ChatGPT

Building a macOS Screenplay Editor with Teatro, FountainAI, and MIDI2

Welcome to this comprehensive tutorial on creating a **screenplay editor** for macOS that combines text-based screenwriting with multimedia playback. We will build a SwiftUI GUI application from the ground up, leveraging several powerful components:

- **Teatro** – a declarative view engine (with its own DSL) for rendering screenplay content and timeline animations [1] [2].
- **Codex** – prompt-driven code generation (using AI) to help create Teatro views and SwiftUI components.
- **FountainAI & FountainStore** – a local AI-backed system and embedded database for managing screenplay content and metadata [3] [4].
- **MIDI2** – a MIDI 2.0 library to coordinate audio tracks or score playback in sync with the screenplay's timeline [5].

By the end of this tutorial, you will have a working macOS app that can display and edit screenplay text (in .fountain format), persist data locally, and play back scenes with synchronized audio. We'll cover everything from project setup and running the app, to generating UI code with AI assistance, to integrating audio/visual playback. Both **manual coding** and **AI-assisted (Codex) generation** approaches will be demonstrated, so you can see how to build features by hand and how to accelerate development with AI.

Prerequisites: This guide assumes you have **Swift 6.1** (or newer) installed, along with the Swift Package Manager, and are running **macOS 14+** (Teatro's render engine targets macOS 14 for full SwiftUI support [6]). No Xcode is required for building or running the app, as we will use SwiftPM and command-line tools for a streamlined workflow [7].

Let's get started!

1. Project Setup and Running Locally

First, we'll set up the project structure and ensure we can build and run a basic macOS SwiftUI app using these components. There are two ways to proceed: using a **scaffold script** provided by FountainAI for a quick start, or setting up the Swift package manually. We'll outline both:

A. Quick Scaffold Method (using FountainAI script):

FountainAI's repository includes a script to scaffold a new GUI app with a SwiftUI template. This will create a new Swift package target under the FountainAI monorepo, preconfigured with some defaults like an LLM service integration. To use this method:

1. **Clone the FountainAI Monorepo** – If you haven't already, clone the main FountainAI repository (which includes various services and GUI scaffolding). In a terminal:

```
git clone https://github.com/Fountain-Coach/the-fountainai.git
cd the-fountainai
```

Ensure submodules or package dependencies (like Teatro, FountainStore, etc.) are fetched if required.

2. **Run the Scaffold Script** – Use the provided script to generate a new app target. For example, to create an app called *ScreenplayEditor*:

```
bash Scripts/new-gui-app.sh ScreenplayEditor
```

This will create a new Swift package target under `apps/ScreenplayEditor` with some starter code (an App struct and basic views) [8]. It also updates the root `Package.swift` to include the new executable product. The scaffold sets up a minimal UI with an **OnboardingView** (for API keys) and a **MainView** using a default `AskViewModel` (for AI Q&A) [9] [10]. We will customize this for our screenplay editor soon.

3. **Build, Bundle, and Run** – Without opening Xcode, you can compile and launch the app using SwiftPM and a helper script. Run the following in the repository root:

```
swift build --product ScreenplayEditor
bash Scripts/make_app.sh ScreenplayEditor
open dist/ScreenplayEditor.app
```

This sequence **builds** the executable, **bundles** it into a macOS `.app` container, and then **launches** it [11]. The app should open a window on macOS. (At this point, the UI is still the template UI from the scaffold – we'll modify it in later steps.)

Tooling note: The `make_app.sh` script simply creates a `.app` bundle for the SwiftUI executable, making it double-clickable in Finder [7]. If you get a "permission denied" error on the script, either run it with `bash` explicitly or `chmod +x Scripts/*.sh` once [12].

B. Manual Swift Package Setup (if not using scaffold):

If you prefer to create a standalone project (outside the FountainAI monorepo), you can use Swift Package Manager to set up the SwiftUI app and include the required dependencies:

1. **Initialize a Package** – In an empty directory, run:

```
swift package init --type=executable
```

This creates a Package.swift and a basic main file. Adjust the Package manifest to include our dependencies. Open `Package.swift` and add the following to the dependencies array:

```
dependencies: [
    .package(url: "https://github.com/Fountain-Coach/Teatro.git",
branch: "main"),
    .package(url: "https://github.com/Fountain-Coach/Fountain-
Store.git", from: "0.1.0"),
    .package(url: "https://github.com/Fountain-Coach/midi2.git", from:
"0.3.0")
],
```

Also add target dependencies for `Teatro`, `FountainStore`, and `MIDI2` as needed (e.g. `.product(name: "Teatro", package: "Teatro")`, etc.). If you have access to FountainAI's Swift packages (like `FountainAICore` and `FountainAIAdapters` for LLM integration), you may add those too, but they are optional for core functionality.

2. **Add an App Main** – Replace the contents of `Sources/<YourPackage>/main.swift` with a minimal SwiftUI App structure. For example:

```
import SwiftUI
import Teatro
import FountainStore
import MIDI2

@main
struct ScreenplayEditorApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

struct ContentView: View {
    var body: some View {
        Text("Screenplay Editor")
            .frame(minWidth: 600, minHeight: 400)
    }
}
```

This is a placeholder UI; we will expand it later. The imports bring in Teatro (for UI rendering and DSL), FountainStore (for persistence), and MIDI2 (for multimedia). We will verify these build correctly in the next step.

3. **Build and Run** – Use SwiftPM to build and run the app:

```
swift run
```

> On first run, SwiftPM will fetch the packages and compile everything. If successful, it should launch the app which shows a window with "Screenplay Editor" text. If you prefer to see a bundled app, you can similarly run the bundle script approach: after build, create an app bundle manually or by writing a simple script (the scaffold's `make_app.sh` can serve as a reference). At this stage, the app doesn't do much – our next steps will add real functionality.

