**CelebFitLife Functional Specification Document (FRD)**

**Project:** CelebFitLife  
**Author:** Aaron Hazzard | Senior Software Engineer  
**Date:** March 30, 2025  
**Version:** v1.1 (Revised)

**Twilio Docs**: <https://www.twilio.com/docs/video/javascript-getting-started>

## **Table of Contents**

1. Introduction
2. Key Terms / Glossary
3. Purpose & Scope
4. User Personas  
    4.1. End User / Viewer  
    4.2. Streamer (Celebrity / Fitness Instructor)
5. Requirements Overview  
    5.1. User (Viewer) Journey  
    5.2. Streamer Journey
6. Functional Requirements  
    6.1. Landing & Subscription  
    6.2. Registration & Authentication  
    6.3. Payment & Subscription Management  
    6.4. Streaming & Live Interaction  
    6.5. Moderation & Access Control  
    6.6. Notifications & Scheduling
7. Non-Functional Requirements (High-Level)
8. System Architecture, Assumptions & Dependencies
9. Open Questions
10. Conclusion
11. Next Steps

## 

## **1. Introduction**

CelebFitLife is a **live-streaming fitness platform** that connects celebrities and fitness instructors (Streamers) with fans (Users) in real time. The vision is to create a fun, engaging fitness experience by allowing fans to work out virtually alongside their favorite celebrities or instructors, essentially a **“Twitch for fitness”**.

This platform addresses a growing business opportunity at the intersection of the fitness industry and the creator economy. **Problem Being Solved:** Many fitness enthusiasts struggle to stay motivated with pre-recorded videos or impersonal classes. Meanwhile, celebrities and influencers seek new ways to engage fans and monetize content. CelebFitLife solves these problems by providing **interactive live workouts** – fans get personal, real-time interaction and motivation, and streamers get a new channel for engagement and revenue.

**What Makes CelebFitLife Unique:** Unlike traditional fitness apps or one-way video content, CelebFitLife offers **bi-directional engagement** (live chat, polls, reactions) in a high-energy environment. Fans can cheer, ask questions, and feel a sense of community, while working out **simultaneously with a celebrity**. This real-time participation and celebrity access differentiate the platform, creating a unique value proposition in the market.

## **2. Key Terms / Glossary**

* **FRD (Functional Requirements Document):** A document outlining the functions that a system or component must be able to perform, detailing business and technical requirements. (This document serves as an FRD, expanding into a formal functional specification.)
* **Next.js 15+:** A React-based web application framework enabling server-side rendering (SSR), static site generation, and seamless building of scalable web apps. Chosen for its performance, SEO benefits, and ability to easily integrate backend logic via API routes.
* **Firebase (Already Setup):** A Backend-as-a-Service (BaaS) platform by Google offering authentication, real-time database/Firestore, cloud storage, hosting, and serverless functions. Used for quick development of auth and data features without managing infrastructure.
* **Email.js (Already Setup):** A cloud-based email delivery service used to send and manage large volumes of email reliably (e.g. transactional emails, notifications). Ensures high deliverability for things like verification emails and announcements.
* **Stripe (Already Setup):** An online payment processing platform enabling subscription management, recurring billing, and secure transactions. Used for handling credit/debit card payments globally with robust fraud protection and subscription APIs.
* **PayPal (Planned Integration):** A widely-used online payment system that supports payments from bank accounts or PayPal balance and recurring subscriptions. Adding PayPal provides flexibility for users who prefer it, increasing global reach and payment options.
* **Twilio (Already Setup):** A cloud communications platform providing APIs for real-time communications – voice, video, and messaging. Often used for live streaming or SMS. Here, Twilio enables low-latency live video streaming and possibly chat integration for the workouts.
* **Zustand (To-Do):** A small, fast, and flexible state-management library for React applications. It allows storing and managing global UI state outside of the component tree with minimal boilerplate, chosen for its simplicity over heavier alternatives.
* **Tailwind CSS:** A utility-first CSS framework for rapidly building custom user interfaces. Used to ensure a consistent design system and speed up UI development with predefined classes, while allowing flexibility in styling.

## **3. Purpose & Scope**

**Purpose:** Define the functional requirements, core user flows, and system behaviors for both Users and Streamers. This document ensures the development team and stakeholders have a clear, shared understanding of what the system will do and how it will behave, before moving into implementation. It also outlines why certain technologies are chosen to meet these requirements.

**Scope:** This functional specification covers the core features of the CelebFitLife platform, including user registration and authentication, subscription purchases and trials, live streaming sessions with real-time interaction (video, chat, polls), content access control (free previews, subscription enforcement), chat moderation tools, scheduling and notification mechanisms, and basic platform administration considerations. It also touches on the integration of **third-party services** (Firebase for auth/data, Twilio for streaming, Stripe/PayPal for payments, EmailJS for emails) and the rationale behind these choices. Non-functional requirements such as scalability, security, and performance are outlined at a high level. Anything beyond core streaming and subscription features (e.g. advanced recommendation algorithms, social features like adding friends, detailed analytics) is generally outside the scope of this MVP specification, though mentioned as future considerations.

## 

## **4. User Personas**

To design an effective platform, we consider two primary personas:

**4.1. End User / Viewer:** The typical viewer is a fitness enthusiast or fan of a celebrity instructor, likely between 18-50 years old. They seek an engaging, interactive workout experience with minimal friction. This user may have tried home workout videos or fitness apps but craves more excitement and personal connection. They value ease of use (quick sign-up, easy access on any device), motivation (being cheered on or noticed by their idol), and community (seeing other fans in chat). Their goal is to stay fit while being entertained. They might join via a mobile phone in their living room or a laptop at home, expecting a smooth, high-quality streaming experience without technical hassles.

**4.2. Streamer (Celebrity / Fitness Instructor):** The streamer is a celebrity or professional fitness instructor who provides live workout content. They could be a famous actor, musician, athlete, or online fitness influencer with a large fan following. Their needs include simple tools to schedule sessions, stream video, interact with viewers, handle moderation, and monetize their content effectively. They are not necessarily tech experts, so the platform’s broadcast interface must be intuitive and reliable. Streamers are motivated by engaging their fanbase in a new way, building their personal brand, and creating a revenue stream from subscriptions. They need features like viewer metrics (to see their reach), chat moderation (to maintain a positive environment), and feedback mechanisms (polls or reactions) to keep the sessions lively. Streamers will likely use a desktop with a good camera or a high-end smartphone to broadcast, and they expect the platform to handle the heavy lifting of video streaming and viewer management seamlessly.

## 

## **5. Requirements Overview**

This section provides a high-level overview of how Users and Streamers will experience the platform, from start to finish.

