Capstone Mentored Project Proposal

ComicSeen

Platform Feature Improvements for Comedy Industry Workflows

Students	Yusra Ashar Sam Numan Shane Ritter Vivian Le
Mentor	Justin Greenberg

1. Understanding the Problem or the Need

ComicSeen is a comedy platform created to help comedians and producers manage comedy opportunities more efficiently. The platform supports open mic signups, showcases, comedian profiles, messaging, notifications, and availability management for both comedians and producers. Comedy booking and communication has historically been scattered across disconnected tools; social media groups, text messages, email chains, spreadsheets, and personal websites. This creates real operational friction for everyone in the industry.

Who Is Affected

- Comedians need a professional way to present themselves, share their availability, and stay updated on booking and event communication
- Producers need a better way to discover comedians, review their information, communicate with them, and manage shows without relying on disconnected systems
- The ComicSeen platform itself is limited in its ability to grow and retain users if these workflows remain fragmented

Consequences of Not Addressing This

- Comedians may continue using scattered links or informal communication methods with no centralized professional presence
- Producers may spend unnecessary time tracking performer information manually, reducing platform adoption
- ComicSeen risks losing relevance to informal tools like Facebook groups and spreadsheets

- Technical debt from the current tightly-coupled notification architecture will limit scalability as the platform grows

How This Project Addresses the Problem

Our project improves ComicSeen by building and enhancing four core platform features, each directly addressing a gap in the current comedy workflow:

- EPK Profile — gives comedians a professional, producer-facing profile page
- Notification System Refactor — decouples notifications from business logic using an event-driven architecture
- In-App Messaging System — enables direct communication between producers and comics inside the platform
- Calendar of Availability — allows comics to show availability and producers to find and book talent

2. Stakeholders

1) Comedians (End Users)

Comedians are a primary end user group. They stand to gain a better way to present themselves professionally, manage their availability, and communicate with producers. The EPK Profile gives comedians a dedicated place to display their bio, headshot, media links, performance clips, comedy style, and booking contact. If these features are not useful or easy to use, comedians may not adopt the platform and will continue relying on informal tools.

2) Producers (End Users)

Producers are the other primary end user group. They need an organized way to review comedians, communicate with them, and book them for shows. Features such as the availability calendar, in-app messaging, notifications, and EPK profiles allow producers to make faster, better-informed booking decisions. If these features are incomplete or confusing, producers may continue using outside tools instead of ComicSeen.

3) Justin Greenberg (Client / Mentor / Product Owner)

Justin Greenberg is the founder of ComicSeen and the project mentor. As both a comedian and a producer himself, he has direct personal experience with the problems the platform is solving. He stands to gain a stronger, more complete platform that better supports real comedy workflows and its roadmap toward subscription membership tiers.

4) Development Team and Future Contributors

The development team — Yusra Ashar, Sam Numan, Shane Ritter, and Vivian Le — are stakeholders because the work completed in this project directly shapes the product they are responsible for delivering.The architectural decisions made now will also determine how easy or difficult it is for future contributors to build on top of what we leave behind.

3. Environmental Scan

To understand where ComicSeen fits in the current landscape, we looked at existing platforms that comedians, producers, and event organizers already use. What we found is that while several tools exist for event management and entertainment booking, none of them are built specifically for the workflows that stand-up comedy requires. Most platforms are either too broad, too focused on ticketing, or simply not designed with the producer-to-comedian relationship in mind.

Existing Platforms

1) **Eventbrite**

Eventbrite is one of the most widely used event platforms in the country. It allows organizers to create events, sell tickets, and manage registrations. Some comedy producers do use Eventbrite for public-facing ticket sales, but that is essentially where its usefulness ends for the comedy industry (Eventbrite, 2024). There is no way to manage a lineup of performers, handle open mic signups, communicate with comedians, or track who is on a show. It is built for the audience side of events, not the production side. A comedian cannot create a profile, share their availability, or receive booking requests through Eventbrite. It solves a ticketing problem, not a comedy coordination problem.

2) **GigSalad**

GigSalad is closer to what the comedy industry needs because it is specifically designed for booking performers (GigSalad, 2024). It allows entertainers to create profiles and receive booking inquiries from event planners. However, GigSalad is a general marketplace built for all types of entertainment — musicians, magicians, face painters, and comedians are all listed side by side. It is not built around the specific workflows of stand-up comedy, such as open mic signups, showcase lineups, or the informal but ongoing relationship between a producer and the comics they regularly book. There is also no concept of a producer managing a recurring show or a comic building a professional presence specifically within a comedy ecosystem.

3) **Facebook Groups and Events**

Realistically, Facebook is still the most widely used tool for comedy coordination, even though it was never built for that purpose (Meta, 2024). Most open mic producers run their signups through Facebook groups, and comedians rely on these groups to find shows. The problem is that Facebook has no structure for this — signups happen through comments or DMs, there is no way to manage a lineup, notifications are unreliable, and there is no professional profile system for comedians. Everything is informal and scattered. The fact that the comedy community has adapted Facebook for this purpose is actually evidence of how much a dedicated platform is needed, because the workarounds people use are inefficient and do not scale.

4) **Meetup**

Meetup supports community event discovery and RSVPs but lacks any performer-facing features, booking infrastructure, or show management tools (Meetup, 2024). It is not used meaningfully in professional comedy contexts. It is primarily a discovery tool for attendees, not a coordination tool for the people running shows.

Industry Context

Beyond just looking at individual platforms, it is worth understanding how much the live comedy industry has grown and why that makes the need for better tooling more urgent. According to Pollstar, a leading trade publication that provides news, data, and analysis for the concert and live entertainment industry, grosses from stand-up comedy shows have nearly tripled over the last decade, generating over $900 million in 2023 alone (Pollstar, 2023). The number of comedy clubs in the United States has also grown significantly, from around 1,200 in 2000 to approximately 1,800 by 2023 (Zipdo, 2023). As more comedians enter the space and producers

run more shows, the manual coordination methods that work for a small community start to break down. The reliance on informal tools like Facebook and spreadsheets becomes a real bottleneck as volume increases. A growing industry needs infrastructure that can scale with it, and right now that infrastructure does not exist in a form built specifically for comedy (Greenberg, 2025).

How ComicSeen Fills the Gap

What is clear from this review is that there is no existing platform purpose-built for the operational side of stand-up comedy. Eventbrite handles ticketing. GigSalad handles general entertainment booking. Facebook handles informal communication. None of them handle the full workflow: a producer creating and managing shows, comics discovering and signing up for those shows, both parties communicating directly, comics maintaining a professional profile, and the whole system sending the right notifications at the right time. ComicSeen is designed to fill exactly that gap by combining all of these workflows into a single platform built specifically for the comedy community (Greenberg, 2025).

4. Prototype with Key Features

After meeting with Justin Greenberg and confirming our approach with Dr. Su, our team decided that each member would create an AI-assisted prototype for their individually assigned feature area, rather than all members prototyping the same feature. Dr. Su confirmed this approach is appropriate given our project structure. Each member documented their prompts, AI tool used, what was generated, what worked and did not work, and where human engineering judgment was required beyond the AI output. The individual prototypes are documented below.

Yusra Ashar — Notification System Refactor

AI Tool Used: Claude Code (claude.ai) for codebase analysis, architecture design, and prototype generation

Prototype Branch: notification-refactor

AI Workflow — Six Phases

The workflow followed six phases: three analysis phases to understand the existing codebase and document the problem, followed by three architecture generation phases to design and refine the solution.

