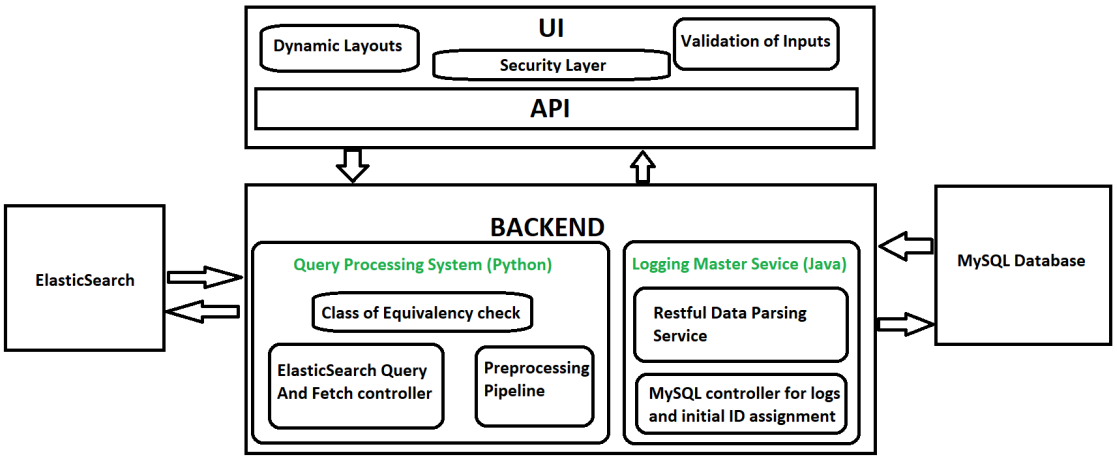
Architectural Document

Introduction

Brief overview of the project

The Recipe Finder web application simplifies discovering and accessing a wide array of recipes, tailored to users' preferences and ingredient availability. With unique features designed to cater specifically to user needs, it offers a user-friendly interface for exploring, filtering, and selecting recipes that align with individual tastes.

Fig1. Communication flow diagram



Purpose of the architectural document

The architectural document is a crucial blueprint for the successful execution of a software project. It outlines the system's structure, components, interactions, and behaviors from a technical viewpoint. By facilitating clear communication, offering design insight, setting guidelines, aiding decision-making, and documenting the architecture, this document significantly contributes to the project’s success. It ensures that every team member has a clear understanding of their role and the system's overall framework.

Coding Guidelines

Python

Framework

- Utilize the framework for Python: Flask Documentation

PEP 8 - Style Guide for Python Code

- **Line Length:** Limit lines to a maximum of 79 characters to enhance readability.
- **Blank Lines:** Utilize blank lines strategically to demarcate logical sections within your code.
- **Indentation:** Adhere to an indentation level that consists of 4 spaces.
- **Import Formatting:** Import statements should be placed at the top of the file, after module-level docstrings and file comments.
- **Naming Conventions:** Follow consistent naming conventions, file names all lowercase, for variables, functions, and classes, all while using meaningful and descriptive names.

Indentation Rule

- Indent code blocks with four spaces.
- Ensure all Python code adheres to correct indentation practices.
- For more guidance, refer to : PEP 8 – Style Guide for Python Code

Comments and Documentation

- **PyDoc Comments:** Use PyDocs to write detailed explanations of code blocks, functions, classes, and modules. Figure 1 demonstrates a PyDoc comment used to explain the `upload()` function within a Flask route. PyDocs should be placed

directly beneath the function or class signature and enclosed in triple-double quotes.

- **In-Line Comments:** Write comments to the right of specific lines only when referring to complex logic. Basic functionality should be documented in the PyDocs, as shown in Figure 1 where the function's purpose and usage are described within a PyDoc string.
- **Automated Documentation:** By using PyDocs comments it facilitates automated documentation generation, providing a structured approach to understanding the code base. The code excerpt in Figure 1 serves as an example of how to effectively document a Flask route handling file uploads.

```python
@app.route('/upload', methods=['POST', 'GET'])
def upload():

    """
    upload(): uses the upload.html to create a drag and drop area that allows a user to upload
    their file into the app directory. This gives us the capability to acces data and parameters
    to ultimately send.
    """
    if request.method == 'POST':
        f = request.files.get('file')
        username = session.get('username')
        subfolder_path = os.path.join(app.config['UPLOADED_PATH'], username)
        os.makedirs(subfolder_path, exist_ok=True)
        file_path = os.path.join(subfolder_path, f.filename)
        f.save(file_path)
        upload_file(file_path)

    return render_template('upload.html')
```

Figure 2: Example of PyDoc Usage for the upload Function in a Flask Application

Use Virtual Environments

- Use virtual environments to create isolated Python environments for each project, ensuring dependencies and package versions are managed separately.
- **Docker Integration:** Utilize containerized platforms like Docker for consistent deployment environments and easier collaboration among development teams.

Error Handling

- Implement comprehensive error handling mechanisms throughout the project.
- Provide clear and informative error messages and logs for an improved user experience and developer efficiency.
- **User Redirection:** In the event of an error, redirect users to appropriate pages, such as the index web page, while logging the error details.
- **Testing:** Rigorously test error handling functionality to identify and address potential issues, ensuring robustness and reliability.

Javascript

Framework

The application is built using **React** to create a modular and interactive front-end experience. React's component-based architecture promotes maintainability and allows for easy updates.

Naming Conventions

Maintaining a consistent naming convention is crucial for code readability and reducing errors caused by naming inconsistencies. The following guidelines outline best practices for naming variables, functions, classes, and other code elements:

Variables and Functions: Use camelCase for variable and function names. This convention uses a lowercase first letter with uppercase letters for new words within the name. For example, recipeList, fetchUserData, handleRecipeSearch.

Classes: Use PascalCase for class names. This convention uses uppercase letters for the start of each word. For example, RecipeFinder, IngredientValidator.

Boolean Variables: Start boolean variable names with "is" to indicate their true/false nature. This makes the purpose of the variable clear immediately. For example, isUserAuthenticated, isIngredientValid.

Functions: Functions that perform specific actions or handle events should have descriptive names. Include a verb that indicates the action or event they handle. For example, handleUserLogin, validateUserInput.

Avoid Magic Numbers: Avoid using hard coded numbers within the code. Instead, define constants with descriptive names. This makes the code easier to understand and maintain. For example, instead of if (x > 10), use const MAX_LIMIT = 10; if (x > MAX_LIMIT).

Consistent Naming for Related Elements: When naming related elements, maintain a consistent pattern to indicate their relationship. For example, for CRUD operations, use create, read, update, delete.

Descriptive Names: Ensure that all names are descriptive and convey the purpose or behavior of the variable, function, or class.

