

Final Year Project Report

Full Unit - Final Report

Personal Finance Tracker Application

Benjamin Thomas 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

April 10, 2025

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 These have been acknowledged where other published and unpublished source materials have been used.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: 13312

Student Name: Benjamin Thomas Etheridge

Date of Submission: 10/04/2025

Signature:

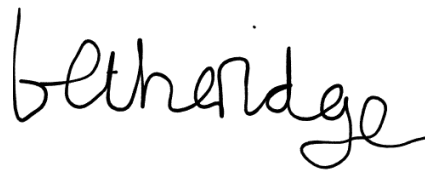
A handwritten signature in black ink, reading "betheridge". The signature is written in a cursive, lowercase style with a long horizontal flourish at the end.

Table of Contents

1	Introduction	4
1.1	Abstract	4
1.2	Project Motivations	5
1.3	Project Aims	5
1.4	Relevance To Future Career	5
1.5	Literature Survey	6
2	Literature Review	7
2.1	State-of-the-art web development	7
2.2	Architectural paradigms and applicable design patterns	9
3	The system to be implemented	12
3.1	Chosen tools and design	12
3.2	Planning	19
4	Software Engineering	26
4.1	Methodology	26
4.2	Testing	28
4.3	Documentation	29
5	End System	30
5.1	Architecture	30
5.2	Proof of Concept	31
5.3	Milestones	34
6	Assesment	35
6.1	Self Evaluation	35
6.2	Further Development	35
6.3	Professional Issues	36

6.4	Conclusion	37
7	Appendix	38
7.1	Diary	38

Chapter 1: Introduction

Software Demo Video Link: <https://youtu.be/KF9wTZsFXGk>

GitLab Repository Link: <https://gitlab.cim.rhul.ac.uk/zlac163/PROJECT.git>

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 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. The project is a state-of-the-art web application using the latest and optimal technologies. It will explore the various frameworks and tools available, as well as use modern architectures and designs to be a scalable and efficient financial tracking application.

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.

1.2 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 [16]. 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 [16].

Studies have shown that visual information is retained more effectively than written or verbal [9] 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 financially stable people, or to at least raise awareness, to help manage and track their finances, especially with students. Giving them a greater sense of financial awareness and hopefully encouraging them to start thinking for the future.

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 Relevance To Future Career

Working on this project has greatly expanded my knowledge of commonly used industry frameworks and standards. My research has raised my awareness and brought me up to date on modern industry practices. Using tools such as React, Spring boot, and MongoDB have provided me with invaluable experience in designing and creating web applications, which itself has a large share of the industry. I have also studied and used modern architectures and ideas such as Micro Services, which aim to improve scalability and flexibility. These concepts are some of the most important within the development of almost any application and will be invaluable tools I can apply as a software engineer.

1.5 Literature Survey

Before embarking on the design and implementation of web applications, it is necessary to understand the current landscape of technologies, methodologies and best practices.

Front end development focuses on the user experience of web applications. Over the years, the common solution has shifted from static HTML/CSS pages to dynamic interfaces, such as React, Angular and more. Due to their component-based design, they support building single-page applications (SPA) [15]. According to A. O’rinboev(2023)[13], React can offer better performance due to its virtual DOM. However, Angular provides a complete development ecosystem. Research comparing such frameworks highlights the tradeoffs made in the learning curve, performance and community support.

Backend development handles the server-side logic and data processing. It is often the control link between the Model and View in an MVC architecture. Frameworks such as Spring Boot, Django and more all offer their own advantages over each other. Recent studies L. Lauretis.(2019) [6] suggest a Micro Service architecture is increasingly becoming favoured over monolithic approaches due to its flexibility and scalability in large applications.

Data storage is a critical part of web applications; traditional relational databases like MySQL are still widely used for structured data and can support complex queries. However NoSQL databases like MongoDB and Firebase have gained substantial popularity for their flexibility with unstructured data, scalability and easy integration. Literature comparing these databases emphasises the choice of a trade-off, which is, again, dependent on the requirements of the application, such as data consistency, scalability, and flexibility.

The landscape of web development has a wide array of tools and approaches. Each offers its own unique strengths and weaknesses. Every decision made on choosing how to develop must be thoughtful as each involves tradeoffs in terms of performance, scalability, performance speed and more. As highlighted in the literature, there is no "best" technology; the effectiveness of the solution is heavily dependent on the specific requirements of the application being developed. Ultimately, a successful web application relies on selecting the right technologies that align with the goals it is trying to achieve.

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." [13]. This in particular would apply to a visual financial application, where charts and graphs would dynamically be updating and changing with applied filters or new data being added. 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." [19]. 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" [10]. 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, [3] 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 [18]. 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 [5], 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.

Many tools are at the disposal to assist with financial data; charting libraries such as charts or Chart.js can create dynamic graphs to represent spending trends or bar charts for monthly summaries. find ref for echarts They also support interactive elements such as hover tooltips and drill-downs into data. Component-based frameworks that can apply reusable UI components to transaction lists, filter controls, and account summaries are steadily available. These also allow for responsive, real-time updates for new data being added for filter changes. As such data is so sensitive, it is also important to consider the authentication they provide. Libraries such as Firebase Auth or JWT can secure user data to personalised financial dashboards. And protect financial data using sessions/tokens. Aside from security, efficiency is also one of the most important aspects of financial data processing. Applications will handle thousands upon thousands of data requests and processes. As such, an application should be designed to scale as efficiently as possible. Services such as SpringBoot provide options to optimise this by using stateless APIs for elimination session tracking between requests and asynchronous request handling for long-running calculations. Databases such as MongoDB are also optimised for this. Supporting sharding to allow data to be split across multiple servers, a flexible schema that can adapt to new additions to data without downtime. It can be optimised further with indexing and aggregating pipelines for performance under high query load. "In the realm of MongoDB, indexes serve a pivotal role in ensuring query execution efficiency." [11]

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. [15]. 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. [6].

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 [6]. 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.