Phase 1 — Codebase Structure Analysis

Prompt used:

"Look through this entire codebase and give me a summary of: the overall project structure and main packages, what technologies and frameworks are being used, how the backend is organized (controllers, services, repositories), how the frontend is organized (main components, routing)."

Phase 2 — Notification System Deep Dive

Prompt used:

"Focus only on the notification system. Find and analyze: every place in the codebase where email or notifications are sent, which services directly call notification or email logic, what notification types currently exist, what the EmailNotificationService (or equivalent) looks like, any notification preference settings and how they are checked, any notification-related classes, events, or listeners that already exist, what is clearly missing based on the code."

Phase 3 — Tight Coupling Problem Documentation

Prompt used:

"Show me specifically which business logic services are directly calling notification or email methods inline. For each one show: the service name and method name, the exact line or block where the notification call happens, what that notification is doing. This is to document the tight coupling problem before the refactor."

Phase 4 — Event-Driven Architecture Generation

Prompt used:

"I'm refactoring the notification system in a Spring Boot app called ComicSeen. Current problem: EmailNotificationService is called directly and synchronously inside ComedianShowServiceImpl (signup/drop), ProducerShowServiceImpl (show cancellation — currently a TODO), and processApproval (slot approval — currently unnotified). I want to refactor this using Spring ApplicationEvents and @EventListener so business logic no longer calls EmailNotificationService directly. Design the following: (1) A domain event class for each of these triggers: Comic signs up for open mic (ComicSignedUpEvent), Comic drops from open mic (ComicDroppedEvent), Producer cancels a show (ShowCancelledEvent), Comic's slot approval status changes (SlotApprovalEvent). (2) How ApplicationEventPublisher should be injected and used inside ComedianShowServiceImpl and ProducerShowServiceImpl to publish these events instead of calling EmailNotificationService directly. (3) A NotificationEventListener class with @EventListener methods for each event, that looks up the user's NotificationSettings, checks the relevant boolean preference before sending, and calls EmailNotificationService only if preference is true. Show me the complete code for each class."

Phase 5 — Showcase Bug Fix

Prompt used:

"In ComedianShowServiceImpl, the dropOpenMicSignup method always sends OPEN_MIC_DROP_NOTIFICATION regardless of show type. There is a SHOWCASE_DROP_NOTIFICATION enum value defined in Constants.java and a notifyShowcaseDrop preference in NotificationSettings, but they are never used. As part of the event-driven refactor, show me how the ComicDroppedEvent should include the show type, and how the NotificationEventListener should use it to send the correct notification type and check the correct preference flag."

Phase 6 — Async Resilience

Prompt used:

"Currently all EmailNotificationService calls are synchronous and blocking. If Resend API is down, the comic's signup transaction fails. Show me how to add @Async to the NotificationEventListener so email sending is non-blocking, including: the @Async annotation on the listener methods, the @EnableAsync configuration needed, why this is safer than the current inline synchronous calls."

Problems Found in the Existing Codebase

The codebase analysis revealed five specific tight coupling violations:

- AuthService.registerNewLocalUser (line 118) — email verification sent inline inside @Transactional registration, blocking the response thread on Resend's HTTP call
- AuthService.resendVerificationEmail (line 54) — entire method body is a single email call with no other business logic

- PasswordResetService.sendResetEmail (line 55) — email sent immediately after DB token write in the same @Transactional method, with a hardcoded production URL baked into the service
- ComedianShowServiceImpl.signupForOpenMic (line 62) — fires OPEN_MIC_SIGNUP_NOTIFICATION unconditionally after the DB write regardless of approval status or show type
- ComedianShowServiceImpl.dropOpenMicSignup (line 70) — always fires OPEN_MIC_DROP_NOTIFICATION even for Showcase drops, meaning SHOWCASE_DROP_NOTIFICATION and the notifyShowcaseDrop preference are permanently dead code

What the AI Generated — Event-Driven Architecture

Claude Code generated a complete event-driven refactor including four domain event record classes, a refactored ComedianShowServiceImpl using ApplicationEventPublisher, a refactored ProducerShowServiceImpl resolving the long-standing show cancellation TODO, a NotificationEventListener with @TransactionalEventListener(phase = AFTER_COMMIT) handlers for each event, and new EmailNotificationService methods for each notification type. The @TransactionalEventListener(phase = AFTER_COMMIT) annotation ensures no email fires if the database transaction rolls back. The REQUIRES_NEW propagation gives each listener its own transaction for the repository reads it performs. This means a failed signup produces no misleading confirmation email — a guarantee the current inline design cannot provide.

Showcase Bug Fix

The AI identified and fixed the SHOWCASE_DROP_NOTIFICATION bug by adding a showType field to ComicDroppedEvent. The refactored dropOpenMicSignup method now fetches the show before dropping and passes show.getShowType() into the event. The NotificationEventListener then branches on showType at a single point to both select the correct preference flag (notifyShowcaseDrop vs notifyOpenMicDrop) and call the correct email method — ensuring the two decisions can never get out of sync.

Async Resilience

Claude Code generated an AsyncConfig class with @EnableAsync and a named ThreadPoolTaskExecutor (2 core threads, 5 max, queue capacity 100) scoped specifically to notification work. Adding @Async to each listener method moves email sending entirely off the HTTP request thread. Under the current design, if Resend is slow or down, the comic's signup response is delayed by the full HTTP timeout. After the refactor, the signup returns a 200 immediately after the transaction commits, and any Resend failure is contained in the background notification thread with no impact on the user-facing response.

Critical Evaluation of AI Output

Claude Code performed well for codebase analysis, identifying bugs and architectural gaps that would have taken significantly longer to find manually. The generated architecture was technically sound and consistent with the existing project structure. However, several areas required human engineering judgment:

- AI suggested using @TransactionalEventListener but did not explain the implications — human judgment was needed to understand that plain @EventListener would fire even on rolled-back transactions, meaning a failed signup would still send a confirmation email

- AI assumed findByUserId exists on NotificationSettingsRepository — this needs to be verified against the actual repository interface before implementation
- AI generated new EmailNotificationService method signatures that need to be reviewed for compatibility with the existing Resend HTML email template structure
- AI occasionally proposed broader changes than necessary — human judgment was needed to keep the scope focused on what Justin specified

Human Value Added Beyond AI

AI tools can analyze code and generate architecture, but several critical decisions required human judgment that the AI could not make on its own:

- Scoping to Spring EventListeners instead of Kafka — Justin's dev plan specifically said to scope to EventListeners for a single running app instance, with Kafka as a long-term goal. AI would not have known this constraint without being told. The decision to use Spring ApplicationEvents specifically was made based on Justin's direction, not AI's suggestion
- Identifying the SHOWCASE_DROP_NOTIFICATION bug — this required actually reading the codebase and understanding that SHOWCASE_DROP_NOTIFICATION was defined in the enum, handled in EmailNotificationService, and stored as a preference, but never triggered anywhere. AI identified the bug after being directed to look at the coupling map, but a human had to recognize that this was a real production bug worth fixing
- Deciding which callers to refactor and which to leave alone — AuthService and PasswordResetService also call EmailNotificationService directly, but scoping the

refactor to ComedianShowServiceImpl and ProducerShowServiceImpl first was a deliberate human decision to keep the change minimal and safe for a production codebase