Comments

Comments enhance code readability, improves team collaboration, and makes maintenance easier. Below are coding practices for writing clear and effective inline documentation in this React Project:

Use JSDoc for Function and Component Comments: The purpose of functions, components, and their parameters and return values. This documentation is typically placed above the relevant code section to maintain clarity and context. An example of a React functional component documented with JSDoc:

```
/**
 * A Button component that triggers a callback when clicked.
```

```
 *
 * @param {Object} props - The component properties.
 * @param {string} props.label - The text to display on the button.
 * @param {Function} props.onClick - The function to call when the button is clicked.
 * @returns {JSX.Element} The rendered button element.
 */
const Button = ({ label, onClick }) => {
  return <button onClick={onClick}>{label}</button>;
};
```

Key Aspects

The @param tag describes the component's properties (props).

The @returns tag defines the expected return type of the function or component.

The description above the code provides a clear overview of the component's purpose and its expected behavior.

UI Design Guidelines

Consistency

The project utilizes Tailwind CSS to ensure a consistent look and feel across all components. This framework maintains uniformity in design elements like buttons, forms, and layout throughout the application.

Responsiveness

The user interface is designed to be responsive, adapting to various screen sizes and devices, from desktops to mobile phones. This ensures a consistent user experience across platforms. Media queries and CSS grids are used to achieve this responsiveness.

Accessibility

To meet accessibility standards, the project adheres to ISO/IEC 40500:2012 and the Web Content Accessibility Guidelines (WCAG) 2.0. Key features include:

- Alternative Text for Images: All images contain descriptive alt text to support screen readers and visually impaired users.
- Keyboard Accessibility: All functions are operable via keyboard, supporting users with mobility impairments.
- Contrast and Color Use: The color scheme uses high-contrast elements and avoids color combinations that could be problematic for users with color blindness.
- Headings and Structure: Proper heading tags (h1, h2, h3) and semantic HTML elements ensure a logical and navigable structure.
- Focus Indicators: Interactive elements, such as links and buttons, have visible focus indicators for keyboard navigation.

Navigation

The application features intuitive navigation with clear menus and a search function for ease of use. This design choice reduces user confusion and streamlines interactions. The key component of the navigation system is the top-level menu that provides links to primary sections of the application, such as Home, Profile, and Login. It remains visible on all pages, providing easy access to the main sections.

Feedback

To enhance user interaction and provide immediate responses to user inputs, the application incorporates various feedback mechanisms, including:

- Visual Cues: Buttons and interactive elements provide visual feedback on hover, click, or focus. This feedback may include color changes, shadows, or animations.
- Loading Indicators: When an action requires loading or processing time, a spinner or progress bar indicates that the system is working on the request.

- Error Messages: Clear and consistent error messages guide users in correcting mistakes. Errors are displayed in a prominent but non-intrusive manner, often in red text or with an alert icon.

Performance

To ensure optimal performance, the application minimizes load times and prioritizes essential content. The application is optimized for performance through several key techniques:

- Code Splitting: Code is divided into smaller chunks, allowing only the necessary parts to load, reducing initial load times.
- Lazy Loading: Resources, such as images and additional scripts, are loaded on demand, minimizing the initial payload and improving performance.
- Caching: The application uses caching strategies to store frequently accessed data, reducing the need to fetch it from the server repeatedly.
- Minification and Compression: JavaScript and CSS files are minified and compressed to reduce file sizes, speeding up page load times.

Error Handling

Proper error handling is crucial to maintain a smooth user experience and avoid unexpected application crashes. The following guidelines outline best practices for managing errors within the application:

- Exception Handling: Implement try...catch blocks to capture and manage exceptions at critical points in the code. This prevents application crashes and provides an opportunity to display meaningful error messages to users.
- Uniform Error Messages: Ensure error messages are consistent throughout the application. They should be clear, concise, and helpful to users, guiding them on how to resolve the issue.

- Logging: Use a centralized logging system to record exceptions and errors. This aids in debugging and troubleshooting by providing a historical record of errors. Send logging data to Java Server for a centralized logging system.
- User-Friendly Feedback: When displaying error messages to users, use non-technical language and provide guidance on how to correct the error.
- Boundary Checks: Validate user inputs and check for edge cases to prevent common errors, such as array out-of-bounds or division by zero.
- Client-Side Validation: Validate user inputs on the client side before sending them to the server to avoid unnecessary server requests and reduce the load on backend resources.
- Security Considerations: When handling errors, ensure that no sensitive information is disclosed. This is particularly important when providing feedback to users, to avoid revealing details that could be exploited for security breaches.

Security

- **Input Validation:** Validate user inputs to prevent malicious attacks and ensure data integrity.
- **Secure Data Handling:** Employ HTTPS for secure data transfer and sanitize input data to prevent injection attacks.
- **Client-Side Validation:** Implement client-side validation using JavaScript to check user inputs before sending them to the server. This can help in preventing unnecessary server requests and providing immediate feedback to users.
- **Regular Expressions:** Use regular expressions (regex) to define and enforce specific patterns for user inputs, such as email addresses, passwords, and phone numbers, to validate and sanitize data effectively.
- **Input Sanitization:** Sanitize user inputs by removing or escaping potentially malicious characters, HTML tags, and scripts to prevent injection attacks and protect against Cross-Site Scripting (XSS) vulnerabilities.

- **Cross-Origin Resource Sharing (CORS):** Implement CORS policies to restrict and control the domains and origins that can access and interact with your web application's resources, reducing the risk of unauthorized access and data exposure.

File Structure

This section provides a high-level overview of the file structure for a React-based website's client-side. It explains the purpose of various directories and files, providing a guide for developers to understand the codebase and follow best practices. Inside the root_folder of the project:

src/

This is the main source code directory. It contains all the React components, pages, styles, assets, and other necessary code files for the website.

- **components/:** This directory contains reusable React components. Components are organized by feature or purpose. For example, Button.js, Navbar.js, etc.
- **pages/:** This directory contains individual page components. Each page represents a specific route or view in the website.
- **hooks/:** Custom React hooks are stored here. Hooks are reusable logic that can be shared across different components.
- **assets/:** This directory contains static assets like images, fonts, and other media files.
- **stubs/:** This directory is designed to contain stub functions that simulate API calls to the backend. These functions serve as stand-ins for real network calls during development and testing. They are to be used in scenarios where the backend is not yet available or when you want to test frontend components without actual backend dependencies.
- **tests/:** This directory holds all test-related resources and code. It contains two main folders, unit/ and integration/.

    - **unit/**: This folder contains unit tests for individual React components or functions. The purpose is to test specific logic or functionality without considering external dependencies.
    - **integration/:** This folder contains integration tests, which test the interaction between multiple components or modules.
- **StateContext.js:** This file contains React context providers for global state management, such as authentication, user data, etc.
- **utils.js:** Utility functions and helper scripts are stored here for common tasks like data formatting, validation, etc.
- **App.js:** The root React component that defines the main structure of the application.
- **index.js:** The entry point of the React application, where the React DOM is rendered.
- **Styles.css:** This file holds global CSS. This can help enforce consistent design and theming across the application.

public/

This directory contains public assets that do not require Webpack processing. These files are served as-is by the web server.

- index.html: The main HTML file where the React application is injected.
- favicon.ico: The website's favicon.
- manifest.json: The web app manifest for Progressive Web Apps (PWAs).
- robots.txt: The file that provides instructions to web crawlers.