Running the App: Whether you used the scaffold or manual setup, you should now have a basic macOS app that you can launch. Running via `swift run` is convenient during development (it opens the GUI after compiling). For a release build, you could do `swift build -c release --product ScreenplayEditor` and then bundle if needed. Throughout development, you can rely on the CLI tools and avoid Xcode, which keeps iteration fast [7]. (If you do open the package in Xcode, note that SwiftPM executables might appear as "Command Line Tool" in the scheme selector, but will still launch a GUI window as long as they use SwiftUI's App lifecycle [13].)

Now that our project is set up and running, let's move on to building out the features: the GUI, data persistence, multimedia playback, and AI integration.

2. Designing the GUI with Teatro's View Engine

Our screenplay editor's GUI will be built using **Teatro**, which provides a declarative DSL for SwiftUI-like views and integrates tightly with multimedia. We'll create a user interface that includes: - A text editor or viewer for screenplay content (formatted in Fountain syntax). - Controls for playback (e.g. a play/pause button to start/stop multimedia scenes). - Possibly a timeline or placeholder for visuals when playing a scene (for example, an area where scene images or text overlays appear during playback).

Teatro can be used in two ways: 1. **Direct SwiftUI integration** – using `TeatroPlayerView` and other Teatro components in SwiftUI views. 2. **Teatro DSL (Domain-Specific Language)** – constructing views, scenes, and animations using Teatro's own builders (which resemble SwiftUI's DSL but with added capabilities like Scenes and Transitions).

We will demonstrate a bit of both.

2.1 Manual UI Layout in Swift (using Teatro DSL within SwiftUI)

Let's start by manually coding a simple layout. Open your `ContentView` (or equivalent main view file) and replace its contents with a more elaborate interface:

```
struct ContentView: View {
    @State private var screenplayText: String = ""   // holds the Fountain
screenplay text
    @State private var isPlaying: Bool = false       // playback state

    var body: some View {
        HStack {
            // Left side: Script Editor
            TextEditor(text: $screenplayText)
                .font(.system(.body, design: .monospaced))
                .padding()
                .frame(minWidth: 400, minHeight: 300)
```

```
            // Right side: Playback/Preview area
            VStack {
                if isPlaying {
                    // When playing, show the TeatroPlayerView for the
current scene
                    if let playerView = playerViewForCurrentScene() {
                        playerView
                    } else {
                        Text("No scene to play")
                    }
                } else {
                    Text("Playback stopped")
                        .foregroundColor(.gray)
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                }
            }
            .frame(minWidth: 400, minHeight: 300)
        }
        .toolbar {
            Button(action: togglePlayback) {
                Label(isPlaying ? "Stop" : "Play", systemImage: isPlaying ?
"stop.fill" : "play.fill")
            }
        }
    }

    func togglePlayback() {
        if isPlaying {
            // Stop logic
            isPlaying = false
        } else {
            // Start playback logic
            isPlaying = true
            // (We'll initiate scene rendering and MIDI playback here)
        }
    }

    func playerViewForCurrentScene() -> some View? {
        // This function will create a TeatroPlayerView for the current
screenplay or scene.
        // We'll implement it in the multimedia section.
        return nil
    }
}
```

In this UI: - We use a `TextEditor` bound to `screenplayText` to allow editing of the screenplay in plain text. Using a monospaced font makes it easier to align screenplay formatting. - A toolbar button is added to toggle playback on and off. When not playing, the preview area just shows “Playback stopped”. When playing, we will display a `TeatroPlayerView` with the scene animation (to be created later). -

The `playerViewForCurrentScene()` function is a placeholder that will generate a TeatroPlayerView for the currently active scene or sequence. We'll fill this in once we cover the multimedia integration.

This is a basic two-pane layout. You can run the app now to see the UI scaffold: on the left, an empty text editor (try typing to confirm it works), and on the right, a gray message "Playback stopped" plus a Play button in the window toolbar. The Play/Stop button toggles state, but we haven't implemented the actual playback content yet.

Using Teatro DSL for Custom Views: We could further enhance the UI using Teatro's DSL for fine-grained control. For example, if we wanted to render a styled title page or scene headers using Teatro (which can apply consistent formatting), we could do something like:

```
import Teatro

func renderTitlePage(title: String, author: String) -> some View {
    // Use Teatro's view types (e.g., VStack, Text from Teatro) to create a
styled title page
    let teatroView = VStack(alignment: .center) {
        Text(title, style: .title)    // Teatro Text with a predefined style
        Text("by \(author)", style: .subtitle)
    }
    .padding(24)
    // Convert Teatro view to SwiftUI
    return teatroView.swiftUIView
}
```

Here we use `VStack` and `Text` from Teatro (note: Teatro likely has its own `Text` type for its DSL). The `.swiftUIView` property (assuming such exists via an extension) would embed the Teatro DSL-defined view into SwiftUI. Another approach is to use `TeatroRenderer` to render the view to an SVG or image and then display that, but that's more relevant for static previews. In our case, since our main UI is straightforward, we might not need to use Teatro DSL for static UI elements – we will heavily use it for **animated sequences and rendering the screenplay content** in a rich way.

Tip: Teatro's DSL is designed to be *Codex-controllable*, meaning you can generate or manipulate these views via AI. We'll see this in action shortly. The DSL covers typical view types (text, images, stacks) and extends to timeline constructs (scenes, frames). If you're familiar with SwiftUI's syntax, you'll find Teatro's view builder syntax very similar. The advantage is that Teatro views can be rendered deterministically to SVG (for snapshot tests or AI reasoning) and integrated with MIDI timing for animation.

