# CST2550 Group Coursework – 'Fundify'

Qais Abdelshah • Ahmed Elbehri • Arash Ahangar • Ben El Ansari • Wayne Kuvi

## Introduction

This project involved the development of '**Fundify'**, a simple and user-friendly crowdfunding platform tailored specifically for product systems. The website allows new users to register for an account, create and manage their own campaigns, donate to others, and interact with the community by commenting on active campaigns. The main goal was to design an intuitive, accessible platform while ensuring the underlying system remains efficient, secure, and scalable.

This report covers the key stages of the project, beginning with the design decisions made around core functionalities, including the use of wireframes to create quick mock-ups and guide the layout and structure of each page. It also discusses the algorithms and data structures selected to support these features. The report then outlines the testing approach used, along with specific test cases implemented to validate system behaviour. Finally, it concludes with a critical reflection on the project, highlighting its limitations and offering suggestions for improvement in future development. Additionally, we held two meetings per week to the best of our ability while always staying in contact via emails and messages.

## Design

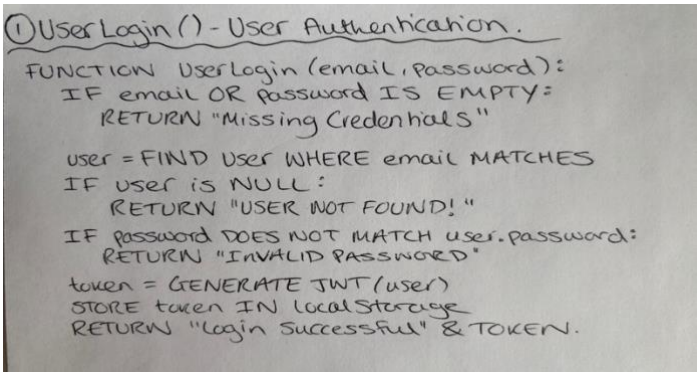
At the start of our project, we focused heavily on planning the user experience and ensuring our website would be intuitive and functional. Together, we created several wireframes and mockups using Balsamiq, which are included below. These mockups helped us visualise the layout of essential pages such as the homepage, login/register, and campaign viewing sections. They also served as early references for both frontend and backend development.

To design a realistic and user-friendly crowdfunding platform we made thorough research by examining existing platforms including Kickstarter, Indiegogo, and Crowdcube. Ben and Ahmed analysed core features such as campaign browsing, filtering, contribution methods, and campaign creation. We extracted and discussed the most useful and consistent patterns from these platforms to replicate in our own application.

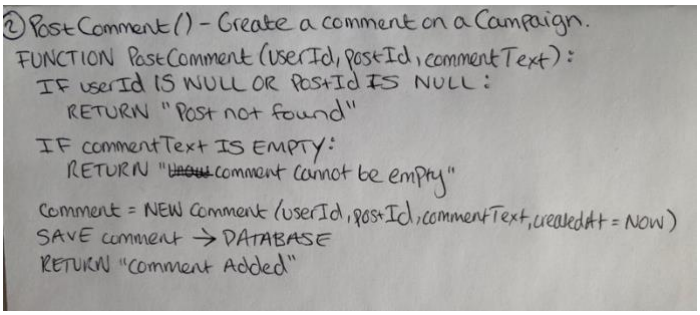
Through regular sprint meetings we collaboratively decided on the core functionalities of our site: User registration and login with authentication, launching new campaigns, viewing and donating to campaigns and posting comments.

Once we finalised these features, we began creating some pseudo code for the major backend operations. The aim was to map out logic before implementation to make the transition into code smoother. Examples of pseudo code created during team meetings shown below:

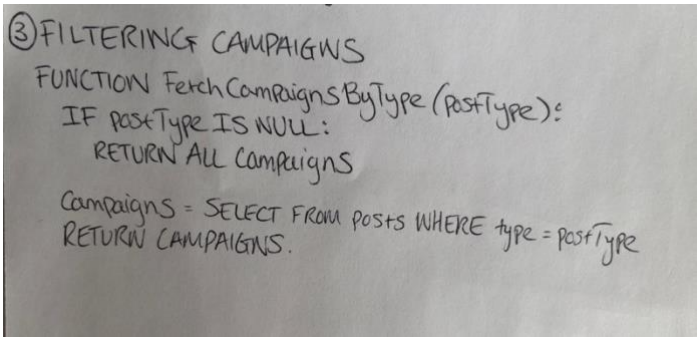
# Psuedo-Code



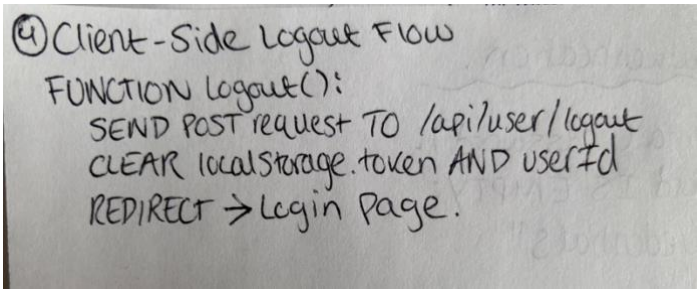

The UserLogin function is responsible for handling the authentication process when a user attempts to log in. It begins by checking if either the email or password fields are empty in which case it returns a "Missing Credentials" error. If both fields are filled it searches for a user in the database whose email matches the input. If no such user is found, it returns a "User Not Found" message. If the user exists, the function then compares the input password with the stored password. If the passwords don't match, it returns "Invalid Password." If all validations pass, a JWT is generated for the user, which is then stored in the browser's localStorage. This token is used for authenticating future requests. Finally, the function returns a "Login Successful" message along with the token.



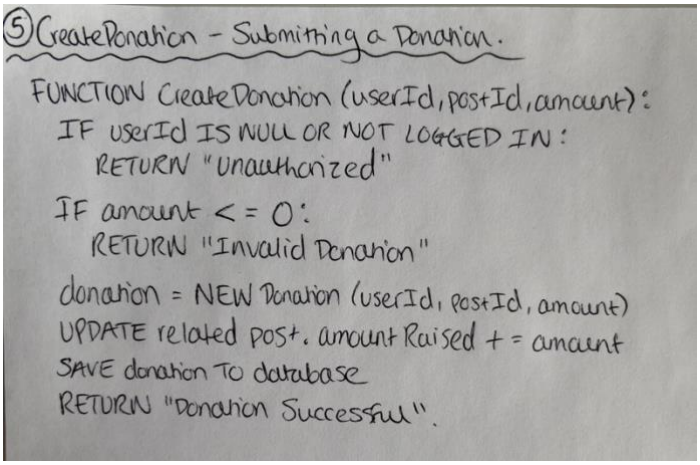

The PostComment function handles the logic for adding a comment to a specific campaign post. It first checks whether the userId or postId is missing, indicating that either the user is not logged in or the post does not exist; in that case, it returns an "Post Not Found" message. If both IDs are valid, it then checks whether the comment text is empty. If it is, the function returns "Comment Cannot Be Empty" to prevent blank submissions. If all input is valid, a new comment is created using the provided user and post IDs along with the comment text and the current timestamp. This comment is then saved to the database, and function returns a message: "Comment Added."




The FetchCampaignsByType function is designed to retrieve campaigns based on their assigned category or type. It begins by checking if the postType value is null. If no specific type is provided, it simply returns all campaigns from the database. However, if a type is specified the function filters the campaigns and selects only those whose type matches the given postType. The filtered list of campaigns is then returned, allowing users to easily browse content based on interest or category.



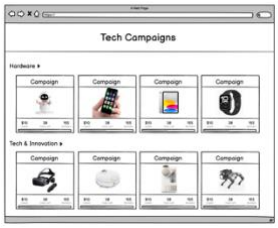

The Logout function manages the process of securely logging a user out of the application. It begins by sending a POST request to the backend's /api/user/logout endpoint, which can handle any necessary server-side cleanup. After this, the function clears the user's session data stored in the browser's localStorage, specifically removing the JWT token and user ID. Finally, it redirects the user to the login page, effectively ending the session and preventing further access to protected features until the user logs in again.




The CreateDonation function handles the process of allowing a logged-in user to donate to a specific campaign. It first checks whether the userId is null or if the user is not authenticated; if so, it returns an "Unauthorised" message to prevent unregistered users from donating. Next, it validates the donation amount, ensuring it is greater than zero, otherwise it returns an "Invalid Donation" message. If all checks pass, a new donation is created using the provided user ID, post ID, and donation amount. The function then updates the related campaign's amountRaised field by adding the donation value and saves the donation record to the database. Finally, it returns a confirmation message: "Donation Successful."

# Balsamiq Designs (Wireframes for UI)




# Data Structures & Algorithms Used

As our project was primarily a full-stack web application we didn't require complex or custom data structures. However, we made effective use of built-in structures provided by C# and .NET, especially through Entity Framework, which allowed us to model and interact with relational data using C# classes linked to database tables.

