GetIssues()
{
    List<Issue> issues = new List<Issue>();
    GetIssuesFromTree(root, issues);
    return issues;
}
```

Efficiency Contributions

Dynamic sizing is a key feature that automatically handles varying numbers of service requests. This eliminates the need for manual memory management and provides efficiency for iterations and LINQ operations. Additionally, the GetIssuesFromTree functionality performs in-order traversal, which maintains the sorted order of issues by ID. This traversal process has an O(n) time complexity for full traversal of the data structure.

3. Issue Class Structure

Implementation



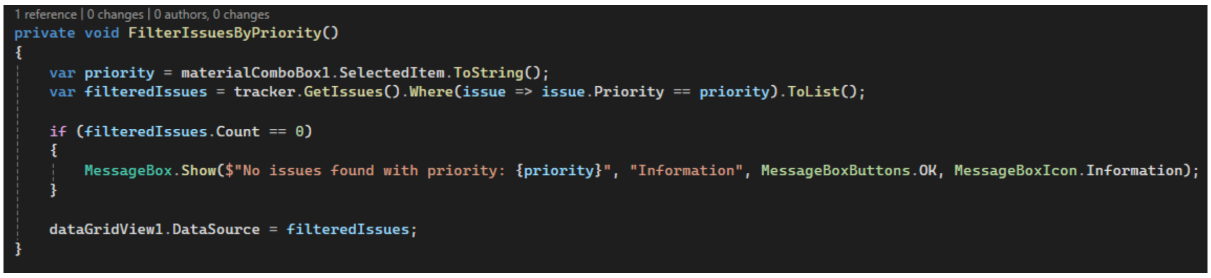
```
17 references | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
public class Issue
{
    6 references | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
    public int Id { get; set; }
    6 references | 0 changes | 0 authors, 0 changes
    public string Priority { get; set; }
    2 references | 0 changes | 0 authors, 0 changes
    public string Status { get; set; }
    1 reference | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
    public string Location { get; set; }
    1 reference | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
    public string Category { get; set; }
    1 reference | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
    public string Description { get; set; }
    1 reference | Fisokuhle Mkhize, 59 days ago | 1 author, 1 change
    public string ImagePath { get; set; }
}
```

Efficiency Contributions

Property encapsulation is implemented through auto-implemented properties that provide clean access to data. The system achieves efficient memory usage through proper data typing, while also facilitating easy serialization and deserialization of objects (GeeksForGeeks, 2022).

4. Filtering Mechanisms

Implementation

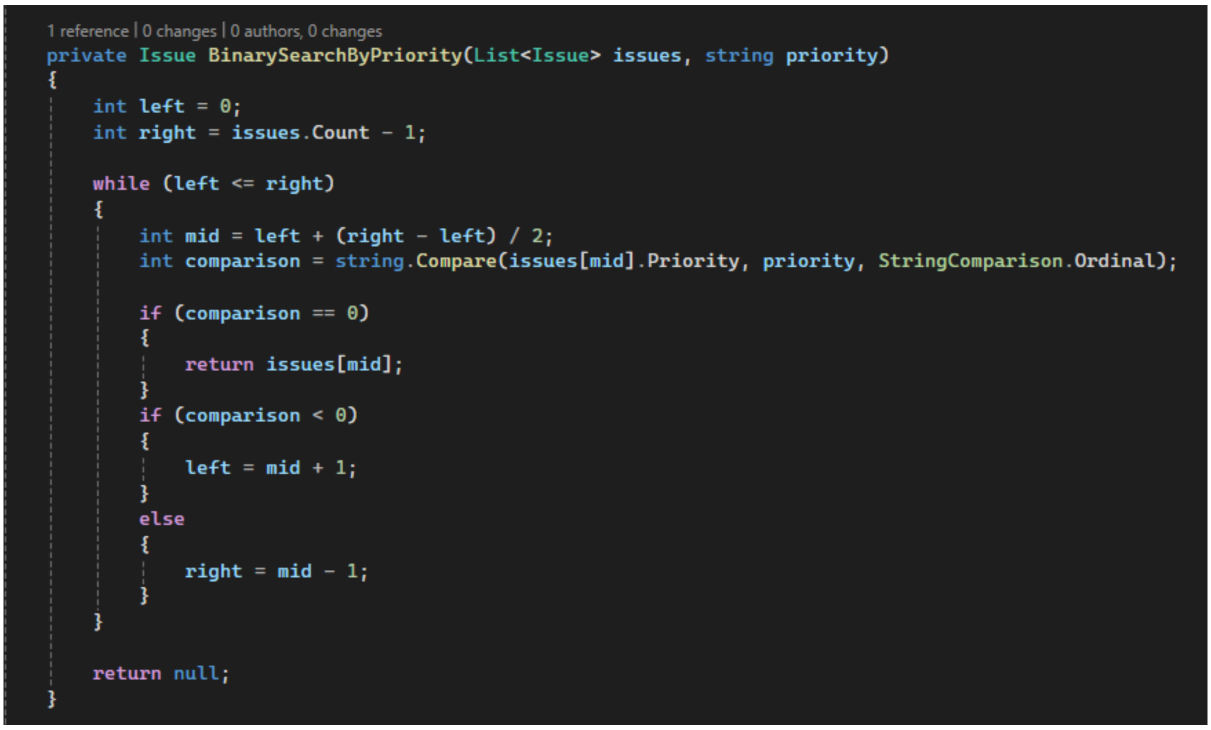