2.2 Generating UI Code with Codex (AI Assistance)

Instead of writing all UI code manually, we can leverage AI (like OpenAI's Codex or GPT-4) to generate SwiftUI/Teatro code from natural language prompts. This can speed up development and offer creative ideas. There are two scenarios where Codex can assist: - **Layout Generation:** e.g., "Create a view with a two-pane layout, left side for text editor, right side for a preview with a play button." - **Teatro DSL Generation:** e.g., "Define a Teatro Storyboard with two scenes that fade from a title card to a scene snippet."

FountainAI is geared to facilitate such prompt-driven development. In fact, Teatro provides a `CodexStoryboardPreviewer` to help AI models reason about UI sequences by generating text descriptions of frames [14]. Let's illustrate how one might use Codex to generate parts of our UI:

Example Prompt (Layout Generation):

> *"Design a SwiftUI view for a screenplay editor. It should have a TextEditor filling the left half for the script, and on the right a vertical stack that shows either a placeholder text 'Playback stopped' or a player view if playing. Include a play/pause toolbar button."*

When fed to a code-generation model, the returned Swift code might look similar to what we wrote above – a `HStack` with a `TextEditor` and a conditional `VStack` for the preview, plus a toolbar button. In fact, the code we ended up with in `ContentView` could very well have been produced by such a prompt to GPT-4 or Codex. If you have an interactive development environment with Codex integration (for example, using FountainAI's GUI with an "Ask" function), you could try such prompts and insert the generated code, then refine it.

Example Prompt (Teatro DSL Generation):

> *"Using Teatro's DSL, create a storyboard with two scenes: Scene 'Intro' showing text 'Opening Scene' centered, and Scene 'Scene1' showing a character name and a dialogue line. Transition with a crossfade over 10 frames. Provide the Swift code."*

An AI response might produce something like:

```
let storyboard = Storyboard {
    Scene("Intro") {
        VStack(alignment: .center) {
            Text("Opening Scene", style: .title)
        }
    }
    Transition(style: .crossfade, frames: 10)
    Scene("Scene1") {
        VStack(alignment: .leading) {
            Text("JOHN", style: .bold)
            Text("Hello there.")
        }
    }
}
```

This is exactly in line with Teatro's storyboard DSL syntax [15] [16]. The AI might not get everything 100% correct on first try (you might need to tweak property names or import statements), but it provides a solid starting point. In this case, the structure matches the documented example: we have Scenes and a Transition defined in a closure builder.

In practice, you can use the FountainAI app (or any environment where you can talk to an LLM) to iteratively develop your UI. For instance, once our base UI is running, we might press a special "Ask Codex" button (similar to what FountainAI's scaffolded `AskViewModel` does) and prompt it to adjust the interface or add a component. This can be done locally if you have an LLM API key configured. The

key is to treat Codex as a pair-programmer: you describe the change, it provides a code diff or snippet, and you integrate it.

We will continue with manual coding for the remaining sections, but keep in mind that **every step moving forward could be accelerated with AI generation**. We'll highlight a few more opportunities to use Codex (like generating model code or test data) as we proceed. Next, let's tackle persistence: saving and loading screenplay content using FountainStore.

3. Persistence with FountainStore – Reading, Writing, and Extending Fountain Scripts

To make our editor useful, it needs to save screenplay content and related data. We'll use **FountainStore**, a pure-Swift embedded database designed for FountainAI, to persist data locally. FountainStore functions as a high-performance, ACID-compliant store with support for full-text search (FTS) and vector search, and it serves as the "memory core" of FountainAI [4]. In our app, FountainStore can store screenplay texts, their metadata (titles, timestamps, etc.), and even additional artifacts like analyses or multimedia references.

3.1 Setting Up FountainStore in the App

First, we'll configure a FountainStore instance. We can create it when the app launches (e.g., in `AppEntry` or in an `ObservableObject` that lives throughout the app). For simplicity, let's initialize it in our App struct:

```
import FountainStore

@main
struct ScreenplayEditorApp: App {
    // Initialize the store at a fixed path (in Documents or a temp directory
for demo)
    let store: FountainStore = {
        let documentsURL = FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask).first!
        let storeURL = documentsURL.appendingPathComponent("FountainAIStore")
        let options = StoreOptions(path: storeURL)
        // Opening the store is async, so we'll do it synchronously for
simplicity (not recommended for real UI app init)
        return try! await FountainStore.open(options)
    }()

    // Define a data model for screenplay entries
    struct Screenplay: Codable, Identifiable {
        var id: UUID
        var title: String
        var content: String  // the full fountain text
        var lastEdited: Date
    }
    // Collection handle for screenplays
    let screenplayCollection: Collection<Screenplay>
```

```
    init() {
        // Create or open the collection
        screenplayCollection = store.collection("screenplays", of:
Screenplay.self)
        // (Optional) define a full-text index on content for search
        try? await screenplayCollection.define(Index<Screenplay>(name:
"fulltext", kind: .fts(\Screenplay.content)))
    }

    var body: some Scene {
        WindowGroup {
            ContentView(store: store, collection: screenplayCollection)
        }
    }
}
```

A few important points about this setup: - We open the FountainStore at a specified directory (here using the user's Documents for persistence; adjust as needed). The first time, it will create the necessary files (Write-Ahead Log, manifest, etc.), and on subsequent runs it will reload the saved data [17]. - We define a `Screenplay` struct that conforms to `Codable & Identifiable`. It stores an `id` (UUID), a `title`, the `content` of the screenplay (as raw Fountain text), and a `lastEdited` timestamp. You can extend this with more fields (author, etc.) as needed. - We obtain a `Collection<Screenplay>` from the store by name `"screenplays"`. Collections in FountainStore are like tables for a specific data type [18]. If data was previously saved, the collection will load existing items into an in-memory index. If not, it starts empty. - We add a full-text search index on the `content` field using `Index.fts` [19] [20]. This means we can later perform keyword queries on screenplay text efficiently (e.g., find all scripts containing "INT. HOUSE"). This uses FountainStore's optional FTS module (we included it by adding the package; under the hood it tokenizes text and builds postings lists [21] [22]). - We pass the store and collection into `ContentView` for use (so that our UI can load and save screenplays).