- Validating the @Async and @TransactionalEventListener interaction — AI generated the annotation combination but understanding why the order matters (AFTER_COMMIT guarantee must apply inside the async thread, not on the original request thread) required reading Spring documentation and reasoning about transaction propagation
- Understanding the production impact — the existing sendEmail method already swallows ResendException silently, but the HTTP thread still blocks on the Resend timeout before returning. Recognizing this as a real user-facing latency problem, not just a theoretical concern, required understanding how the production signup flow actually behaves

Testing Plan

- Unit test NotificationEventListener.handleComicSignedUp — mock notificationSettingsRepository to return notifyOpenMicSignup=false and verify emailNotificationService is never called
- Unit test handleComicDropped with showType='Showcase' — verify notifyShowcaseDrop is checked and sendShowcaseDropEmail is called, not sendOpenMicDropEmail
- Unit test handleComicDropped with showType='Open Mic' — verify notifyOpenMicDrop is checked
- Integration test the full signup flow — POST signup → transaction commits → AFTER_COMMIT fires → email sent
- Integration test that a rolled-back signup (e.g. duplicate slot) produces no email

- Verify @Async behavior — confirm signup endpoint returns 200 before email is sent

Planned Improvements Beyond the Prototype

- Implement the notifyReminders workflow — the preference exists in NotificationSettings but the reminder scheduler does not exist yet
- Enable comic-facing notification preferences in the frontend — the UI currently renders 'Coming Soon' for all comic notification settings
- Add notifyShowCancelled preference to NotificationSettings — show cancellation notifications currently have no preference flag
- Stretch goal: SMS notification channel via the same event-listener architecture

The following screenshots document the current state of the notification system before the refactor. Figure 1 shows the signupForOpenMic method in ComedianShowServiceImpl, where emailNotificationService.sendEmailNotification is called directly and synchronously as the final line of the business method:

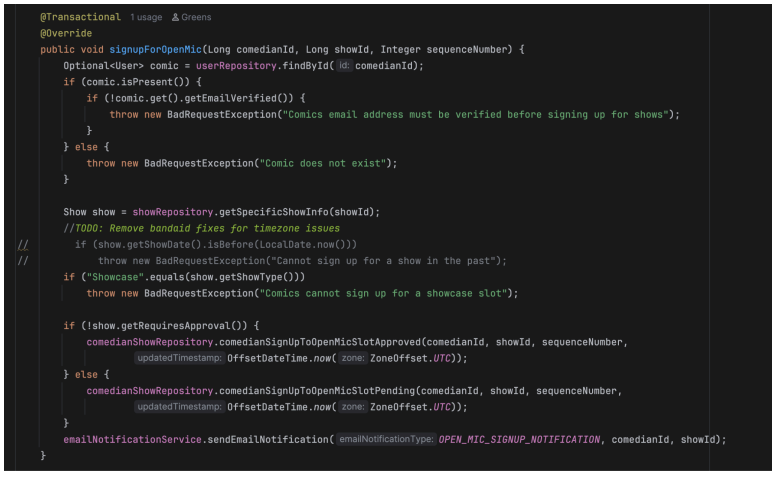


Figure 2 shows the dropOpenMicSignup method. The entire method body is a single repository call followed immediately by an inline email notification call, with no show type check:

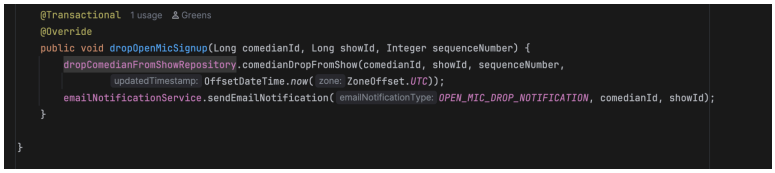


Figure 3 shows the EmailNotificationService class structure, confirming it is a concrete @Service class directly injected into all callers with no interface abstraction:

The following screenshots show the AI-generated event-driven architecture.

Figure 4 shows the new domain event record classes generated by Claude Code, including ComicSignedUpEvent and ComicDroppedEvent:

1. Domain Event Records

New package: `com.openmicservice.demo.account.notifications.events`

../open-mic-service/src/main/java/com/openmicservice/demo/account/notifications/events/ComicSignedUpEvent.java

```
package com.openmicservice.demo.account.notifications.events;

import java.time.LocalDate;

public record ComicSignedUpEvent(
        Long comedianId,
        Long showId,
        String showName,
        LocalDate showDate
) {}
```

../open-mic-service/src/main/java/com/openmicservice/demo/account/notifications/events/ComicDroppedEvent.java

```
package com.openmicservice.demo.account.notifications.events;

import java.time.LocalDate;

public record ComicDroppedEvent(
        Long comedianId,
        Long showId,
        String showName,
        LocalDate showDate,
        String showType   // "Open Mic" or "Showcase" — drives which preference flag is checked
) {}
```

Figure 5 shows the generated NotificationEventListener class with its dependencies, confirming EmailNotificationService is now only referenced inside the listener and no longer injected into business services:

```
public class NotificationEventListener {

    private final NotificationSettingsRepository notificationSettingsRepository;
    private final UserRepository userRepository;
    private final ShowRepository showRepository;
    private final ShowTemplateRepository showTemplateRepository;
    private final EmailNotificationService emailNotificationService;

    // AFTER_COMMIT ensures no email fires if the DB transaction rolls back.
    // REQUIRES_NEW gives the listener its own transaction for the repository reads below.

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void handleComicSignedUp(ComicSignedUpEvent event) {
        Long producerId = resolveProducerId(event.showId());
        if (producerId == null) return;

        NotificationSettings settings = notificationSettingsRepository.findByUserId(producerId);
        if (!settings.isNotifyOpenMicSignup()) return;

        User producer = userRepository.findById(producerId).orElse(null);
        User comedian = userRepository.findById(event.comedianId()).orElse(null);
        if (producer == null || comedian == null) return;

        emailNotificationService.sendOpenMicSignupEmail(
                producer.getEmail(),
                comedian.getFirstName() + " " + comedian.getLastName(),
                event.showName(),
                event.showDate()
        );
    }
```

Figures 6, 7, and 8 document the SHOWCASE_DROP_NOTIFICATION bug fix.

Figure 6 shows the AI identifying the root cause — the show is never fetched in

dropOpenMicSignup, making showType permanently unknown and SHOWCASE_DROP_NOTIFICATION permanently dead code:

The Bug — Current `dropOpenMicSignup`

../open-mic-service/src/main/java/com/openmicservice/demo/shows/comic/ComedianShowServiceImpl.java

```
@Transactional
@Override
public void dropOpenMicSignup(Long comedianId, Long showId, Integer sequenceNumber) {
    dropComedianFromShowRepository.comedianDropFromShow(comedianId, showId, sequenceNumber,
            OffsetDateTime.now(ZoneOffset.UTC));
    // Always OPEN_MIC_DROP — show type is never checked, showType never fetched
    emailNotificationService.sendEmailNotification(OPEN_MIC_DROP_NOTIFICATION, comedianId, showId);
}
```

The show is never fetched here, so `showType` is never known. `SHOWCASE_DROP_NOTIFICATION` and `notifyShowcaseDrop` are permanently dead code under this design.

Figure 7 shows the fix: ComicDroppedEvent now carries a showType field, and dropOpenMicSignup fetches the show before the drop to populate it:

Step 1 — `ComicDroppedEvent` carries `showType`

```
public record ComicDroppedEvent(
        Long comedianId,
        Long showId,
        String showName,
        LocalDate showDate,
        String showType   // "Open Mic" or "Showcase"
) {}
```

`showType` is the only field that distinguishes these two notification paths. Everything else (`showName`, `showDate`) is denormalized into the event at publish time to avoid redundant DB lookups in the listener.

