Use Cases

Preston, Lawrence, Steven and Nathan
Manager: Aditi

Tech Stack (PERN)

- Frontend: React
  - CSS/HTML, JavaScript
- Backend: Node.js, Express
  - JavaScript
  - Node.js: Handles server-side logic, asynchronous operations, and API endpoints.
  - Express: Simplifies routing, middleware integration, and request/response handling.
- Database: PostgreSQL
  - Stores structured data such as user profiles, posts, and rankings.

Use Case 1: Posting a Service on MinuteMatch

Goal: Allow users to post services for others to view and inquire about.

Key Databases

- User Database: Stores user credentials and profiles.
- Post Database: Stores service posts (e.g., category, description, price).

Functional Flow

Frontend Setup

- User Actions:
  - User selects "Post a Service" from the navbar.
  - React triggers createPostRequest() to validate inputs and send data to the backend.
- Form Fields:
  - Service Type (Dropdown)
  - Category (Dropdown)
  - Group(Dropdown optional)
  - Description (Text Box)
  - Picture (Optional File Upload)

Backend

- validatePostData(): Express middleware ensures all required fields are filled and meet constraints
- generatePostID(): Node.js allows the creation of unique ID using the User ID and post count.
- Database Interaction:
  - Data is stored in the Post Database (fields: category, description, price, timestamp).

ValidatePostData() Function

Purpose: Send the validated post or help request data from the frontend to the backend via API call

Inputs: From the React Frontend we get the object {userID, postID, category, group, text, time stamp}

Outputs:

- success returns a response object confirming post creation
- failure returns and error response if the data is invalid or backend encounters and issue

Testing:

- Frontend:
  - Ensure the function triggers the API call when the form is submitted.
  - Verify the correct data structure is sent to the backend.
  - Test user feedback for success (e.g., confirmation message) and failure (e.g., error message).
- Frontend → Backend:
  - Confirm the API call sends the correct payload to the backend.
  - Ensure the backend response (success or error) is handled correctly by the frontend.
- Backend:
  - Verify the backend processes the data and stores it in the database.
  - Ensure error handling for invalid data or database issues.

CreatePostRequest()

Purpose: Sends the validated post or help request data from the frontend to the backend via an API call.

Inputs: postData object containing {userID, postID, category, group, text, time stamp}

Outputs:

- Success: Returns a response object confirming the post was created (e.g., postID).
- Failure: Returns an error response if the data is invalid or the backend encounters an issue.

Testing:

- Frontend:
  - Ensure the function triggers the API call when the form is submitted.
  - Verify the correct data structure is sent to the backend.
  - Test user feedback for success (e.g., confirmation message) and failure (e.g., error message).
- Frontend → Backend:
  - Confirm the API call sends the correct payload to the backend.
  - Ensure the backend response (success or error) is handled correctly by the frontend.
- Backend:
  - Verify the backend processes the data and stores it in the database.
  - Ensure error handling for invalid data or database issues.

GeneratePostID()

Purpose: Creates a unique identifier for each post or help request using the us

Inputs:{userID: string, postCount: number}

Outputs: A string combining the userID and an incremented post count (e.g., user123-11).

Testing:

- Backend:
  - Verify the function generates unique IDs for each post.
  - Ensure the format of the ID is correct (e.g., userID-postCount).
  - Test edge cases, such as very high post counts or invalid userIDs.
- Backend → Database:
  - Confirm the generated ID is stored correctly in the database.
  - Ensure duplicate IDs are not created for the same user.
- Integration:
  - Verify the ID is returned to the frontend after the post is created.
  - Ensure the frontend displays the correct ID in confirmation messages or UI element

Use Case 2: Requesting Help on MinuteMatch

Goal: Allow users to request help in specific categories and receive responses from others.

Key Databases

- User Database: Stores user credentials and profiles.
- Post Database: Stores help requests (e.g., category, description, urgency).

Functional Flow

Frontend Setup

- User Actions:
  - User selects "Post a Service" from the navbar.
  - React triggers createPostRequest() to validate inputs and send data to the backend.
- Form Fields:
  - Service Type (Dropdown)
  - Category (Dropdown)
  - Description (Text Box)
  - Price (Number Input)
  - Picture (Optional File Upload)

Backend Setup

- validatePostData(): Express middleware ensures all required fields are filled and meet constraints
- generatePostID(): uses Node.js, Creates unique ID using the User ID and post count.
- Database Interaction:
  - Express handle API request so Data can be stored in the Post Database (fields: category, description, price, timestamp).
- markAsCompleted(): for when user views that they have received the help for the post. Post is then removed from the Post DB

ValidatePostData() & createPostRequest() & generatePostID()

The functions we mentioned above and used in Use Case 1 will be implemented again into the Use Case 2 function.