In financial services, security and scalability are the primary concerns; it is important, therefore, to properly design and implement a solution that aligns well with the demands of modern financial applications, where user trust and efficiency are critical to success. The architecture plays an important role in separating the interface from the server services. A greater separation of concerns can enhance security and manageability, making it easier to implement features like user authentication, transaction history and such. The MVC design pattern ensures that there is a clear division between client, logic and data storage. Creating more organised code that's easier to debug and maintain. A must-have for large financial services.

In most modern financial apps, SPA's are becoming increasingly popular due to their performance [15]. They can provide a smoother and seamless user experience. Users are able to navigate through different sections, like their spending breakdown or setting up budgets, without the webpage reloading. As these applications tend to scale fast, a MicroServices architecture can provide many benefits. Being able to divide different tasks, such as handling user transactions and managing user budgets, into separate smaller and more manageable services. Each of which is responsible for its own section. Further pushes the idea of separation of concerns. Again, this allows for easier management and processing. It also allows for better horizontal scalability, optimising the app for handling spikes in certain services, such as during tax season when users will be rushing to bring up their monthly summaries. So, developers can commit more resources at these peak times without affecting performance. Design patterns are also widely used to manage complexity in financial apps. For instance, the observer pattern can allow real-time updates to financial dashboards. Users can be notified of changes, such as new transactions or updates to budgets. The command pattern can also be used for operations like undoing a transaction or reverting any changes. These patterns can help with functionality while keeping the app modular and maintainable. The design and architecture of an application will heavily dictate how secure it is. As such, financial applications need to take great care when choosing. Authentication features like JWT or 2FA before a user is even allowed access to their account is commonplace. Data encryption at every stage is also paramount for such applications, not only for user trust. But also comply with regulations such as GDPR. This is often done by using HTTPS tokens to encrypt data in transit; AWS can also safeguard data at rest. Combining these can help to protect data. Following an architecture such as MicroServices will also allow for security at each individual service. "The microservice style of highly isolated, easily deployable distributed components also implies new opportunities for better security." [20]. Offering several layers of security an attacker would have to breach to gain full access to a user's data.

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 [4]. 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 [8].

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 [17]. 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. [19]. 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." [7]. 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.

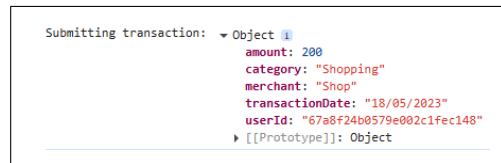


Figure 3.1: JSON Object sent from front end.

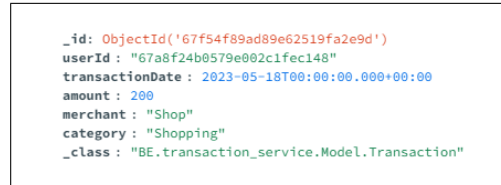
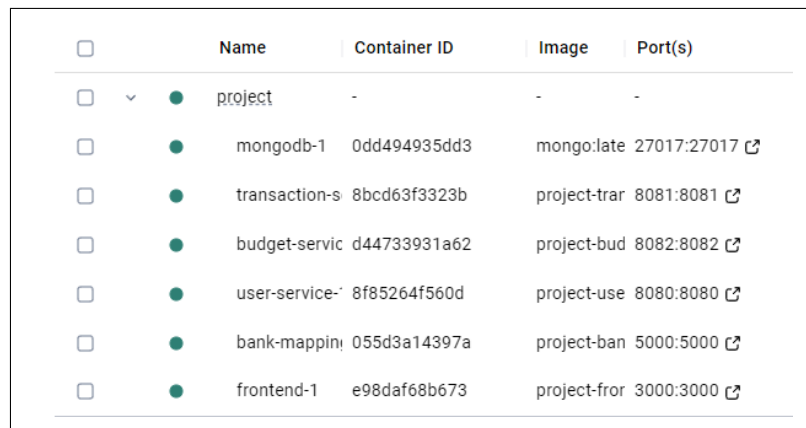


Figure 3.2: Document object stored in database shown on MongoCompass.

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.

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.



<input type="checkbox"/>	Name	Container ID	Image	Port(s)
<input type="checkbox"/>	project	-	-	-
<input type="checkbox"/>	mongodb-1	0dd494935dd3	mongo:late	27017:27017
<input type="checkbox"/>	transaction-s	8bcd63f3323b	project-trar	8081:8081
<input type="checkbox"/>	budget-servic	d44733931a62	project-bud	8082:8082
<input type="checkbox"/>	user-service-	8f85264f560d	project-use	8080:8080
<input type="checkbox"/>	bank-mappin	055d3a14397a	project-ban	5000:5000
<input type="checkbox"/>	frontend-1	e98daf68b673	project-fror	3000:3000

Figure 3.3: View of Docker Containers in Docker Desktop.

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 Planning

The Application development process will be split into two major sections; the interim section will focus on building the architecture and design of the application. The second stage will be set up to heavily push out core functionality that the User will see. It will also be where the security and optimisation will be added. Each section will also be further split into three sprints, allowing for continuous development and a consistent timed plan throughout the whole process.

3.2.1 Initial Plan

The initial part of the project focuses on establishing the core components of the system, ensuring a strong foundation for further development.

The first sprint will be to create the application itself and establish communication between the front, back, and Database. It will initially be a monolithic design while I create the initial user-specific structure. By the end of this, a user will be able to log in to their account with data stored in a database specific to user details. All necessary validation will also be included.

The second sprint will transfer to a MicroService architecture with the introduction of the transaction service; this service will be able to parse preformatted csv files into transaction objects and store them on a separate transaction database, keeping user data and transaction data separate but linked through the userID as a foreign key for the transaction database. At the end of this sprint, users will be able to upload a preset CSV file and add transactions to their account, with the application being able to display said data onto a list of transactions.

The third sprint initially was to introduce filtering functionality; however, with the deadline for the early deliverables approaching, I decided to adjust the goal to display the User's data onto a basic graph. I thought that this was a more important deliverable to produce than filtering, as it would help showcase the intention of my application.

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. These deliverables were then placed onto a Miro Board to help me visualise the progression path that I needed to follow.