Step 2 — `dropOpenMicSignup` fetches the show and publishes with `showType`

```
@Transactional
@Override
public void dropOpenMicSignup(Long comedianId, Long showId, Integer sequenceNumber) {
    // Fetch before the drop — gives us showType, showName, showDate for the event
    Show show = showRepository.getSpecificShowInfo(showId);

    dropComedianFromShowRepository.comedianDropFromShow(comedianId, showId, sequenceNumber,
            OffsetDateTime.now(ZoneOffset.UTC));

    eventPublisher.publishEvent(new ComicDroppedEvent(
            comedianId,
            showId,
            show.getShowName(),
            show.getShowDate(),
            show.getShowType()   // "Open Mic" or "Showcase" — now carried into the event
    ));
}
```

`getSpecificShowInfo` is already called in `signupForOpenMic` on the same class, so this pattern is consistent with existing usage.

Figure 8 shows the listener branching on showType at a single point to both select the correct preference flag and call the correct email method:

Step 3 — Listener branches on `showType` for both preference check and send

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void handleComicDropped(ComicDroppedEvent event) {
    Long producerId = resolveProducerId(event.showId());
    if (producerId == null) return;

    NotificationSettings settings = notificationSettingsRepository.findByUserId(producerId);

    // Single branch point — showType drives both which preference to check
    // and which email method to call
    boolean shouldNotify = "Showcase".equals(event.showType())
            ? settings.isNotifyShowcaseDrop()
            : settings.isNotifyOpenMicDrop();

    if (!shouldNotify) return;

    User producer = userRepository.findById(producerId).orElse(null);
    User comedian = userRepository.findById(event.comedianId()).orElse(null);
    if (producer == null || comedian == null) return;

    String comicName = comedian.getFirstName() + " " + comedian.getLastName();

    if ("Showcase".equals(event.showType())) {
        emailNotificationService.sendShowcaseDropEmail(
                producer.getEmail(), comicName, event.showName(), event.showDate());
    } else {
        emailNotificationService.sendOpenMicDropEmail(
                producer.getEmail(), comicName, event.showName(), event.showDate());
    }
}
```

Figures 9 and 10 document the async resilience addition. Figure 9 shows the generated AsyncConfig class with @EnableAsync and a named ThreadPoolTaskExecutor scoped to notification work:

1. Async Configuration

New file: `../open-mic-service/src/main/java/com/openmicservice/demo/config/AsyncConfig.java`

```
package com.openmicservice.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import java.util.concurrent.Executor;

@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "notificationExecutor")
    public Executor notificationExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(5);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("notification-");
        // If the queue fills up and max threads are busy, the calling thread runs the
        // task itself rather than dropping it silently
        executor.setRejectedExecutionHandler(new java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }
}
```

`@EnableAsync` must live on a `@Configuration` class — putting it on the application main class works but mixes concerns. `AsyncConfig` keeps it isolated alongside `SecurityConfig`, `WebSocketConfig`, and `S3Config` which is consistent with how this project is already organized.

`SimpleAsyncTaskExecutor` is Spring's default when you add `@Async` without a named bean — it creates a new thread per invocation with no pooling, no queue, and no backpressure. The named `ThreadPoolTaskExecutor` above gives you bounded concurrency and a queue so a spike in signups doesn't spawn unbounded threads.

Figure 10 shows the NotificationEventListener with @Async added to each handler method, moving email sending entirely off the HTTP request thread:

```java
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.event.TransactionPhase;
import org.springframework.transaction.event.TransactionalEventListener;

@Slf4j
@Component
@RequiredArgsConstructor
public class NotificationEventListener {

    private final NotificationSettingsRepository notificationSettingsRepository;
    private final UserRepository userRepository;
    private final ShowRepository showRepository;
    private final ShowTemplateRepository showTemplateRepository;
    private final EmailNotificationService emailNotificationService;

    @Async("notificationExecutor")
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void handleComicSignedUp(ComicSignedUpEvent event) {
        Long producerId = resolveProducerId(event.showId());
        if (producerId == null) return;

        NotificationSettings settings = notificationSettingsRepository.findByUserId(producerId);
        if (!settings.isNotifyOpenMicSignup()) return;

        User producer = userRepository.findById(producerId).orElse(null);
        User comedian = userRepository.findById(event.comedianId()).orElse(null);
        if (producer == null || comedian == null) return;

