

Final Year Project Report

Full Unit - Interim Report

Personal Finance Tracker Application

Benjamin Etheridge

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dr Donggyun Han



Department of Computer Science
Royal Holloway, University of London

December 12, 2024

Declaration

This report has been prepared based on my own work. These have been acknowledged where other published and unpublished source materials have been used.

Word Count: 6025

Student Name: Benjamin Etheridge

Date of Submission: 12/12/2024

Signature:

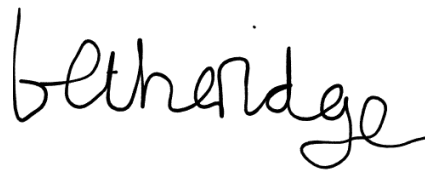
A handwritten signature in black ink that reads "betheridge". The signature is written in a cursive, lowercase style with a long horizontal stroke at the end.

Table of Contents

1	Introduction	3
1.1	Abstract	3
1.2	Project Description	3
1.3	Project Aims	4
1.4	Project Motivations	4
2	Literature Review	5
2.1	State-of-the-art web development	5
2.2	Architectural paradigms and applicable design patterns	6
3	The system to be implemented	9
3.1	Chosen tools and design	9
3.2	Initial plan	15
4	Completed Work	18
4.1	Conclusion	19
5	Project Diary	20

Chapter 1: Introduction

Software demo video link: <https://youtu.be/Oq4R4ooSdEQ>

1.1 Abstract

The evolution of web development constantly creates opportunities and challenges for software engineers. This report evaluates web development tools and frameworks. Comparing capabilities and trade-offs to give informed decision-making by integrating principles like performance optimisation, maintainability and security. The report identifies the best technologies suited for different project requirements. It also looks at the architectural paradigms, design patterns, deployment methods, and security of such projects. Offering insights into efficient system design and operation. It concludes with justifications for the chosen technologies and a detailed plan to implement a robust, modern solution. The project's progress during the interim has derived from a heavy amount of research and putting into practice the new tools I have learned. These tools have helped me design and create my application to match successful websites used by millions of people. It has taught me the importance of keeping up with modern solutions and managing an application.

1.2 Project Description

The project is a state-of-the-art web application using the latest and optimal technologies. It will explore the tools and methods currently used in web development. Web development has become a key part of modern software engineering. With lots of optionality, which frameworks to use, ranging from how they deploy their project to which programming language they want to use. Each has its benefits and drawbacks. It is an essential skill for a software developer to be able to evaluate and choose the appropriate tools to help build their project. This report provides a detailed exploration of the latest web development tools, focusing on the technologies, frameworks, and platforms that influence the web development landscape. A comparison of multiple tools and approaches should be made when developing websites, explaining their strengths and weaknesses and their suitability for different types of projects. The report will consider many principles, such as maintainability, scalability, performance, user experience, and security. Additionally, it will cover architectural paradigms in web development and address key design patterns, security/privacy considerations, and operational aspects of building systems. In addition, deployment methods and standard developer practices emphasise their role in efficiency and application deployment. Finally, the report justifies the tools chosen in this project. The report also provides a comprehensive overview of the system to be implemented, with use cases to describe the functionality and user interactions. It is designed to address and discuss the best technologies and architecture to ensure a solution that can scale well and is secure.

1.3 Project Aims

The goals of the project are to:

- Evaluate and identify the best Web Development tools and frameworks.
- Compare development approaches.
- Address security and privacy concerns.
- Optimize the deployment process.
- Develop a scalable system that:
 - Allows users to enter transactions.
 - Let users filter transactions based on parameters such as date, amount, and category.
 - Visualizes user data onto graphs, matching the filter parameters.
 - Gives users a better understanding of their finances.
 - Helps users make better financial decisions.

1.4 Project Motivations

Managing Finances is a hard task for many adults, leading to lots of financial stress and poor decision-making. In the U.K., 47% of adults do not feel confident in financial products and services [12]. This lack of confidence can be due to the financial illiteracy they have. And the overcomplication of financial data, which gives a poor understanding of personal finances. For instance, 22% of individuals have less than £100 in savings, and 47% have no financial goals for the next five years [12].

Studies have shown that visual information is retained more effectively than written or verbal [8] data. This project will focus on translating financial information into visual representations, such as graphical interfaces while studying the necessary tools to do so. The emphasis will be on mapping data onto graphs that are both visually appealing and easy to understand.

This is to encourage less fortunate people, especially students, to help manage and track their finances. Giving them a greater sense of financial awareness and hopefully encouraging them to start thinking for the future.