### **5.1. User (Viewer) Journey**

* **Landing Page:** The user arrives at the CelebFitLife homepage and sees basic information about the service, pricing options, and perhaps previews or thumbnails of currently live or upcoming workout streams. The landing page clearly communicates the concept (“work out live with your favorite celebrities”) and presents a call-to-action to join or sign up.
* **Sign Up:** If interested, the user proceeds to register. They enter personal details (username, email, etc. as required) and see an offer for a free trial (one full workout session free). The sign-up process is streamlined to reduce friction – e.g., social login options might be available for convenience.
* **Subscription Selection:** After account creation (or as part of sign-up), the user is prompted to select a subscription plan. There are multiple tiers (for example: access to 1 streamer, 3 streamers, or unlimited streamers). The user picks a tier and enters payment details. Thanks to a free trial, their card (or PayPal) won’t be charged until after they redeem the free session or the trial period lapses. A confirmation is shown that they have a trial active and will be billed later.
* **Stream Selection & Preview:** Once registered (and trial or subscription in place), the user can browse available live streams. They might see a **grid of streamer thumbnails** (each representing a live or upcoming session) possibly categorized (yoga, HIIT, dance, etc.) or a search function to find a specific celebrity. If the user is **not subscribed to a particular streamer** (or on a trial), they can still click into the stream. At that point, a **1-minute free preview** will play. A countdown timer is visible (e.g., “Preview: 00:45 remaining”) so the user knows they are in a limited preview. This lets them sample the content before committing.
* **Subscription Gate (After Preview):** After 1 minute of watching a streamer to whom the user isn’t subscribed, access becomes restricted. The video will blur and the audio mutes. A prompt or modal appears saying “Your preview has ended. Subscribe to continue working out live!” with a button to subscribe (or if they have remaining trial eligibility for another streamer, use that – but by default trial is one-time). The user cannot continue watching that stream until they subscribe (or choose a different streamer to use a trial on, if that were allowed – see Open Questions).
* **Live Stream (Full Access):** If the user **is subscribed** to the streamer (either they chose that streamer as part of their 1-streamer plan, or it falls under their 3/unlimited plan, or they are in their one-time trial session), they get full access to the live stream. They can see the video and hear audio with no time limit. During the live session, they can participate in real-time features like chat (sending messages that the streamer and other viewers can see), polls (answer questions the streamer poses), and sending quick reactions (like an applause or heart emoji that might float on the screen). This interactivity enhances their workout experience.
* **Post-Session & Continued Engagement:** After a workout stream ends, the user might receive a summary or a friendly sign-off message. They can look forward to the next scheduled session from that streamer or even explore other streamers if their subscription tier allows. They might also receive notifications about upcoming streams or community updates via email or in-app notifications to keep them engaged.

### **5.2. Streamer Journey**

* **Streamer Onboarding:** (Administration process) Initially, streamers are onboarded into the platform by the CelebFitLife team. For MVP, there is likely a vetting or invitation process – streamers might receive a special link or credentials to set up their account. Each streamer is configured with a special **streamer role** in the system (e.g., identified by a dedicated email domain like name.streamer@celebfit.com or a flag in their account) to distinguish them from normal users. This ensures that only authorized celebrities/instructors can broadcast content.
* **Streamer Login:** The streamer logs into the platform via a secure interface. This could be the same login page as users (with the system recognizing their streamer status) or a separate streamer portal. Using their credentials (email/password, possibly with 2FA for security), they gain access to streamer-specific functionalities.
* **Dashboard:** Upon login, the streamer is presented with a **dashboard** that is tailored to their needs. The dashboard shows key metrics (e.g., current live viewers count when streaming, total subscribers to their channel), and tools for scheduling and managing streams. It also includes a chat moderation panel for during live sessions and possibly a list of recent subscriber activities (new subscribers, cancellations) or feedback. The layout is designed for clarity since the streamer might access it while also managing a live workout.
* **Scheduling Live Sessions:** From the dashboard, the streamer can schedule upcoming live workout sessions. They select a date and time (with time zone support) for their next class and perhaps add a title or description (e.g., “Monday Yoga Flow with Alice”). When they schedule a session, the system will notify all users who are subscribed to this streamer (for example, via email or push notification: “Alice has scheduled a new live workout on March 5 at 6 PM”). *Upcoming streams also appear on the site for users to see and plan for.*
* **Starting a Live Stream:** When it’s time for a scheduled session, the streamer uses the platform to **launch the live stream**. Behind the scenes, this uses Twilio/Mux’s video streaming service. The streamer likely clicks a “Go Live” button on their dashboard. The app then connects to Twilio/Mux’s network: it either creates or joins a Twilio/Mux video room designated for that streamer’s session. The streamer’s camera and microphone feed start broadcasting to the room. The interface might show them a preview of their own video, the viewer count in real-time, and a live chat window. Streamers can see messages from viewers as they come in, and they can respond verbally on video or type in chat if needed (though likely they’ll focus on instructing verbally). There may also be controls to start/stop the stream, mute audio, or end the session and test their hardware devices(mic, camera, speaker).
* **Live Interaction & Moderation:** During the live class, the streamer can interact with the audience. They might pose questions or create a poll (“Which muscle group should we focus on next?”) using an in-app control. The viewers’ responses or votes are aggregated and shown to the streamer (and possibly everyone once the poll ends). Streamers can also see quick reactions (like emoji floating up) which give them feedback that people are enjoying or keeping up. For moderation, if someone in the chat is disruptive, the streamer (or an assigned moderator) has tools to mute or ban that user. This could be done through the chat interface by clicking on a username and selecting an action (mute for the remainder of session or ban from future sessions). The streamer might also have a trusted partner or assistant logged in as a **Moderator** to handle this so the streamer can focus on the workout.
* **Post-Stream Wrap-up:** After the session, the streamer ends the stream via the dashboard. They might then see a summary of the session on their dashboard: e.g., peak concurrent viewers, total unique viewers, poll results, new subscribers gained during the session, etc. This data helps them understand their engagement. The streamer can then schedule the next session or send a follow-up message to subscribers if needed (for example, a thank-you email or a note about the next session, possibly using the notification tools).

In summary, the streamer journey is about **planning, broadcasting, and managing** their live content, with the platform handling all technical streaming details, so the streamer can concentrate on delivering a great workout and engaging with fans.

## **6. Functional Requirements**

This section details the specific functional requirements of the CelebFitLife platform. Each requirement is labeled with a unique identifier (FRD-XX) for traceability. The requirements are grouped by feature area.

### **6.1. Landing & Subscription**

**Landing Page Display**

* **FRD-L1:** The landing page shall display basic information about the service, including a high-level description of CelebFitLife (e.g., tagline, how it works) and visual highlights of featured streamers or sample workout sessions. This is to quickly inform new visitors about the platform’s value proposition.
* **FRD-L2:** The landing page shall present the available subscription tiers (e.g., *1-Streamer Plan*, *3-Streamer Plan*, *Unlimited Plan*) with clear pricing for each. Users should be able to easily compare what each tier offers (how many streamers they can access, monthly cost, etc.).
* **FRD-L3:** A prominent Call To Action (CTA) button (e.g., “Join a Workout” or “Get Started”) shall be displayed on the landing page. Clicking this CTA directs an unauthenticated user to sign up (or login if they have an account), or if already authenticated, directly to the main streaming interface or their dashboard.

**Free Trial Offer**

* **FRD-L4:** Upon their first registration, a new user shall receive a free trial equivalent to **one full live workout session** at no charge. This trial allows the user to join one streamer’s live session from start to finish, so they can experience the platform without immediately paying. *(Implementation note: The user can choose any one live session to use this free trial on. Once they have fully watched one session using the trial, the trial is considered used up.)* The system should clearly communicate the trial terms (e.g., “Enjoy your first session free!”).

### 

### **6.2. Registration & Authentication**

**User Sign-Up Flow**

* **FRD-A1:** The system shall provide a registration form where users enter required information including Username, Email, Password, Phone Number, Country & City, and Age. Minimal necessary personal details should be collected to personalize the experience (location may help with time zones for scheduling, age could be used for demographic metrics or ensuring appropriate content if needed).
* **FRD-A2:** Users must explicitly accept the platform’s Terms & Conditions and Privacy Policy during sign-up (e.g., via a required checkbox) before the account can be created. This ensures legal compliance and user consent.
* **FRD-A3:** All user credentials and personal data must be stored securely. If using Firebase Authentication (as planned), the platform shall rely on Firebase to securely store passwords (hashed) and manage authentication tokens. No plaintext passwords will ever be stored.
* **FRD-A4:** Upon initial registration, the system shall require email verification. This can be implemented via Firebase’s email verification mechanism (sending a verification email with a link) or via Email.js to send a custom verification email. The user must confirm their email address before gaining full access, ensuring the email is valid for communication.