```
1 reference | 0 changes | 0 authors, 0 changes
private void FilterIssuesByPriority()
{
    var priority = materialComboBox1.SelectedItem.ToString();
    var filteredIssues = tracker.GetIssues().Where(issue => issue.Priority == priority).ToList();

    if (filteredIssues.Count == 0)
    {
        MessageBox.Show($"No issues found with priority: {priority}", "Information", MessageBoxButtons.OK, MessageBoxIcon.Information);
    }

    dataGridView1.DataSource = filteredIssues;
}
```

Efficiency Contributions

1. LINQ integration enables efficient filtering through specialized LINQ methods. Through lazy evaluation, the system prevents unnecessary iterations, while simultaneously maintaining code that is both readable and maintainable.

2. **Binary Search Implementation**



```
1 reference | 0 changes | 0 authors, 0 changes
private Issue BinarySearchByPriority(List<Issue> issues, string priority)
{
    int left = 0;
    int right = issues.Count - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        int comparison = string.Compare(issues[mid].Priority, priority, StringComparison.Ordinal);

        if (comparison == 0)
        {
            return issues[mid];
        }
        if (comparison < 0)
        {
            left = mid + 1;
        }
        else
        {
            right = mid - 1;
        }
    }

    return null;
}
```

The system implements efficient O(log n) search functionality for priority-based queries, specifically optimized for sorted data sets. This approach significantly reduces search time compared to traditional linear search methods (Bazmi, 2024).

From a performance perspective, the time complexity varies across different operations. Tree operations average O(log n), while list operations require O(n) for full traversal. Filtering operations on unsorted fields maintain O(n) complexity, whereas binary search operations on sorted fields achieve O(log n) efficiency. In terms of space complexity, both the BST and list storage require O(n) space for n issues, while temporary collections used in filtering occupy O(k) space, where k represents the size of the filtered result set (GeeksForGeeks, 2024).

The design offers substantial benefits in terms of scalability and maintainability. The BST structure enables efficient scaling as data volumes grow, with balanced tree operations maintaining consistent performance levels. Dynamic list allocation effectively handles varying data sizes. The system's maintainability is enhanced through clear separation of concerns, encapsulated data structures, and well-defined interfaces for all operations (Aslanyan, 2023).

Challenges faced

Task 1

During the creation of the LocalEventsForm, I ran across an issue with implementing a progress bar that updates dynamically dependent on the user's input. The problem developed when attempting to associate the progress bar's value with user activities such as selecting categories, filtering events, or conducting searches. The objective was to provide explicit criteria for progress updates and ensure that the progress bar reflected relevant milestones in the form's workflow. To remedy this, I included code that tracks user interactions and incrementally updates the progress bar's value based on completed activities. This solution necessitated careful management to guarantee that the progress bar would not exceed its bounds or reset improperly, thereby improving the user experience by providing a visible indicator of form completion.

Task 2

I experienced a number of technical hurdles while developing the Local Events functionality. One of the big challenges was building an efficient data structure to manage events, I used a SortedDictionary to keep events organised by date, but this necessitated careful handling of event filtering and sorting when users searched using different criteria. Another key problem was building and deploying a recommendation system that could track user preferences and suggest relevant events while remaining current and useful. Furthermore, delivering a smooth user experience by properly addressing edge circumstances, such as when no events met the search criteria, and maintaining synchronisation between event display and recommendation updates necessitated careful consideration of the UI flow.