        emailNotificationService.sendOpenMicSignupEmail(
                producer.getEmail(),
                comedian.getFirstName() + " " + comedian.getLastName(),
                event.showName(),
                event.showDate()
        );
    }
```

Sam Numan — In-App Messaging System

AI Tools Used: Claude (claude.ai) for troubleshooting, code analysis, and organization; Codex in VS Code (GPT-5.5 Extra High, Default Permissions) for codebase understanding and prototype generation

Prototype Branch: messaging-prototype

AI Workflow — Three Phases

Sam followed a three-phase AI-assisted workflow to analyze the existing codebase before generating prototype improvements.

Phase 1 — Repository Understanding

Prompt used:

"Analyze the overall project structure and give me a comprehensive overview of: the folder structure and what each directory is responsible for, the main technologies and libraries being used (check package.json), how routing is set up in App.js, how authentication and the current user are managed and shared across components, how API calls are made (look at src/util/), any shared constants or configuration values I should be aware of."

Phase 2 — Messaging System Analysis

Prompt used:

"Now focus specifically on the messaging system in this codebase. Analyze src/messaging/DirectMessage.js and src/messaging/DirectMessage.css and give me a detailed explanation of: the overall component structure and how the three components (DirectMessage, SubscribingComponent, and SendingMessages) relate to each other, how the WebSocket/STOMP connection is established and managed using react-stomp-hooks, how chatrooms are loaded, selected, and displayed in the sidebar, how message history is fetched from the REST API, how new messages are sent and received in real time, how a new conversation is started via the user search modal, what the current UI looks like, any known bugs, TODOs, or limitations you can identify in the code."

Phase 3 — Prototype Generation

Prompt used:

"I need to build a prototype improvement of the messaging UI for ComicSeen. The goals are: (1) Fix the scroll-to-bottom behavior so the chat window automatically scrolls to the latest message when new messages arrive or when a new chatroom is selected. (2) Redesign the messaging UI to be more modern and polished while keeping the existing functionality intact. The new design should have a cleaner interface similar to iMessage or WhatsApp, show sender initials avatars, display timestamps, improve sidebar visual hierarchy, use ComicSeen's existing purple brand color (#662c91 / #831fc1), keep all existing WebSocket/STOMP logic completely intact, and remain responsive. Make all changes in DirectMessage.js and DirectMessage.css only. Do not modify App.js, any backend files, or any other frontend files."

Key Features — Two Improvements

1) Scroll-to-Bottom Fix:

The original implementation only attempted to scroll to the latest message on initial component mount, meaning new incoming messages did not automatically scroll the chat window. The fix adds correct useEffect dependencies so scrolling fires whenever the message list updates or a new chatroom is selected.

2) UI Modernization:

The original design used basic React Bootstrap styling with plain gray and purple chat bubbles. The redesigned interface introduces improved visual hierarchy in the chat sidebar, sender initials avatars, message timestamps, and a more polished chat bubble design using ComicSeen brand colors (#662c91 / #831fc1).

Human Contributions Beyond AI Output

- Identified that the /messages route was commented out in App.js and made the decision to re-enable it on the prototype branch
- Diagnosed and resolved multiple local environment issues independently: backend H2/MySQL profile conflict, missing NOT NULL column defaults, email verification bypass, HERE API geocoding errors, and incorrect WebSocket URL configuration
- Wrote SQL scripts to create realistic test users with correct role assignments (COMIC, PRODUCER) including all required related records
- Made the deliberate architectural decision to scope the prototype to UI-layer changes only, keeping all WebSocket/STOMP backend logic untouched to avoid regressions
- Identified that ChatNotification.java exists in the backend but is not wired up, flagging it as a gap for future email alert integration with Yusra's notification work

Technical Challenges and Limitations Identified

- Chat list query only returns conversations where the user is the sender — recipients do not see incoming conversations until they send a reply
- ChatNotification class is defined but never used — email alerts on message receipt are not implemented
- No read/unread message tracking exists
- No role enforcement — any user can message any other user regardless of role (Comic to Comic not currently restricted, per Justin's plan it should be Producer to Comic only)
- created_timestamp field in ChatMessage uses Long instead of LocalDateTime, causing inconsistent time handling

Critical Evaluation of AI Output

Claude was highly effective for troubleshooting environment setup errors and explaining the existing WebSocket/STOMP connection flow. Codex significantly accelerated the analysis phase by identifying bugs and gaps that would have taken much longer to find manually. AI worked less well for resolving issues requiring external API keys (HERE geocoding), occasionally suggested changes outside the scoped files requiring human judgment to reject, and generated UI code that required manual review to ensure props, state references, and useEffect dependencies matched the original component structure exactly.

Planned Improvements Beyond the Prototype

- Restrict messaging to Producer to Comic only via backend role validation
- Fix the chat list query to surface conversations for both senders and recipients
- Wire up ChatNotification to trigger email alerts via the Resend API (coordinating with Yusra's notifications work)
- Add read/unread message status tracking

Testing Plan

- Unit test ChatMessageService.save() to verify chatId is correctly assigned before persistence
- Unit test ChatRoomService.getChatRoomId() to confirm bidirectional room creation
- Integration test the full message send flow: POST via WebSocket → save to DB → broadcast to subscriber

- Frontend manual testing: verify scroll behavior on new message arrival, chatroom switching, and page refresh
- Role enforcement testing: attempt Comic to Comic messaging and confirm it is rejected once restriction is implemented
- Cross-browser WebSocket compatibility testing

The following screenshots document Sam's messaging prototype. Figure 1 shows the original messaging UI before the prototype improvements:

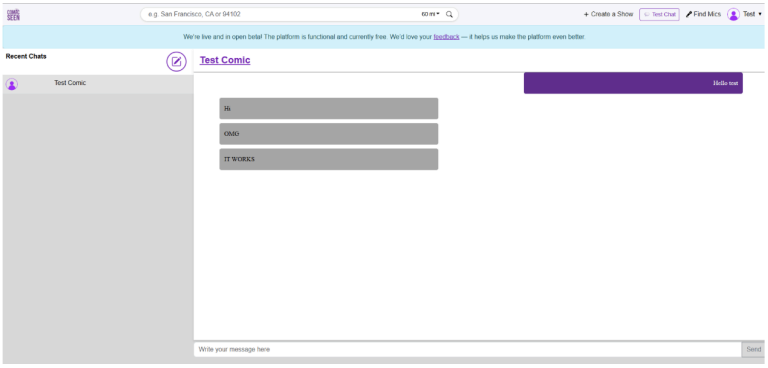


Figure 2 shows the modernized messaging UI after the prototype improvements, including sender initials avatars, timestamps, and improved sidebar hierarchy:

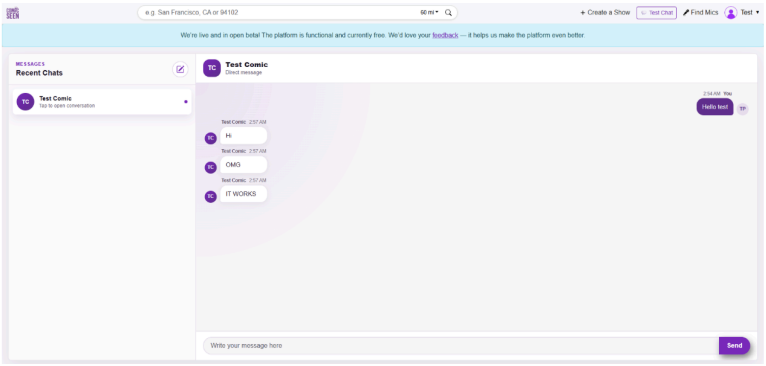


Figure 3 shows the New Message modal allowing a user to search for and initiate a conversation:

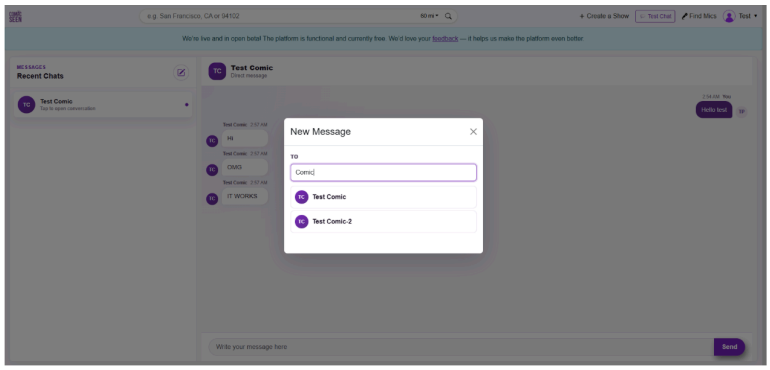
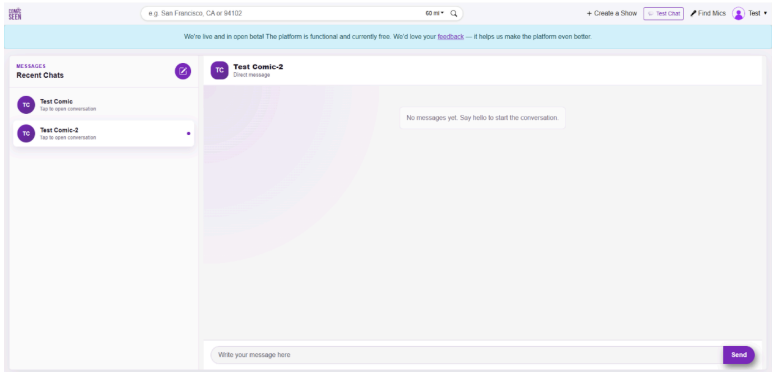


Figure 4 shows a second chatroom selected from the sidebar with the empty state message:



Shane Ritter — Calendar of Availability

AI Tool Used: Codex (GPT-5.5, High Intelligence, Default Permissions)

Prototype Branch: calendar-prototype

What Is Being Built

The Calendar of Availability feature addresses a significant gap in the current platform: there is currently no way for comedians to communicate when they are available, and no way for

producers to search for or filter comedians by availability. Per Justin's project plan, this feature is implemented in two phases. Phase 1 focuses on the comic-facing calendar implementation. Phase 2 introduces producer-facing booking request workflows.