**Login Methods**

* **FRD-A5:** The platform shall support multiple login methods. At a minimum, email and password login is supported. Additionally, social login options should be provided for convenience, such as **Google** and **Facebook** OAuth, allowing users to authenticate using those accounts. (Firebase Auth can facilitate these social logins.)
* **FRD-A6:** The platform shall offer “passwordless” login via email link as an alternative. For example, a user can request a magic link to be sent to their email; clicking that link will log them in (Firebase Email Link authentication can be used for this). This provides flexibility for users who prefer not to remember passwords.
* **FRD-A7:** (Twilio/Mux Integration Gate) When a user attempts to access a live stream, the system shall verify the user’s subscription status **before** providing any Twilio/Mux streaming credentials (tokens). In practice, this means the backend will only generate a Twilio/Mux Access Token for joining the video room if the user is authorized (has an active subscription or trial for that streamer). This prevents unauthorized viewers from sneaking into a stream via the Twilio API directly.

**Streamer Onboarding & Auth**

* **FRD-A8:** The system shall support a distinct onboarding process for Streamers. Streamer accounts can be created by an admin or through a controlled registration that marks them as streamers. Only accounts flagged with a *Streamer role* (or using a special verified email domain) shall have access to broadcasting features. Regular users cannot arbitrarily become streamers without approval.
* **FRD-A9:** Streamers shall authenticate similarly to users (using email/password or supported SSO), but upon login the system shall recognize their streamer status and redirect them to the streamer Dashboard (instead of the viewer interface). The platform must enforce authorization checks so that streamer-only pages (like scheduling or streaming control panels) cannot be accessed by normal users. Conversely, streamers should also have a way to see the site as a user if needed (e.g., they might also subscribe to others), but primary navigation will be their dashboard.

### **6.3. Payment & Subscription Management**

**Payment Integration (Subscription Purchase)**

* **FRD-P1:** Users shall enter payment details through a secure form during the subscription sign-up process (immediately after registration or when upgrading). If using Stripe, this will be a Stripe Elements form or Checkout page ensuring PCI-compliant handling of card data. **No sensitive payment data** ever touches our servers directly; it’s handled by Stripe’s secure widgets.
* **FRD-P2:** The system shall **not charge the user until the free trial is utilized or expires**. Concretely, if the trial is “one free session”, the user’s card will be charged **the day after** they use that session, or if a time-based trial (e.g., 7 days) is offered instead, then at the end of that period. For the one-session trial, we interpret that as essentially the next day after they have attended a session, their billing cycle starts. (E.g., user signs up on Jan 1, uses their free session Jan 3, then billing begins Jan 4 for the plan they selected.) The exact billing trigger will be defined (either usage-based or time-based).
* **FRD-P3:** Users can manage their subscription from their profile/account settings. This includes the ability to **cancel** their subscription (preventing future renewals), **upgrade or downgrade** to a different plan (e.g., from 1-streamer to 3-streamer plan or vice versa). Upgrades should take effect immediately or on the next billing cycle per business rules, and downgrades typically on the next cycle. The UI should make the effects clear (e.g., “Your new plan will start on date X”).
* **FRD-P4:** The system shall handle recurring payments on the defined interval (likely monthly). This means automatically charging the user’s saved payment method each month (or period) for the active subscription. Stripe’s subscription APIs can manage this schedule, and the platform must respond to webhook events (like successful payment, failed payment, cancellation) to update user access accordingly.

**Subscription Packages & Limits**

* **FRD-P5:** **Package 1 – Single Streamer:** The user’s subscription allows them to have full access to *one* streamer’s live sessions. They must choose which streamer’s channel they subscribe to (typically at sign-up or via a selection interface). They can still preview others for 1 minute, but full access is only for their chosen streamer.
* **FRD-P6:** **Package 2 – Three Streamers:** The user’s subscription allows full access to up to *three* streamers of their choice. The user will designate which 3 streamer channels they want to subscribe to. They can watch any of those three without restriction.
* **FRD-P7:** **Package 3 – Unlimited:** The user’s subscription allows unlimited access to all streamers on the platform. There is no limit to how many different live sessions they can join (one at a time) or which streamers they follow. This is the premium tier.
* **FRD-P8:** The system shall track the number of streamers each user is subscribed to (especially for limited plans) and enforce those limits. For example, a user on the 1-streamer plan cannot accidentally subscribe to a second streamer without upgrading their plan first. Attempting to do so would prompt an upgrade.
* **FRD-P9:** On the streaming page, if a user who is not subscribed to a particular streamer tries to watch that stream, the system shall allow the **1-minute preview** (if they haven’t exhausted that for the month for that streamer) and then present a prompt to subscribe. This prompt should clearly indicate that subscribing (or using their free trial, if not yet used) will grant full access to the rest of the session. Essentially, unsubscribed or free users should always see a CTA to subscribe either overlaying the blurred stream or in the modal when the preview ends.

**Multiple Payment Options**

* **FRD-P10:** The platform shall support **PayPal** as an alternative payment method alongside Stripe. Users should be given a choice, for example, “Pay with Credit/Debit (Stripe)” or “Pay with PayPal,” when entering payment info. If PayPal is chosen, the user will be redirected to PayPal’s secure checkout to approve a recurring subscription agreement. The system will need to handle PayPal IPN/webhooks to know when the subscription is created, active, or cancelled. Providing PayPal support ensures users in regions or with preferences where PayPal is more convenient can still subscribe.
* **FRD-P11:** The subscription management features (upgrade/downgrade/cancel) must work seamlessly for both Stripe and PayPal subscribers. For instance, if a user who paid via PayPal cancels from our app, our system should trigger the cancellation in PayPal’s system via API or guide the user appropriately. Similarly, upgrading a PayPal user might involve creating a new subscription tier in PayPal. The user’s subscription status in our database should always reflect the truth from either payment provider. In short, multiple payment providers are abstracted into one uniform subscription status for the user.
* **FRD-P12:** Users on limited streamer plans (1 or 3 streamers) shall be able to manage which streamers they are subscribed to. The system should provide a UI (perhaps on the user’s profile or subscription settings page) to **select or change their chosen streamers** up to their plan’s limit. For example, if on a 3-streamer plan, the user can have 3 active subscriptions and can swap one out if they want to follow a different streamer next month. Business rule could allow changes immediately or only at the start of a new billing cycle to prevent abuse; this needs to be defined. Initially, we assume they can make changes at any time, but once they exceed their limit, they must deselect one before adding another. The platform will enforce that no more than the allowed number of streamers are active at once for that user.

### **6.4. Streaming & Live Interaction**

**Browsing and Accessing Streams**

* **FRD-S1:** If a user is **not logged in**, the Streaming page (or main dashboard) shall show a grid of streamer thumbnails (with perhaps a snapshot or name of the class) that are live or upcoming, along with categories/tags and a search bar. However, clicking on any of them should prompt login or registration, since viewing streams requires an account (even for a preview, we might require at least an account so the preview usage can be tracked – this detail can be decided, but likely yes to prevent infinite incognito previews). Essentially, guests can see what’s available but need to sign in to watch.
* **FRD-S2:** If a user is **logged in**, the streaming homepage will personalize content. They will see the same grid of live streams, plus possibly a section or slider of “Recommended” or “Your Subscriptions” at the top – highlighting streamers they are subscribed to or might be interested in (similar to Twitch’s followed channels). This helps users quickly find their subscribed content. There should also be a clear indication of which streams they have full access to (maybe a badge like “Subscribed”) versus which would be a preview (with a “Preview” label or lock icon).