Chapter 2: Literature Review

2.1 State-of-the-art web development

Multiple frameworks have emerged as popular choices in a constantly changing web development landscape for many reasons. The more popular are typically better choices because they are generally better with many types of applications. However, aside from the framework's benefits, they foster large communities that can support developers through library documentation, tutorials or forums, and third-party tools. React.js is one of the most highly used frameworks. Primarily a Javascript-based development library for user interfaces, it focuses on creating fast, scalable, and dynamic interfaces for applications with multiple architecture types. React's component-based architecture allows developers to build reusable UI components with their style and structure. The primary benefit of React is the use of a virtual DOM to optimise rendering. "The virtual DOM, with its intelligent diffing algorithm, minimises unnecessary updates and ensures faster rendering, making it a cornerstone of React's efficiency." [10]. Other frameworks also provide their benefits. Angular is another popular front-end framework. Primarily because it is a comprehensive framework with much built-in functionality. Moreover, it tends to be preferred for large-scale applications at the enterprise level. Like react, a large part is reusable components, "At its core, Angular is about building reusable user interface components. We can then control it with Angular and combine it with other members to create an entire user interface from those Angular-controlled components." [15]. Angular is often called an All-in-One solution, meaning developers do not need to rely on multiple external libraries for simple tasks. Also, having a stricter opinionated framework encourages developer teams to enforce more coding standards and structure, which would be advantageous to large teams. Vue is considered a middle point between React and Angular regarding flexibility. Giving developers more freedom to structure as they need. Vue doesn't enforce strict separation of concerns as Angular does. Yet its template syntax and reactivity tend to be intuitive, making it an easier option for developers with basic HTML, CSS, and Javascript knowledge. Even compared to React, developers would need a fair understanding of concepts such as states and props to effectively use it.

Most business logic is handled on the server side or backend of the stack. Following the separation of concerns principle, it is responsible for processing data and managing interactions with databases and external services. It ensures security and validation and handles computationally intensive tasks. Again, developers can choose from multiple backend frameworks, all with their benefits over others. Spring boot is built on Java, making it highly efficient and scalable. Especially with Java Virtual Machine (JVM), which allows for optimised performance, would benefit enterprise-level applications that need high levels of throughput with low latency. It also has a large set of tools, such as Load Balancing and Service Discovery letting, allowing for high reliability, even when under high-stress environments, "The test application developed using the Spring Boot framework exhibited a high level of reliability" [1]. Aside from this, Spring Boot is also developer-friendly. Since being widely used for a long time, spring has been given a large set of security or data management libraries, especially "Spring boot is straightforward for building RestFul Web services and APIs, making it simpler for the developer to complete the job" [9]. Allowing for easy integration and communication with front-end frameworks is an essential component of web development. Django is considered for its high levels of security, especially in terms of protection from attacks like Structured Query Language (SQL) injection or cross-site scripting. It also has a well-supported and active community. However, due to its lack of tools, it does not scale well with high-level traffic or complex systems. It doesn't fare well in microservices, cloud

integration, and distributed systems. The choice of where and how to store data depends entirely on what the application needs and there is no more apparent, better solution.

Databases tend to be the most chosen option for web development. SQL and NoSQL are the two most popular choices for this type of database. SQL is often used for datasets that follow a relational model with rows, columns and predefined schemas. It is ideal for a structured dataset with lots of relationships between tables. SQL also works better for access control, [2] making it easy to assign user permissions, roles and privileges. NoSQL databases are more dynamic and can be shaped to the developer's needs. Even after the database has been created, helping developers to avoid unneeded complexity [14]. Often using document, graph, or key-value pair models to manage its data. Querying mechanisms can vary by the datatype and have no standard, such as document models using JSON-like queries, which can simplify transmitting data through the stack. NoSQL can scale significantly better horizontally [4], by distributing data across multiple servers or clusters. This makes them significantly faster with applications with massive data volume or high read/write operations. It also makes them more efficient for architectures such as large and unstructured microservices. Between NoSQL databases, there are still many options, depending on what the developer is after. MongoDB is popular due to its document style model, which offers incredibly high flexibility and makes objects practically schema-less if wanted. Documents will contain nested structures such as arrays. Even though it is not SQL, there is still a very high level of query capability, such as filtering and aggregations. It is often best to use documents that can vary in their parameters. Neo4j is another type of NoSQL database that uses graph structures to model and store data. Like an SQL database, it primarily focuses on their relationships and works best for complex queries that involve connections. However, it is still flexible for dynamic data models and has no required set structure. It is not suited for massive datasets and won't scale horizontally as well as others.

2.2 Architectural paradigms and applicable design patterns

Web development has evolved from static and straightforward to dynamic and complex. Developers have countless processes and patterns to follow depending on what they want for their application. These patterns and paradigms can guide the developer to structure efficient applications. Key aspects that these patterns cover are security, privacy, deployment, and development practices. All of which and more are crucial to an effectively designed application. Client-server architecture has a basic split of the model, where the client (web browser) and server are separate entities that can communicate to exchange and manage resources. This model is easy to follow and allows for an easy separation of concerns, assigning roles to each part of the model. This model is often embedded in others and is a key design aspect in many applications. MVC (Model View Controller) is a common architecture type to follow; this splits the stack into data, UI, and controller components. Most applications that follow this structure consist of a front-end (View), backend(controller/model), and database(model). It has a stricter separation of concerns, allowing for a more consistent application structure. This leads to easier development, testing, and maintenance. These features become more relevant the larger an application gets, especially maintenance.