Key Features Planned

- Comic availability calendar UI — mark available dates, times, and general recurring availability
- Seeking bookings indicator that surfaces a comic in producer search results
- Producer search and filter by comic availability and location
- Booking request workflow — producer sends a request, comic receives a notification and can accept or decline
- Integration with the notification system for booking request alerts
- Last-minute spot filling — producers can flag urgency, comics can opt in to last-minute alerts

AI Workflow

Prompt used:

“Create a calendar that has a management system. This system will have two users / phases. The first user will be a comic. They will need to mark their availability, and what times they are booked and where on the calendar. The second user is a producer that can search up a comic to seek bookings. The producer can search and filter by comic availability and location. There will be an alert / notification system. This system will work by having the producer send a request, which the comic receives a notification and can accept or decline. Producers can flag urgency,

comics can opt in to last-minute alerts. Write this code with the front-end using the React framework and the back-end using the Java coding language.”

Critical Evaluation of AI Output

The output given is a useful starting point on where to start the full implementation into the Comic Seen codebase. Codex was highly effective in creating most of the systems that it was instructed to create in a simple and modern layout that is easy to understand. Human judgement will have to come in the form of evaluating generated code including tradeoffs between existing and generated code, inserting sensitive data including API keys, evaluating that the code fits with the existing SQL database and codebase, and to change suggestions to keep the project more in line with what is requested.

Planned Improvements Beyond the Prototype

- Turn calendar into a month format instead of only displaying the next five days
- Color code booking to easily display different comics or bookings
- Remove toggle for user and have it depend on user’s role after they sign in
- Fix “Failed to fetch” bug displayed in Figure 1, which is caused by not fetching SQL database correctly
- Fully integrate with preexisting notification system instead of creating one
- Fit CSS styling into the preexisting Comic Seen styling

Human Value Added Beyond AI

Shane gave additional context to the AI program of the different users of the Comic Seen website, and to how each user will interact with the feature that he is implementing. Furthermore, Shane evaluated not only if the generated code could properly fit into the existing codebase, but also the inconsistencies between the output of the prototype and the desires of the founder, Justin Greenberg. With those inconsistencies in mind, Shane can pivot his implementation or create additional prototypes that address the issues. Shane will also have to make sure that the variables, routes, etc created in the generated code matches what is already preestablished within the Comic Seen codebase.

Testing Plan

Shane plans to create unit tests for the following before starting development:

- Back-end
    - Calendar returns availability and booking events
    - Producers can filter comics by all following criteria / filters
    - Valid availability saves correctly
    - Valid manual booking saves and overlapping booking is rejected
    - Producer can create a request
    - Comic accepting or denying a request
- Front-end
    - Calendar is rendered correctly
    - Forms are rendered correctly
    - Availability and booking form submits while calendar refreshes

- Results of producer form renders comic stage names, locations, tags, and availability

The following screenshots document Shane's availability calendar prototype.

Figure 1 displays the simple, sleek, and modern UI design of the prototype through the view of the comic. It displays the current day and following four days on the calendar, the name of the profile of whom is logged in, and the notifications of the comic.

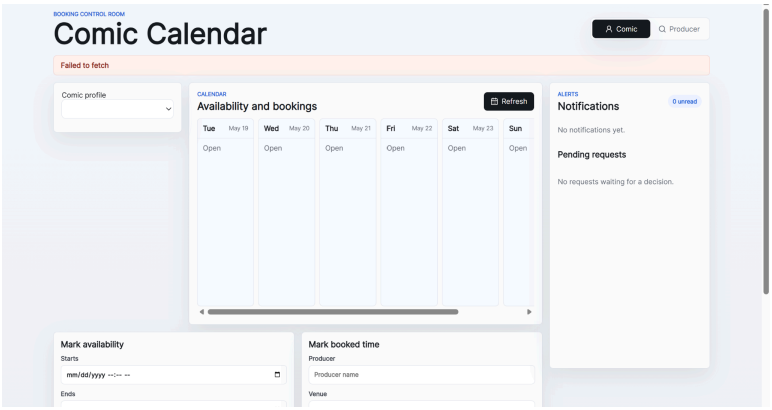


Figure 2 is a continuation of the comic view displaying features, such as a section where the comic can enter their availability into the calendar and a section where the comic can enter their bookings into the calendar.

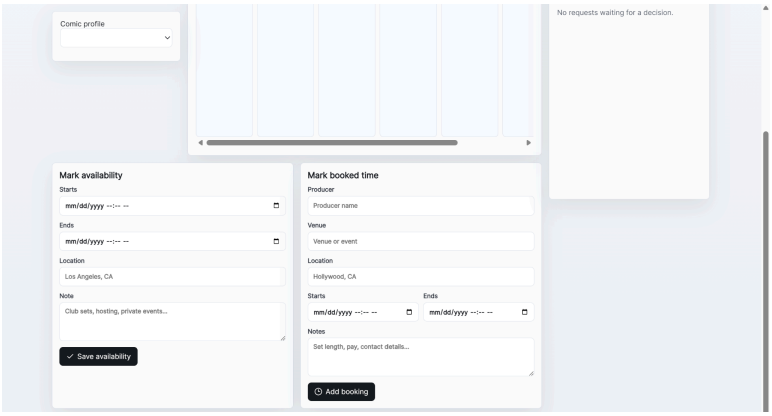
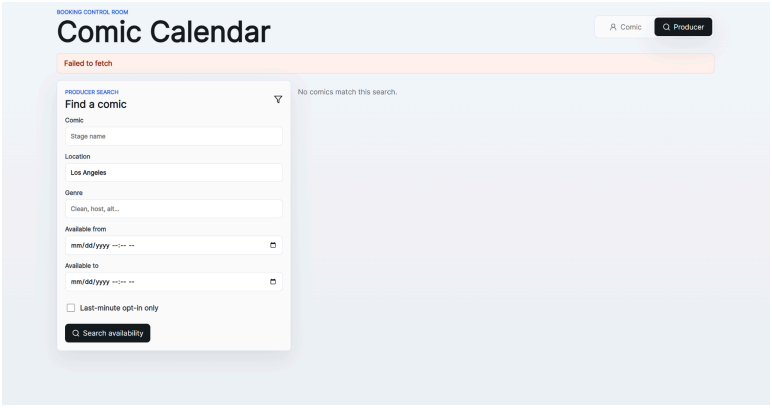


Figure 3 displays the simple, sleek, and modern UI design of the prototype through the view of the producer. Features include the search engine where they can search for a specific comic or comic near them. Search results display on the right side of the screen.



Vivian Le — EPK Profile

AI Tool Used: Codex and ChatGPT

Prototype Link/Screenshot: Screenshots included below.

What Was Built

Vivian's assigned feature was the EPK Profile. EPK stands for Electronic Press Kit. The purpose of this feature is to give comedians a professional public profile that producers can review when deciding whether to book a comedian for a show, showcase, open mic, or other comedy opportunity. The profile needs to feel professional enough to work like a performer version of LinkedIn, while still matching the comedy industry and showing the comedian's personality.

AI Prompts Used

For this prototype, Vivian used Codex and ChatGPT to help develop an improved EPK Profile layout and workflow. AI was used to generate design ideas, improve the structure of the profile, identify useful producer-facing information, and create example HTML prototypes for both the public EPK Profile and the Edit EPK form. The goal was not only to make the page look better, but also to make it more useful for the real decision-making process producers go through when deciding whether a comic is a good fit for a spot. An important note is that an official design will be provided to Vivian for implementation, and this prototype serves as an early exploration of ideas for both design and code.

Key Features Prototyped

The EPK Profile prototype included a profile header with the comedian's name, headshot, location, and short tagline. It also included a featured performance clip, short producer summary, comedy style tags, experience information, social media links, and a booking/contact action. The layout was organized so producers could quickly scan the profile and understand the comedian's background, performance style, experience level, and potential fit for a show.

The prototype also included an improved Edit EPK form where comedians can update their own profile information. This is important because a comic's EPK needs to stay current. A comedian may need to update their bio, add a stronger clip, change social media links, update booking contact information, or add recent performance experience. The form was redesigned to collect more useful information than the original form, including performance location, years performing, comedy style, available set lengths, performance credits, audience-facing description, and booking email. The form also included helper text explaining why each field matters, so comedians understand how completing the information can improve their chances of being reviewed or booked by producers.

The prototype screenshots show both the mobile-focused public EPK Profile and the updated Edit EPK form. The public profile screenshots show the profile header, featured clip, producer quick-view information, about section, comedy experience, social links, and booking call-to-action. The Edit EPK screenshots show the improved mobile form layout, profile basics section, featured clip section, performance details, and contact/social link fields.