**Stream Access Control (Previews vs Full Access)**

* **FRD-S3:** A user who is not subscribed to a given streamer (including a logged-in free user or a subscriber who has not chosen that streamer in their plan) is allowed to watch that streamer’s live session for **1 minute as a free preview**. This preview limitation applies **per streamer** basis. (E.g., a user could preview Streamer A for 1 minute and Streamer B for 1 minute, but not the same streamer repeatedly without subscribing.) Each preview session should be tracked. The timer starts when the video stream begins playing for that user.
* **FRD-S4:** A countdown timer shall be displayed for users in preview mode. It should be unobtrusive but visible (for example, a small timer in the corner of the video player) showing the remaining seconds of free viewing (e.g., “Free Preview: 0:45”). This gives the user clear feedback on how much time they have before the preview ends.
* **FRD-S5:** When the preview time is almost up (e.g., at 5 seconds remaining), the system shall display a notification or modal warning the user that the free preview is about to end. For instance, a message “Preview ending in 5...4...3...” with a suggestion to subscribe to continue watching. This prepares the user and encourages them to decide to subscribe quickly if they want to keep watching without interruption.
* **FRD-S6:** Once the preview timer reaches zero, the system shall immediately enforce the restriction: the video area becomes blurred or hidden, and audio is muted. A modal or overlay takes over the screen, informing the user that the preview has ended. The overlay will include a **CTA to subscribe** (or log in, if somehow a guest got that far) to continue watching. The user cannot see or hear more of that stream until they act (subscribe or close the modal – closing would likely return them to the stream selection page). If the user closes the modal without subscribing, the stream remains inaccessible (we might show a static screen with subscription options or simply not allow re-entry for that stream).

**Anti-Abuse: One-Time Preview Limits**

* **FRD-S7:** The system shall implement measures to prevent users from abusing the free preview mechanism. It should track a combination of the user’s identity and environment, such as IP address and device/browser fingerprints or cookies, to ensure a person cannot simply refresh or use incognito mode or switch devices to get infinite 1-minute previews. For example, if a user not logged in tries to watch 1 minute of Streamer A and then opens a new private browser window to do it again, the system should detect this via IP and possibly other fingerprinting and block the second attempt (or require login to enforce one preview per account). Similarly, a logged-in user should not get more than one preview for the same streamer in the defined period (see FRD-S8). While IP tracking alone is a start, we anticipate using browser local storage or fingerprinting techniques for more robustness.
* **FRD-S8:** Free preview access is limited to **once per streamer per month per user** (whether the user is logged in or just tracked via device). This means if User X already used their 1-minute preview for Streamer Y this month, they cannot get another preview of Streamer Y until the next month. They would need to subscribe to watch more of Streamer Y’s content. This rule resets monthly to allow users to sample again later, but prevents continuous free watching. This limitation applies equally to logged-in users without a subscription for that streamer and to anonymous sessions (though anonymous usage will be curbed by requiring login as noted). The backend should record preview usage events (with timestamp) to enforce this.

**Live Chat, Polls, and Reactions**

* **FRD-S9:** Logged-in users who are viewing a live stream (and have access beyond a preview) can participate in live chat. Chat messages shall appear in real-time to all participants (the streamer and other viewers). The chat interface will display messages in chronological order with the username and message. The system should handle potentially thousands of messages across many viewers, so it must be optimized (likely using Twilio’s data channels or a real-time database). If a user is banned from chat (see moderation), their messages will not appear.
* **FRD-S10:** Streamers (and possibly moderators) can create polls during a live session. When a poll is created (e.g., a question with multiple choice answers), it shall appear for all viewing users in an interactive panel or pop-up. Users can vote once per poll. The system collects votes in real-time and, when the streamer closes the poll, shows the results to everyone (percentages or winning option). Poll results could also be updated live as votes come in, to build excitement. This feature helps increase interactivity and audience engagement.
* **FRD-S11:** Quick reaction mechanisms (such as sending an emoji like a thumbs-up, heart, or clapping icon) shall be available to viewers. When used, these reactions may briefly float up or flash on the video overlay or appear in a dedicated part of the UI, visible to all viewers and the streamer. These are not exactly chat messages but lightweight feedback. The system should aggregate or throttle these if the audience is large (to avoid overwhelming the streamer visually). Reactions happen in real-time as well. They provide a way for viewers to participate even if they don’t send chat messages.

**Stream Security & Authorized Access**

* **FRD-S12:** Live streams shall be secured such that only authorized users can access them. The platform will use unique **Twilio Access Tokens** tied to each user session to join the streaming video room. This means that simply knowing a stream’s URL or an identifier is not enough – you must have a valid token issued by our backend (which checks your subscription). Tokens are short-lived and cannot be re-used. This ensures that the video feed cannot be pirated or shared freely outside the platform. Additionally, the video and audio streams are transmitted securely (Twilio uses DTLS/SRTP for encryption in WebRTC), protecting against eavesdropping. Any attempt to join a Twilio room without a valid token will fail, thus preventing unauthorized viewing.
* **FRD-S13:** Users who have subscriptions that include multiple streamers (e.g., the 3-streamer or unlimited plan) may switch between different live streams of their subscribed streamers freely. For example, a user on the 3-streamer plan can watch Streamer A’s session for a while, then leave and join Streamer B’s ongoing session, and so on, without any additional payment or restriction (aside from the general limitation that one account shouldn’t stream multiple videos concurrently on one device for bandwidth reasons). The system should handle granting access tokens to any stream they are entitled to. This requirement simply clarifies that the platform will not charge or block a user from switching among streams they have paid for. (If a user tries to join two streams at the exact same time on different devices, we may allow it if technically feasible, but it’s not a typical use-case to emphasize.)

### **6.5. Moderation & Access Control**

**Moderation Roles & Privileges**

* **FRD-M1:** Each streamer can designate one or more **Moderators** for their channel. Moderators are usually trusted individuals (maybe a friend, assistant, or community manager) who help oversee the chat during live sessions. A moderator has the power to manage chat participants but typically cannot start/stop the stream or do things reserved for the streamer. The system shall allow a streamer to assign or revoke moderator status to other user accounts (likely done via the streamer’s dashboard, either by entering the user’s email or selecting from their subscribers).
* **FRD-M2:** The streamer shall be able to manage their list of moderators via the Dashboard. This includes adding a moderator (which might send an invite or automatically grant that user mod privileges in future streams) and removing a moderator. Only the streamer (and possibly platform admins) can assign moderators for a streamer’s channel. Moderators’ privileges are limited to that streamer’s sessions (they are not global moderators for other channels, unless separately assigned there).

**Ban & Mute Functionality**

* **FRD-M3:** If a moderator or streamer bans a user during a live session, that banned user shall immediately lose the ability to send chat messages in that streamer’s chat. The ban for MVP is on a per-streamer basis: a banned user cannot participate in *any* future live chats of that particular streamer. (They might still be able to watch the video if they are subscribed, unless we decide to also remove their viewing privileges as a ban – but initially, ban is just chat-ban. Hard bans from the platform would be an admin-level action.) The system will persist the ban in the database so that on subsequent streams, that user’s chat input is blocked or simply not shown to others.
* **FRD-M4:** A *mute* is a temporary silence. If a user is muted by a mod or streamer, the user’s chat messages will not appear for others for the remainder of that live session. Mute does not necessarily notify the user (we might or might not tell them they are muted, to avoid confrontation). After the session (or a set mute duration), the mute lifts automatically. This is useful for handling spammy users without banning them permanently. The system should handle this by essentially ignoring/hiding messages from muted users on the client side (with the server not broadcasting them).