Now update `ContentView` to accept these as parameters and use them. For instance, we can add functionality to load an existing screenplay into the editor and save edits back:

```
struct ContentView: View {
    let store: FountainStore
    let collection: Collection<ScreenplayEditorApp.Screenplay>
    @State private var screenplayText: String = ""
    @State private var currentScript: ScreenplayEditorApp.Screenplay? = nil

    // ... (rest of ContentView code from earlier) ...

    func loadScreenplay(withId id: UUID) async {
        if let script = try? await collection.get(id: id) {
            currentScript = script
            screenplayText = script.content
        }
    }

    func saveScreenplay() async {
```

```
        guard var script = currentScript else {
            // If no script loaded, we could treat the current text as a new
script
            let newScript = ScreenplayEditorApp.Screenplay(
                id: UUID(),
                title: "Untitled",
                content: screenplayText,
                lastEdited: Date()
            )
            currentScript = newScript
            try? await collection.put(newScript)
            return
        }
        // Update existing script
        script.content = screenplayText
        script.lastEdited = Date()
        try? await collection.put(script)
    }
}
```

Here: - `loadScreenplay(withId:)` fetches a script by UUID from the collection [23] and populates the editor. - `saveScreenplay()` either creates a new script entry (if none loaded) or updates the current one, then writes it using `collection.put` [24]. The `.put` method will add or overwrite the value in the store, and FountainStore's MVCC mechanism will keep track of version history under the hood. (Proper error handling omitted for brevity; in a real app you'd handle failures.) - We would call `saveScreenplay` whenever needed, e.g., when the user hits Cmd+S or toggles away from the document, etc. For this tutorial, you can tie it to some UI action or simply call it when stopping playback (to simulate auto-save).

Running a quick test: To verify persistence is working, you might programmatically create a screenplay, save it, then reload:

```
Task {
    let sample = ScreenplayEditorApp.Screenplay(id: UUID(), title: "Test
Script", content: "INT. HOUSE - DAY\n\nJOHN\nHello there.", lastEdited:
Date())
    try? await collection.put(sample)
    print("Saved screenplay with id \(sample.id)")
    // Now retrieve it
    if let fetched = try? await collection.get(id: sample.id) {
        print("Loaded screenplay titled: \(fetched.title), content:\n\
(fetched.content)")
    }
}
```

This code (which you could run in `ContentView.onAppear`) would save a script and immediately load it, printing to console. You should see the content come back, proving FountainStore's end-to-end storage. Also note that FountainStore is crash-safe – even if the app terminates unexpectedly, the WAL ensures no data loss [25].

3.2 Overriding and Extending the .fountain Format

The app deals with **Fountain** screenplay format (a plaintext markup for screenplays). Our editor currently just treats it as text, but we can do more: - We can **parse** Fountain text to identify elements (scene headings, character cues, dialogues, etc.) and potentially display them with special styling or interactive features. - We can **extend** the Fountain spec with custom annotations for multimedia cues, since our app supports playback.

Parsing Fountain: Teatro actually includes a Fountain parser and renderer. For example, Teatro's render API can take Fountain text and produce an SVG representation or a synopsis in Markdown [1]. In our app, we might use this to show a formatted preview or to quickly extract scene headings:

```
import TeatroRenderAPI

func fountainSynopsis(from text: String) -> String? {
    let input = SimpleScriptInput(fountainText: text)
    if let result = try? TeatroRenderer.renderScript(input) {
        return result.markdown  // e.g., "- Opening scene" as synopsis for
first scene
    }
    return nil
}
```

The `TeatroRenderer.renderScript` function parses the fountain text and renders a script. It returns an object with an `svg` (for visual representation) and `markdown` (for text outline) [26]. You could use `result.svg` to show a nicely typeset image of the script, but since we want editable text, using the Markdown outline might be more appropriate for, say, a sidebar of scenes.

We won't replace our editor with a fully rendered view just yet (to keep it editable), but this shows you can integrate the Fountain parser. Additionally, Teatro's `FountainParser` can likely be extended – there is mention of a `FountainParser+FountainAI.swift` which suggests customizing parsing rules for FountainAI's needs (perhaps to support AI-specific annotations). While we won't dive into extending Teatro's parser in code, we can achieve custom behavior by pre-processing the text ourselves.

Custom Fountain Markup for Multimedia: Let's say we want the writer to mark a point in the script where a sound effect or music should play. Standard Fountain doesn't have a dedicated syntax for this (though writers often just put it in action lines or notes). We can invent a simple convention, for example: - Use a comment note `[[SOUND: <description>]]` in the Fountain text to denote an audio cue. - Or use an action line that starts with a special keyword, e.g., `AUDIO: track1.mp3 @ 1:23`.