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

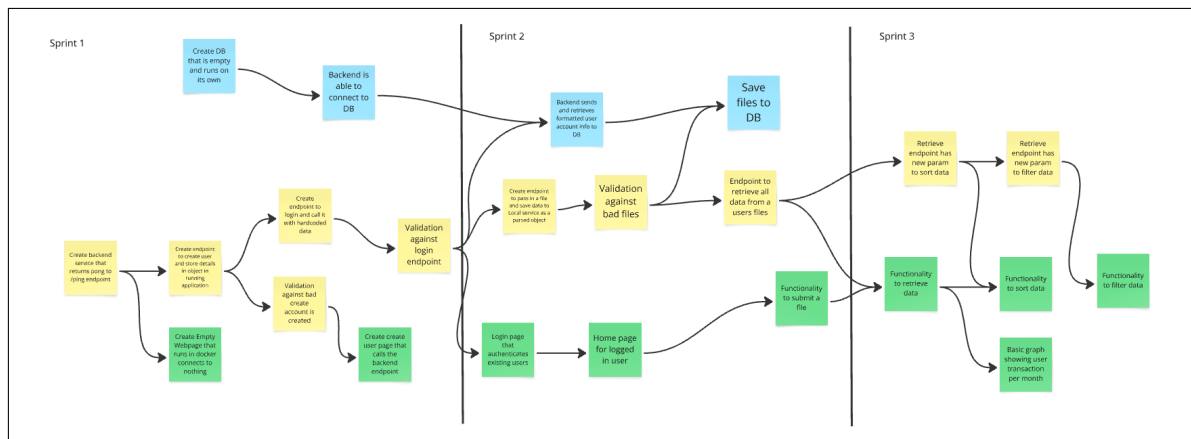


Figure 3.4: Miro Board Initial Progression Plan.

Initial Plan Timeline

- **Week 1–2:**
 - Complete the first 5 yellow tickets.
 - Implement account creation and login via backend API endpoints.
 - Ensure the login endpoint returns/logs messages indicating successful login.
- **Week 3:**
 - Focus on the next 3 green tickets.
 - Enable account creation and login through the web interface.
 - Data is temporarily stored in the backend; the database should not exist yet.
- **Week 4:**
 - Set up the database.
 - Store account credentials in the database instead of backend memory.
- **Week 5:**
 - Implement backend endpoint for file uploads.
 - Add input validation logic.
 - Create an endpoint to retrieve stored data.
 - Store uploaded data in the database.
- **Week 6–7:**
 - Add frontend functionality to upload files and retrieve all file data.
 - Ensure full integration with the backend and database.
- **Week 8–9:**
 - Add backend functionality to filter data based on parameters.
 - Implement corresponding frontend features to apply and display filters.

3.2.2 Final Plan

It is important to note that the plan for the final deliverable changed drastically based on the progress of the early deliverables. For instance, I decided to push filtering further into the progress as it was missed during the early stage, and I felt I wanted to introduce more important functionality first. This part of the project will focus on developing the core components, creating the remaining features, and implementing the overall design. The deliverables aim to give full functionality to the User, further develop current components and optimise the security and scalability of the design. The following goals are again split into three sprints and further split by the three core components. Once completed, the application should be able to provide financial data visualisation in an efficient and secure manner. All use cases should be met, and the system should have an appealing design.

In the first sprint, I wanted to fully split the services I had into separate microservices that are fully independent and communicate with each other using APIs. I also wanted to dockerise these services so each could be separately built during development and not have to rebuild the entire set.

The second sprint brought most of the functionality. It introduced the category parameter that is ready for filtering, manual transaction adding, and editing. I also need the microservice to map actual bank statements to the necessary data I need on my application.

The final sprint was intended to bring the user interaction element to the application. Users were able to filter, view graphs and set budgets. This brought the whole application together, especially from the User's point of view. This meant that all use cases were achieved.

Final Deliverable Objectives

Backend

- Implement APIs for CRUD operations on transactions, including categorisation.
- Design a microservice architecture.
- Develop APIs for communication between services.
- Define a **BankMapping** model to store mappings between bank-specific CSV columns and a standard format.
- Integrate a CSV parsing library to support different formats.
- Map CSV columns to the standardised format.
- Add robust error handling for malformed CSV files.
- Integrate a basic machine learning model to infer categories when not set.
- Implement API endpoints to support data filtering by parameters.
- Develop API endpoints for managing budget data.
- Ensure budgets remain within acceptable limits and handle edge cases.
- Design and implement basic algorithms to analyse transaction data and generate insights.
- Create an endpoint to retrieve insights data.

Frontend

- Implement a cyclical structure to support user login and logout.
- Develop a transaction page supporting CRUD operations on transactions.
- Allow users to manually update categories for preset transactions.
- Create UI components for filter criteria.
- Implement dynamic updates to the transaction list based on applied filters.
- Build a budget setting page for users to allocate budgets per category.
- Develop a graph charting page.
- Enable users to set and apply filters to graphs.
- Add an insights tab to the main page.
- Polish and refine the UI.

Database

- Set up server-specific databases.
- Add a `category` field to the transaction model.
- Create a separate database for bank mapping data.
- Create indexes on commonly used filters to optimize query performance.
- Create a `Budget` table with appropriate fields.
- Establish relationships between budgets and categories.
- Store insights logs in a separate database related to the user.
- Optimize queries and indexes for efficiency.

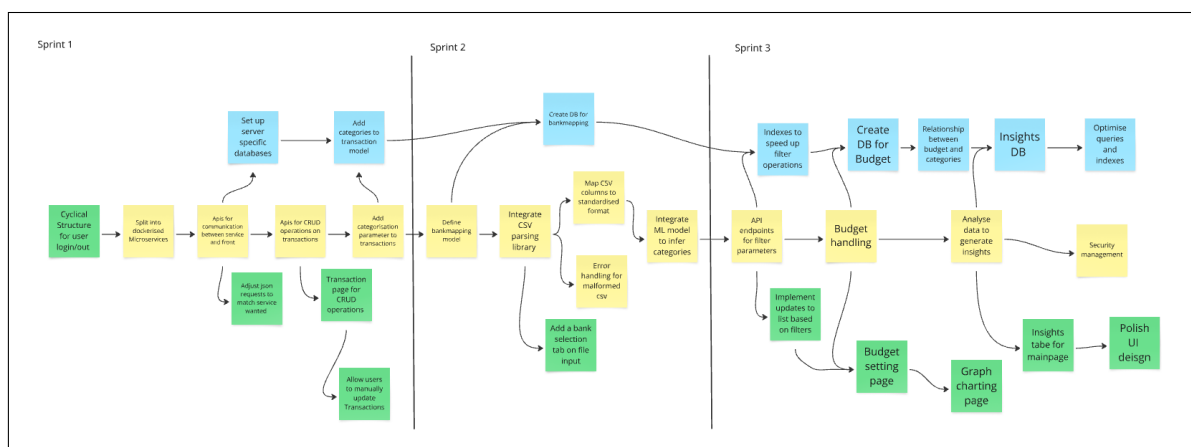


Figure 3.5: Miro Board Final Progression Plan .

