*SkillSwap Mobile App – SRS*

## *Software Requirements Specification (SRS)*

### *Introduction*

***Purpose:*** *This SRS describes the* ***SkillSwap*** *mobile application, a platform where users (students) can exchange skills through mutual learning sessions without monetary transactions. It details the system’s intended features, behavior, and constraints to guide developers and stakeholders. In essence, “SkillSwap is a platform that facilitates skill trading among individuals, allowing them to exchange knowledge without monetary transactions.*

***Scope:*** *The SkillSwap app is aimed at university students who want to teach and learn skills from each other in a peer-to-peer manner. The scope covers a cross-platform mobile application (built with React Native) and a backend service (e.g. Node/Express with a MongoDB database). Key functionalities include user account management, creating/viewing skill offer posts, requesting skill swap sessions, in-app messaging, and admin oversight. This SRS addresses the requirements for an MVP release, focusing on core features for skill exchange.*

***Definitions, Acronyms, Abbreviations:***

* *SkillSwap: The application/platform for swapping skills.*
* *Student/User: A regular user of the app who can offer and request skills.*
* *Admin: A special user role with privileges to manage and moderate the system.*
* *Skill Offer: A post created by a user advertising a skill they can teach.*
* *Skill Swap Session: An agreed exchange where two users meet (online or in person) to trade skills.*
* *MVP: Minimum Viable Product, a basic functional version of the app with essential features.*
* *SRS: Software Requirements Specification, this document outlining requirements.*

### *Overall Description*

***Product Perspective:*** *SkillSwap is a standalone mobile application, complementing or replacing informal skill-sharing arrangements among students. It does not require legacy integration. The system uses a client–server architecture: a React Native mobile frontend and a cloud backend (with RESTful API and MongoDB database).*

***User Roles:***

* ***Student (Normal User):*** *The primary actor who uses the app to learn or teach skills. Each student can create a profile, post skill offers, browse others’ offers, request skill swap sessions, chat with peers, and give reviews.*
* ***Admin:*** *An administrative user who oversees the platform. Admins can manage user accounts and moderate content (e.g. remove inappropriate posts or ban users). Admins ensure the platform remains safe and effective. (The Admin role can be implemented as a user with elevated permissions.)*

***User Stories:*** *(each described from an end-user perspective)*

* *As a student user, I want to register an account easily so that I can start using the SkillSwap app.*
* *As a student user, I want to create a profile listing my skills and what I want to learn, so that others know what I can offer.*
* *As a student user, I want to browse and search for available skill offers, so that I can find skills I’d like to learn.*
* *As a student user, I want to post an offer for a skill I can teach, so that others can request to learn it from me.*
* *As a student user, I want to request a skill swap with another user’s offer, specifying what I can teach in return, so that we can exchange knowledge.*
* *As a student user, I want to receive notifications or messages when someone requests a swap or responds, so that I stay updated.*
* *As a student user, I want to chat with the other user after a swap request is accepted, so that we can coordinate the details of our session.*
* *As a student user, I want to mark a skill swap session as completed and leave a review for my partner, so that we build trust and reputation on the platform.*
* *As an admin, I want to view and manage all user accounts, so that I can remove or block problematic users.*
* *As an admin, I want to moderate skill offers and reviews, so that inappropriate content is removed from the platform.*

*These user stories guide the functional requirements of the system. The* ***Overall Product Functions*** *can be summarized as enabling profile management, content posting (skill offers), discovery (browsing/searching offers), matchmaking (requests and acceptances for swaps), communication (messaging), and feedback (reviews), along with admin oversight. According to the project case study, key features of SkillSwap include user profile setup, skill swap requests, a messaging system, and an intuitive onboarding process to help new users get started. This MVP will focus on implementing these core features.*

### *Functional Requirements*

*The system’s functional requirements are enumerated below. (Each requirement is labeled for reference, but numbering does not imply priority.) At minimum, the system shall support the following functions:*