For demonstration, we'll use the double-bracket annotation since Fountain's spec treats `[[ ... ]]` as a **boneyard** (comment) and will ignore it in normal rendering. Our app, however, will detect it. For instance:

```
func extractAudioCues(from text: String) -> [(time: TimeInterval, file:
String)] {
    var cues: [(TimeInterval, String)] = []
```

```
    let pattern = #"\[\[SOUND:\s*([^@]+)@([^]]+)\]\]"#  // e.g. [[SOUND:
track.mp3 @ 01:23]]
    let regex = try? NSRegularExpression(pattern: pattern,
options: .caseInsensitive)
    let nsText = text as NSString
    if let matches = regex?.matches(in: text, range: NSRange(location: 0,
length: nsText.length)) {
        for match in matches {
            if match.numberOfRanges >= 3 {
                let file = nsText.substring(with: match.range(at:
1)).trimmingCharacters(in: .whitespaces)
                let timeString = nsText.substring(with: match.range(at:
2)).trimmingCharacters(in: .whitespaces)
                if let time = parseTimecode(timeString) {
                    cues.append((time, file))
                }
            }
        }
    }
    return cues
}

// Helper to parse "MM:SS" or "M:SS" into seconds
func parseTimecode(_ str: String) -> TimeInterval? {
    let parts = str.split(separator: ":").compactMap { Double($0) }
    guard parts.count == 2 else { return nil }
    return parts[0] * 60 + parts[1]
}
```

This function scans the screenplay text for patterns like `[[SOUND: track.mp3 @ 1:23]]` and extracts the filename and timestamp. The `parseTimecode` converts a minute:second string into seconds. We can call `extractAudioCues(screenplayText)` whenever we start playback, to know what audio files and timings to load.

(In a real application, you might integrate this with how you construct the MIDI timeline, or even extend Teatro's parser to treat `[[SOUND:...]]` as a token. Since we have the text and control of the timeline, doing it manually as above works fine.)

Writing Custom Data to FountainStore: We might also want to save these cues or other metadata in the store. We could extend our `Screenplay` model to have an array of cues or store them in a separate collection (e.g., a `MediaCue` collection referencing screenplay ID and containing the cue info). For simplicity, let's say we decide not to store cues separately but always derive them from the text for now. However, keep in mind FountainStore can handle multiple collections, so storing structured data (like cues, or breakdown of scenes) is entirely possible. FountainStore's embedded design means everything is saved locally in our app's database, with crash recovery, etc. [25] .

At this stage, our app can load and save screenplay text, and we've laid the groundwork to parse and extend the fountain content for multimedia. The editor remains text-centric, but that's expected – many screenwriters prefer working directly in the fountain text. Next, we'll focus on the multimedia playback:

how to take a screenplay (or part of it) and play it with synchronized visuals and audio using **Teatro + MIDI2**.

4. Embedding Multimedia Playback with Teatro and MIDI2

One of our app's key features is the ability to play back scenes with audio – think of it as a lightweight "pre-visualization" or table-read tool, where the screenplay text can be accompanied by background score or sound effects at the right moments. We will accomplish this by using **Teatro's Storyboard DSL** for visual sequencing and **MIDI2** for audio timing and playback.

4.1 Creating a Storyboard for Scenes

A **Storyboard** in Teatro is a sequence of frames (views) grouped into scenes, possibly with transitions [27]. For example, we might want to animate the display of a scene's text or switch between different screens (title card, scene content, etc.). In our context, one straightforward use is to show the script text line by line or paragraph by paragraph in sync with audio.

For simplicity, let's create a storyboard for a single scene: It will display the scene heading and then the dialogue line by line. We'll use that to visualize the script during playback. If the script is long, one could break into multiple scenes or have the storyboard only cover a portion. Here, we'll assume the user triggers playback for the current scene or current selection only.

Example: Suppose our fountain text has:

```
INT. HOUSE – DAY

JOHN
Hello there.

JANE
Hi!
```

We want to create a storyboard that maybe first shows the scene header "INT. HOUSE – DAY", then fades into John's line, then Jane's line.

Using Teatro's DSL:

```
import Teatro

func makeStoryboard(for scriptText: String) -> Storyboard {
    // Very naive parsing: split lines and find dialogue lines
    let lines = scriptText.components(separatedBy: "\n")
    var storyboard = Storyboard {
        // Scene for scene heading
        if let heading = lines.first, heading.hasPrefix("INT") ||
heading.hasPrefix("EXT") {
            Scene("Heading") {
                Text(heading, style: .italic)  // style it differently
```

```
            }
        }
        // Iterate subsequent lines for dialogue
        for i in 0..<lines.count {
            let line = lines[i]
            if line.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty
{ continue }
            // If it's a character name (usually in all caps)
            if line == line.uppercased(), i+1 < lines.count {
                let charName = line
                let charLine = lines[i+1]
                Scene(charName) {
                    VStack(alignment: .leading) {
                        Text(charName, style: .bold)
                        Text(charLine)
                    }
                }
            }
            // We could also add transitions between each scene
            Transition(style: .crossfade, frames: 5)
        }
    }
    return storyboard
}
```

This code builds a storyboard with multiple scenes: one for the scene heading, and one for each character's speech. We inserted a Transition of 5 frames between each scene for a simple crossfade. Note: This is a simplified parsing logic (it doesn't handle multiple dialogues properly, it will even insert a transition after every line which might be a bit too frequent). But it illustrates the approach.

Better approach: In a real scenario, you'd use a proper Fountain parser to identify sections like scene headings, action lines, character cues, etc. Teatro's `FountainScreenplayEngine` (as referenced in docs) likely does this parsing. If we tapped into that, we could get a structured representation and then feed that into a storyboard. For now, assume we manage to identify segments we want to visualize.