Final Plan Timeline

Week 1

- Complete the first 3 tickets and the first blue ticket.
- Application should now have a cyclical structure and be split into microservices.

Week 2

- Add categories as a parameter for manual transactions, including categories in the transaction database.
- Create the transaction page where users can perform CRUD operations on their transactions.

Week 3

- Implement bank mapping. Data handling should be dynamic as banks will format data differently.
- Files should be parsed, bank format detected, and the necessary columns mapped to the standardised format.
- Add error handling for any inconsistencies or issues with file formats.
- At this point, users should be able to add transactions from any bank statement, and the data will be parsed and formatted according to application parameters (excluding categories, which should be null for now).

Week 4

- Add automatic categorisation for transactions from bank statements.
- Begin with preset categories for the most common merchant names.
- Unrecognised merchant names will be assigned to a “MISC” category, which users can change manually.
- Implement a basic machine learning model to determine the best-suited category for a transaction based on previous transactions (only considering the merchant name).

Week 5

- Implement filtering functionality by any transaction parameter.
- The transaction list should update dynamically based on the applied filters.

Week 6

- Focus on implementing budget functionality and management.
- Users should be able to set budgets for each category as well as an overall budget.
- Establish relationships between budgets and categories for budget charts and filtering.

Week 7

- Implement the graph charting page.
- Allow users to apply filters to both budget and expenditure charts.

Week 8 (IF NOT BEHIND)

- Move to the insights tab on the main page.
- Create basic algorithms to analyse user data and provide insights (e.g., "You tend to spend £80 a month over on your essential shopping budget").

Week 9/10

- All core functionality should be complete by this point.
- Spend time focusing on UI improvements and polishing the frontend.
- Add security features to the backend.
- Optimise queries and indexes for the database.

Chapter 4: Software Engineering

4.1 Methodology

The decision to adopt an agile approach was driven by the dynamic nature of the project. Given that it required flexibility in its design and iterative development, it helped me to see that often needed adjusting based on new insights. Even though I was working alone, the principles allowed me to focus on creating functional chunks of the application that can be tested and improved constantly. The approach provided me with a structure while allowing me to adapt as requirements arose. Also, although I didn't have a team, I found that applying this structure to my own workflow definitely helped me to stay organised and on track. To implement agile into my project, I divided the work into 3-4 week sprints, each focused on specific milestones. For example, the initial plan for Sprint 2 focused on parsing a CSV file with transaction objects (Figure 4.1). Each sprint had a clear set of objectives, which allowed me to measure progress and deliver incremental value to my project.

Backlog grooming was one of the most important aspects I engaged in throughout the project. It ensured that the tasks I was working on were aligned with the project's evolving needs and that I didn't get ahead of myself. For instance, towards the end of the interim phase, I noticed that my project didn't yet have much to show. So, I focused on the graphical representation of data before filtering, which was mostly backend work. A key aspect of Backlog grooming is ensuring I didn't get overwhelmed by the scope of the project. I would constantly go back to evaluate tasks and break down main use cases into the smaller actionable deliverables seen on the miro boards (Figure 3.4, Figure 3.5). This made it easier to progress without feeling stuck and overwhelmed during the more complex parts of the project.

I also used the principle of continuous improvement. Each sprint allowed me to reflect on my work and identify which areas needed improvement. For example, I initially underestimated the difficulty of mapping CSV bank statements due to a lack of testing, which led to delays in the final plan sprint 2. So, I adjusted my approach to include more unit testing to help find errors better and earlier for further sprints.

The more the project progressed, the more I became efficient in planning, estimating and delivering functional chunks. Following the principles of agile, I was able to prioritise my tasks better and improve the overall quality.

Interim Sprint 2**Duration:** 30/10/2024 – 21/11/2024**Goal:** Split into Microservice Architecture and Formatted CSV Parsing.**Overview:**

Formatted CSV files can now be parsed into transaction objects, allowing users to bulk import transactions as well as manual transactions.

Completed Use Cases so far:

- User Signup
- User Login
- Add transactions
- (Half completed) Add CSV file

Backlog Review:

- Transaction and User not fully split into microservices, although separate, they still need to rely on each other to build properly. Services should be able to run without the other.
- Users can only add preformatted CSV files or "fake" bank statements. Need a service to map actual bank statements to the desired format, which can then be processed.

Challenges:

- Validation issues when testing mock bank statements. Mockito tests would struggle with parsing CSV formatted files.

Retrospective Items:

- What went well – parsing CSV file into transaction objects and then storing in Database.
- What didn't go well – Testing of CSV file often failed due to incorrect usage of Mockito.

Next Sprint Additional Planning:

- Taking longer than anticipated – filtering may be pushed back to introduce basic graph on frontend before interim due date.

Figure 4.1: Initial Plan Sprint 2 Overview

4.2 Testing

Testing is essential to ensure the software works as expected and is kept reliable, especially when dealing with financial data. The primary process followed during development was Test Driven Development. I would design and write tests for the next function of my application before writing the code for the function itself. This would ensure that the code I'm writing is serving its intended purpose and detect any bugs or errors as early as possible. To keep my testing as efficient as possible, I followed the testing pyramid concept, "addressed by these sorts of tests and be implemented at the level considered most efficient and effective." [2] where I stuck to many unit tests compared to fewer integration and end-to-end tests.