The only additional function needed for this implementation is so users can add groups to post to on MinuteMatch

markAsCompleted() Function

Purpose: Marks a help request as completed once the user has received satisfactory assistance

Inputs: {postID: string, userID: string}

Outputs:

- Success: Returns a confirmation that the post has been marked as completed and removed from the database.
- Failure: Returns an error message if the postID or userID is invalid, or if the post cannot be found.

Testing

- Frontend:
  - Verify that the "Mark as Completed" button triggers the function.
  - Ensure the correct postID and userID are sent to the backend.
  - Test user feedback for success (e.g., "Request marked as completed") and failure (e.g., "Unable to complete request").
- Frontend → Backend:
  - Confirm the API call sends the correct payload (postID, userID) to the backend.
  - Ensure backend responses (success or error) are displayed correctly to the user.
- Backend:
  - Validate the postID and userID against the database to ensure they exist and match.
  - Ensure the post is removed from the database upon successful completion.
  - Test error handling for invalid IDs or database issues.
- Database Interaction:
  - Verify the post is deleted from the Post Database after being marked as completed.
  - Ensure no other posts are affected during the process

Use Case 3: Searching for Groups on MinuteMatch

Goal: Allow users to search for and join relevant group.

Key Databases

- User Database: Stores user credentials and profiles.
- Group Database: Stores group details (e.g., name, category, description).
-

Code Examples

Frontend

- User Action:
  - User selects "Groups" from the navbar.
  - React triggers the fetchGroups() function to populate the dropdown with all available Groups.
  - User selects a Group from the dropdown.
- Dropdown Fields:
  - Group Name (Dropdown populated dynamically with group names)
    i. Implemented with React
  - CreateGroup(): if the user wants to add something not in the list

Backend

- validateInput(): Express Middleware uses this function. Ensures the selected group/category does not exists in the Group Database.
  - Express assists in API request to search/create groups
  - Node.js handles the validation errors and edge cases
  - If no suitable group is found, user can create a new group. The group is then added to the database if not present

fetchGroup() Function

Purpose: Retrieves all existing groups from the Group Database to populate the dropdown menu for users to search and join groups

Inputs: None

Outputs:

- Success: Returns an array of group objects (e.g., {groupID, groupName, category, description}).
- Failure: Returns an error message if the database query fails

Testing

- Frontend:
  - Verify that the dropdown menu is populated with the correct group data.
  - Test user feedback for success (e.g., groups displayed) and failure (e.g., "Unable to fetch groups").
- Frontend → Backend:
  - Confirm the API call is triggered when the user accesses the "Groups" section.
  - Ensure the backend response (success or error) is handled correctly by the frontend.
- Backend:
  - Validate the function retrieves all groups from the database.
  - Test edge cases, such as an empty database or database connection issues.
- Database Interaction:
  - Ensure all group data is fetched accurately and matches the database records.

createGroup() Function

Purpose: Allows users to create a new group or category if no suitable group exists

Inputs:{name: string, description: string}

Outputs:

- Success: Returns a confirmation message that the group was successfully created.
- Failure: Returns an error message if the input validation fails or the group already exists.

Testing

- Frontend:
  - Verify that the "Create Group" button triggers the function.
  - Ensure the correct input data (e.g., name, category, description) is sent to the backend.
  - Test user feedback for success (e.g., "Group created successfully") and failure (e.g., "Group already exists").
- Frontend → Backend:
  - Confirm the API call sends the correct payload (category, name, description) to the backend.
  - Ensure backend responses (success or error) are displayed correctly to the user.
- Backend:
  - Validate the input fields (e.g., name is required, description length).
  - Ensure the new group is added to the database only if it does not already exist.
  - Test error handling for invalid inputs or database issues.
- Database Interaction:
  - Verify the new group is stored correctly in the Group Database.
  - Ensure duplicate groups are not created.

validateInput() Function

Purpose: Ensures that all input fields for creating a group are properly filled and meet constraints

Inputs: {name: string, description: string}

Outputs:

- Success: Returns a confirmation that the input data is valid.
- Failure: Returns an error message specifying the missing or invalid fields.