One of the key data structures we used throughout the backend was the List<T> collection, especially in logic handled by Ben. These lists were essential for storing and working with collections of data such as posts, comments, donations, and users. For example, when retrieving all comments for a post or displaying paginated campaigns, we used List<Comment> and List<Post> to store the results from our queries.

We decided to use List<T> over structures like Dictionary<T > because our operations were mostly focused on iterating through collections and fetching subsets of data, rather than performing fast lookups by a specific key. Since the data we retrieved (like posts or comments) was typically displayed in order and filtered by attributes such as post type or page number, a list was more appropriate in this situation. A dictionary would only have been useful if we were repeatedly accessing items by a unique key, which wasn't the case in our design.

Other important elements in our code included:

• **DTOs (Data Transfer Objects):** Used to securely pass structured data between the client and server and to prevent over-posting from users.

• **Filtering Algorithms:** Applied to retrieve posts by type or category using simple conditional queries (example: WHERE type = "Education")

• **Pagination Logic:** Implemented and used to control how many items appear per page and improve performance, rather than displaying all campaigns on one page which is a bad practice and made the site look cleaner.

# Testing

Unit Test Case Table:

| Test Case ID | Feature | Test Description | Input Data | Expected Outcome |
|---|---|---|---|---|
| TC01 | User Registration | Register a user with valid input | Name, Email, Password | User is registered successfully |
| TC02 | User Registration | Attempt to register with existing email | Existing Email | BadRequest - Email already exists |
| TC03 | User Registration | Attempt to register with weak password | Password = 'weak' | BadRequest - Password must be at least 8 characters |
| TC04 | User Login | Login with valid credentials | Valid Email & Password | Returns JWT token (200 OK) |
| TC05 | User Login | Login with incorrect credentials | Wrong Email or Password | BadRequest - Invalid credentials |
| TC06 | User Login | Login with empty credentials | Email = '', Password = '' | BadRequest - Missing credentials |
| TC07 | Get Profile | Fetch user profile by valid ID | UserId = 1 | Returns User object (200 OK) |
| TC08 | Post Creation | Create a project with valid input | Title, Description, Goal | Project created successfully (200 OK) |
| TC09 | Post Creation | Create a project with missing/invalid data | Missing Title or Goal < 0 | BadRequest - Invalid data |
| TC10 | Post Retrieval | Get existing project by ID | ProjectId = 1 | Returns Project (200 OK) |
| TC11 | Post Retrieval | Get non-existing project | ProjectId = 999 | NotFound |
| TC12 | Post Listing | Get all projects | None | Returns list of projects (200 OK) |
| TC13 | Post Deletion | Delete existing project | ProjectId, UserId | Project deleted successfully (200 OK) |
| TC14 | Post Update | Update project with valid input | ProjectId, Updated Data | Project updated successfully (200 OK) |
| TC15 | Comment Creation | Submit a valid comment | Comment content, PostId, UserId | Comment added (200 OK) |
| TC16 | Comment Retrieval | Fetch comments for a post | PostId | Returns list of comments (200 OK) |
| TC17 | Comment Deletion | Delete existing comment | CommentId, UserId | Comment deleted successfully (200 OK) |
| TC18 | Comment Deletion | Attempt to delete non-existing comment | CommentId = 999 | NotFound |
| TC19 | Donation Submission | Make a valid donation | Amount > 0, ProjectId, UserId | Donation successful (200 OK) |
| TC20 | Donation Submission | Attempt to donate with invalid amount | Amount = -100 | BadRequest - Invalid donation |
| TC21 | Donation Retrieval | Fetch donations for a project | ProjectId | Returns list of donations (200 OK) |
| TC22 | User Donation History | Fetch user's donation history | UserId | Returns list of user's donations (200 OK) |

# Conclusion

Overall, our team successfully planned, designed, developed, and tested a full-stack crowdfunding web application. The process was smooth, with very few major issues along the way. When problems did arise, they were addressed and resolved quickly thanks to our strong communication and teamwork. Throughout the project, we maintained regular collaboration, with Ahmed playing a key role in resolving technical issues and keeping the team informed at every stage of development.

Despite the positive outcome, we did encounter some limitations. One notable challenge involved handling JWT authentication via cookies, while it worked in tools like Thunder Client, it failed in the browser due to cross-domain cookie restrictions. This required us to adjust our approach and store the token in localStorage instead. Additionally, we faced a few bugs and merge conflicts during integration and due to time constraints, our frontend testing was more limited than we would have liked.

If we were to take on a similar project in the future, we would start the testing phase earlier to allow for more time across both frontend and backend. Overall, this project was a great learning experience that helped us build confidence in full-stack development, collaborative coding, and real-world problem-solving.