Other architectures can be more specific, such as SPA(Single Page Application), which could be used to define the front-end side of an application. The developer may want their view or UI component to be a dynamically updating single-page application that doesn't need to reload a page when traversing to a new part of an application. Allowing for a more fluid and desktop-like design. [11]. Certain frameworks, such as React.js, work well with this due to its Component base architecture and Virtual DOM, allowing for easy rendering of changes. MicroService architecture turns larger applications into smaller independent services, and developers can design each component for a specific task. The split of services allows for horizontal scaling that can be split among multiple servers, benefiting applications on a large scale. These services use APIs to communicate with each other. This is opposed to the monolithic structure, which groups all functionality and services. All are running in a tight-knit, integrated single unit. It is more beneficial to a smaller, less complex application, using a centralised database and typically a single codebase. This, however, doesn't scale well, and eventually, the overhead from a MicroServices approach will be negligible compared to the complications caused by a monolithic architecture on a large scale. [5].

Design patterns can often be split into three key types: Creational, structural and behavioural. Often dictating how the object is made for the situation, how objects conform to the larger structure, and the communication between objects, respectively. The singleton pattern ensures that only one instance of a class exists, as well as providing a point of access to it. This is simple to implement and ensures controlled access to resources. This would be used for instances such as a database connection manager where only one class should exist. A Factory Pattern defines interfaces for creating objects but allows subclasses to alter the type of object that is being created. It, therefore, delinks the object creation from the client code. This would, however, require additional classes to create the objects and be more intensive. Abstract Factory patterns can provide interfaces for related or dependent objects without specifying their classes. This would typically be used in front-end UI frameworks like React.js for widget creation, such as buttons or text fields. The adapter pattern links two interfaces to make them compatible; it simplifies the integration of systems, such as user authentication services for payment or logging in. Keeping them decoupled and also reusable. However, it adds an extra abstraction layer, increasing complexity and potentially reducing performance. Often used in UI customisation, the Decorator pattern adds extra functionality to an object without modifying its structure and altering the original object. It can lead to more complexity with many small classes, making the system harder to understand. If ordered incorrectly, it can also lead to unintended behaviours if they are dependent on others. The observer pattern is where an object maintains a list of its dependants (observers) and notifies them of state changes through method calls. It is used for event-driven systems where many objects, such as a SPA front-end application, need to be notified of state changes. The subject and observer are loosely coupled, meaning new observers can be dynamically added without changing the subject. This comes at a performance cost, as notifying observers can become resource-intensive, especially if the update process is complex. The command pattern encapsulates a request or operation into an object. This can allow decoupling of the send and receiver, creating a more dynamic relationship between processes. It also supports undo functionality, such as changing data sets to be added to a database. Different types of receivers can also act differently upon the data sent, depending on the state. This, again, increased complexity, and an increased number of objects can increase the overhead and memory usage.

Security is always a primary concern for web applications, and there are many methods to protect the user's data and the application from malicious attacks. Security can be managed at every level of an application, from encrypting data to managing access control. Security can be extended by having separate considerations at different levels, such as in microservices architecture [5]. Attacks on personal data and websites will ever evolve to overcome current methods of protection. Applications must keep updated with security, as most attacks can easily overcome outdated security.

Attacks such as cross-site scripting inject scripts into webpages viewed by other users, which can steal sensitive information or deface websites. Or NoSQL injection attacks that allow attackers to bypass authentication, change other users' data or gain access to their information. This happens when user inputs are improperly handled and inserted directly into queries. Many authorities have compliance requirements such as GDPR that require websites to legally have minimum security, which is often enough to deter most attacks and protect user data. These regulations are essential to have and follow to create a standard of privacy and security for users, especially among websites that use sensitive data.

Authentication is another fundamental aspect; it verifies the identity of users/devices through passwords, two-factor authentication, and even biometric authentication. With correct authentication, strict access control can be easily ascertained. Safeguarding the confidentiality and integrity of personal data and financial transactions from potential fraudulent activity. It is also crucial to comply with regulations from GDPR and HIPAA and will need to be in any web application.

Encryption can often be considered the last line of defence in data breaches. Transforming data into an unreadable format at rest and in transit should always be used when managing data, especially if it's sensitive to the user. Data can be protected in transit using HTTPS/TLS certificates, which authenticate the identity and secure communication between the browser and server. It is also essential to build trust with users for a web application, as in the modern day, an application without these certificates will not be trusted. Even having their security features and best intentions. Encrypting data at rest is just as important. It protects against physical security breaches against unauthorised personnel and limits data leakage if a breach does occur.

Deployment is a critical phase of the web development life cycle, moving the application from a development environment to live production. Choosing the correct environment for the application to be deployed is just as paramount to the success of the application as any other consideration. Most web applications are deployed to a cloud-based system such as AWS or Azure. They often function as a pay-as-you-go model that can scale according to the application's demands. This allows for easy horizontal scaling, multiple region deployments and elasticity, as the developer would only pay for the directly used resources. And not unneeded excess resources during downtime. This also brings more of a reason to maximise any application's performance and resource usage, as it will directly bring down the runtime costs.