1. ***User Registration:*** *The system shall allow a new user to create an account by providing required information (e.g. name, email, password). Duplicate emails/usernames must be checked, and passwords stored securely (hashed). Upon successful registration, the user becomes a registered member.. (If email verification is in scope, the system should send a verification email before activation.)*
2. ***User Login:*** *The system shall allow a registered user to log in with their credentials. The app must validate the email/username and password, and establish a session or token upon successful authentication. (For MVP, a simple in-app check with dummy credentials is used.)*
3. ***User Logout:*** *The system shall allow an authenticated user to log out, ending their session. (In the MVP this can simply be a local state reset; in a full system it would revoke the auth token or session.)*
4. ***Profile Management:*** *The system shall provide each user a profile where they can add or edit information such as bio, contact details (if any), and skills offered and/or skills desired. Users can navigate to a “Edit Profile” section to update their bio or list of skills.*
5. ***View User Profile:*** *The system shall allow users to view other users’ profiles (public information), including their name, bio, and list of skills/offers. For example, tapping on a user’s name or offer in the feed will show their profile details and any reviews. (This helps users decide if they want to learn from that person.)*
6. ***Create Skill Offer (Post):*** *The system shall allow a user to create a new skill offer post. The user provides details such as the skill name/title, a description, and optionally a category or availability info. The post is saved under the user’s account and becomes visible on the platform. Users can also edit or delete their own offers.*
7. ***Browse Skill Offers (Feed):*** *The system shall display a feed or list of all available skill offers posted by users. This feed can be the home screen, showing recent offers or featured skills. Users shall be able to scroll through the list to discover skills being offered by others. By default, all active offers are shown.*
8. ***Search and Filter Offers:*** *The system shall provide search functionality so that users can find specific skills or categories. Users can enter keywords to search skill titles/descriptions and filter results by category or other criteria (e.g. filter by skill category, or sort by highest rated mentors). This helps users discover available skills efficiently.*
9. ***View Offer Details:*** *The system shall allow a user to tap on a specific skill offer to view its details. This includes the full description, the name and profile of the offering user, and possibly the number of swap requests or reviews related to that offer. From this detail view, the browsing user can choose to send a swap request.*
10. ***Send Skill Swap Request:*** *The system shall allow a user (User A) to request a skill exchange with another user (User B) based on B’s offer. When sending a request, User A can include a message and specify which of their own skills they are willing to teach in return. The system will record the request and notify User B of the incoming swap proposal. (For example, User A selects B’s “Guitar lessons” offer and proposes an exchange by offering “Spanish tutoring” in return.)*
11. ***Accept/Reject Swap Request:*** *The system shall allow a user to respond to an incoming skill swap request. The user who posted the original offer (User B) can view the request details (skill offered in exchange, message from requester) and choose to accept or reject it. If accepted, the system marks the swap as confirmed (creates a Session record) and notifies both parties. If rejected, the requester is notified of the denial.*
12. ***In-App Messaging:*** *The system shall provide a messaging or chat feature for users who have an active pending swap or confirmed session. Once a swap request is accepted (or possibly even before acceptance, as needed), the two users can exchange messages to coordinate logistics (when and how to meet, etc.). This chat should be accessible only to those two participants and remain available until the session is completed. (For MVP, a simple text messaging interface can be simulated; full real-time chat could be a future enhancement.)*
13. ***Notifications:*** *The system shall notify users of important events. For example, when a user receives a new swap request, when their request is accepted or rejected, or when they get a new message, the app should provide a notification (in-app pop-up or push notification). This ensures timely awareness of updates. (In the MVP, this can be a simple indicator in the UI since push notifications may require additional setup.)*
14. ***Complete a Skill Swap Session:*** *The system shall allow users to mark a skill swap session as* ***completed*** *once the exchange has taken place (offline or online). After a session is accepted, users will have an option (e.g. a “Complete Session” button) to indicate they have finished the swap. The system updates the session status accordingly. This could trigger the ability to review the partner.*
15. ***Rate and Review Partner:*** *The system shall enable users to rate and review each other after a completed skill swap. Each user can give a star rating (e.g. 1 to 5) and a short textual feedback about their experience with the other user . These reviews are attached to user profiles (so future participants can see a user’s reputation). A user can only review another user if they had a confirmed session together.*
16. ***Admin – Manage Users:*** *The system shall provide administrative functions for managing the user base. An Admin user can log in to an admin interface (or use an admin mode in the app) to view all user accounts. Admins can deactivate or ban users who violate terms, and update user roles (e.g. promote a user to admin).*
17. ***Admin – Moderate Content:*** *The system shall allow an Admin to review and moderate content on the platform. This includes the ability to remove or edit any skill offer posts that are inappropriate and possibly remove offensive reviews. Admins might also monitor ongoing sessions or intervene if needed. (Optional extension: Admin receives reports of abuse and takes action.)*