Task 3

In Task 3, the goal was to make the data displayed in the Service Request Status Form dynamic. By enabling users to report issues, having the user display the issues they've logged as well as the progress in the Municipality addressing the issues would be important. After reworking Task 1, to migrate from storing the issues in a dictionary to utilizing a binary tree and binary search tree, this would enable users to store data, and in addition, they would be able to filter the data fast. Now the issue with this change was ensuring that the data persists in the runtime of the program. Initially, when trying to navigate to a form from the main menu, that would mean creating an instance of the called window, the issue with this is, each time a window is called, everything contained in it is a different instance, thus when calling something like the Report Issues window, I would be calling a new instance of the report issues window, thus everytime an issue is logged, the issue would be saved in a new instance of the IssueTracker class.

This was all addressed by creating instances of the IssueTracker class in the main form as a first step. With this, parameters for the constructors of each form would require the tracker instantiated in the main form and then the instance of the main form itself to ensure that 1 instance of the IssueTracker class persisted, and thus not resulting in data loss.

Technology Recommendations

At present, the application utilizes a basic in-memory data structure, specifically a SortedDictionary ,HashSet and a Binary Search Tree , for managing event data. While this approach is sufficient for small-scale applications, it may become inefficient when scaling to accommodate larger datasets or when persistent data storage is required. In such cases, a more robust solution is needed for optimal performance, data integrity, and scalability (GeeksforGeeks, 2024).

Recommendation: To address these issues, consider switching to a relational database system, such as SQL Server or SQLite, and using Entity Framework Core for data persistence. This will allow for more efficient data storage, querying, and management of relationships between events, categories, and other relevant entities. Using a relational database, the application will be better suited to manage enormous amounts of events.

provide a more scalable and maintainable infrastructure, enabling smoother integration with other systems or applications as the need for expansion arises (GeeksforGeeks, 2024).

Conclusion

The incorporation of these three aspects marks a big step forward in municipal service administration and community participation. The Report Issue Feature's organised approach to issue tracking, along with a material design interface and extensive validation, assures accurate data capture and user delight. The Local Events Feature's sophisticated filtering and recommendation mechanism illustrates the potential for increased community participation through personalised information distribution. The Service Request Status Feature's effective usage of binary search trees demonstrates how careful data structure design may have a major impact on system performance and scalability.

References

Aslanyan, V., 2023. *Data Structures Handbook – The Key to Scalable Software*. [Online]
Available at: https://www.freecodecamp.org/news/data-structures-the-key-to-scalable-software/
[Accessed 18 11 2024].

Bazmi, M., 2024. *O(log N) Algorithm Example*. [Online]
Available at: https://tutorials.eu/olog-n-algorithm-example/
[Accessed 18 11 2024].

GeeksForGeeks, 2022. *Complexity of different operations in Binary tree, Binary Search Tree and AVL tree*. [Online]
Available at: https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/
[Accessed 18 11 2024].

GeeksforGeeks, 2024. *Advantages and Disadvantages of SQL*. [Online]
Available at: https://www.geeksforgeeks.org/advantages-and-disadvantages-of-sql/
[Accessed 18 11 2024].

GeeksForGeeks, 2024. *Binary Tree Data Structure*. [Online]
Available at: https://www.geeksforgeeks.org/binary-tree-data-structure/
[Accessed 17 11 2024].