Critical Evaluation of AI Output

Using AI to design and prototype the EPK Profile was helpful, but the AI output still required human judgment and technical direction. The first AI-generated ideas were generally useful, but they were not specific enough to the actual ComicSeen product, the expected user behavior, or the needs of comedy producers.

One important example was the mobile-focused design. The existing ComicSeen interface was already designed with mobile users in mind, but the AI did not know that context on its own. Without being told, AI may generate a layout that looks polished but does not fully match how

users are expected to interact with the site. After that context was added, the prototype became more useful because it focused on mobile scanning, shorter sections, and easy access to the most important actions.

AI was also useful for suggesting additional fields, but those suggestions still needed to be reviewed. Some fields were helpful because they directly supported producer decision-making, such as comedy style, set length, location, performance clip, and booking contact. Other possible ideas had to be considered carefully because adding more fields can also make the form feel too long or create extra implementation work. This showed that AI can generate options quickly, but a person still needs to decide which options are realistic, useful, and worth including.

There are also technical concerns that AI may not handle correctly without engineering review. The generated prototype would still need to be checked against the existing ComicSeen codebase. For example, the final implementation would need to follow the project's React component structure, styling conventions, routing patterns, form validation approach, backend data model, and authentication flow. AI may generate code that looks good visually but does not fit the existing architecture, duplicates logic, or creates fields that are not yet supported by the backend.

Because of this, the prototype should be treated as a design and workflow guide, not final production code. It helps show what the EPK Profile could become, but the final feature would still require engineering decisions, integration work, testing, and review.

Human Value Added Beyond AI

AI can generate a useful starting point, but human input is still needed to make the prototype realistic, useful, and aligned with the actual product. A person working on the feature can add

value by giving the AI important project context, such as who the users are, how they are expected to use the site, what the existing product already looks like, and what information producers actually need when reviewing a comic.

Human judgment is also needed to decide which AI suggestions are practical and which ones should be changed or removed. For example, AI may suggest a layout or extra fields that look useful, but those ideas still need to be checked against the existing codebase, backend data model, timeline, and overall feature scope. An engineer can add value by turning the prototype into something that is technically feasible, maintainable, testable, and consistent with the rest of the application.

Evaluation Plan

The EPK Profile prototype can be evaluated by testing whether it helps producers understand a comic quickly and clearly. A producer should be able to scan the profile on a mobile device and understand the comic's location, comedy style, experience level, available set lengths, and booking fit within a short amount of time. The booking or contact action should also be easy to find without requiring the user to search through the whole page. Since the site is expected to be used mainly on mobile devices, the profile and edit form should be readable, organized, and usable on a phone screen without horizontal scrolling or crowded sections.

The Edit EPK form can also be evaluated by checking whether a comedian can complete the main EPK fields without confusion. The helper text should make it clear why each field matters and how that information can help producers evaluate the comic. From a system perspective, the feature should allow a comic to edit and save EPK information, then view the updated public profile with the new information displayed correctly. If possible, the team can also ask the

mentor or a producer to review the prototype and provide feedback on whether the profile gives enough information to consider booking the comic.

The following screenshots show Vivian's EPK Profile prototype. Figure 1 shows the public EPK Profile view designed for producer review on mobile:

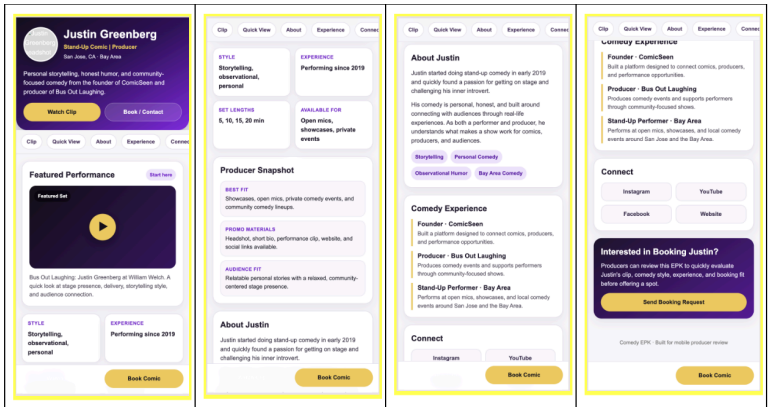
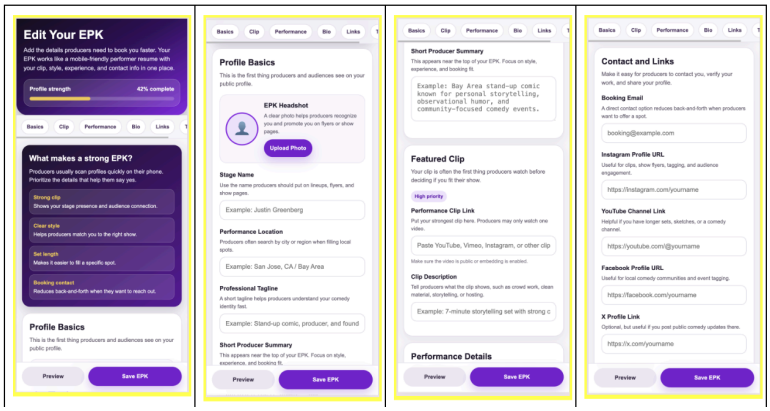


Figure 2 shows the redesigned Edit EPK form with tabbed sections and helper text for each field:



Overall

After reviewing all four prototypes as a team, we found that each one demonstrated a different strength of AI-assisted development. Yusra's notification refactor showed the deepest codebase

analysis and architectural planning. Sam's messaging prototype showed the clearest before-and-after UI improvement. Shane's calendar prototype showed a full system generated from a single prompt. Vivian's EPK prototype showed the strongest design and user experience thinking. Rather than selecting one as the single best, we determined that each prototype best represents its own feature area and that combining all four gives the most complete picture of the platform improvements we are building. Additionally, we found the different strengths and weaknesses between different AI models on how it generated and debugged code as stated in the team's evaluations.

5. Individual's Intended Contribution to the Project

Yusra Ashar

Yusra will focus on the Notification System Refactor. Her core contribution is the architectural decision-making involved in moving from a tightly-coupled notification model to an event-driven system using Spring ApplicationEvents. This requires understanding how business logic and notification workflows currently interact, identifying where decoupling needs to happen, and designing a handler architecture that is extensible for future channels. This work requires engineering judgment that AI tools alone cannot provide, specifically the decisions about transaction boundaries, async behavior, and preference checking logic that must be validated against the real codebase before implementation.

Sam Numan

Sam will focus on the In-App Messaging System. His contribution includes deep analysis of the existing WebSocket/STOMP infrastructure, identifying architectural gaps including role

enforcement, bidirectional conversation visibility, and email alert integration, and making scoped implementation decisions that improve the system without introducing regressions. Sam brings practical environment troubleshooting, SQL data scripting, and the judgment to stay within a defined implementation scope when AI tools suggest broader changes.

Shane Ritter

Shane will focus on the Calendar of Availability. His contributions include designing a calendar UI workflow that is genuinely useful for both comics and producers, implementing the Phase 1 comic availability system, and planning the Phase 2 producer booking request workflow. This requires understanding how availability data should be structured, stored, and surfaced in search results, and how the feature integrates with the notification system for booking alerts.

Vivian Le