***Note:*** *Requirements 1–15 cover core* ***student user*** *functionalities, while 16–17 are* ***admin*** *functionalities. The MVP implementation will prioritize the student user features (registration, posting, browsing, requests, etc.). Some advanced functions like real-time chat or push notifications may be simplified in MVP. The requirements above align with the fully-dressed use cases identified for SkillSwap, ensuring a user-friendly experience for skill exchange.*

### *4. Non-Functional Requirements*

*In addition to the above functional capabilities, the system must meet several quality and constraint requirements:*

* ***Usability & UX:*** *The app should be intuitive and easy to navigate for students. A first-time user should understand how to create a profile and start swapping skills with minimal guidance. The design will follow mobile app usability best practices, with a clean interface and an “intuitive onboarding process” to help new users get started.*
* ***Performance:*** *The application should be reasonably fast and responsive. Screens like the home feed (offer list) should load within a couple of seconds even with dozens of offers. Searching or filtering should feel instantaneous on the client side (leveraging local filtering for loaded data or quick server queries). The system should handle at least a few hundred concurrent users in the MVP phase without degradation.*
* ***Scalability:*** *The architecture should allow scaling up to more users and data. Using MongoDB for data storage and a stateless API server means the system can be scaled horizontally (more server instances, database clustering) as user count grows. The design should support increasing the number of skill offers and messages without major rewrites (e.g. using pagination or infinite scroll in lists).*
* ***Security:*** *User data must be protected. Passwords will be stored securely (hashed and salted). API communication should be over HTTPS to protect credentials and personal information. Each user’s private data (messages, email) should be accessible only to authorized users. The system should enforce authorization checks (e.g. only owners can modify their offers; only participants can access a session’s chat).*
* ***Reliability:*** *The app should handle expected error conditions gracefully. For example, if the server is unreachable, the app should show an error message. Data operations should be atomic where needed (e.g. a swap acceptance should update all relevant records consistently). The system should avoid crashes; any crash or serious bug should not compromise data integrity.*
* ***Maintainability:*** *The codebase (both frontend and backend) should be organized and documented to allow future enhancements. Using React Native ensures a single codebase for iOS and Android, simplifying maintenance. The team should follow best practices (modular code, meaningful naming, version control) so new developers can understand the system.*
* ***Compatibility:*** *The mobile app must run on modern Android and iOS devices. The UI should be responsive to different screen sizes. For MVP, targeting Android 10+ and iOS 13+ devices is acceptable. The app relies on minimal device-specific features, ensuring cross-platform consistency.*
* ***Privacy:*** *The system should collect only necessary information from users. Profiles are public to other users, but sensitive data like full name or contact info is optional. Any personal data is used solely within the skill swap context. The app should comply with relevant privacy policies (e.g. not sharing data with third parties without consent).*
* ***Extensibility:*** *It should be relatively straightforward to introduce new features in the future, such as adding a “points” system or integrating calendar scheduling. The initial design will keep such potential extensions in mind (e.g. a session could have a scheduled time field added later).*