Codex Tip: You could ask Codex to generate this storyboard code by providing a few lines of script as example input. The AI could infer that each character block should be a scene, etc. Always double-check the logic though, as parsing human language reliably might need iterative refinement.

4.2 Building a MIDI Track for Timing and Audio

With our storyboard defined, we need a corresponding **MIDI sequence** that dictates the timing of frames and also triggers actual audio playback. We'll use the **MIDI2** library's facilities to create a MIDI timeline: - We can consider each **frame** in the storyboard to correspond to a musical beat or a fixed time interval. - We will insert MIDI **notes or markers** to delineate frames. - We will also add MIDI **SysEx or Meta events** for things like lyrics or markers which we can use to carry text tokens or signals (Teatro can even send SSE events over MIDI as tokens [5] [28], but here we'll keep it simple). - For playing an actual audio file (like a background music MP3), MIDI by itself can't play it – but we can either: - Use a MIDI **marker** event that our code listens for and then trigger an audio player in Swift when that marker is reached. - Or if the audio is a musical score (MIDI notes), embed those notes directly.

To keep it straightforward, let's assume we have a background score as a MIDI melody (or at least we'll create a simple one) that lasts the duration of our scene. We'll generate a dummy melody that aligns with the number of frames.

Using AppleSequencerBridge: The MIDI2 package provides `AppleSequencerBridge` which helps build a sequence and export to a Standard MIDI File [29]. We can use it to add notes and other events. For example, to create a simple sequence:

```
import MIDI2
import TeatroAppleBridge  // this is part of the MIDI2 package for CoreMIDI
integration

func buildSequence(frameCount: Int, tempo: Double = 120.0) ->
AppleSequencerBridge {
    let seq = AppleSequencerBridge()
    seq.setTempoMap([TempoEvent(beat: 0, bpm: tempo)])       // set tempo
    seq.addMarker(beat: 0, text: "Start")                    // marker at
start
    // Add a note for each frame as a timing cue (each note = one frame
duration)
    for frame in 0..<frameCount {
        let beat = Double(frame)
        seq.addNote(track: 0, channel: 0, note: 60, velocity: 80, startBeat:
beat, durationBeats: 1)
        // Optionally, add lyric events carrying text of that frame (e.g.,
the dialogue line)
        // seq.addLyric(beat: beat, text: "Frame \(frame)")
    }
    // End marker
    seq.addMarker(beat: Double(frameCount), text: "End")
    return seq
}
```

This creates a sequence where: - Tempo is 120 BPM (so each beat = 0.5 seconds, if quarter note = 0.5s when tempo=120). - Each frame corresponds to 1 beat (so frames advance every 0.5 seconds). Adjust tempo or duration if you want different pacing. - We add a middle C note (MIDI note 60) per frame, with velocity 80, each 1 beat long. These notes don't necessarily produce sound by themselves unless connected to a synth, but they serve as timing events. - We could use lyrics or markers to embed frame-specific data (e.g., to show the text content of that frame as an overlay, which TeatroPlayerView can use as overlays as mentioned in docs [30]). For brevity, we only added a generic lyric commented out.

Now, crucially, we want to actually **hear** something. If we have actual MIDI musical content (notes that should be heard, e.g., a melody or chords), we could encode that similarly by adding notes of varying pitches and perhaps on a second track. For demonstration, let's play a simple scale or repeated note:

```
    // ... continuing from buildSequence above ...
    // On a second track (track: 1), add a simple melody (e.g., C major scale
spanning the frames)
```

```
    for frame in 0..<frameCount {
        let notePitch: UInt8 = 60 + UInt8(frame % 8)  // cycle through 8
notes
        seq.addNote(track: 1, channel: 0, note: notePitch, velocity: 100,
startBeat: Double(frame), durationBeats: 1)
    }
```

This will cause track 1 to play a scale (C, D, E, … looping) along the timeline. If we route this to an instrument, it would produce sound.

Audio Playback: There are a few ways to actually get audio output: - **Use TeatroPlayerView with a Sampler:** According to Teatro docs, `TeatroPlayerView` can directly drive audio via a `TeatroSampler` if given a `SampleSource` [31]. This implies we can hand it a MIDI sequence and it will output sound (likely using AVAudioUnitSampler or Core MIDI internally). - **Use Apple's CoreMIDI** to send to the default synth (the Apple DLS Synth). The `teatro-play` CLI example shows piping to an Apple sampler on macOS [32], which suggests the code chooses an output sink. In our app, we'd want the sound in-app. - **Use AVFoundation**: Alternatively, we could export our sequence to a MIDI file and use AVMIDIPlayer or similar to play it with a soundfont, or use AVAudioEngine with MIDISampler. But since Teatro presumably already solved this with TeatroSampler, we'll use that.

So the easier path: **TeatroPlayerView** – this view, when constructed with a storyboard and a MIDI sequence, and given a sample source, should handle syncing and playback:

```
import Teatro
import SwiftUI

func playerViewForStoryboard(_ storyboard: Storyboard, sequence:
AppleSequencerBridge) -> some View {
    // Convert AppleSequencerBridge into a MIDISequence type if needed
    // (TeatroPlayerView might accept the AppleSequencerBridge directly or we
get Data from export)
    let player = TeatroPlayerView(storyboard: storyboard, midi: sequence)
    return player
}
```

From the docs: `TeatroPlayerView(storyboard: storyboard, midi: melody)` creates a player [33]. We might have to ensure `AppleSequencerBridge` or the underlying sequence format is compatible. If not, we might do `sequence.exportSMF` to get a MIDI file and then load that into an Apple MusicSequence. But since both Teatro and AppleSequencerBridge use CoreMIDI, likely `TeatroPlayerView` has internal logic to iterate through the sequence's events (maybe by reading the `.ump` or `.smf` data).