Unit Tests

Most of the testing was done on the unit level; using JUNIT and Jest, I was able to create a wide array of tests to target each specific part of logic within my code base. These tests are automated and quick to run, inexpensive and will run during every build, allowing for early detection of bugs.

Integration Tests

These tests focused on testing interactions between components. As the application followed a microservice architecture, it was essential that this was rigorously tested. Most of the testing in this section was done manually using PostMan. This lets me create API calls with every format of data I want and see the return. It was mainly used to check that the validation worked correctly between services.

End to End Tests

These tests would interact with the entire application on a high level. Intended to simulate real user interactions. They would often follow an entire user story, such as a user logging in, adding a transaction and then seeing new data adjust the graph.

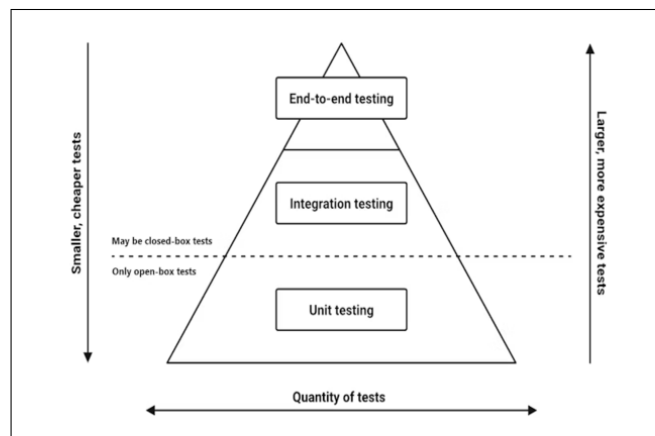


Figure 4.2: Testing Pyramid Example [14].

4.3 Documentation

Documentation plays a vital role in helping end users and also other developers understand the system. The project comes with documentation of overall design concepts, as well as Schema for the databases. This, along with the user manual and code documentation, should allow users or developers to easily understand the purpose of the project and all of its code. Code documentation will help developers understand the purpose of each piece of code, as well as its expected behaviour. Although it is not expected that other developers will expand on this project, it is still fully documented with JavaDoc for the Spring Boot applications and JSDoc for the React components. This has also assisted me in easily understanding previous functions I may have created months prior, such as old API calls, helping along the development process. APIs are also documented to include signatures, expected inputs and outputs, and error codes. Allowing for easy integration of services. On the user side, a user manual will assist users in understanding how the application works and how to use it effectively. Although it has been designed to align with UX design principles, it is still good practice to include a full user manual describing every aspect of usage.

Chapter 5: End System

Overall, the project followed a structured approach, utilising the agile methodology to ensure flexibility and continuous improvement. Throughout the project, the focus was on creating a system to efficiently manage financial data, provide insights and visual representations, and ensure security. By using a combination of technologies and architectural ideas, along with extensive testing and refinement, the project evolved into a functional application that met most of the specified requirements.

5.1 Architecture

The final architecture design of the project consisted of a single-page application front end, several microservice backends, and multiple databases for each type of data. This allowed for easy separation of concerns in all aspects of the application. Letting each service focus on specific responsibilities makes development and deployment easier. Also, the project would be more maintainable and scalable if it were to extend to large quantities of users. Each service is built in its own docker container, letting the application be deployed over multiple services if needed and making it easier to scale horizontally.

The SPA front end built in React.js has ensured a smooth and responsive user experience, with dynamic updates and navigation without having to reload the page, which was a primary goal of my application. Each MicroService backend can handle its distinct business logic, such as transaction management, budgeting, bankmapping and more. This modularity simplifies the codebase and gives the ability to scale each separate container based on the demand at a specific instance. The flexibility of the application allows transaction documents to be submitted while potentially not being completed. The use of MongoDB documents meant I could separate databases for different types of data such as User, transaction, and budget. This lets me easily categorise the data and maximise performance through data isolation. I found it not necessary to have a separate database for categories, as I decided to have a set of premade categories. This is similar to other financial applications and will reduce the complexity of dynamically adjusting the frontend charts to suit User-made categories.

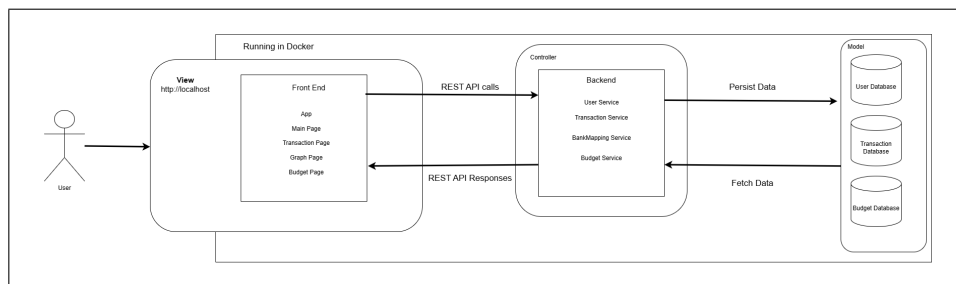


Figure 5.1: MVC Architecture UML diagram.

5.2 Proof of Concept

The final system application can be split into four core functionalities. Each feature corresponds to specific services and components. These features work together to deliver a seamless user experience for managing and visualising the User's financial data. While the current implementation is focused on core functionality, it provides a strong proof of concept for what could easily become a full-scale financial platform. The application demonstrates how modular services, clean interfaces and efficient data flow can lead and scale to support a much larger and more complex system. Being able to handle thousands of users and millions of transactions. And providing real-time data processing and analytics.

It is important to note that the transactions in the following images are mocked.

User Login/Sign Up

This feature serves as the entry point into the application, enabling secure user access and identity management. It is responsible for handling the user credentials, registration data and authentication. Upon a successful login or signup, it will grant access to the User's dashboard and fetch the unique `userId` to which the User's financial data will be linked. This prepares the application ready to process the specific User's data. Its functionality is implemented across the frontend interface, the user service and its dedicated user database.