Mocks

The React_Mocks branch on GitLab provides code and stub functions designed for client-side testing in the recipe finder system. The goal is to simulate different scenarios to ensure the system behaves correctly under various conditions. Stubs are used to mimic server responses, allowing developers to test functionality without relying on a live backend. This ensures faster and more isolated testing.

Stub Functions

Here is an overview of the stubs included in the project and their purposes:

fetchRecipesByIngredientsStub:

This stub simulates a successful response when querying recipes based on a list of ingredients. It returns an array of recipe objects containing sample data including recipe names, descriptions, and other relevant information. This stub is useful for testing the core functionality of the application, ensuring that it retrieves and displays recipes as expected when given valid input.

fetchRecipesInvalidStub:

This stub mimics a scenario where the input ingredient list is invalid such as “laptop, chair”. It returns an empty array along with an error message indicating the issue. This stub is helpful for testing error handling and ensuring that the application properly communicates validation errors to the user.

fetchRecipesNotFoundStub:

This stub represents a scenario where no recipes match the given ingredients. It returns an empty array with an error message stating that no recipes were found. This is ideal for testing how the application responds to empty search results and displays appropriate feedback to the user.

fetchRecipesWithFiltersStub:

This stub simulates a successful response when applying additional filters (such as dietary restrictions or cuisine types) to a recipe search. It returns an array of recipes that meet the specified criteria without errors. This stub is valuable for testing more complex queries and ensuring the application supports a variety of filters.

How to Use the Stubs

To use these stubs in your testing environment:

Import the Stubs: Import the desired stubs from the root_folder/src/stubs directory into your test file. For example, to import fetchRecipesByIngredientsStub, you would use the following import statement:

```
import { fetchRecipesByIngredientsStub } from '../stubs;
```

Mock the API Calls:

Use a mocking library, Jest, to replace actual API calls with the corresponding stub. This way, when the component or function under test makes a call to the mocked endpoint, it will use the stub's data instead of reaching out to a real backend.

Testing

Jest Configuration: Jest is configured using a jest.config.js file at the root of the project. This file contains settings like test environment, coverage thresholds, and module resolution.

Test Files Location: Jest automatically looks for test files with .test.js extension. These files are located in the tests/ directory, with unit/ and integration/ subfolders.

Using Stubs in Jest Tests:

In your Jest test files, import the stub functions and use them to simulate API responses. You can mock API calls with jest.fn( ) function , replacing the actual API call with your stub.

```
jest.mock("relative/path/to/api", () => ({
  fetchUser: jest.fn(() => Promise.resolve(getUser())),
}));

test("renders user information", async () => {
  render(<MyComponent />);
}
```

Running Tests

To run the tests in the React project, use the following command in the terminal at the project root:

```
npm test
```

- Watch Mode: Run tests continuously during development by adding --watch to the command.
- Code Coverage: To generate a code coverage report, use --coverage with the test command.

Setting Up the Environment:

To establish a functional development environment for a React project, it is crucial to have the right tools and frameworks in place. This guide provides a step-by-step outline for setting up a React project, from installing Node.js to starting the development server.

Step 1: Install Node.js and npm

Node.js is a JavaScript runtime that allows you to execute JavaScript code outside of a web browser, while npm (Node Package Manager) is used to manage project dependencies.

Download Node.js:

- Visit the Node.js website.
- Download the latest Long-Term Support (LTS) version.
- Install Node.js on your system following the installation instructions for your operating system.

Step 2: Clone the GitLab Repository

To get started with the project, clone the repository from GitLab to your local development environment.

Clone the Repository:

Open a terminal or command prompt.

Run the following command to clone the repository:

```
git clone https://gitlab.com/csuncompsci/2024-spring-recipe-governor-s1.git
```

Change into the newly created project directory:

```
cd 2024-spring-recipe-governor-s1/recipe-client
```

Step 3: Install Node Modules

With the repository cloned, you need to install the project's required node modules. This step ensures that all dependencies specified in package.json are installed.

Install Node Modules:

In the terminal, run the following command at the project root to install the necessary packages:

```
npm install
```

Step 4: Start the Development Server

Once all dependencies are installed, you can start the development server to run the React application.

Start the Server:

In the terminal, run the following command at the project root:

```
npm start
```

This command starts the development server on port 3000.

Access the Application:

Open a web browser and navigate to http://localhost:3000.

The React application should be running on the machine.

Java

Reference Document: JAVA Architectural Document

Naming Conventions

- For readability use clear, consistent, concise, meaningful names
- Use underscores in lengthy numeric literals
- Use of single quotes and double quotes for static values based on datatype

Security

- Ensure class members are private for better encapsulation.
- Implement thorough input validation and sanitization mechanisms to mitigate injection attacks such as SQL injection, Cross-Site Scripting (XSS), and command injection. Utilize techniques such as whitelisting, regular expressions, and input parameter validation to ensure the integrity of user inputs.
- Safeguard data transmission over networks by employing secure communication protocols such as HTTPS/TLS. Encrypt sensitive data in transit to prevent eavesdropping and man-in-the-middle attacks, and adhere to best practices for encryption key management.
- Implement secure session management techniques to prevent session-related vulnerabilities such as session hijacking or fixation. Utilize secure session tokens, enforce session expiration policies, and employ techniques to regenerate session identifiers upon authentication.
- Avoid hardcoding sensitive information within the codebase and adopt secure configuration management practices. Store sensitive data, such as passwords and API keys, securely outside the application source code, utilizing encrypted configuration files or secure key management systems.
- Implement robust error handling mechanisms to gracefully manage unexpected errors and exceptions. Ensure that error messages do not expose sensitive information and follow secure logging practices to record relevant security events while avoiding logging of sensitive data.

Memory Management

- Prevent memory leaks.
- Avoid redundant initializations and unnecessary object creation.
- Use `StringBuilder` or `StringBuffer` for string concatenation.

Code Efficiency

- Maintain loose coupling between services.
- Opt for enhanced for-loops over traditional for-loops with a counter.
- Enhance code reusability by creating multiple methods for each functionality.

Error Handling

- Implement proper exception handling, especially for `NullPointerExceptions`.
- Avoid empty catch blocks and return empty collections or use `Optional` instead of `null`.

Logging

- Document each method with proper JavaDoc.
- Log informational messages and errors adequately.

Testing

- Utilize JUnit for executing tests to ensure expected behavior.
- Write comprehensive unit tests for maximum code coverage.

Technical functionalities

- Use lambda expressions and streams for efficient collection processing.
- Leverage Spring Boot for service and API endpoint creation.
- Utilize JSON for API request and response formats.
- Employ Spring WebClient or RestTemplate for external API calls.
- Return appropriate response codes when results are not found.
- Manage application configurations for default values and connection details.
- Follow a microservices architecture pattern.

- Implement in-memory caching where necessary.
- Containerize the service with Docker for deployment.