*(The above non-functional requirements ensure that the SkillSwap app not only works correctly, but also provides a good user experience and a solid foundation for future growth.)*

### *5. Client Questionnaire*

* *Who’s the primary audience (year, department), and which platforms do we launch on first (Android, iOS)?*
* *Should sign-up be email only, or include Google/Apple SSO and university email verification?*
* *Which profile fields are mandatory at signup (name, photo, bio, skills offered/wanted)?*
* *What info must every Skill Offer include (title, level, duration, tags, location, availability)?*
* *How should discovery work at MVP: keyword search, category filters, sort by rating or recent?*
* *Do tutors set fixed time slots, or do learners propose times for booking?*
* *Do we need calendar integration or just in-app reminders for now?*
* *What’s the reschedule/cancel policy (who can do it, timing, any limits)?*
* *Is simple in-app messaging enough for MVP, or do you want real-time chat?*
* *Which notifications are required at MVP (new request, accept/reject, reminders, review prompts)?*
* *Are reviews public and tied to identities, and can users be blocked or reported?*
* *What admin powers are needed (remove content, suspend users, view reports)?*
* *Any privacy/compliance requirements (university policy, GDPR-style data export/delete, retention period)?*
* *What’s explicitly out of scope for MVP (payments, video calls, groups, advanced analytics)?*
* *How will we measure success in the first term (MAU, number of offers, request→session conversion, average rating)?*

### *6. Database Schema*

*The SkillSwap app will use a MongoDB (NoSQL) database. Data is stored in JSON-like documents within collections. Below is the proposed schema structure with major collections and their fields (in a MongoDB-style format):*

* ***Users Collection:*** *Stores user account and profile info. Each document might look like:*

*{*

*\_id: ObjectId("..."), // unique user ID*

*name: "Alice Doe", // user's full name*

*email: "alice@example.com", // login email (unique)*

*passwordHash: "<hash>", // hashed password*

*role: "student", // "student" or "admin"*

*bio: "I love learning languages and music.", // profile bio*

*skillsOffered: ["French", "Guitar"], // list of skills user can teach*

*skillsWanted: ["Painting", "JavaScript"], // list of skills user wants to learn (optional)*

*rating: 4.8, // average rating from reviews (e.g. 1-5 scale)*

*numReviews: 5, // number of reviews received*

*createdAt: ISODate("2025-09-01T...") // account creation timestamp*

*}*

* ***Offers Collection:*** *Stores skill offer posts created by users. Each offer document represents one skill a user is offering to teach. Example:*

*{*

*\_id: ObjectId("..."),*

*userId: ObjectId("..."), // reference to Users.\_id (the author of this offer)*

*title: "Guitar Lessons for Beginners",*

*description: "Acoustic guitar basics for absolute beginners. 1-hour sessions.",*

*category: "Music", // category or domain of the skill*

*tags: ["guitar", "music", "beginner"], // keywords or tags for search*

*location: "Campus", // e.g. "Campus" or "Online" (how/where the skill can be taught)*

*availability: "Weekends", // availability of the offering user (optional free-text)*

*createdAt: ISODate("2025-09-05T..."),*

*status: "active" // e.g. "active" (open to new requests) or "inactive"*

*}*

* ***Sessions Collection:*** *Stores records of agreed skill swap sessions (a match between two users). A session begins when one user’s request is accepted by the other. It can be pending, confirmed, or completed. For simplicity, each session involves two users and each provides one skill (from either an offer or their profile) to the other. For example:*

*{*

*\_id: ObjectId("..."),*

*userAId: ObjectId("..."), // the user who initiated the swap (requester)*

*userBId: ObjectId("..."), // the user who accepted the swap (offer owner)*

*skillFromA: "Spanish", // skill taught by User A to User B*

*skillFromB: "Guitar", // skill taught by User B to User A*

*requestId: ObjectId("..."), // optional: reference to a Request if separate collection (MVP may merge with Session)*