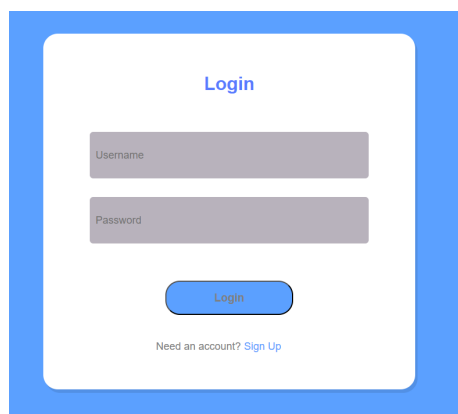
The image shows a mockup of a login interface. It is a white rounded rectangle with a blue border. At the top, the word "Login" is written in blue. Below it are two grey input fields: the first is labeled "Username" and the second is labeled "Password". Below these fields is a blue rounded button with the word "Login" in white. At the bottom, there is a link that says "Need an account? Sign Up" in blue.

Figure 5.2: Login Tab.

The image shows a mockup of a sign-up interface. It is a white rounded rectangle with a blue border. At the top, the words "Sign Up" are written in blue. Below it are three grey input fields: the first is labeled "Email", the second is labeled "Username", and the third is labeled "Password". Below these fields is a blue rounded button with the words "Sign Up" in white. At the bottom, there is a link that says "Already have an account? Login" in blue.

Figure 5.3: Sign Up Tab.

Main Page

This feature served as the central hub linking all the other features, it let users navigate to other pages and tabs, log out, as well as showing a list of the users recent transactions and a small summary graph of their overall spending and budget for the most recent 6 months of their transactions. This brought the whole application together and as such would be interacting with every other page, aswell as most backend services.

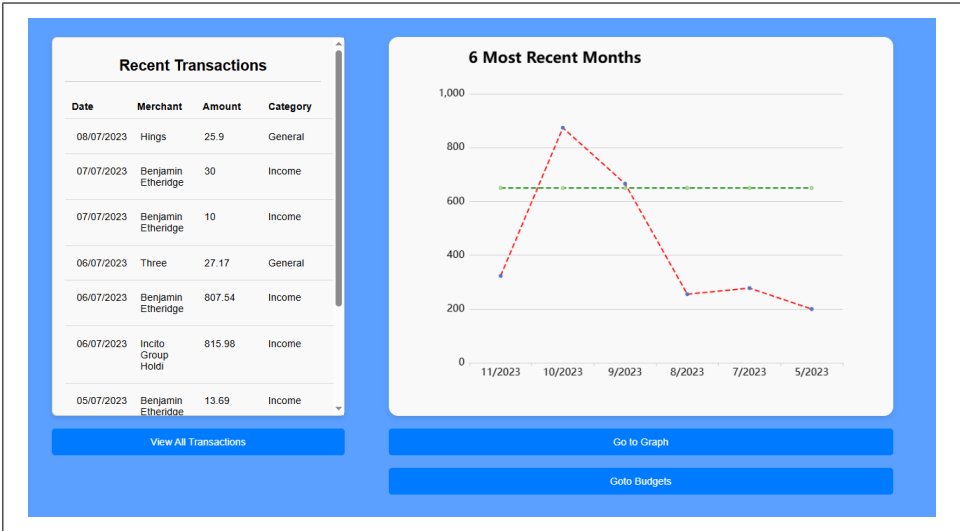


Figure 5.4: Main Page.

Transaction Page

The transactions feature forms the central part of the application’s financial tracking. It manages all user-related transactions, allowing users to perform CRUD operations on their records. This involves coordination between the transaction and bank mapping services, front end, with a separate transaction database. The system ensures each transaction is correctly associated with a user via their User, supporting all CRUD operations and giving a comprehensive view of the User’s financial activity.

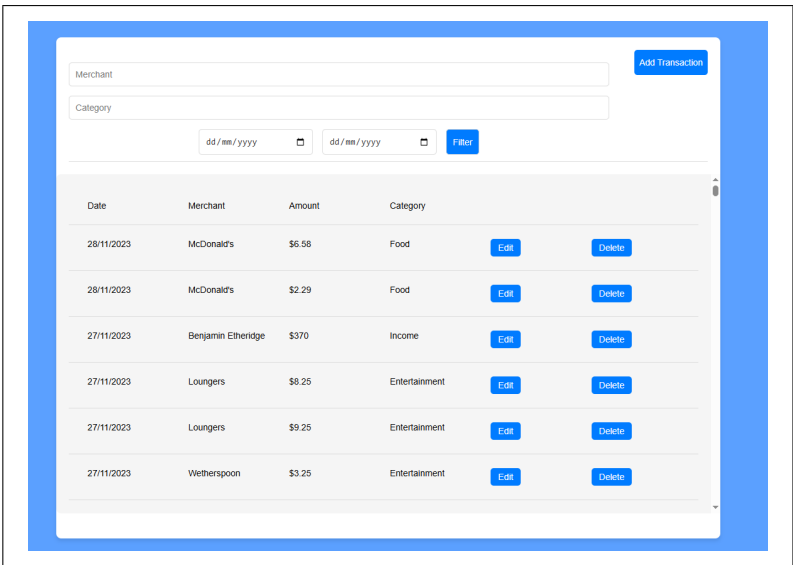


Figure 5.5: Transaction Page.

Budget Page

The budget feature allows users to set, update and view their monthly spending limits across the preset categories. Its logic is handled through the budget service, frontend components and a separate budget database. This functionality will ensure that users can effectively plan and monitor their finances by comparing their spending against their budgets.

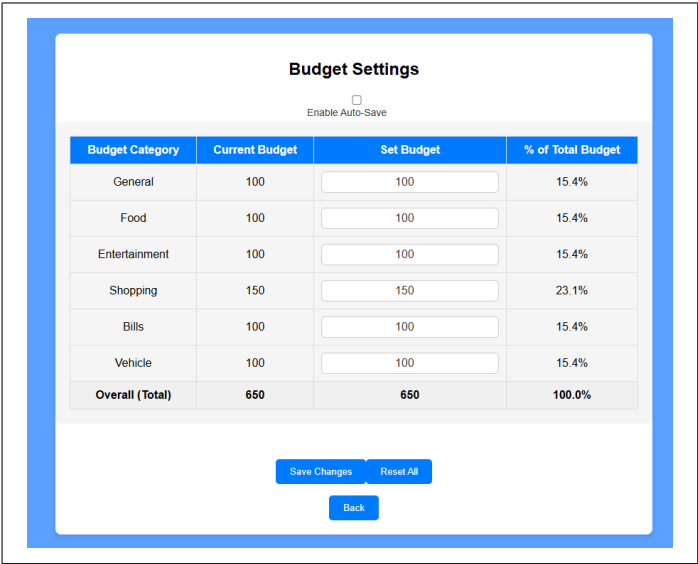


Figure 5.6: Budget Page.