**Reporting & Content Moderation**

* **FRD-M5:** *(Initial implementation optional)* The platform should allow users or moderators to **report** malicious or inappropriate behavior in chat. This could be a simple UI option on a message (e.g., a flag icon to report a message/user). Reports would ideally be sent to a backend or admin interface for review. In the MVP, this might not be fully built out for users (since moderators can directly ban), but we plan for an eventual admin dashboard to handle user reports. For now, we note this as a future consideration: the data model should accommodate storing reported incidents for later review by the platform administrators.
* **FRD-M6:** The chat system should employ basic **automated moderation filters** to assist human moderators. For example, messages containing certain profanity or hate speech keywords can be automatically filtered or blocked from being posted. Similarly, flooding the chat (too many messages too quickly) could trigger a temporary auto-mute. These automated rules will help maintain a positive environment especially in fast-moving chats. The implementation might use a list of banned words and simple rate-limiting on message sending. This is a functional requirement to ensure baseline community standards even if a moderator is not quick enough to catch something.

### **6.6. Notifications & Scheduling**

**Streamer Scheduling & Announcements**

* **FRD-N1:** Streamers can set specific dates and times for upcoming live sessions via their dashboard. The scheduling interface should allow picking a date/time (with time zone awareness). Once scheduled, the session info (time, title, duration, etc.) is saved in the system and visible to users.
* **FRD-N2:** The system shall display upcoming scheduled streams to users, especially those who have subscribed to the respective streamers. For example, a user’s “Subscribed Streamers” section might show “Upcoming: Yoga with Alice at 6 PM” so they know what’s coming. Additionally, there could be a global schedule page showing all upcoming sessions (useful for discovery). This ensures users can plan ahead to attend live or decide which sessions to join.
* **FRD-N3:** Subscribed users shall receive an **email notification** when a streamer they follow schedules a new session or is about to go live. For instance, an email like “Don’t miss out! Alice goes live in 30 minutes.” We can leverage Email.js or Firebase’s email service to send these programmatically. The timing of notifications might be configurable (e.g., immediately on scheduling, and a reminder 30 minutes before the event). The content will include session details and a link to join.
* **FRD-N4:** Streamers shall have the ability to send announcements or updates to their subscribers via the platform. This could be done through push notifications (if we have a web push or mobile app in the future) or via email. Examples include: a streamer needs to **delay or reschedule** a session (“Tonight’s workout will start 30 minutes late”), or they want to hype an upcoming session (“Special guest joining my next workout!”). The system should provide a way for streamers to craft these messages in their dashboard and dispatch them to all their subscribers. For MVP this might simply trigger an email to subscribers. Moderation or rate-limiting of these announcements might be needed to prevent spam.
* **FRD-N5:** All scheduled times displayed to users shall be converted to the user’s local time zone. The system will use the user’s provided location (Country & City from sign-up) or browser settings to show times in a friendly format (e.g., “Today at 7:00 PM your time” instead of UTC or streamer’s time). This is crucial for usability given a potentially global user base – users shouldn’t have to do time conversions to figure out when to join.
* **FRD-N6:** If a streamer cancels or changes the time of a scheduled session, the system shall update all relevant displays (so the session is marked as cancelled or the time is updated) and notify all subscribers of the change. This could be via an email like “Update: Alice’s workout has been rescheduled to 7:00 PM.” Keeping subscribers informed of changes maintains trust and reduces confusion or missed sessions.

## 

## **7. Non-Functional Requirements (High-Level)**

Beyond specific features, the platform must meet certain overarching standards:

* **Scalability:** The system must handle an increasing load as the user base grows. For year one, anticipate up to ~1,000 concurrent users per celebrity stream (in popular sessions) and scale beyond in year two. The architecture (using serverless and managed services) should allow adding more capacity easily. For example, Twilio’s infrastructure should support large audience rooms, and our front-end and Firebase backend should handle high concurrent connections (using Firestore or realtime database scaling, CDN for static assets, etc.). We also plan for horizontal scalability of any custom servers (though Next.js on Vercel and Firebase are inherently scalable).
* **Reliability:** Live streaming demands high uptime and stability. The platform should target **99.9% uptime** for the core service during active hours. Twilio’s service reliability is crucial – it must handle multiple concurrent streams and associated chat sessions without degradation. There should be minimal disconnects or crashes. We will rely on Twilio’s proven infrastructure for streaming; on our side, we should ensure proper error handling (e.g., if a stream fails, the system can allow the streamer to reconnect quickly, and viewers are informed). Critical flows like payment processing and authentication should also be highly reliable, with fallback options if third-party services have issues (e.g., a graceful message if Stripe is down during checkout).
* **Security:** The platform will deal with personal user data and payment information. We must protect user data by design. All communications will occur over HTTPS with TLS encryption. Firebase Authentication and JWT tokens will secure user sessions. Access to video streams is secured by Twilio tokens as described (FRD-S12). Payment processes are secured by Stripe/PayPal – we never store raw card details. We will enforce secure password policies and possibly 2FA for streamers or admins. Additionally, we’ll guard against common web vulnerabilities (SQL injection is largely mitigated by using Firebase, XSS and CSRF protections via frameworks, etc.). Content security: ensure that stream URLs or Twilio credentials cannot be easily extracted to bypass our app. Also, the platform should conform to GDPR or other privacy laws for data handling (users can request data deletion, etc., likely facilitated by Firebase).
* **Performance:** Users should experience minimal latency and fast load times. The site (Next.js front-end) should load quickly – initial content should display within 2-3 seconds on a typical connection. SSR in Next.js will help with fast first paint and SEO. For the live video, Twilio’s **interactive live streaming** is expected to have ~2 seconds or less of latency, making the experience feel real-time. Chat messages should appear virtually instantaneously (<100ms network transit ideally). We will optimize images and assets with Next’s optimization features and use CDN caching for the landing page and static content. The video and chat handling will be optimized by offloading to Twilio and possibly using efficient WebRTC and data channels. The application should also handle less ideal network conditions gracefully (e.g., video quality might auto-adjust via Twilio’s bandwidth optimizations, and the UI should indicate if a user’s connection is weak).
* **Usability:** The platform should be **easy to navigate and use** for both viewers and streamers. The user interface will be clean, with clear calls to action (sign up, subscribe, join stream, etc.). Complex tasks (like subscribing or managing a subscription, joining a stream, using chat) should be as simple as possible. We will use consistent design patterns (with Tailwind CSS ensuring a cohesive look and feel). The site must be **responsive**, working well on various screen sizes (mobile phones, tablets, desktops) since users or even streamers might occasionally join on mobile. For streamers, the dashboard and streaming controls will be intuitive – large buttons for start/stop, clear labeling for metrics, etc. Additionally, the platform should provide feedback for user actions (e.g., if you click subscribe, show a loading state and confirmation). We also consider accessibility: using proper HTML semantics, labels, and considering features like subtitles or transcripts in future to not exclude users with disabilities (not a hard requirement in MVP, but we aim to not design in an inaccessible way).