A developer has many tools at their disposal for optimising the development of their application. Almost all projects use a version control system (VCS) such as Git to manage and collaborate on changes to the project efficiently. This is especially necessary for projects with multiple developers, where workflow can be distributed and later merged. And any conflicts between developers' changes can easily be resolved. VCS also allows for saving the history of commits and reverting if necessary. Automated workflows are also beneficial to developers; they can assist with automating aspects such as testing, building and deployment. Following continuous integration / continuous deployment allows developers to detect bugs early, improve code quality and receive immediate feedback on their code.

Chapter 3: The system to be implemented

3.1 Chosen tools and design

There were many options for my web application, and I had many considerations to make before deciding which structure I would follow. However, some concepts I had to consider proved more important than others. As a web application that would let users import bank statements, security had to be one of my biggest concerns regarding protecting such sensitive information. Scalability was also a primary concern if my application grew large, with many users all sending multiple requests to the database simultaneously. Then, my service would have to scale effectively to keep up with demand. The system architecture also played a large role in which frameworks I would use.

A microservice architecture seemed the best option, as it not only scaled horizontally well. However, it could also provide extra security through each layer of service. For my database, I chose MongoDB as my best option. Compared to SQL databases, NoSQL is more scalable, especially with transactions such as insertion, query, update, and deletion [3]. If my application were to scale larger, this would become more noticeable and effective; hence, I decided to do it initially rather than later when it became an issue. My data structure also didn't have many relationships; most of what would have been the tables of the database would only have a relationship with the user table. So, the benefits of an SQL database would be negligible to my application. This also ruled out any graph NoSQL designs. Compared to other NoSQL databases, MongoDB offers the best average time for reading, from small collections to large amounts of requests. Databases such as RavenDB and CouchDB were slower overall. CouchBase had a faster time managing tens of thousands of requests in tests; however, it was only by a small amount. It was also significantly slower when comparing relevant times for a few requests [7].

As the backend would be responsible for most business logic, security and managing data. I decided to go with SpringBoot. This framework has many security features and dependency injection [13]. It helps with architecture patterns such as separation of concerns, allowing me to easily assign controller, service, model, and repository classes. SpringBoot also has a lot of support for the microservice architecture type, making it easy for me to separate responsibilities into separate containers, such as login, transactions, analytics, etc. For my front end, I decided to go with React.js. There were many reasons for this, the primary ones being ease of use and code reusability through states and props. Customisability and its virtual DOM make rendering significantly faster, allowing my application to progress faster. [15]. Another large influence of choosing React was its libraries, which allowed compatibility with charts. Letting me "provide flexibility while preserving simplicity, ECharts provides a special component series which allows users to modify a predefined chart by changing its rendering process." [6].]. The compatibility with the parts of the stack I decided upon was also an important feature, such as the document style of the database being JSON, which meant I didn't have to alter the JSON object sent from the front end and could store it directly in the database, without too much manipulation in the backend service. SpringBoot also has libraries for MongoDB databases with built-in functions, making it easier to make simple modifications or requests to and from the database.



Figure 3.1: JSON Object sent from front end.



Figure 3.2: Document object stored in database shown on MongoCompass.

For my application, a MicroService architecture approach would work best. It will allow me to isolate errors more easily and is more scalable than a monolithic approach. A cloud deployment service would also benefit from this approach as some services do not require as many resources, such as the one for operations on user login details, compared to user transactions, which can have thousands per user. My front end will follow the SPA architecture as I build it on the React.js library. It will be more fluid and dynamic when loading components and global parameters that need to be stored for the runtime, which won't need to be passed to a new state every time the user loads a different part of the application. Another aspect to consider is how to protect data. As users will submit their transaction information to my application, multiple layers of security will be needed. Including encryption in transit and at rest, multiple methods to protect data from attacks like XSS or CSRF will need TLS certificates such as HTTPS/SSL. Not only would this let users trust the application, but it is also necessary for compliance with data protection regulations due to the sensitivity of the data. Being a Microservice architecture would also help with this, offering security and protection at each separate service and layer of the application. Cloud-based deployment services also offer their protection and security methods, further justifying that deployment method. This application is designed to be a personal finance management and tracking tool. Its primary target is students, helping them to manage their limited budgets effectively by providing features such as transaction tracking, budgeting and data visualisation. It also allows users to import bank statements to bulk add transactions and apply filters to gain insights into spending patterns. By simplifying and visualising financial management, this application helps to foster good financial habits and awareness, letting them make better decisions about their expenditures. The system follows a Microservice architecture comprising several core services with their purpose and tasks. Each component is dockerised and uses API calls to communicate with each other and the databases.

```

@RequestMapping("/transactions")
@PostMapping("/bulk")
public ResponseEntity<String> addBulkTransaction(
    @RequestParam("file") MultipartFile file,
    @RequestParam("userId") String userId) {
    try {
        // Pass the file to the service for processing
        transactionService.parseFile(file, userId);
        return ResponseEntity.status(HttpStatus.CREATED).body("Created");
    } catch (IOException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error proces
    } catch (IllegalArgumentException e) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Failed to ad
    }
}

```

Figure 3.3: Example of an API call in TransactionController class.