Graph Page

The graph feature serves as the visual representation of the User’s financial information. It brings together information from both the transaction and budget services to generate dynamic and informative graphs. This allows the users to easily track their spending patterns, compare expenses against budgets, and gather insights into their financial behaviour over time. The logic for this resides mainly in the front end but relies on backend services for data retrieval in real time.



Figure 5.7: Graph Page.

5.3 Milestones

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.

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. Although not all of the separate tasks were achieved, the application provides most of the functionality planned and all but the last use case was met.

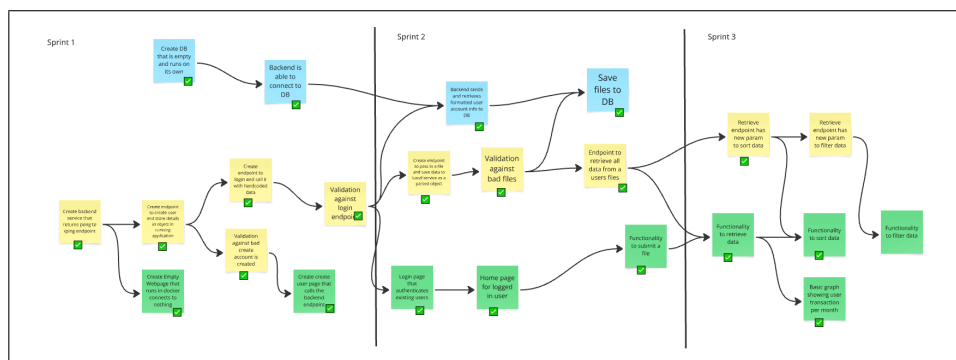


Figure 5.8: Completed Tasks for Interim.

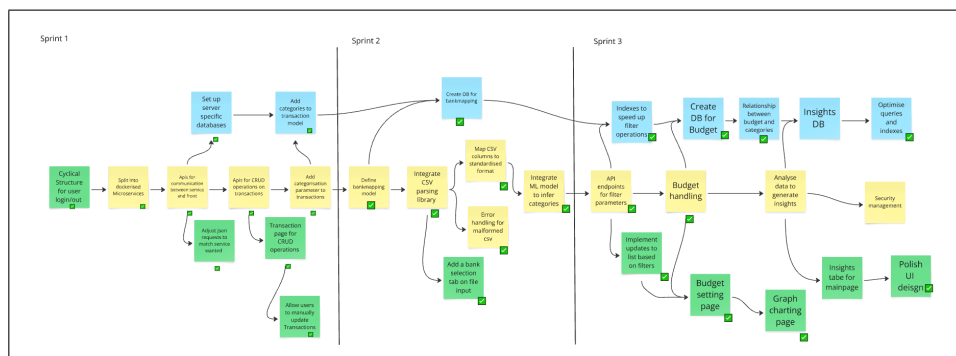


Figure 5.9: Completed Tasks for Final.

Chapter 6: Assessment

6.1 Self Evaluation

Throughout this project, I have followed industry standards and agile methodologies. The application itself has become a robust and scalable solution, from front-end design to backend microservices, letting it be a fully modular design. I flowed agile practices with dedicated sprints and regular reviews. This helped keep development aligned with the core goals. Although the project has core functionality, it lacks significant security. The application, as it stands, only offers basic protection and would need to be enhanced on a full-scale application. This should have been a higher priority based on the sensitivity of the data; however, I decided to commit to functionality to develop a solid proof of concept in time.

Like any complex software project, the process came with a range of challenges, both technical and structural. These obstacles helped to test the robustness of the architecture chosen and offered learning opportunities that helped with decisions moving forward. One significant issue was the slower feedback loop created when trying to render the react page in a dockerised environment. While Docker does provide consistency across multiple environments, it introduced an overhead with React on my project. React benefits the developer by fast reloading the page while developing. However, running the app inside a container meant the application had to be rebuilt every time, costing around 60 seconds every time a new addition was made. This couldn't be mitigated by running the front end outside of Docker due to the application being connected and dependent on multiple microservices inside the container.

6.2 Further Development

Given more time and before considering a full-scale deployment, there are several features I would see completed. These features would be absolutely necessary to have an end-product that is more than able to withstand the difficulties it would face in an environment such as the web, where attacks and high-volume traffic would be an absolute certainty if the app were to grow in popularity.

The application would need to have at least a computationally secure design due to the nature of its data. This would involve protection against attacks such as NoSQL injections, cross-site scripting and many more. Alongside this, the application would need an SSL certificate, which brings in the aspect of having to pay for security from providers. These sorts of measures, however, are necessary in the real world, aside from the security they provide. They also are a symbol that the User can see so they know their data is being protected.

Security would be the highest priority not only for user trust but also for data protection and compliance. With regulations like GDPR that will require protecting this type of personal data, my app will have to "implement appropriate technical and organisational measures to ensure a level of security appropriate to the risk" [12]

The architecture of my project has been set up to suit a cloud-based deployment. The Micro Service design means the project can scale up and down efficiently when there is demand. Cloud-based services like AWS would be able to manage operations like this, ensuring that demand is always met while being as cost-effective as possible. Like certain security features, this would also cost money and is only a realistic option if the application was on a full-scale deployment. These services also provide some additional security features.

I would also like to finish my intended implementation and have an insights page; this would greatly boost the user experience and assist users in understanding their finances better, which is the entire point of the application. Regrettably, this feature of the project was not completed in time for the final deliverable product due to time constraints.

Although the application is already set up to scale efficiently, there are other features that could be added to further enhance the project's flexibility. A load balancer would be a good feature to introduce. It would assist with traffic management and resource optimisation and even help protect against DDoS attacks.