Vivian will focus on the EPK Profile feature. She will review the current codebase, understand how profile information should be displayed and edited, implement or improve the profile display page and edit flow, and ensure the feature matches the official design provided by Justin. Her contribution includes thinking through the user experience for both comedians and producers and making implementation decisions about which prototype ideas are technically feasible within the existing React architecture.

6. Final Deliverables

- EPK Profile — comedian profile page with bio, headshot, media links, performance clips, comedy style, and booking contact, implemented to match Justin's official design

- Notification System Refactor — event-driven architecture using Spring ApplicationEvents decoupling notifications from business logic, supporting signup confirmations, drop notifications, producer updates, and cancellation alerts
- In-App Messaging Improvements — UI modernization, scroll-to-bottom fix, role enforcement (Producer to Comic only), bidirectional conversation visibility, and email alert integration
- Calendar of Availability — Phase 1 comic availability calendar with seeking-bookings indicator; Phase 2 producer booking request workflow with notification integration
- GitHub pull requests and code contributions submitted through the team's GitHub workflow for Justin's review
- Prototype screenshots and AI tool documentation for each member's feature area
- Testing documentation — unit tests, integration tests, and usability testing notes
- Final demo and documentation — walkthrough of completed features with written documentation
- Mentor approval — final feedback and sign-off from Justin Greenberg

7. Approach / Methodology

The team will use an Agile development process because this project involves an existing real-world production codebase and requirements may evolve as the team learns more about the platform. Agile allows the team to work in smaller, reviewable increments, receive feedback from Justin between iterations, and adjust scope based on what is discovered during implementation.

Development Process

- Step 1 — Codebase review and feature scoping: each member reviews the existing ComicSeen code related to their assigned feature, identifying integration points, existing infrastructure, and gaps
- Step 2 — AI-assisted prototyping: each member uses AI tools to generate design ideas, prototype UI, and analyze existing code, evaluating outputs critically with human engineering judgment
- Step 3 — Prototype review: the team reviews each prototype together for consistency, feasibility, and potential integration conflicts
- Step 4 — Implementation: each member implements their assigned feature on dedicated branches, communicating via Discord and GitHub
- Step 5 — Testing and debugging: features are tested individually and as part of the integrated platform
- Step 6 — Final documentation and demo: each member documents their feature and the team prepares a final walkthrough for Justin's review and approval

Coordination

The team communicates asynchronously through Discord and coordinates code through GitHub branching and pull requests. Weekly check-ins with Justin provide mentor feedback throughout the development process.

8. Timeline / Resources

Timeline	Milestone	Work Description	Assigned To

Weeks 1–2	Team planning and project understanding	Meet with Justin, review scope, confirm feature ownership, understand ComicSeen platform and existing codebase.	Entire Team
Week 3	AI-assisted prototyping	Each team member creates an AI-assisted prototype for their assigned feature area.	Entire Team
Weeks 4–5	Prototype feedback and feature planning	Review prototype feedback, identify changes before implementation, confirm design direction with Justin. For the EPK Profile, this includes reviewing design mockups, profile form fields, and prototype feedback.	Entire Team
Weeks 6–7	Codebase review and feature setup	Each member reviews the codebase related to their feature and identifies integration points. For the EPK Profile, this includes identifying where profile-related components, routes, backend data, or API endpoints may fit.	Entire Team
Weeks 8–10	Feature implementation	Each member implements their assigned feature. For the EPK Profile, this includes building the profile display page with comedian information	Individual

		such as bio, headshot, media links, clips, and performance details.	
Weeks 11–12	Data connection and feature completion	Connect features to real or test data, complete main workflows. For the EPK Profile, this may include the edit profile form, saved profile information, and handling missing fields	Individual
Weeks 13–14	Testing and refinement	Test features, fix issues, check integration problems. Mobile layout, permissions, error handling. For the EPK Profile, this includes testing display, editing, user permissions, mobile layout, and error handling.	Entire Team
Weeks 15–16	Final demo and documentation	Prepare final demo, screenshots, documentation, and mentor approval. Each member documents their assigned feature, completed work, and any remaining limitations.	Entire Team

Tools and Resources

- Development: IntelliJ IDEA, VS Code with Codex
- Frontend: React, React Bootstrap, WebSocket/STOMP (react-stomp-hooks)
- Backend: Spring Boot, Spring ApplicationEvents, MySQL

- Infrastructure: Docker, Kubernetes, GitHub
- AI tools: Codex, ChatGPT, Claude, Claude Code
- APIs: Resend API (email notifications), HERE API (geocoding)
- Communication: Discord, Google Docs
- Design: Official ComicSeen design mockups provided by Justin

9. Testing Plan

The team will test both functionality and usability across all four feature areas.

Unit Testing

Validate individual method-level behavior. Examples include testing notification preference filtering logic, EPK form validation rules, ChatMessageService.save() for correct chatId assignment, and calendar availability data handling.

Integration Testing

Validate full backend-to-frontend workflows including the complete message send flow via WebSocket, notification event publishing and handler dispatch, and EPK profile save and retrieval through the REST API.

Role and Permission Testing

Verify messaging is restricted to Producer to Comic only. Verify a comic cannot edit another comic's EPK. Verify notification preferences correctly suppress or allow notifications per user settings.

Usability Testing

Team members and Justin Greenberg will review each feature. Key questions: Can a producer scan an EPK and understand booking fit quickly on mobile? Can a comic mark availability without confusion? Are notifications clear and actionable?

Mobile and Layout Testing

All features will be tested on mobile screen sizes given ComicSeen's mobile-first user base.

Error Handling Testing

Test behavior with missing or invalid data, incomplete EPK fields, no availability set, empty chat history, notification preference edge cases, rolled-back transactions producing no emails.

10. References

Eventbrite. (2024). How Eventbrite works. Eventbrite. https://www.eventbrite.com/l/how-it-works/

GigSalad. (2024). About GigSalad. GigSalad. https://www.gigsalad.com/about

Greenberg, J. (2025). ComicSeen platform development plan [Internal project document]. ComicSeen.

Meta. (2024). Facebook Events. Meta Platforms. https://www.facebook.com/events

Meetup. (2024). How Meetup works. Meetup. https://www.meetup.com/about/

Pollstar. (2023). The year in comedy: Belly laughs are booming business. *Pollstar News*. https://news.pollstar.com/2023/12/11/the-year-in-comedy-belly-laughs-are-booming-business/

Spring Framework Documentation. (2024). WebSocket support. https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#websocket

STOMP Protocol. (2012). STOMP simple text oriented messaging protocol. https://stomp.github.io/

Zipdo. (2023). *Comedy industry statistics*. https://zipdo.co/comedy-industry-statistics/