Comments and Documentation

- Establish guidelines for writing meaningful comments.
- Set documentation standards for code, including detailed function and module descriptions.
- Ensure thorough documentation of APIs or libraries, covering functions, classes, modules, parameters, return types, exceptions, and other relevant information.
- Use `TODO` comments sparingly to mark areas needing future attention, and regularly address these points.
- Reserve inline comments for explaining the rationale behind certain code segments.
- Begin each function or complex logic block with block comments to describe its purpose, inputs, outputs, side effects, exceptions, and any algorithms or logic used.

Best Practices

The following general guidelines aim to ensure the project meets delivery goals and achieves success.

Code Quality

- **Consistency:** Adhere to coding style guides (e.g., PEP 8 for Python, Google's Java Style Guide for Java).
- **Clarity:** Use meaningful variable names and comments for complex logic.
- **Modularity:** Break down functions into manageable pieces; refactor for reusability.
- **Optimization:** Avoid duplication; utilize whitespace and indentation effectively.

Version Control

- **Goal:** Create a master branch as the main, with each task assigned to a group. Each group should have a main task branch that merges into master. Use personal

branches for individual task work, merging progress into the group's main task branch. This ensures tested code and maintains quality and consistency in the master branch.

- **Strategy:** Use names as feature, bug, etc. to describe branch/task.
    - Name main group branches by `group_feature_xxxx`
        - Ex. `hulks_feature_loginFuncionality`
    - Name personal branch by `name_bug_xxxxx`
        - Ex. `Grant_bug_fixedAuth`
- **Commit Messages:** Create descriptive messages for each commit.
- **Integration:** Regularly sync feature branches with the master to avoid conflicts.
- **Automation:** Implement Git hooks for tasks like linting and tests.

Continuous Integration and Deployment (CI/CD)

- **Docker:** Containerize applications using Docker to ensure consistency across different environments and facilitate easier deployment.
- **Automation:** Utilize outlined Git practices to facilitate as close to automated processes as possible.

Documentation

- Write comprehensive documentation covering installation instructions, usage guidelines, and API references.
- Utilize PyDocs to generate documentation directly from source code comments.
- Keep documentation up-to-date with the codebase by incorporating it into the development workflow.
- Provide examples and use cases to illustrate how to use different parts of the codebase effectively.

Test-Driven Development (TDD)

- Implement integration tests to verify interactions between different components or services.
- Monitor test coverage metrics and aim for high coverage to ensure code reliability and maintainability.

Code Reviews

- Only team managers will be allowed to merge into main branch, they will review groups code to be merged into main group branch as well
- Encourage constructive feedback and discussions during code reviews to promote knowledge sharing and collaboration.
- Rotate code review responsibilities among team members to distribute knowledge and ensure accountability.

Use All Available Resources

- Engage with the developer community through forums, blogs, or social media platforms to learn from others' experiences and best practices.
- Participate in open-source projects or contribute to libraries and frameworks relevant to your project's technology stack.
- Leverage online resources such as documentation, tutorials, and code repositories to accelerate development and troubleshoot issues.
- Foster a culture of innovation and experimentation within the team by encouraging exploration of new tools and approaches.

Specific best practices relevant to the project

These are guidelines to help developers meet project specifications.

Design

- When creating the UI, make sure to refer to the Figma design, as this will help make sure all front-end developers are on the same page regarding what users should see when using the recipe app.
- The design of the recipe app can be reviewed here: Food Web Recipe - Figma

Tools

Familiarity with the project's technology stack is essential for efficient task execution. Developers should acquaint themselves with the relevant tools prior to commencing work on the project. The toolset varies depending on the specific role:

- **Front-end Developers:** Mastery of ReactJS is required for creating dynamic user interfaces.
- **Back-end Developers:** Proficiency in Node.js, Express.js, MongoDB, and Firebase is necessary for server-side and database management.
- **Full-stack Developers:** A comprehensive understanding of both front-end and back-end technologies is beneficial.

Common tools essential for all developers include:

- **Version Control:** Platforms like Trello, GitHub, and GitLab are integral for tracking changes and managing collaborative efforts.
- **Integrated Development Environments (IDEs):** Tools such as Visual Studio Code are recommended for their versatility and support for multiple languages and frameworks.

Research API

- Research and understand the responses from the Edaman recipe API call (specific queries and responses). Find out what information this API gives and what format the information is provided in.
- The API will only be utilized to scrape data about recipes and to store as indices in the elasticsearch.

Communication

- Considering the team's size, it's advisable to communicate information and issues effectively. Managers should relay relevant details and issues from their group to other managers. These managers, in turn, should pass this information and issues to members of their group who would find it pertinent.

Coding Languages

- **JavaScript:** This is the primary programming language used in your project. React JS is a library built on JavaScript, enabling developers to create user interfaces.
- **JSX (JavaScript XML):** JSX is a syntax extension for JavaScript commonly used with React. It allows you to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and `appendChild()` methods.
- **CSS:** For styling your components. Although not a programming language, it's essential for defining the presentation of your web pages.
- **HTML:** HTML is the backbone of web pages, and it is used to structure content. Even though React manages the DOM for you, the ultimate output is HTML.

Reasoning for Language Selection

- **JavaScript:** As a cornerstone of web development, JavaScript is ubiquitous for client-side scripting. Its vast ecosystem and compatibility with modern web browsers make it a natural choice.

    - **React JS:** Component-Based Architecture: React's component-based structure allows for reusable, maintainable, and scalable code and aids in organizing the user interface.
    - **Virtual DOM:** React employs a virtual DOM that optimizes rendering, improving performance and user experience.
    - **Rich Ecosystem and Community:** React has a vast ecosystem, including many libraries and tools, and is supported by a large community.
    - **Flexibility and Integration:** React can be integrated with other libraries and frameworks, making it versatile for different projects.
- **JSX:** It simplifies the process of writing and adding HTML in JavaScript. JSX makes the code more readable and more accessible for debugging.
- **CSS:** To provide a rich, engaging user interface with custom layouts and styles.
- **HTML:** As the standard markup language for web pages, it's fundamental for rendering content.

Development tools and IDEs used in the project

- **Visual Studio Code (VS Code)**: This IDE is commonly used for various types of development, including mobile app development. It provides a wide range of extensions for different programming languages and frameworks.
- **Firebase Console:** If the app requires backend services like *user authentication*, database, or cloud functions, Firebase Console offers a suite of tools for development and management.
- **Git and GitHub/GitLab/Bitbucket**: *Version control* is essential for collaborative development. Git is the version control system, and platforms like GitHub, GitLab, or Bitbucket provide hosting services for Git repositories.
- **Trello:** We will need to clearly *track our tasks* and address issues to solve our goals.

Tools for testing, debugging, and deployment

- **Emulators/Simulators:** allow you to test your app on virtual devices during development without needing physical devices.

- **Real Devices:** Testing the app on real devices is crucial for detecting device-specific issues and ensuring compatibility. We may need a variety of devices to cover different screen sizes, resolutions, and OS versions.
- **Unit Testing and Integration Testing Frameworks:** Depending on the programming language and framework we are using, we'll choose appropriate testing frameworks such as Jest for JavaScript, Pytest for Python, and JUnit for Java.
- **Crash Reporting and Analytics:** Integrating tools like Firebase Crashlytics or Sentry provides insights into app crashes and user behavior, helping us identify and fix issues quickly.