This allows for a seamless integration to a cloud-based service when ready. The front end is composed of a React.js SPA, which makes it responsive and interactive to the user. The back end is in a Spring Boot framework and is responsible for logic, data processing and database integration. The database type will be a NoSQL MongoDB, responsible for storing necessary data, such as transaction information, in a document format. Matching the JSON structure of the front API calls. The key features of the applications are sign-up, login, add a transaction, add CSV, filter transaction, budgeting, and display on the graph.

Use Cases

1. Sign Up

Goal: To allow new users to create an account and access the system.

Actors: User

Preconditions: User is not already registered in the system.

Steps:

1. Sign Up component loads.
2. User enters required details (username, email, password).
3. System validates input.
4. If valid, the system creates a new account for the user.
5. User is redirected to the main page.

Flow:

- *Basic Flow:* User submits valid details and account is created successfully.
- *Alternative Flow:* If any input is invalid, the system prompts the user which and how to solve.

Postconditions: User account is created and stored in the system.

2. Login

Goal: To authenticate and provide access to registered account.

Actors: User

Preconditions: User is already registered and has valid credentials.

Steps:

1. User navigates to the login component.
2. User enters their username and password.
3. System verifies the credentials.
4. If valid, user is granted access to their dashboard.
5. If invalid, system displays an error message.

Flow:

- *Basic Flow:* User enters valid credentials and is redirected to their dashboard.
- *Alternative Flow:* If the credentials are incorrect, the system prompts the user to retry.

Postconditions: User is logged in and has access to their account.

3. Add Transaction

Goal: To allow users to add and track individual financial transactions.

Actors: User

Preconditions: User is logged in.

Steps:

1. User navigates to the Transaction component.
2. User clicks add single transaction button.
3. User enters transaction details (date, merchant, amount, category).
4. System validates the input.
5. If valid, the transaction is saved in the system.
6. User is presented with a success message.

Flow:

- *Basic Flow:* Transaction is successfully added to the system.
- *Alternative Flow:* If any details are missing or incorrect, the system prompts the user to correct the input.

Postconditions: Transaction is added to the user's transaction records and available for viewing.

4. Add CSV (Bulk Transactions)

Goal: To allow users to upload bank statements in CSV format.

Actors: User

Preconditions: User is logged in and has a properly formatted CSV bank statement.

Steps:

1. User navigates to the Transaction component.
2. User clicks add bank statement button.
3. User uploads a CSV statement containing transaction data.
4. System parses the file and validates the data format.
5. If valid, the system adds the transactions to the database.
6. User is presented with a success message.

Flow:

- *Basic Flow:* Transactions are successfully uploaded and added.
- *Alternative Flow:* If the CSV file has invalid data, the system displays an error message about the issues.

Postconditions: Multiple transactions are added to the user's transaction records and available for viewing.

5. Filter Transactions

Goal: To allow users to view transactions based on specific filters.

Actors: User

Preconditions: User is logged in and has existing transactions.

Steps:

1. User navigates to the "Filter Transactions" section.
2. User applies filters such as date range, category, or amount.
3. System processes the filter and displays matching results.

Flow:

- *Basic Flow:* Filtered transactions are displayed based on the user's criteria.
- *Alternative Flow:* If no transactions match the filter criteria, the system tells the user that no results were found.

Postconditions: Transactions are filtered and displayed according to the user's preferences.

6. Budgeting

Goal: To allow users to create and manage budgets for specific categories and time periods.

Actors: User

Preconditions: User is logged in.

Steps:

1. User navigates to the Budget component.
2. User sets a total or category budget.
3. System validates the budget input.
4. If valid, the system saves the budget and tracks spending against it.
5. User is presented with the status of their budget.

Flow:

- *Basic Flow:* Budget is successfully created, and the user can track spending.
- *Alternative Flow:* If the budget exceeds the total amount, the system prompts the user to adjust it.

Postconditions: Budget is set and linked to the user's account for ongoing tracking.

7. Display on Graph

Goal: To provide users with visualizations of their financial data through graphs.

Actors: User

Preconditions: User is logged in and has transactions or budget data.

Steps:

1. User navigates to the Graph component.
2. User selects the type of graph and filters for the data to visualize.
3. System processes the data and generates the selected graph.
4. User can interact with the graph, and it adjusts as the user changes filters.

Flow:

- *Basic Flow:* The graph is displayed, and the user can see their trends and insights.
- *Alternative Flow:* If there is insufficient data to display, the system notifies the user.

Postconditions: A visual representation of the user's financial data is generated and displayed.

8. Notify User with Insights

Goal: To notify users with insights about their financial activity, such as spending trends, budget status, or potential savings.

Actors: User

Preconditions: User is logged in and has transactions or budget data in the system.