*initiatedAt: ISODate("2025-09-10T..."), // when the request was sent*

*acceptedAt: ISODate("2025-09-11T..."), // when the request was accepted*

*status: "completed", // "pending", "confirmed", or "completed"*

*completedAt: ISODate("2025-09-20T..."), // when both users marked session done*

*chatThreadId: ObjectId("...") // optional: reference to a messages thread for this session*

*}*

*(Note: In a more normalized SQL design, this might be split into separate “Requests” and “Sessions” tables. In MongoDB, we might keep a single collection and use the status field to indicate if it’s just requested or confirmed.)*

* ***Reviews Collection:*** *Stores user reviews and ratings given after sessions. Each review is linked to the session and the users involved:*

*{*

*\_id: ObjectId("..."),*

*sessionId: ObjectId("..."), // reference to the session that concluded*

*reviewerId: ObjectId("..."), // user who wrote the review*

*revieweeId: ObjectId("..."), // user who is being reviewed*

*rating: 5, // rating given (e.g. 5 stars)*

*comment: "Great experience, I learned a lot!", // textual feedback*

*createdAt: ISODate("2025-09-21T...")*

*}*

*In this schema, references like userId, userAId/userBId, sessionId link documents across collections. For example, Offers.userId ties an offer to the User who created it. A Session references two Users (participants) and possibly an Offer (not explicitly shown above, but skillFromB could be inferred from an Offer). Reviews reference which session and which users are involved.*

*Additional collections (if needed later) could include* ***Messages*** *(for chat threads), and* ***Notifications****. For MVP, we might handle messages in-memory or via a simple structure, but a Messages collection could have fields like { threadId, fromUser, toUser, text, timestamp }. Notifications could similarly be stored or just handled in real-time.*

***Database Design Rationale:*** *Using a document database (MongoDB) is suitable as it allows flexible embedding of data. For instance, we could embed an array of skillsOffered in the User profile (as shown). We could also embed reviews under each User document for quick access to a user’s ratings. However, for clarity and to avoid unbounded document growth, a separate Reviews collection as defined is used. This schema is designed to support the core operations of the app efficiently. For example, to show the home feed, the app can query the Offers collection for recent offers. To retrieve all info about a session for a chat, the app can join (lookup) user details and related review after completion.*

*(In summary, the above SRS outlines* ***what*** *the SkillSwap system should do (functional requirements) and the constraints on* ***how*** *it performs (non-functional requirements). It also defines the data model that will support these features. The next sections will present the corresponding UML diagrams and an MVP implementation outline.)*

## *7. UML Diagrams*

### *Use Case Diagram – SkillSwap App*

***Actors:***

* ***Student (User):*** *Primary actor who uses the app’s features to swap skills.*
* ***Admin:*** *Secondary actor with special privileges for maintenance.*

***Use Cases for Student:***

1. ***Register/Sign Up*** *– Create a new account in the system.*
2. ***Login*** *– Authenticate and start a session.*
3. ***Manage Profile*** *– Create or update profile (bio, skills offered/wanted).*
4. ***Post Skill Offer*** *– Create a new skill offer listing.*
5. ***Browse Offers*** *– View the list of available skill offers (feed).*
6. ***Search/Filter*** *– Search offers by keyword or filter by category.*
7. ***View Offer Details*** *– View full details of a selected skill offer (including info about the offering user).*
8. ***Send Swap Request*** *– Propose a skill swap to another user for a specific offer.*
9. ***Receive Request*** *– (Passive) Get notified of incoming swap request from another user.*
10. ***Accept/Reject Request*** *– Respond to a received request (confirm or decline the swap).*
11. ***Messaging/Coordination*** *– Communicate with the partner to discuss details once a request is accepted.*
12. ***Complete Session*** *– Mark an agreed skill swap session as completed after execution.*
13. ***Rate/Review Partner*** *– Leave a rating and review for the other user after a completed session.*