## **8. System Architecture, Assumptions & Dependencies**

This section outlines the overall architecture of the CelebFitLife platform and provides a breakdown of the technology stack, justifying each component and noting key dependencies and how they interact. The design leverages modern web technologies and third-party services to meet the requirements efficiently.

At a high level, **CelebFitLife’s architecture** is a serverless-oriented web application:

* The front-end is a **Next.js 15** application (React), which delivers the user interface and runs both client-side and server-side as needed.
* The app relies on **Firebase** for many backend functions (authentication, database, and possibly cloud functions), reducing the need for a custom server.
* **Twilio/Mux** services are integrated for live video streaming (and possibly chat), handling the heavy lifting of real-time communication.
* **Stripe and PayPal** are integrated for payments, each through their secure APIs, to handle subscription billing.
* The state management on the front-end uses **Zustand** for simplicity, and styling is done with **Tailwind CSS**.
* **EmailJS** is used to send transactional emails (though Firebase Auth can send verification emails too, EmailJS might be used for custom notifications or marketing emails).
* The entire system is broken into components: the web app (UI + Next.js server functions), the third-party services (Twilio, Stripe/PayPal, Firebase, EmailJS), and data storage (Firestore/Realtime DB for app data).

**Key Components & Technologies:**

* **Next.js 15 (React Framework)** – *Why Next.js:* We chose Next.js for its ability to render pages on the server side, providing fast initial loads and SEO-friendly pages (useful for the landing page and any public profiles). Next.js 15+ also supports the latest React features and a robust development ecosystem. The front-end is a single-page application experience for logged-in users, but with the power of Next.js, we can have serverless API routes to handle backend logic (like generating Twilio tokens or processing webhooks) without maintaining a separate server. The project likely uses Next.js’s file-based routing for pages (e.g., pages for landing, login, stream, dashboard) and may use the new App Router for improved routing and layout handling. Next.js will also make deployment easier (e.g., on Vercel or another Node hosting, auto-scaling as needed).
* **React 18 & Client-Side Logic:** The user interface is built with React components. This allows a dynamic, interactive UI (for chat updates, live viewer counts, etc.). React’s declarative nature and component reuse will speed development. With React 18, we can use features like concurrent rendering if needed. The app likely uses functional components and React Hooks extensively (including custom hooks for things like useAuth or useSubscriptionStatus).
* **Zustand (State Management):** Zustands is a lightweight state management library used on the front-end to handle global state outside of React’s context. We chose Zustand over heavier solutions (like Redux) because the state needs (tracking user info, current stream data, UI toggles) are relatively straightforward. Zustand allows us to create a global store that components can subscribe to, with very little boilerplate. For example, when a user logs in, we can store the user profile and subscription status in the Zustand store so any component (like the nav bar or the streaming page) can access it. It’s fast and plays well with React’s concurrent mode. Zustand will be particularly handy for things like maintaining the list of online streamers, chat messages (if we store some in state), or the status of the Twilio connection.
* **Firebase Authentication:** We rely on Firebase Auth for user identity management. This provides secure, out-of-the-box handling of user sign-up, login, password hashing, and social logins. The Next.js app will integrate with Firebase via the Firebase JS SDK. When a user logs in, Firebase issues a JWT that we can use to identify the user on the client and also secure any API routes or Firestore data. Using Firebase means we don’t have to build our own auth system from scratch or manage our own user credential database. It’s also easy to use Firebase’s features for email verification and password reset emails, which are important parts of the auth flow. Assumption: Firebase Auth is already set up for email/password and Google/FB login methods.
* **Firebase Firestore / Realtime Database:** For application data (beyond auth), we use Firebase’s cloud database. Firestore (or the Realtime Database) will store data such as user profiles (e.g., what plan they’re on, which streamers they subscribe to), streamer schedule info, subscription records (or at least references to Stripe/PayPal subscription IDs), chat transcripts or messages (if we choose to store them), and moderation records (bans, etc.). The advantage is real-time syncing – if we use the Realtime DB or listen to Firestore updates, viewers could get live updates (like chat could technically be done through this as well). Firestore provides a scalable, noSQL datastore that requires no server maintenance and can be directly accessed securely from the client with rules. We will define security rules to ensure, for example, users can only read/write certain collections (like they can read public streamer schedules, but only update their own profile, etc.). We assume Firebase can handle the scale of simultaneous connections for chat if needed (though Twilio is another option for that). We also assume we might use Firebase Cloud Storage if streamers upload any images (profile pictures, etc.), though not heavily needed for MVP except maybe storing thumbnails.
* **Twilio Programmable Video (Live Streaming):** Twilio is a core dependency for the live streaming feature. The way it works in our app: when a streamer clicks "Go Live", our system will use the Twilio Node SDK (or a Twilio REST API call) to create a video **Room** (if not already created) for that session. We likely use a Group Room or Twilio’s low-latency “Twilio Live” product if available, which allows one (or few) broadcaster(s) and many viewers. Each viewer who joins triggers our backend to generate a Twilio **Access Token** containing a VideoGrant (permission to join that specific room) and possibly a Chat grant if using Twilio for chat. This token generation is done securely on the server side (for example, via a Next.js API route or Firebase Cloud Function). It uses our Twilio Account SID and Auth Token (kept secret) plus the Twilio API Key/Secret for signing JWTs. The token also encodes the user’s identity (maybe their username or an ID) and role. For instance, the streamer’s token might mark them as a *publisher* (with permission to publish video/audio), while viewer tokens might be *subscriber* only (to just receive video/audio). Twilio’s client-side SDK (twilio-video.js) is used in the React app to connect to the room using the token. Once connected, Twilio takes care of peer-to-server (or peer-to-peer in small rooms) communication. All viewers essentially subscribe to the streamer’s video track that Twilio relays, with minimal delay. Twilio handles scaling the media server as more viewers join.  
  + *Twilio Chat Integration:* Twilio also offers either a data channel in the Video API or a separate **Twilio Conversations** service for chat. We have two possible implementations: (1) Use Twilio’s **data tracks** within the video room for chat messages – the streamer or a viewer sends a message over the data channel, and Twilio distributes it to everyone in the room. This keeps video and chat in one network session, but might need custom handling to display and persist messages. (2) Use **Twilio Conversations (Programmable Chat)**, which is a dedicated chat service where a chat room (channel) is created for each stream, and participants send messages through that. Twilio Conversations provides message persistence and even moderator roles, but it’s a separate API to integrate. (3) Alternatively, not use Twilio for chat at all and use Firebase realtime DB for chat messages since we already have it. This was an open design point.
* For MVP simplicity, we lean toward using Twilio’s own capabilities to reduce the number of separate moving parts. Twilio’s data track can transmit chat in real-time along with video, ensuring messages are delivered in sync with the stream and only to those in the room. The main drawback is data tracks are transient (no built-in storage of chat history). That might be acceptable for now (chat scrolls live and is gone when the session ends). If we want persistence, we could log messages to Firestore when they arrive. The architecture will include a decision here (see Open Questions). Regardless, Twilio’s role is to provide the live connectivity.  
  + *Room Management:* The system may choose to create a fresh Twilio video room for each scheduled session (like room name "streamer123\_session\_2025-03-01"), or have a persistent room per streamer (like room name "streamer123\_live" that they use every time they go live). A persistent room simplifies things (one room per streamer channel), but Twilio might keep resources open if not ended properly. We might create and destroy rooms for each session for cleanliness. Twilio’s API allows us to end a room when the streamer disconnects.
  + *Latency and Quality:* Twilio is designed for real-time interactivity, which is key for our “live” feel. The video quality can be configured (720p or 1080p depending on bandwidth). We assume most users have sufficient internet for video; Twilio will adapt streams to lower quality if a user’s bandwidth is low (to avoid buffering). This adaptation and the heavy network lifting is all handled by Twilio’s infrastructure, meaning we don’t have to build our own media server.
* In summary, Twilio integration means our architecture includes a **Twilio Token Service** (server side) and the **Twilio Client SDK** (front-end). The token service ensures only eligible users join streams, and the client SDK handles capturing streamer video, displaying video to viewers, and sending/receiving chat or other real-time signals. We assume Twilio’s service is already set up with necessary API keys, and perhaps Twilio Live (if needed for scaling beyond 50 participants in a room by using a media server approach).
* **Stripe Payments:** We use Stripe to handle credit/debit card payments and subscription billing. The architecture here is: on the front-end, when a user chooses a plan and enters card details, we either redirect to a **Stripe Checkout** page (hosted by Stripe) or use **Stripe Elements** (an embeddable card input field) in our React app. In both cases, the card data goes directly to Stripe. Stripe then returns a token or sets up a subscription based on how we implement. Likely we use Stripe’s Subscription API: we create a Customer in Stripe for the user, attach their payment method, and subscribe them to a Plan (we will have defined our subscription plans in Stripe’s dashboard with prices corresponding to 1-streamer, 3-streamer, etc.).  
    