Source Control

When it comes to working on our project, keeping a log of our development journey is important. We want to make sure that we're not just throwing in changes randomly but are actually documenting our steps so we can understand what we did, why we did it, and how we can fix things if they go sideways. Things we can do to keep our codebase neat and efficient:

- Commit to a branch often when making changes that way if an issue arises locally you wouldn't lose hours of progress. It is important to make sure we are committing to a branch so we don't accidentally mess up the main before merging.
- Peer review other people's branches, often a new set of eyes will make it easier to spot mistakes. If you are going to merge your branch with main you should make sure to review so that nothing goes wrong.
- If you and your peers are working on different branches and try to merge there will be conflicts and you will have to be able to deal with the overlaps of commits and the changes you have to make to handle them.
- It's important to set goals large or small on a timeline for your project. This will result in 2 things: You are more likely to stick to finishing a goal within a set time

frame, and you are able to look back at the progress you've made in the project's history.

Guidelines

- **Build and Test locally:** For each change you want to make, first test it on your local machine and make sure it works without faults.
- **Branch name:** Name the branch according to the issue you are fixing and make it descriptive.
- **Reviewers:** Assign people working on the same issue as reviewers.

Supported platforms

- **Browsers:** Focus on supporting the five most commonly used browsers (Firefox, Safari, Chrome, Edge, Opera/Brave). Prioritize these platforms without expending excessive resources on less common options.
- **Platform-Specific Considerations:** Must be aware of potential cross-browser compatibility issues. Differences in JavaScript feature implementation and rendering behaviors among browsers could pose challenges.

Conflict Resolution Policy

- **Open Communication**
    - **Team Dynamics:** Encourage open and honest communication within the team to prevent misunderstandings. All team members should feel comfortable voicing their opinions and concerns.
    - **Managerial Transparency:** Managers must maintain open lines of communication, discussing all relevant problems and details to avoid misunderstandings.
- **Timely Resolution**
    - Address any misunderstandings or differing perspectives promptly before proceeding with tasks.

    - Ensure clarity on expectations and responsibilities among developers to prevent conflicts.

- **Review and Analysis**
    - **Understanding Conflicts:** Encourage both sides to understand the root causes of conflicts by examining code, consulting with colleagues, and understanding the rationale behind changes.
    - **Peer Review:** Utilize peer reviews for conflict resolution. Assign at least two reviewers to each pull request relevant to the topic in question.
- **Resolution Process**
    - **Attempted Self-Resolution:** Initially, developers should try to resolve disputes independently, seeking compromise or middle-ground solutions.
    - **Third-Party Intervention:** If self-resolution is not possible, involve a neutral third party to assist.
    - **Final Approval:** Require approval from all assigned reviewers for a pull request to be merged. Reviewers have the authority to request code changes until satisfactory resolution is achieved.
- **Resolution for Code conflict**
    - In case of a code conflict, two (or more) developers might make changes in the same script leading to conflicts. The Gitlab Merge Conflicts wiki documents the method to merge such conflicts.
    - If two developers alter the same line of code (same line and same functionality) then a code review is required in order to close the conflict.

Debugging Guidelines

These guidelines provide strategies for effectively identifying and resolving code issues, fostering a culture of methodical problem-solving and efficient use of tools.

- **Systematic Approach to Debugging**

- **Issue Replication:** Begin by replicating the issue to understand the conditions under which it occurs.
- **Divide and Conquer:** Use divide-and-conquer techniques to pinpoint the section of the codebase where the problem originates.
- **Incremental Development:** Write and test code in small increments to detect bugs early.
- **Preventive Measures:** Regularly conduct code reviews and engage in pair programming to prevent the introduction of bugs.

- **Tools and Techniques for Effective Debugging:**
  - **Version Control System:.**
    - **Git:** Utilize Git to manage code changes and track when and how bugs are introduced.
    - **Branch Policies:** Implement branch policies to maintain code quality and simplify bug tracking.
  - **Integrated Development Environment (IDE) Tools:**
    - **Debugging Features:** Make full use of the debugging capabilities of your IDE, such as breakpoints, step-through execution, variable inspection, and call stack analysis.
    - **Training Resources:** Provide resources or training sessions to maximize the effective use of IDE debugging tools.
  - **Automated Testing and Continuous Integration:**
    - **Automated Frameworks:** Encourage the adoption of automated testing frameworks for early bug detection.
    - **Continuous Integration (CI):** Implement CI systems to run tests automatically upon each code check-in, providing immediate feedback on bugs' introduction.

JSON Formatting Guidelines (if applicable)

Guidelines for formatting JSON data

- **Indentation:** Use consistent indentation to represent the hierarchy of data and indent each nested level by a consistent number of spaces or tabs.
- **Use Line Breaks:** Break long JSON objects or arrays into multiple lines for readability.
- **Avoid Trailing Commas:** Omit trailing commas in objects and arrays to ensure maximum compatibility.
- **Use Quotes for Keys and Strings:** Enclose keys and string values in double quotes (`"`).
- **Use Arrays and Objects Appropriately:**
    - Use arrays (`[]`) for ordered lists of items.
    - Use objects (`{}`) for key-value pairs.
- **Use Descriptive Keys:** Use descriptive and meaningful keys that accurately represent the data they hold. Avoid cryptic or ambiguous key names.
- **Define Data Types:** Clearly define the data types of JSON values (e.g., string, number, boolean, array, object) to ensure consistency and prevent type-related errors.

Naming conventions for JSON keys

- **Use Descriptive Names:** Use clear, descriptive names that reflect the key's purpose. Aim for names that are self-explanatory, reducing the need for comments.
- **Avoid Reserved Keywords:** Avoid using reserved keywords from programming languages to prevent conflicts.
- **Choose Naming Convention:** Choose a consistent naming convention (e.g., camelCase or snake_case) for JSON keys throughout the project. Stick to one convention to maintain uniformity.

- **Avoid Conflicting Keys:** Avoid using reserved keywords or language-specific terms as JSON keys to prevent conflicts and ensure compatibility with various platforms and languages.

Feature-based Development

Approach for organizing code based on features

- **Identify Features:** Understand the requirements of your application and identify distinct features. Each feature should represent a cohesive piece of functionality that can be developed and tested independently.
- **Create Feature Modules:** Create separate modules or directories for each feature in your codebase. Group related components, services, and assets within their respective feature modules.

Guidelines for implementing and testing features

- **Define Clear Requirements:** Clearly define the requirements and acceptance criteria for each feature before implementation begins.
- **Implement Incrementally:** Break down features into smaller tasks and implement them incrementally. Use version control to track changes and ensure code maintainability.
- **Unit Testing:** Write unit tests for individual components, services, and other units of functionality within a feature. Test all edge scenarios to ensure robustness.
- **Documentation:** Document each feature, including its purpose, functionality, and usage, to facilitate understanding and collaboration among team members.
- **Reusable Components:** Encourage the creation of reusable components within feature modules to promote code reusability and consistency across the application.