Steps:

1. System displays the user's financial activity (transactions, budget adherence, etc.).
2. Based on defined thresholds or patterns such as overspending, nearing budget limit, or unusual spending, the system creates relevant insights.
3. In the insights tab on the main page, the system creates insights based on spending trends, budget status, or potential savings.
4. User sees the insights tab and can view relevant insights.

Flow:

- *Basic Flow:* System detects insights based on user data and updates the insights tab to the user accordingly.
- *Alternative Flow:* If no significant data is given, no update is provided.

Postconditions: The user is notified of important financial insights and can take action to adjust their spending or budget.

3.2 Initial plan

The initial part of the project focuses on establishing the core components of the system, ensuring a strong foundation for further development. These deliverables aim to give the core functionality, test basic integration, and ensure objectives are being met. By breaking down the tasks into manageable components, these deliverables will be stepping stones to the completed system. The Following goals are split into three core components. The frontend, backend, and database. By focusing on these goals, the project will have these essential building blocks that are functional and reliable, allowing for more advanced features to be added in later phases.

3.2.1 Early Deliverable Objectives

Backend

- Create endpoint to login and call it with hardcoded data
- Create endpoint to pass in a file and save data to Local service as a parsed object
- Save files to DB
- Create backend service that returns pong to /ping endpoint
- Create endpoint to create user and store details in object in running application
- Validation against bad login is created
- Validation against bad files
- Retrieve endpoint has new param to sort data
- Retrieve endpoint has new param to filter data
- Validation against bad create account is created
- Connect to DB and save account details there
- Endpoint to retrieve all data from submitted files

Frontend

- Create empty webpage
- Create create user page that calls the backend endpoint. Backend logs user details
- Create login page that calls login endpoint and authenticates a user
- Create file submission and calls endpoint to add files in

Database

- Create empty DB that runs on its own
- Backend is able to connect and send data to DB
- Backend sends formatted user account info to DB
- Backend sends formatted user file info to DB

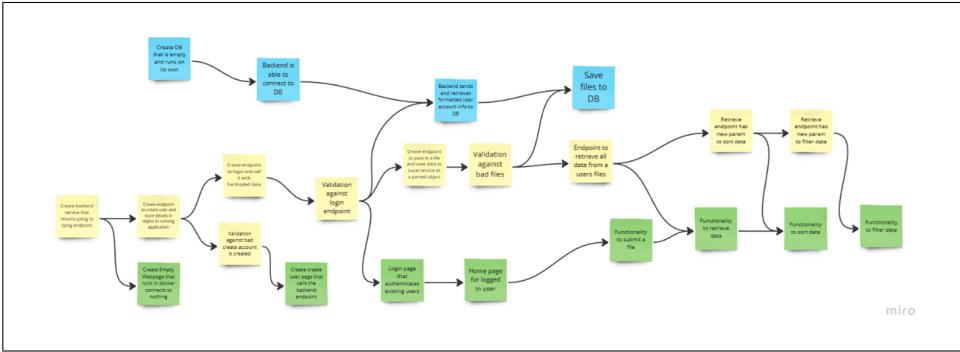


Figure 3.4: Miro Board plan.

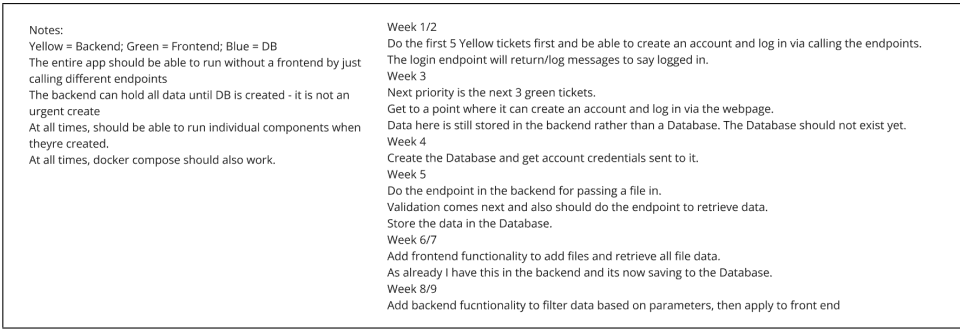


Figure 3.5: Timeline plan on Miro page.

Chapter 4: Completed Work

Each task was mapped to a dedicated branch on the Git repository, ensuring a modular development and minimizing conflicts. This allowed isolated testing and debugging of individual tasks, and easy integration of completed tasks into the main project. Following the Test-Driven Development methodology, tests were written prior to implementation to describe expected behaviors.

```
@Test
public void testParseFile() throws IOException {
    //Given          Use mock file for testing in test/resources
    MockMultipartFile file = new MockMultipartFile(
        "file", // The name of the file field
        "mockTransactionsFile.csv", //filename
        "text/csv", // File type
        new FileInputStream(ResourceUtils.getFile("classpath:File.csv"))
    );
    String userId = "userId123";
    //When
    List<Transaction> lines = transactionService.parseFile(file, userId);

    //Then
    assertNotNull(lines);
    assertFalse(lines.isEmpty(), "File should contain at least one line");
    assertEquals(100, lines.get(0).getAmount());
    assertEquals(200, lines.get(1).getAmount());
    assertEquals(300, lines.get(2).getAmount());
}
```

Figure 4.1: Example of a test case in TransactionServiceTest class.