   Our backend (Next.js API or Firebase Function) will handle the webhook events from Stripe. For example, when a subscription is created or a trial is started, Stripe sends an event (which we catch in a webhook handler) – we then mark the user in our database as subscribed to that tier, starting from a certain date. When a payment succeeds monthly, we might not need to do much (maybe extend their next billing date if we track it). If a payment fails or the user cancels via Stripe’s billing portal, we get an event and we should update the user’s status (maybe downgrade them after end of paid period, or flag payment issues).  
    
   We justify Stripe because it provides a robust, secure solution for global payments with minimal dev effort. It handles currency conversion (if needed), stores cards securely, and has features like built-in invoices, receipts, and the Customer Portal (which we could optionally allow users to manage their payment method through). Stripe’s reliability and documentation reduce errors in this critical revenue component.
* **PayPal Payments:** In addition to Stripe, the architecture includes PayPal integration for subscriptions. PayPal integration typically works by redirecting the user to PayPal to approve a billing agreement. On our side, we might use PayPal’s Subscription APIs: we create a Product and Plan in PayPal’s system corresponding to our offerings, then when user chooses PayPal, we initiate the creation of a Subscription and direct the user to approve it. Once approved, PayPal will notify us (via webhook or direct return) that the subscription is active. We then treat it similar to Stripe – update the user’s status in our DB. PayPal will send recurring payment notifications via webhooks (IPN Instant Payment Notifications or their newer webhook system) for renewals or cancellations. Our system’s backend must listen for those and handle them akin to Stripe events.  
    
   Having both Stripe and PayPal does add complexity in the backend (two sets of webhook handling), but it dramatically increases accessibility for users. Many international users or those without credit cards prefer PayPal. We assume the app’s serverless functions or API routes will include endpoints for PayPal IPN processing. We justify PayPal as a strategic choice to maximize user conversions and not lose users who don’t have a card ready but do have a PayPal account.
* **Firestore Integration for Subscriptions:** We will likely maintain a record in Firebase (Firestore) that indicates a user’s subscription status, plan tier, and which streamers they have picked (for limited plans). This record gets updated when webhooks from Stripe/PayPal come in. The front-end will use this to determine what the user can access (e.g., populate a list of subscribed streamer IDs for that user on login).
* **EmailJS (Email Service):** The platform will send various emails (account verification, password resets - though Firebase can do those, notifications for going live or scheduling - which we have as requirements). EmailJS is a service that we have set up to handle sending these emails reliably. The architecture likely involves our serverless backend calling EmailJS’s API to trigger an email template. For instance, when a streamer schedules a session (FRD-N3), our code will send an email to each subscriber via EmailJS (or potentially aggregate or use a mailing list). We justify EmailJS because it abstracts away SMTP and ensures high deliverability (important for things like verification emails not ending up in spam). It’s already integrated, meaning templates for welcome emails or notifications might already be configured.
* **Tailwind CSS (UI Library):** On the front-end, we use Tailwind CSS to style the application. Tailwind lets us apply utility classes (like bg-blue-500 text-white p-4 rounded) directly in JSX, which speeds up development and enforces consistency (we define a color palette, spacing scale, etc.). It’s highly maintainable and works well with React’s component structure. We assume a configured Tailwind setup in the Next.js app (with PostCSS) and possibly some custom theme values to match our branding. This choice is justified by the need for rapid UI iteration and a consistent design without writing a lot of custom CSS from scratch.
* **Deployment and Hosting:** Although not a specific requirement, an assumption is that we will deploy the Next.js application on a scalable hosting environment (such as **Vercel** or another Node.js hosting). Vercel is a good match for Next.js, offering serverless deployment of API routes and static site optimization. Alternatively, since we use Firebase, we could deploy the front-end on Firebase Hosting and use Firebase Cloud Functions for the API pieces (there’s also the possibility of using Firebase’s integration with Cloud Run for SSR). For now, we assume using Vercel or similar, which handles scaling for us. The Twilio and payment webhooks would be configured to hit our deployed endpoints.
* **Assumption – Device Compatibility:** The application will run in modern web browsers (Chrome, Firefox, Safari, Edge) and should be compatible with mobile browsers. Twilio’s video SDK supports WebRTC in mobile Safari/Chrome, so users on mobile devices can join streams through the browser (though experience may vary). We are not building native mobile apps for MVP, but that could be a future enhancement (the architecture can later incorporate a React Native app reusing some logic). We assume users have relatively up-to-date devices for video streaming; we’ll mention recommended specs in documentation.

In summary, the CelebFitLife tech stack is composed of **React/Next.js** for the UI, **Firebase** for user management and data, **Twilio** for streaming, **Stripe/PayPal** for payments, and supporting libraries like Zustand and Tailwind to glue it together. This stack was chosen to maximize development speed (lots of out-of-box functionality), scalability (managed services that scale automatically), and reliability (leveraging proven platforms for critical components).

All these components integrate to fulfill the functional requirements described earlier. The architectural approach also minimizes the need for a traditional always-on backend server – we delegate to cloud services and use serverless functions triggered by events (like user actions or webhooks). This reduces maintenance and costs, and allows the system to handle spikes in usage (e.g., a very popular live class) by scaling out horizontally.