***Use Cases for Admin:*** *A.* ***Admin Login*** *– Authenticate as an admin (could be via a separate admin interface or the same app with admin credentials).  
B.* ***Manage Users*** *– View all users; block or remove users who violate policies.  
C.* ***Moderate Content*** *– Review skill offers, sessions, or reviews; remove or edit content that is inappropriate. (This could include responding to user reports or automated flags.)  
D.* ***View System Metrics*** *– Optional: View statistics like number of swaps, user growth, etc. (Nice-to-have for admin, not core MVP).*

### *Class Diagram – Key Entities and Relationships*

*The class diagram models the static structure of the SkillSwap system. It highlights the main data entities (classes) and how they associate with each other. The critical classes are* ***User****,* ***Offer****,* ***Session****, and* ***Review*** *as identified in the requirements. Below is a description of these classes and their relationships (see* ***Figure 2*** *for a visual representation):*

* ***User:*** *Represents a platform user (typically a student, but can also be an admin).*
  + *Attributes: userId (unique identifier), name, email, passwordHash, role (e.g. “student” or “admin”), bio, skillsOffered (list of skills), skillsWanted (list of skills), and aggregate rating info (rating, numReviews).*
  + *Associations: One* ***User*** *can create many* ***Offer*** *posts (relationship 1-to-many). A User can also initiate or participate in many* ***Session*** *exchanges. Users write* ***Review****s for other users and receive* ***Review****s from others. If modeling in UML:*
    - *User* ***–<creates>–>*** *Offer (a user owns/offers many offers).*
    - *User* ***<–participates in–>*** *Session (a session involves two users; this can be shown as an association class or two associations for user1 and user2 roles).*
    - *User* ***–<writes>–>*** *Review (a user can author many reviews).*
    - *User* ***<–<receives>–*** *Review (a user can receive many reviews).*
  + *Specialization: An* ***Admin*** *could be represented as a subclass of User or simply indicated by User.role = “admin”. In practice, no separate Admin class is needed; it’s a user with elevated permissions.*
* ***Offer:*** *Represents a skill offer posted by a user.*
  + *Attributes: offerId, title, description, category, tags[], location, availability, status, and a reference to the author (userId).*
  + *Associations:* ***Offer*** *is associated with the* ***User*** *who created it (many offers per one user). Additionally, an Offer might be linked to* ***Session*** *or* ***Request*** *– e.g. if a particular offer was exchanged, the session could reference that offer. For simplicity, we treat offers as standalone listings; the session will record the skill names rather than a direct link. (In an extended model, we could have a relationship: Offer* ***–<is basis for>–>*** *Session to denote that a session was arranged for that offer.)*
* ***Session:*** *Represents an agreed skill swap session between two users.*
  + *Attributes: sessionId, references to the two participants (userA and userB or similar), details of the skills each side is giving (skillFromA, skillFromB), status (pending/confirmed/completed), and timestamps (requested/accepted/completed dates). It may also include a reference to a related offer (e.g. offerId of the skill that was originally requested) and possibly a scheduled time for the session if known.*
  + *Associations:* ***Session*** *is fundamentally an association between two* ***User****s (many sessions for each user, each session has exactly 2 users). We can denote this as a multiplicity: a Session has 2 User participants; each User can have 0.. many Sessions over time.* ***Session*** *may also have a one-to-many with* ***Review*** *(a session can generate up to 2 reviews, one from each participant). In the class diagram:*
    - *Session* ***–*** *1..\** ***User*** *(depending on notation, could show two separate associations, or label roles like “User (teacher)” and “User (learner)” connected to Session).*
    - *Session* ***–<includes>–>*** *Review (one session can result in two reviews; each review belongs to one session).*
    - *(Optional) Session* ***–<based on>–>*** *Offer (if we track which offer the session originated from).*
* ***Review:*** *Represents a rating/review one user gives to another after a session.*
  + *Attributes: reviewId, rating (numeric score), comment, and references to reviewerId (the author User), revieweeId (the User being reviewed), and sessionId (the Session after which the review was made).*
  + *Associations:* ***Review*** *is linked to one* ***Session*** *and to two* ***User****s (one author, one recipient). In UML, this can be shown as: Review* ***–*** *Session (many reviews per session, though typically 0, 1 or 2), Review* ***–*** *User (each review has one author; each review targets one user). From the User side, as mentioned, a user can author many reviews and receive many reviews.*