ElasticSearch

Elasticsearch is a distributed, RESTful search and analytics engine designed for horizontal scalability, reliability, and real-time search capabilities. It is built on top of the Apache Lucene library and provides a distributed multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

ElasticSearch is the perfect technology for implementing recipe lookups. Multiple indices can be created within elasticsearch for faster fetching and appropriate queries can be made too. A few of the suggested indices are:

Ingredient Index

The ingredient index is the first index that needs to be set up. It would contain a list of the ingredients and this index can be used for spell checks and class of equivalency checks.

Fig 3 (a, b)- Sample ingredient index and structure of the elasticsearch index:

```json
{
  "name": "tomatoes",
  "equivalent_class": "tomato"
}
{
  "name": "white mushrooms",
  "equivalent_class": "mushroom"
}
{
  "name": "chicken thigh",
  "equivalent_class": "chicken"
}
{
  "name": "red cabbage",
  "equivalent_class": "cabbage"
}
```

```json
{
  "mappings": {
    "equivalent_item": {
      "properties": {
        "name": {
          "type": "text"
        },
        "equivalent_class": {
          "type": "text"
        }
      }
    }
  }
}
```

Here's the structure breakdown:

- Index Name: `equivalency_index` (for example)
- Document Type: `equivalent_item`
- Fields:
    - `name`: Text field containing the original name.
    - `equivalent_class`: Text field containing the equivalent class.

Fig 4- Sample query to lookup class of equivalency.

```json
{
  "query": {
    "match": {
      "name": {
        "query": "ingredient_name",
        "fuzziness": "AUTO"
      }
    }
  }
}
```

Recipe Index

A recipe index can be used to fetch multiple recipe details. This index would consist of the name of the recipe and the list of ingredients used in the recipe.

Fig 5. - Sample index showing recipe with a list of ingredients.

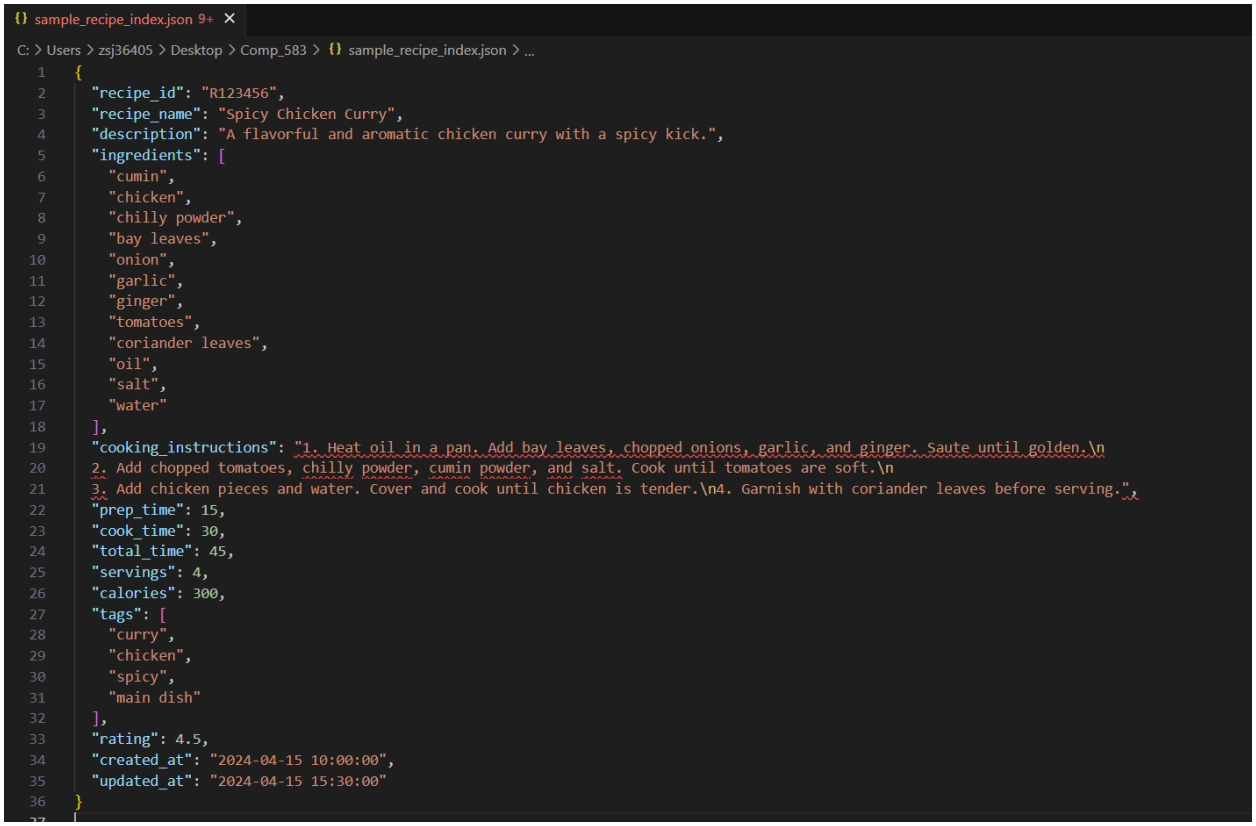


Fig 6 - Index structure:

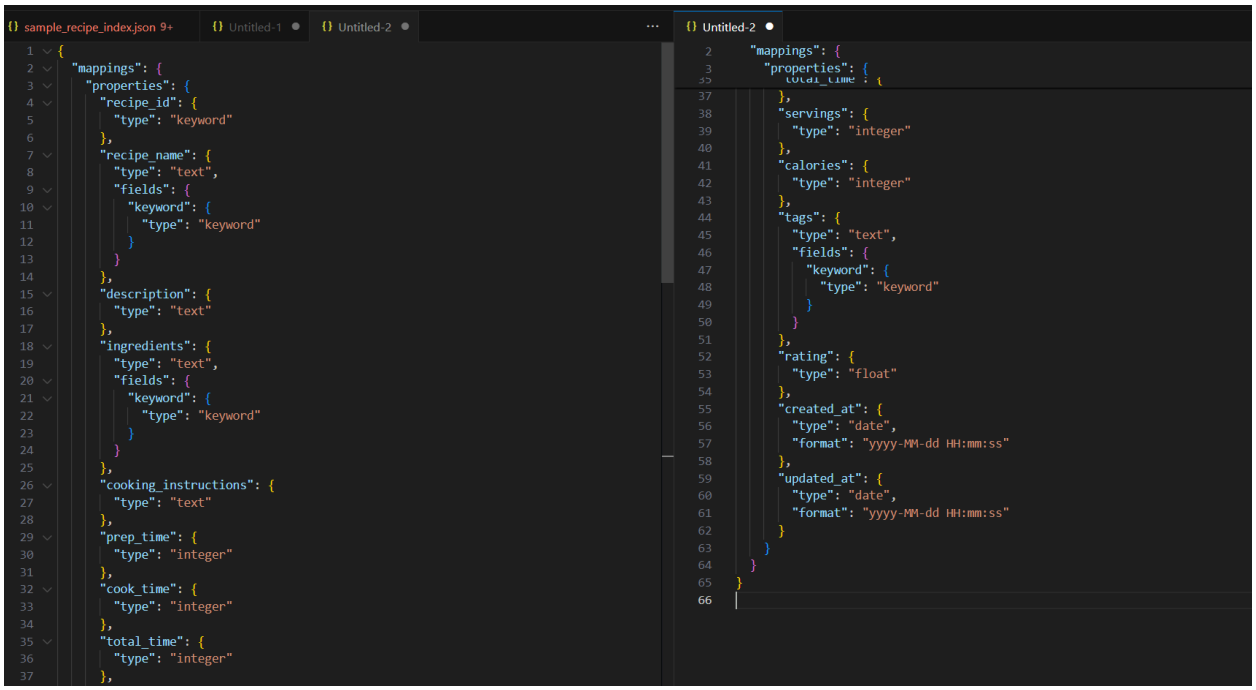


Fig 7 - Query to search for direct match:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "ingredients": "cumin" } },
        { "match": { "ingredients": "chicken" } },
        { "match": { "ingredients": "chilly powder" } },
        { "match": { "ingredients": "bay leaves" } }
      ]
    }
  }
}
```

This query searches for recipes where all of the specified ingredients ("cumin", "chicken", "chilly powder", "bay leaves") are present in the `ingredients` field. Elasticsearch will return documents (recipes) that match all of the specified ingredients.

Fig 8 - Query to search for partial match:

```
{
  "query": {
    "bool": {
      "should": [
        { "match": { "ingredients": { "query": "cumin", "fuzziness": "AUTO" } } },
        { "match": { "ingredients": { "query": "chicken", "fuzziness": "AUTO" } } },
        { "match": { "ingredients": { "query": "chilly powder", "fuzziness": "AUTO" } } },
        { "match": { "ingredients": { "query": "bay leaves", "fuzziness": "AUTO" } } }
      ]
    }
  }
}
```

In this query:

- The **`should`** clause is used instead of **`must`**, which means that at least one of the specified ingredients should be present in the recipe.
- The **`fuzziness`** parameter with a value of "**AUTO**" allows for approximate matching by considering typos, misspellings, and other minor differences in the text.

Elasticsearch will return recipes that contain at least one of the specified ingredients and rank them based on how closely they match the query.

More Use Cases

The processing and querying discussed above is basic and highly useful when the values are present within elasticsearch. There are a few scenarios where it may become difficult to transform the data due to mismatch of spellings.

Consider the input example - **`"chiicken thieigh"`** which should be mapped to "**`chicken`**". This is a complex scenario and can be handled as discussed below:

Fig 9 - Query to search for “chiicken thieigh” and equate it to “chicken”:

Fuzzy Logic

- The fuzzy lookup allows the queries to handle spelling mistakes.
- The fuzzy query with `"fuzziness": "AUTO"` will handle minor spelling mistakes.
- The fuzzy query with `"fuzziness": "2"` will allow up to 2 changes, insertions, or deletions.

Phonetic Algorithms

- We can use a phonetic algorithm like Metaphone to transform the input into a phonetic code.
    - For "chiicken thieigh":
        - "chiicken" might be transformed to "CHK"

        - "thieigh" might be transformed to "TH"
- This phonetic transformation can be stored in the elasticsearch to lookup data. On transforming each ingredient by the phonetic transformation pipeline, we may be able to convert ingredients using their syllables.
- There's an improved version called Double Metaphone, which handles more complex cases and non-English words better. Python libraries like `python-Levenshtein` provide an efficient implementation of the Double Metaphone algorithm.

n-grams to match substrings of characters

N-grams are a sequence of **`n`** characters from a given text, where **`n`** can be any integer. They can be used to capture the context in which a particular term appears, which can be helpful in fuzzy matching and searching.

First, we define the mapping for the index to use n-grams. One can use the **`edge_ngram`** tokenizer to generate n-grams from the beginning of the terms.

- The **`edge_ngram`** tokenizer in Elasticsearch is a tokenizer that breaks text into a stream of n-grams, which are contiguous sequences of `n` characters from a given text.
- The "edge" in **`edge_ngram`** refers to the fact that these n-grams are generated from the beginning of the terms.
- **Min Gram and Max Gram:** Onecan specify the minimum and maximum sizes for the n-grams. For example, setting **`min_gram`** to 2 and **`max_gram`** to 15 will generate n-grams ranging from 2 to 15 characters long.
- **Token Characters:** This parameter specifies which characters should be considered for tokenization. In the example provided earlier, we used `["letter", "digit"]`, which means only letters and digits will be considered for generating n-grams.

- Let's say you have a text "chicken" and you use an **`edge_ngram`** tokenizer with **`min_gram`** set to 2 and **`max_gram`** set to 5.
    - The generated n-grams will be:
        - "ch"
        - "chi"
        - "chic"
        - "chick"
        - "chicke"
        - "chicken"

Fig 10 - Defining mappings for n gram

```
{
  "settings": {
    "analysis": {
      "analyzer": {
        "autocomplete_analyzer": {
          "tokenizer": "autocomplete_tokenizer"
        }
      },
      "tokenizer": {
        "autocomplete_tokenizer": {
          "type": "edge_ngram",
          "min_gram": 2,
          "max_gram": 15,
          "token_chars": ["letter", "digit"]
        }
      }
    }
  },
  "mappings": {
    "equivalent_item": {
      "properties": {
        "name": {
          "type": "text",
          "analyzer": "autocomplete_analyzer",
          "search_analyzer": "standard"
        },
        "equivalent_class": {
          "type": "text"
        }
      }
    }
  }
}
```

Fig 11 - Query with n grams