For now, assume `TeatroPlayerView` can accept the AppleSequencerBridge instance (or there might be an extension or conversion, e.g., `sequence.musicSequence` or `sequence` has the data).

Integrating into the App: We tie it all together in the `togglePlayback` and `playerViewForCurrentScene` we set up earlier:

```
@State private var storyboard: Storyboard? = nil
@State private var midiSequence: AppleSequencerBridge? = nil

func togglePlayback() {
    if isPlaying {
        // Stop playback
        isPlaying = false
        storyboard = nil
        midiSequence = nil
    } else {
        // Start playback: build storyboard and sequence
        guard !screenplayText.isEmpty else { return }
        storyboard = makeStoryboard(for: screenplayText)
        if let sb = storyboard {
            let framesCount = sb.frames().count   // total frames in
storyboard timeline
            midiSequence = buildSequence(frameCount: framesCount, tempo:
120.0)
        }
        isPlaying = true
    }
}

func playerViewForCurrentScene() -> some View? {
    if let sb = storyboard, let seq = midiSequence {
        // Provide the player view with a sampler for audio (if needed)
        return TeatroPlayerView(storyboard: sb, midi: seq)
            .frame(maxWidth: .infinity, maxHeight: .infinity)
    }
    return nil
}
```

Now, when the user hits Play: - We generate a `storyboard` from the current screenplay text (or a portion of it if desired). - We generate a `midiSequence` with matching number of frames. - Set `isPlaying = true`, which triggers SwiftUI to re-render the right pane and call `playerViewForCurrentScene()`, returning a `TeatroPlayerView` that will start playback.

If everything is correct, the `TeatroPlayerView` will begin stepping through the storyboard frames in sync with the MIDI sequence's timing. Each frame will last as long as the corresponding MIDI note (with our settings, 0.5 seconds per frame at 120 BPM). The crossfade transitions we added in the storyboard mean overlapping content between frames for smooth visual transitions [34].

Audio Output: The sequence on track 1 (if connected to a default instrument) will play the scale notes we inserted. By default, on macOS this might route to the Apple default synth. If not, one may need to supply a `SampleSource`. For instance:

```
let sampleURL = URL(fileURLWithPath: "/System/Library/Audio/Sounds/
Submarine.aiff")
```

```
let sampler = try? TeatroSampler(unit: .apple)  // hypothetical initializers
sampler?.loadSoundFont(named: "GeneralUser")    // if we have a SF2
TeatroPlayerView(storyboard: sb, midi: seq, sampleSource: sampler)
```

The above is speculative as the actual API might differ (you'd consult Teatro docs or intellisense for `TeatroSampler`). Since our focus is the integration rather than low-level audio, we'll assume the default works or requires minimal setup.

Syncing Sound Effects (Cues): If we extracted sound cues like `[[SOUND: Thunder.wav @ 00:10]]` earlier, how to play them? We could: - Use the `extractAudioCues` to get a list of (time, file) pairs. - When starting playback, schedule timers or use an AVAudioPlayer to play those files at the specified offsets from start. - Or encode these cues as part of the MIDI sequence (e.g., as markers at specific beats, then have a callback in the MIDI playback to trigger the sound). Since implementing a callback in `TeatroPlayerView` might not be directly exposed, an easier method is to handle it in Swift: start an `AVAudioPlayer` for each cue with a delay equal to `time` (or using a scheduled player node in AVAudioEngine).

For brevity, we won't code a full AVAudioEngine here. But as an outline:

```
let cues = extractAudioCues(screenplayText)
for (time, file) in cues {
    if let url = Bundle.main.url(forResource: file, withExtension: nil) {
        let player = try AVAudioPlayer(contentsOf: url)
        player.prepareToPlay()
        // schedule to play after 'time' seconds
        DispatchQueue.main.asyncAfter(deadline: .now() + time) {
            player.play()
        }
    }
}
```

This would play the sound file at the intended script moment.

Now we have a functioning multimedia playback feature: - Visually, it uses Teatro to show scripted content frame by frame. - Timing-wise, it's governed by a MIDI sequence, ensuring deterministic synchronization (if the system lags, the frames and MIDI stay locked with timestamps). - Aurally, it produces simple music (from MIDI notes) and could trigger sound effects.

Try running the app and clicking Play: you should see the right pane start showing the sequence of scenes (e.g., scene heading italic, then character lines) with crossfades, and hear the notes playing. If you have multiple scenes and dialogues, the storyboard will progress through them. TeatroPlayerView also reportedly can overlay token streams or reliability stats in real-time [28], which is a feature used when streaming AI outputs (more on that next).

4.3 Diagrams: How it All Fits Together

It's helpful to visualize the data flow in our app: - **Screenplay Text** (Fountain) –> parsed by our code (or Teatro) into a **Storyboard** (sequence of view frames). - **Storyboard Frames** – each corresponds to a unit of time (frame/beat) in the **MIDI Sequence**. - **MIDI Sequence** – contains timing (tempo, note durations)

and can also carry data events (markers, lyrics). This sequence is fed into Teatro's player and also drives audio output via a sampler. - **TeatroPlayerView** – takes the storyboard and MIDI. It advances the storyboard according to MIDI timing (e.g., when each note's duration elapses, it moves to the next frame) [35]. It also can display overlays (like lyric events or SSE token text) on top of frames. - **FountainStore** – sits on the side to save the screenplay text whenever changes are made, and to retrieve it later. (Not directly in the playback loop, but ensures persistence.) - **FountainAI (services)** – optional for now, but if integrated, the LLM might generate new dialogue or provide feedback which then becomes new screenplay text or annotations.