6.3 Professional Issues

With the increasing reliance on web applications to manage personal financial information, user privacy is an ever-expanding ethical and professional concern. Web developers working on platforms that handle sensitive financial information must address a wide range of legal, technical and moral responsibilities to protect their User's information. Data such as income, spending habits, and transaction histories are highly sensitive. Unauthorised access to data like this can lead to identity theft and financial fraud. As such, developers need to have strict data handling policies, including encryption through transit and at rest. Secure authorisation and access control based on the "principle of least privilege" [12] are also essential. In addition to a moral obligation to protect data, developers must comply with technical safeguards and privacy regulations. Acts such as GDPR are essential legislation and minimum guidelines that developers must legally follow. These regulations set the expectations of how data should be handled and command user protection through transparency in data usage, user consent and the right to access and delete personal data. From a professional standpoint, respecting user privacy also means avoiding unnecessary data collection and making sure that the data that is collected is only used for its state purpose. Developers must remain vigilant about third-party integrations, such as cloud-based services, that could potentially introduce external privacy risks. To summarise, handling personal financial data in web applications will require a privacy-first approach that is centred around ethical principles and legal obligations. Developers have the duty to protect users' trust and design a system that is secure and transparent and respects the users' rights over their data.

One of the most prominent examples of privacy violations in the tech industry was the Facebook Cambridge Analytica scandal. In 2018, it was revealed that Cambridge Analytica had harvested the personal data of millions of Facebook users without consent. It used this data for targeted political advertising for the U.S. Presidential election. The scandal raised significant questions about Facebook's responsibility to ensure data privacy and the extent to which third-party applications were able to access private information. The failure of Facebook's data protection measures resulted in a large settlement as well as a wide global backlash. This did, however, help to bring awareness to greater scrutiny surrounding privacy laws such as GDPR and even served as a catalyst for the Data Protection Act (2018) in the U.K. It is important to note that the data harvested wasn't even specifically financial data, and so it brings further importance for a developer to ensure the protection of their users' financial data.

My project, which involves building a web application that specifically handles financial data, I have encountered several privacy and ethical challenges related to how sensitive user information is managed. The application relied on tracking user spending habits as well as their income to their bank accounts. Both of which are heavily sensitive. During planning, I had to decide between creating a functional application within the given time frame or creating a secure application that would have minimal, if any, core functionality. Due to the limited resources and time at my disposal and discussions with my supervisor and peers. I opted for a functional application that does not have an acceptable standard of security. The created application itself is a proof of concept for what could be a highly scalable and efficient personal finance tracker. Multiple security features would need to be implemented, such as cross-site scripting protection, NoSQL injection protection and more. Some essential features would also require a payment or subscription to protect the website's data if deployed. Certificates such as SSL/HTTPS are needed to keep data encrypted at all stages.

6.4 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. 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. The completed project is a demonstration of incorporating modern architectures and technologies to address a real world problem. Future work may focus on more optimization, user feedback-based updates, and incorporating new technologies as they emerge.

Chapter 7: **Appendix**

7.1 **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.

13/01 - 19/01:

- Set up plan for second term, establish goals, and research ideas on how to progress.

20/01 - 26/01:

- Established microservice architecture, separating backend into services and establishing communication between them.

27/01 - 02/02:

- Created transaction page, cleared up main page and linked pages.

03/02 - 09/02:

- Not much done this week, basic bug fixing and test building.

09/02 - 21/02:

- Research into Machine Learning algorithm to automatically categorize transactions, might not be doable in the timeframe.
- Redesign and building of transaction page to handle file submits.

22/02 - 30/02:

- Decided against a machine learning algorithm, not enough time, and just used basic highest chosen for automatic categorization.
- Finished off CRUD operations for transactions as well as general bug fixes for transaction page and login page.

01/03 - 07/03:

- Created the bank mapping service, users can now add bank statements from Starling Bank, and it will process into transactions.
- Uses hashmap counter to find most popular category with the same merchant to decide on the category for bulk imports.

08/03 - 18/03:

- Nothing done this week, had other priorities with other courseworks.

19/03 - 26/03:

- Enabled filtering by parameters, users can now filter their transactions to show the ones they want.

27/03 - 30/03:

- Budget setting page, users can set budgets for each category.
- Overall budget is also applied to the general chart.

31/03 - 02/04:

- Charting page, just uses filter and charting already created, so it is sorting design and structure.
- Users can apply filters to chart to view different parts such as yearly food transactions.

03/04 - 10/04:

- Finalizing report and also improving general design and bug fixes.

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] A. Contan, C. Dehelean, and L. Miclea. Test automation pyramid from theory to practice. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5. IEEE, May 2018.
- [3] J. R. Groff, P. N. Weinberg, and A. J. Oppel. *SQL: The complete reference*, volume 2. McGraw-Hill/Osborne, 2002.
- [4] 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>.
- [5] 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>.
- [6] 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>.
- [7] 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>.
- [8] 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>.
- [9] R. Moreno and R. E. Mayer. Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist*, 38(1):43–52, 2003.
- [10] 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>.
- [11] M. Nuriev, R. Zaripova, O. Yanova, I. Koshkina, and A. Chupaev. Enhancing mongodb query performance through index optimisation. In *E3S Web of Conferences*, volume 531, page 03022. EDP Sciences, 2024.
- [12] Council of the European Union. Council regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (united kingdom general data protection regulation), 2016. URL <https://www.legislation.gov.uk/eur/2016/679>. Text with EEA relevance.

- [13] 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>.
- [14] Jacob Schmitt. The testing pyramid: Strategic software testing for agile teams, 2024. URL <https://circleci.com/blog/testing-pyramid/>.
- [15] Jr. Scott, E. A. *SPA design and architecture: Understanding single-page web applications*. Apress, 2015.
- [16] Money Advice Service. Financial capability survey, 2018. Data collection, 2018.
- [17] S. Sharma. *Modern API development with Spring and Spring Boot*. Packt Publishing, 2021.
- [18] 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.
- [19] 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>.
- [20] T. Yarygina and A. H. Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, March 2018.