## **9. Open Questions**

Despite the detailed requirements above, a few points remain open for discussion or future decision. These are areas where final decisions will be made with stakeholders or during technical design:

* **Moderator Scope:** Should moderators have any cross-channel capabilities or are they strictly tied to one streamer’s channel? *(Current assumption:)* Moderators are **per streamer**. A moderator for Streamer A cannot moderate Streamer B’s chat unless separately added by B. This keeps communities separate. A global moderation (platform-wide) could be a role for internal admins only, not regular user mods.
* **Multiple Streamer Subscription Switching:** If a user has a plan that allows multiple streamers (3-streamer or unlimited), can they easily switch which streamers they are subscribed to during a billing period? For example, on a 3-streamer plan with A, B, C selected, could they drop A and add D mid-month? And if they do, does that immediately remove access to A and grant to D? *(Assumption:* We will allow users to manage their selections (FRD-P12) on the fly, but we might limit very frequent switching to prevent circumventing the 1-minute preview system by constantly swapping choices. Perhaps a user can change their subscribed streamers but any new addition beyond their limit triggers removal of another, and we might log this to monitor abuse. We might also decide to lock selections for the month once chosen, but that reduces flexibility. This needs product input.)
* **Free Trial Limitations:** Clarification is needed on the free trial usage. We assume the free trial = one full session with any one streamer of the user’s choice. After they use it, all other streams require a paid sub or 1-minute previews. An alternative interpretation could be one free session per different streamer (but that would allow too much free usage, likely not intended). *(Decision:* It will be one session total per new user, not one per streamer. Essentially, the trial is tied to the account, not to each streamer. We should track when a user has consumed their free session. If they try to join another streamer’s session after using it, they’d be treated as unsubscribed with only the 1-minute preview available.)
* **Preview Enforcement – Device Fingerprinting:** How far will we go to prevent preview abuse? Using IP addresses is straightforward but can be insufficient (e.g., two different people on one corporate network might share an IP and get blocked incorrectly, or a savvy user could use a VPN to get another IP). We can set a cookie or use localStorage to tag a browser after a preview. True device fingerprinting (collecting browser/OS characteristics to form an ID) can be complex and raise privacy concerns. *(Plan:* For MVP, we will implement basic measures: require login for preview beyond a trivial teaser, and record previews per account; also set a cookie on the browser after a preview to discourage quick re-try, and possibly use IP as a coarse check. This should be enough for initial launch, and we acknowledge that determined users might get around it, but the effort likely isn’t worth it for them beyond a point. We will revisit if abuse is observed.)
* **Twilio Chat vs Firebase for Chat:** We need to finalize whether the chat feature will use Twilio’s capabilities or Firebase. Each has pros/cons:  
  + *Twilio Data Tracks:* Pro: integrated with video session, low latency, no extra infra needed. Con: no persistence, Twilio’s cost if data usage scales, max message throughput might be limited.
  + *Twilio Conversations API:* Pro: robust chat with user roles, persistence, and even multi-device sync. Con: separate service to integrate, more complex to implement initially, additional cost possibly.
  + *Firebase Realtime DB/Firestore:* Pro: we already use Firebase, can log and retrieve messages easily, and even do simple moderation like deleting from DB. Con: slight added latency (but usually fine, sub-second), and need to ensure the client subscribes/unsubscribes properly to chat data to avoid leaks across streams.

*(Current leaning:* use Twilio’s data track for simplicity in MVP — meaning chat messages are ephemeral and only exist in the context of the live stream, which might be acceptable. We will not have persistent chat history beyond the session. This is similar to how live streaming chats like Twitch are ephemeral. If needed, we can log some messages to Firestore for moderation review. If we find Twilio’s data track insufficient (maybe if scaling beyond a certain number or if ordering gets messy), we will consider switching to Firebase or Twilio Conversations later. This decision will be tested in prototype stage.)

* **Payment Provider Choice UX:** With both Stripe and PayPal supported, how will we present this choice to users and handle any edge cases? *(To decide:* We likely show two buttons or a toggle for “Pay with Card” vs “Pay with PayPal”. Need to ensure the UX is clear. Also, we should decide if we allow switching payment methods later on – e.g., a user subscribed via Stripe wants to switch to PayPal for future bills. That’s complex (would involve canceling one and re-subscribing via another). Possibly we won’t support mid-subscription switching at launch; users could manually cancel and re-subscribe at period end with the other method if they really want. This is a minor edge case to clarify in documentation.)
* **Recording or VOD (Future):** (Not in scope for MVP, but worth mentioning as future question) Will sessions be recorded and available for on-demand viewing? Currently, we assume **no** – the focus is on live experience. But this might be a common question from users (in case they miss a workout, can they watch it later?). Twilio can record streams if configured. For now, this is out-of-scope, but it’s a feature we might consider later. We’d then face questions of storage, access control for recorded content, and additional subscription value (maybe recorded sessions for subscribers only).

These open questions will be resolved in collaboration with the product team and with further technical evaluation. None of them are blockers for initial development, but they will influence fine-tuning of the platform’s behavior and are noted here for completeness.

## **10. Conclusion**

This Functional Specification Document outlines the vision, features, and technical approach for **CelebFitLife**, a live-streaming fitness platform that leverages celebrity instructors to engage users in interactive workouts. We have defined the core user journeys (for both viewers and streamers) and detailed functional requirements covering everything from landing page to live chat moderation. In addition, we justified our technology choices: using Next.js for a scalable web app, Firebase for rapid backend development, Twilio for reliable low-latency streaming, and Stripe/PayPal for secure global payments, with Zustand and Tailwind providing a solid foundation for state management and UI development.

This combination of features and technologies is aimed at delivering a **smooth, engaging user experience** while ensuring the solution is feasible to build, maintain, and grow. By choosing proven third-party services for the heavy lifting (auth, video, payments), the team can focus on building the unique aspects of the platform – the seamless integration of live video with interactive fitness-specific features and an intuitive user interface.

CelebFitLife’s design keeps future expansion in mind: while the MVP focuses on live sessions and subscriptions, the platform can later incorporate additional monetization avenues (e.g., one-off special event classes or merchandise), more advanced personalization (recommendation of classes, friend/follower social features), and possibly an ecosystem of recorded content or integrations with wearables for tracking workout stats. The architecture is flexible enough to accommodate these enhancements down the line.

In essence, CelebFitLife aims to be the **go-to destination for live fitness entertainment**, marrying the excitement of a live celebrity event with the personal benefit of a workout class. With the requirements and architecture defined in this document, the development team has a clear blueprint to start building this vision.

## **11. Next Steps**

* **Stakeholder Review:** Circulate this document to key stakeholders (product owners, engineering leads, possibly some pilot streamers) for review and sign-off. Ensure that the requirements align with business goals and adjust if necessary. Clear up any of the Open Questions with decisions from product/business perspective so the team has firm guidelines (especially on trial rules, chat implementation, etc.).
* **Technical Architecture & Design:** Based on the requirements, create a more detailed technical architecture plan. This would include defining the data model (e.g., Firestore collection schemas for users, subscriptions, schedules, etc.), the specific API endpoints or cloud functions needed (for Twilio token generation, Stripe/PayPal webhooks handling, etc.), and perhaps a basic component diagram showing how front-end components interact (for example, VideoPlayer component, Chat component, etc. and their integration with Zustand store and Twilio SDK). Also consider any devOps setup needed (API keys management, env configuration for dev/prod, etc.).
* **Development Roadmap:** Establish a development timeline with milestones. Likely start with an MVP that includes core functionality: basic signup/login, one payment method (Stripe) to start, basic live streaming with Twilio, and simple chat. Then iterative enhancements: add PayPal support, add more moderation tools, refine UI, etc. Outline phases such as:  
  + *MVP Prototype:* Basic end-to-end working (one streamer to one viewer test).
  + *Alpha:* Internal testing with multiple users and a friendly streamer.
  + *Beta Launch:* Invite a small group of real users/celebs, gather feedback.
  + *Public Launch:* open up subscriptions to general public.
* Each stage will have specific tasks (e.g., “Integrate Stripe webhooks” or “Implement moderator ban feature”). Having this roadmap will help prioritize development efforts and set expectations for delivery.
* **Testing & QA:** Plan for comprehensive testing. This includes unit tests for critical functions (especially around payment processing and token generation), integration testing for the end-to-end flow (maybe using staging environment with Twilio test credentials and Stripe test mode), and user acceptance testing with scenarios (sign up, watch preview, subscribe, chat, etc.). Also test on different devices and network conditions to ensure the performance and responsiveness non-functional requirements are met.