***Relationship Cardinalities:*** *To summarize cardinalities:*

* *User–Offer: 1 User can have 0..\* Offers; each Offer has exactly 1 creator User.*
* *User–Session: 1 User can be in 0..\* Sessions; each Session has 2 Users (exactly two participants, in the current design).*
* *User–Review: 1 User can write 0..\* Reviews; 1 User can receive 0..\* Reviews. Each Review has exactly 1 author and 1 target.*
* *Session–Review: 1 Session can have 0..2 Reviews (at most two, if both participants review each other); each Review is tied to exactly 1 Session.*
* *Admin: (If modeled separately) is a specialization of User; inherits User’s relationships. Alternatively, treat Admin as User with role flag, as noted.*

## *8. MVP Frontend in React Native*

*The Minimum Viable Product frontend will be a React Native mobile application implementing the essential user interface for SkillSwap. We will create the following screens:* ***Login/Signup****,* ***Home Feed****,* ***Create Post****, and* ***Profile****. Each screen uses dummy data or simple logic (no real backend integration) to demonstrate the core functionality and flow. The app structure can be organized using a navigation library (e.g. React Navigation) to switch between screens. For instance, after login, the user enters a tab navigator with tabs for Home, Create Post, and Profile. Below, we describe each screen and provide example code snippets (using functional components and hooks) to illustrate the implementation.*

### *Login/Signup Screen*

*This screen allows the user to enter credentials to log in, or to create a new account. In the MVP, we will use* ***dummy credentials*** *for simplicity – e.g., accept a fixed email/password as correct and bypass actual account creation logic. The UI consists of text input fields for email and password, and buttons for “Login” and possibly “Sign Up”.*

* *If the user enters the predefined dummy credentials (say* ***test@example.com / 1234****), the app will navigate to the Home screen (indicating a successful login).*
* *If the input doesn’t match, we can show an alert or error message (simulating validation feedback).*
* *The Sign Up could be handled by either a separate form or a toggle on the same screen. For MVP, we might not actually store new users; pressing “Sign Up” could simply navigate the user in (or we treat any input as valid and proceed).*

### *Home Feed Screen*

*The Home screen shows a feed of skill offers available from other users. In the MVP, we use* ***dummy data*** *to populate a list of offers. Each item might display the skill name, a short description or the offering user’s name. We will use React Native’s <FlatList> component to efficiently render the list.*

*For example, we can define a static array of offer objects (with fields like id, skill, user, description). The FlatList will iterate over this data and render each item as a card or a simple text row. Users should be able to scroll through the list. (No real data fetching is done in MVP, but this simulates what the feed would look like.)*

*Key UI elements: a list (FlatList), where each list item could be a Touchable component if we want to allow tapping for details (in MVP, tapping might not do much unless we implement navigation to an Offer Detail screen – which is optional for now). We ensure the list has a unique key for each item (FlatList uses id or key field). According to React Native documentation, FlatList requires a data prop (array) and a renderItem prop to define how each item is displayed.*

*The Home Feed screen addresses the* ***Browse Offers*** *and* ***Search*** *functionalities on the frontend. We might also include a search bar at the top (e.g., a TextInput to filter dummyOffers by keyword). Given the small dataset in MVP, filtering could be done in-memory. For brevity, the above code doesn’t show search, but it could be added by maintaining a searchQuery state and filtering the dummyOffers array accordingly.*