It ensures all components meet requirements and maintains high-quality code. By combining the Miro board for task planning and GitLab for version control, the project maintained a balanced development and continuous quality assurance. During development, as tasks were completed, the Miro board would be checked off and the branch merged into the main branch. Not every goal for this stage was completed. The application has all the expected functionality apart from filtering data based on specific parameters. Several reasons could be deduced why not every goal was met; however, I believe it could be due to a lack of knowledge and therefore extra time to understand new tools such as docker and MongoDB. Towards the end of the development period, I also added the ability to portray users' data onto a basic graph. I felt that this was a more important deliverable than being able to filter data, as it fulfills a more valued use case than filtering and improves the quality of the early deliverable.

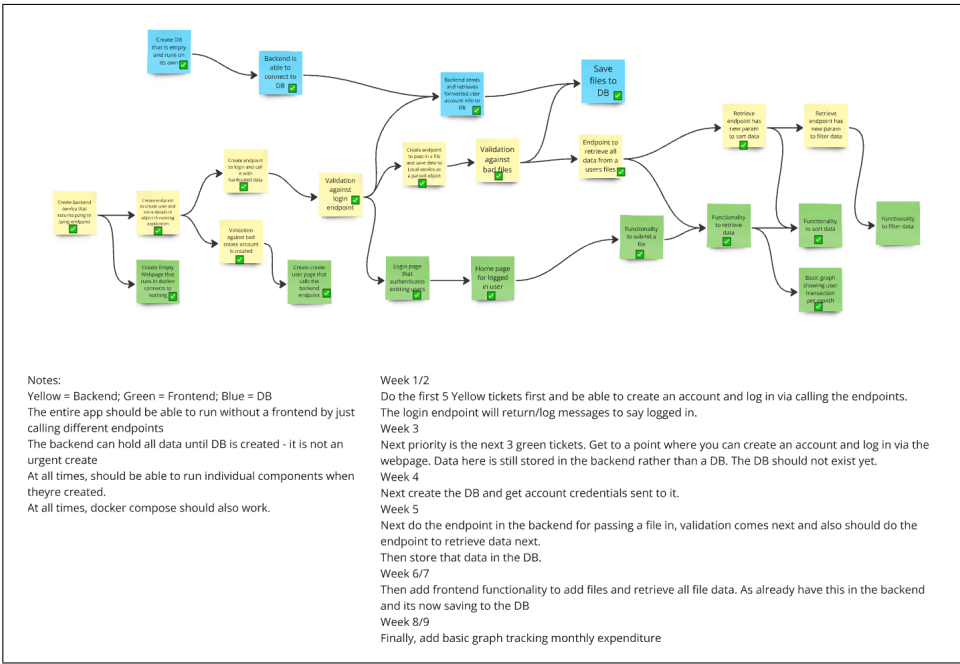


Figure 4.2: Logo of RHUL.

4.1 Conclusion

In this report, we have explored the most modern advancements in web development. Looking at the most prominent technologies and platforms available today. Comparing the strengths and weaknesses of these tools, considering their suitability for different projects, and determining their use based on core principles of web development. Moreover, the architecture types and design patterns discussed have shown the importance of a structured system, especially with the complexity of web applications and the scale they can amount to. Key considerations such as security/privacy and operation aspects were emphasised as components necessary for building competent and efficient systems. Finally, the system to be implemented, is outlined through use cases and justifications for the tools chosen. By taking into account the user needs and the system needs, the proposed application is scalable and secure. Also allowing for a positive user experience. Future work may focus on more optimization, updates based on user feedback, and incorporating new technologies as they emerge.

Chapter 5: Project Diary

30/09 - 3/10:

- Started project plan and researching.
- Risk/mitigation and timetable completed.

3/10 - 11/10:

- Had meeting with supervisor:
 - Main focus will be on security features and how I can show website security. Why would a customer risk info with no proof it will be kept safe?
 - All reports must be done, time allocation is okay.
- Project plan complete.

12/10:

- Created React front, next step is to create a login page.

14/10:

- Created Title Bar and welcome message.
- Chosen colour scheme: light to darker light blue, white background for tabs, grey/black for writing.

16/10:

- Finalized signup page design, need to start backend and create database for signup.

17/10:

- Researching which services to use, as well as backend framework.

18/10:

- Created login form, logic to switch between login and signup. All set to create and store accounts.
- **NEED TO FIX:** Autofill background colour.

21/10:

- Created relation schema diagram for database.
- Chosen Spring Boot for backend:
 - Prefer to work in Java, has built-in support for REST APIs, security, and data access, can create custom APIs.

22/10:

- Started on backend, created Spring Boot Maven project and sorted file structure.

23/10 - 31/10:

- Not much done this week due to coursework on other modules. Have linked backend and frontend, allowing for use of signup.