```json
{
  "query": {
    "match": {
      "name": {
        "query": "chiicken thieigh",
        "analyzer": "autocomplete_analyzer"
      }
    }
  }
}
```

The **`autocomplete_analyzer`** will generate n-grams for the search query, making it possible to match the terms "chicken thigh" even with typos or variations.

Error Handling

Incorporating a standardized approach to error handling is critical for ensuring consistency, reliability, and maintainability throughout our project. This approach minimizes development overhead while enhancing application stability.

General Guidelines

- **Error Classification:** Systematically define and categorize errors based on their occurrence stage and characteristics. This classification aids in understanding and addressing various error types.
- **Error Reporting:** Implement robust error reporting mechanisms. Capturing key error details streamlines the debugging and troubleshooting process.
- **Error Responses:** Design user-centric error responses. Enhancing the user experience during error occurrences by providing clear and helpful feedback.

- **Logging and Monitoring:** Emphasize thorough logging and proactive monitoring. This approach aids in identifying errors, tracking their resolution progress, and ensuring service continuity even when individual features encounter issues.
- **API Error Management:** Prepare for potential API response issues. In cases of no response from the API server, implement a strategy to resend the request, thereby addressing temporary server downtimes.
- **Comprehensive Logging System:** Initiate a master log at the start of each application session. This log should compile and append entries from various scripts, forming an exhaustive audit trail for each user session.
- **Preventive Service Monitoring:** Regularly check the responsiveness of services. Implement automatic service restarts as a fallback mechanism to minimize disruptions during runtime.

Common Error Handling

- **try-except Blocks:** Python utilizes try-except blocks to capture and handle exceptions gracefully, allowing developers to manage and recover from errors effectively. While Java employs try-catch-finally blocks to encapsulate and handle exceptions, allowing for controlled execution and graceful error recovery. JavaScript uses try...catch blocks to encapsulate and handle exceptions, enabling developers to manage errors and prevent application crashes.
- **Null Handling:** Use of Python's built-in data types and structures, such as dictionaries and lists, to handle potential null values and prevent NullPointer exceptions. Java's Optional class and null checks help in handling potential null values and preventing NullPointerExceptions. Utilizing JavaScript's built-in functions, such as Array.prototype.filter() or Array.prototype.find(), and optional chaining (?.) to handle null, undefined, and optional values safely and prevent runtime errors.
- **Logging and Reporting:** All three languages would be utilizing a common logging service created in Java with logs stored in the database. Java would be sending a response on the successful receipt of this log.

Contrasting Error Handling

Python

- **Traceback Function:** Utilize tracebacks in combination with try-except blocks. Python will be creating traceback functions to pinpoint the exact line or error within a try-except block for easier debugging.
- **Check if service is running**: It will also have a check in place to ensure that the services are running correctly. If some service stops running, then it would be restarted by this master checker.
- **Second Request:** If some API initially fails to provide a response, a second request would be sent to ensure that the service is not impacted due to momentary environment issues.

Java

- **Using Optional<T>:** Instead of returning null values or empty collections, consider using Java's Optional<T> class to encapsulate and represent optional values that may or may not be present. This approach encourages more expressive and safer APIs, reduces the risk of null pointer exceptions, and promotes better error handling and null-safety practices throughout the codebase.
- **Avoid Empty Catch Blocks:** Empty catch blocks can mask errors and make debugging and troubleshooting challenging. Instead of leaving catch blocks empty, log the exception details, provide meaningful error messages, and handle the exception appropriately by either recovering from the error, retrying the operation, or throwing a custom exception to indicate the failure and notify the caller or user about the issue.

Javascript

- **Organize Code into Reusable Modules or Components:** Adopt a modular and component-based architecture to organize and structure your JavaScript code. Breaking down the application into smaller, reusable modules or components

enhances maintainability, readability, and scalability, while also promoting code reusability, testability, and collaboration among developers.

Centralized Logging System

To streamline debugging across our application's multiple scripts, modules, and services, a centralized logging system is imperative. Such a system simplifies the diagnostic process by aggregating logs into a cohesive structure.

Figure 12 illustrates the logging template that will be employed throughout our system. Each script, module, or service will populate a set of predefined fields to maintain uniformity and facilitate efficient error tracing and operational audits.

Sno	Session_ID	Search_ID	Timestamp	Stage	Logging Level	Module	Status	Severity	Operation	Comment	End Result
1	1	1	4/2/2024 12:55	2	Module	aby.py	Exception	Critical Error	Addiing a recipe	traceback to line 5	Processing terminated
2	1	2	4/2/2024 12:56	3	Script	xyz.py	Info		Query based on diet	Applied filter for gluten free	Stage 3 completed

Figure 12: Template for Centralized Logging System Fields

Description of Fields

- **S. No:** Acts as an index for log entries.
- **Session_ID:** A unique identifier for each session.
- **Search_ID:** A unique identifier for each search within a session.
- **Timestamp:** Records the time at which the log entry is made.
- **Stage:** Identifies the program's stage (e.g., Input, API Fetch, Displaying).
- **Logging Level:** Distinguishes whether the entry was made by a module, script, or service.
- **Module:** Specifies the script that has logged the entry.
- **Status:** Categorizes log entries into Exceptions, Decisions, or Information.
- **Severity:** Indicates the severity of the event (Critical Error, Error, Warning).
- **Operation:** Describes the operation being performed at the time of logging.
- **Comment:** Provides detailed notes or a description of the log entry.

- **End Result:** Summarizes the outcome, such as whether processing was terminated or completed.

To avoid write locks and ensure accuracy in logging, logs should not be written to a single file by each script. Instead, logging should be managed by a RESTful service or written to a database designed to handle concurrent write operations without conflicts.

Data purging policies will need to be established for the maintenance of log data. Database partitioning can be utilized for logs to enhance retrieval speeds over time.

Scripts, modules, and services should collate logger statements in a data structure like a multilevel list and commit to the database in a single operation after primary processing is complete. The first script to execute upon initiating a search or session will retrieve the last used session and search IDs from the database, increment them by one, and apply these new IDs to the current session or search.

This logging protocol ensures systematic recording within our system and provides an essential audit trail.

Communication Across Teams

Fig. 1 in this document showed the communication domains through multiple stages of the project. Additionally, the Logging System showed the format for JSON structure used for communication between Java and all teams.

- A logging service in Java will make the logging agnostic of whoever is logging some information.
- JSON structure of logging data:

```
{
        "s_no" : "",
        "session_id": "",
        "search_id" : "",
        "timestamp" : "",
        "stage" : "",
        "logging_level" : "",
        "module" : "",
        "status" : "",
        "severity" : "",
        "operation" : "",
        "comment" : "",
        "end_result" : ""
    }
```

- For further receipt of log, Java will send a 200 response code to the services requesting them.
- Python and Javascript, if they don't receive a 200 response code, they can try resending the query once again to mitigate momentary disconnects.
- If no response is received on the second attempt as well, there should be a local log created by either of the services where they can store the details to prevent loss of information.
- The structure of JSON for communication between Javascript and Python (for input ingredients is) (Fig 13):

```
{
        "session_id": "",
        "search_id" : "",
        "ingredients" : [ingredient_1, ingredient_2, .. ingredient_n]
}
```

- The structure of JSON for communication between Python and Javascript (output recipe(s)) (Fig 14):

```
{
 session_id : "",
 search_id : "",
    recipe : [{
        name,
        img_url,
        ingredients,
        instructions,
        tags: ['vegan']
        valid_ingredients : []
    }
    ]
}
```

Technical flow of the application

- The user starts a session and Javascript sends a request to Java and fetches the last used session and search ID from the MySQL database.
- A unique search and session ID are assigned to each query and act as primary keys to identify the results.
- Javascript completes its parsing and validation of data then makes a request to the python services.
- The python services clean the ingredients, apply class of equivalency, preprocess the data as per the user story and search for recipes in the elasticsearch.
- On finding a suitable number of matches, they return the data to Javascript.
- The Javascript modules display the result to the user.
- During this entire process, all services, modules and scripts will feature extensive logs that would be saved in the database using a Java service and also serves as an audit trail.

Feedback

This format provides a structured outline for documenting various aspects of the software architecture, ensuring clarity and consistency throughout the development process. Each

section should be elaborated with relevant details, rationale, and examples where necessary. File structure

Tasks

Start Researching and contributing to One of the sections of the document