Testing

- Frontend:
  - Verify that all required fields (e.g., name, category) are filled before submission.
  - Test error messages for invalid inputs (e.g., missing name, description too short).
- Frontend → Backend:
  - Confirm the frontend sends the correct data structure to the backend.
  - Ensure backend error responses (e.g., "Name is required") are displayed to the user.
- Backend:
  - Validate all required fields are present and meet constraints.
  - Test edge cases, such as missing fields or invalid formats.
- Database Interaction:
  - Ensure invalid data is rejected and not stored in the database.
  - Confirm valid data passes validation and is ready for storage

Use Case 4: Certifying Help on MinuteMatch

- Goal: Allow users to certify helpers for their assistance, increasing their ranking in the User Database.

Key Databases

- User Database: Stores user credentials, profiles, and rankings.

Outline

Frontend Setup

- User selects "Certify Help" from the navbar.
- React triggers submitCertification() to validate and send data to the backend.
- Form Fields:
  - Helper Name: Dropdown populated dynamically from the database.
  - Post ID: Dropdown or text input.
  - Category or Group: dropdown
  - Comment: Text input
  - Numerical Rank: 0-10

Backend Setup

- validateCertification() ensures Post ID and Helper ID are valid.
- Database Interaction:
  - Express handles API requests to DB
  - Updates the helper's ranking in the User Database using updateRanking()
    1. Executed with Node.js
    2. Ranking contains average score and category scores

submitCertification() Function

Purpose: Sends user certification data (e.g., helper details, ranking, comments) from the frontend to the backend via an API call.

Inputs: certificationData object containing {postID, helperID, category, comment, rank}.

Outputs:

- Success: Returns a response object confirming the certification was successfully submitted.
- Failure: Returns an error response if the data is invalid or the backend encounters an issue

Testing

- Frontend:
  - Verify that the "Submit Certification" button triggers the function.
  - Ensure the correct data structure is sent to the backend.
  - Test user feedback for success (e.g., "Certification submitted successfully") and failure (e.g., "Unable to submit certification").
- Frontend → Backend:
  - Confirm the API call sends the correct payload (certificationData) to the backend.
  - Ensure backend responses (success or error) are displayed correctly to the user.
- Backend:
  - Verify the backend processes the data and updates the helper's ranking in the database.
  - Ensure error handling for invalid data or database issues.

validateCertification() Function

Purpose: Ensures all required fields in the certification form are properly filled and meet constraints (e.g., valid postID, helperID, rank within range).

Inputs: {postID: string, helperID: string, category: string, comment: string, rank: int}

Outputs:

- Success: Returns a confirmation that the certification data is valid.
- Failure: Returns an error message specifying the missing or invalid fields.

Testing

- Frontend:
  - Verify all required fields (e.g., helperID, rank) are filled before submission.
  - Test error messages for invalid inputs (e.g., rank out of range, missing postID).
- Frontend → Backend:
  - Confirm the frontend sends the correct data structure to the backend.
  - Ensure backend error responses (e.g., "Rank must be between 0-10") are displayed to the user.
- Backend:
  - Validate all required fields are present and meet constraints.
  - Test edge cases, such as missing fields or invalid formats.
- Database Interaction:
  - Ensure invalid data is rejected and not stored in the database.
  - Confirm valid data passes validation and is ready for storage.

updateRanking() Function

Purpose: Updates the helper's ranking in the User Database based on the certification data submitted by the user.

Inputs: {helperID: string, rankIncrement: int}

Outputs:

- Success: Returns the updated ranking for the helper in the database.
- Failure: Returns an error message if the helperID is invalid or the database update fails.

Testing

- Backend:
  - Verify the function calculates the new average ranking correctly.
  - Ensure the helper's ranking is updated in the database.
  - Test edge cases, such as invalid helperID or rankIncrement out of range.
- Backend → Database:
  - Confirm the updated ranking is stored correctly in the User Database.
  - Ensure no other user rankings are affected during the process.
- Integration:
  - Verify the updated ranking is returned to the frontend after the database update.
  - Ensure the frontend displays the updated ranking correctly in the UI.