### *Create Post Screen*

*The Create Post screen enables the user to submit a new skill offer. It contains a form with inputs for the skill details. At minimum, we need a field for the skill title/name and one for description. We could also have a dropdown or picker for category, but for MVP a simple text field for everything is acceptable. There will be a button to “Post” the offer.*

*In MVP (offline mode), we won’t actually send this data to a server. Instead, upon submission we can simply print it to console, or update the dummy list locally so that it appears in the Home feed if we navigate back. For example, we could push the new offer into the dummyOffers array (if it’s in a higher scope or using a state management store). To keep things simple, we might just show an alert “Posted!” and clear the form.*

*This screen corresponds to the* ***Post Skill Offer*** *functionality. It demonstrates capturing user input for a new offer. In a deployed app, pressing “Post Offer” would trigger an API call to save the offer to the backend and then update the global state or trigger a re-fetch for the Home feed.*

### *Profile Screen*

*The Profile screen displays the current user’s information and their skills/offers. For MVP, we will use a dummy current user object to populate this screen (since we aren’t truly logging in and fetching profile data). The screen can show: the user’s name, email, bio, and a list of skills they offer (and possibly those they want to learn). It can also list the user’s past reviews or ratings if we had that data. To keep it simple, we focus on showing basic info and skill lists.*

*UI elements: We can use <Text> components for each field (Name, Email, Bio) and perhaps a subheading for “Skills Offered”. For listing skills, we can either use another FlatList or simply map over an array of skills to generate a list of <Text> items or bullet points.*

*Additionally, we might include a button to edit profile (which would navigate to a Profile Edit screen – not implemented in MVP, but design-wise we could include a placeholder). For now, the profile is read-only in MVP.*

*The Profile screen addresses showing the* ***Manage Profile*** *info (though not editing here, just viewing). It confirms that our data model works – for instance, the skillsOffered and skillsWanted arrays are directly used to display content. In a full app, this data would be fetched from the server for the logged-in user and stored in state (e.g. via a context or Redux store). For MVP/dummy mode, hardcoding is acceptable.*

***Navigation Consideration:*** *In the MVP, after successful login, the user should be able to access Home, Create Post, and Profile screens. This can be done by using a Tab Navigator (with three tabs: Home, Create, Profile). Each tab would correspond to one of the screens defined above. The Login screen would be a separate stack; once login is done, it switches to the main tab navigator. React Navigation is a common solution for this. For example, one could set up:*

*For MVP, we manually navigate using navigation.replace or similar after login, but in a more structured approach, one would use navigation containers and possibly context for auth state. Given this is a small demo, the simplest is fine.*

***Dummy Data Usage:*** *Throughout the MVP, we rely on hardcoded values (dummyOffers list, currentUser object, dummy credentials). This is intentional to avoid needing a backend for demonstration. It also means the app’s state is not persistent – if you restart, it resets to initial dummy data. This is acceptable for an MVP whose goal is to showcase the user interface and flow. By stubbing the data and interactions, we can validate that the screens and navigation achieve the intended user stories: a user can “log in”, see a feed of offers, create a new offer, and view their profile.*

*In summary, the React Native MVP provides a basic but functional prototype of SkillSwap’s client side:*

* *The* ***Login/Signup screen*** *uses dummy logic to simulate authentication.*
* *The* ***Home Feed*** *displays a list of skill offers using a FlatList (dummy content), fulfilling the browsing capability.*
* *The* ***Create Post*** *screen accepts input and (simulated) posts a new skill offer.*
* *The* ***Profile*** *screen shows user info and their skills, demonstrating profile management.*

*This MVP can be run on a simulator or device to give a feel for the app. It sets the stage for integrating actual backend logic (for storing users, offers, sessions, etc.) in future iterations. Each component is kept simple, focusing only on relevant functionality, making it easier to expand later.*