1/11:

- After meeting with a professional software engineer, decided to restructure the project:
 - Frontend and backend will be run from a Docker container, eventually allowing for easy deployment, keeping file structure to a professional standard.
 - Changed from Maven to Gradle due to better dependency management, flexibility, and dynamic support.
 - Using Groovy (not Kotlin) as it is more compatible with Gradle and Java.
 - Java source 21, React frontend still.
 - Database will now be NoSQL using MongoDB.
 - Focus on functionality this term.

02/11 - 11/11:

- Dockerized entire project and got it working from containers.
- Researched further into using NoSQL database.
- Things to consider/remember:
 - Why change? Most relationships are only to the user, structure comparable to documents/file structure, easier to visualize, better for unstructured data (e.g., data for graphs), can handle high throughput when uploading bank statements, easier to develop and maintain, efficient resource usage in Docker.
 - Risks: Delay in workflow due to security considerations, hardware constraints (RAM with large datasets), duplicate data concerns.

11/11 - 18/11:

- Successfully swapped over to NoSQL database.
- Improved testing quality, and file structure is significantly neater.

19/11 - 27/11:

- Created endpoints for submitting transactions, submitting files, and retrieving transactions based on userId.
- Things to consider: In DB I didn't embed transactions into user document as one user can have thousands of transactions. This allows scalability.
- Especially with the 16MB document limit.

27/11 - 02/12:

- Created visualization of graph on main page, users can view their transactions over time.

02/12 - 09/12:

- Report and presentation writing.

09/12 - 13/12:

- Refactoring code, getting rid of unnecessary or irrelevant dependencies, also modifying tests.

Bibliography

- [1] D. Choma, K. Chwaleba, and M. Dzieńkowski. The efficiency and reliability of backend technologies: Express, django, and spring boot. *Informatyka, Automatyka, Pomiary W Gospodarce I Ochronie Środowiska*, 13(4):75–77, 2023. doi: 10.35784/iapgos.4279. URL <https://doi.org/10.35784/iapgos.4279>.
- [2] J. R. Groff, P. N. Weinberg, and A. J. Oppel. *SQL: The complete reference*, volume 2. McGraw-Hill/Osborne, 2002.
- [3] C. Györödi, R. Györödi, G. Pecherle, and A. Olah. A comparative study: Mongodb vs. mysql. In *Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–6, 2015. doi: 10.1109/EMES.2015.7158433. URL <https://doi.org/10.1109/EMES.2015.7158433>.
- [4] D. Kunda and H. Phiri. A comparative study of nosql and relational database. *Zambia ICT Journal*, 1(1):1–4, 2017. URL <https://ictjournal.icict.org.zm/index.php/zictjournal/article/view/8/3>.
- [5] L. Lauretis. From monolithic architecture to microservices architecture. In *Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96, 2019. doi: 10.1109/ISSREW.2019.00050. URL <https://doi.org/10.1109/ISSREW.2019.00050>.
- [6] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. Echarts: A declarative framework for rapid construction of web-based visualization. *Visual Informatics*, 2(2):136–146, 2018. doi: 10.1016/j.visinf.2018.04.011. URL <https://doi.org/10.1016/j.visinf.2018.04.011>.
- [7] Y. Li and S. Manoharan. A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19, 2013. doi: 10.1109/PACRIM.2013.6625441. URL <https://doi.org/10.1109/PACRIM.2013.6625441>.
- [8] R. Moreno and R. E. Mayer. Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist*, 38(1):43–52, 2003.
- [9] M. Mythily, A. S. Arun Raj, and I. Thanakumar Joseph. An analysis of the significance of spring boot in the market. In *2022 International Conference on Inventive Computation Technologies (ICICT)*, pages 1277–1281, 2022. doi: 10.1109/ICICT54344.2022.9850910. URL <https://doi.org/10.1109/ICICT54344.2022.9850910>.
- [10] A. O’rinboev. The virtual dom, with its intelligent diffing algorithm, minimizes unnecessary updates and ensures faster rendering, making it a cornerstone of react’s efficiency. *Innovative Research in the Modern World: Theory and Practice*, 2(24): 54–57, 2023. URL <https://www.in-academy.uz/index.php/zdit/article/view/20263/13680>.
- [11] Jr. Scott, E. A. *SPA design and architecture: Understanding single-page web applications*. Apress, 2015.
- [12] Money Advice Service. Financial capability survey, 2018. Data collection, 2018.
- [13] S. Sharma. *Modern API development with Spring and Spring Boot*. Packt Publishing, 2021.

- [14] Y. Sheed, M. Qutqut, and F. Almasalha. Overview of the current status of nosql database. *Int. J. Comput. Sci. Netw. Secur*, 19(4):47–53, 2019. URL https://www.researchgate.net/profile/Mahmoud-Qutqut/publication/336746925_Overview_of_the_Current_Status_of_NoSQL_Database/links/5db083434585155e27f8236a/Overview-of-the-Current-Status-of-NoSQL-Database.pdf.
- [15] R. Vyas. Comparative analysis on front-end frameworks for web applications. *Electronics and Communication Engineering*, 2022. URL <https://doi.org/10.22214/ijraset.2022.45260>.