If you imagine this as a pipeline: Screenplay (text) –> **Teatro storyboard** –> **TeatroPlayerView** (with MIDI) –> On-screen animation + Sound.

And separately: Screenplay <–> **FountainStore** (load/save); plus possibly Screenplay –> **FountainAI LLM** (for suggestions).

We've built the core functionality of our app. Next, we will integrate the AI aspect (FountainAI's token-based architecture) to enable streaming content and fallback logic.

5. AI Integration: Streaming and Fallback with FountainAI's Architecture

One of the advantages of building on FountainAI's platform is access to AI-driven features for screenplay writing. This can include: - **Streaming text generation** for auto-completing dialogues or writing descriptions (like having an AI "co-writer"). - **Script analysis or transformation** (e.g., analyzing a scene's sentiment or generating a synopsis). - **Fallback to local models or offline modes** if the primary AI service (like OpenAI API) is unavailable or the user opts out of cloud services.

FountainAI is built with a token streaming architecture – responses from the LLM are received incrementally as tokens (often via server-sent events, SSE) [36]. This means our UI can display the AI's answer word-by-word (or sentence-by-sentence) as it comes in, rather than waiting for a full completion. We already saw that Teatro can handle streaming by overlaying tokens on the view in sync with a "live" MIDI stream [28], which is a very advanced feature (embedding GPT output in a timed way, e.g., to have characters "speak" lines as they're generated).

Here, we'll implement a simpler version: use FountainAI's **LLM Gateway** or OpenAI API to get suggestions for continuing the script, and show the tokens as they arrive in a text view.

5.1 Adding an AI "Continue Scene" Feature

Let's add a button or menu action in our UI to "Continue Writing" the screenplay using AI. For example, if the user has written a few lines and wants the AI to draft the next bit: - We send the current screenplay text (or perhaps just the last scene or prompt) to the LLM. - We stream the completion tokens and append them to the `screenplayText`.

Using FountainAI's structure: - The **LLM Gateway** expects a chat or prompt. We can call it via an HTTP API (if FountainAI's services are running locally) or via the OpenAI adapter. - The scaffolded code we saw uses an `AskViewModel` with an `LLMService`. We can reuse that approach here for simplicity. The `OpenAIAdapter` under the hood calls the OpenAI API.

If we included `FountainAIAdapters` in our package (assuming access), we could do:

```
import FountainAIAdapters
import FountainAICore

let llmService = OpenAIAdapter(apiKey: "<YOUR_API_KEY>")
```

(This would normally be configured in onboarding, which the scaffold handled with Keychain and OnboardingView [37].)

For our demonstration, let's assume we have an `apiKey` stored (maybe the user entered it at start, or we run with a dummy if none). We will call the OpenAI completion API. Pseudocode:

```
func continueScene() {
    let prompt = screenplayText + "\n\nContinue the scene:"
    Task {
        do {
            var streamText = ""
            // Use a streaming API call – pseudocode for OpenAI:
            let stream = try await OpenAIAdapter.streamChat(prompt: prompt)
            for try await token in stream {
                streamText += token
                // Update the UI with partial text
                DispatchQueue.main.async {
                    self.screenplayText += token
                }
            }
        } catch {
            print("AI generation failed: \(error)")
        }
    }
}
```

This outlines connecting to an API that yields tokens. With FountainAI's LLM Gateway, the SSE endpoint would send events that contain tokens [36]. If we had FountainAI running, we could call something like `http://localhost:8000/v1/complete` and listen for SSE. In Swift, we could use `URLSession` with a delegate for SSE (or a third-party SSE library).

However, the details of API calls are beyond our scope. Instead, we'll conceptualize: - The UI shows some kind of progress (maybe we disable input or show a spinner). - As tokens arrive, they appear in the TextEditor (giving the impression of the text being typed out by an invisible hand). - Once done, the user can edit or keep it.

Handling Fallbacks: FountainAI's architecture is designed to easily swap AI providers or use a local model if API fails. In our app: - We already included a basic fallback in the scaffold: if no OpenAI API key, they use a `MockLLMService` that just echoes input [38] [39]. We can adopt similar logic: if the user hasn't configured an API, maybe use a trivial offline model or simply inform that AI is not available

offline (depending on what fallback we want). - Another fallback scenario: if streaming fails mid-way (network glitch), we could attempt to re-try once or complete the rest using a non-streaming endpoint.

Given the complexity, let's implement a simple decision:

```
func getLLMService() -> LLMService {
    if let key = UserDefaults.standard.string(forKey: "OpenAIKey"), !
key.isEmpty {
        return OpenAIAdapter(apiKey: key)
    } else {
        return
MockLLMService()  // from the scaffold code, returns a trivial result
    }
}
```

Then use `service.chat` or `service.stream` accordingly. The scaffold's `AskViewModel` already encapsulates some of this: it calls `llm.chat(model:messages:)` and populates `answer` asynchronously. We might just use that AskViewModel in our ContentView for simplicity if available:

```
@State private var askVM: AskViewModel? = nil
...
.onAppear {
    askVM = AskViewModel(llm: getLLMService(), browser: MockBrowserService())
}
...
Button("Continue Scene") {
    Task {
        if let answer = await askVM?.ask(question: "Continue screenplay") {
            screenplayText += "\n" + answer
        }
    }
}
```

However, the prompt "Continue screenplay" is too vague – ideally, we send the actual script. We could modify `AskViewModel.ask` to use our screenplay text as context. Without diving too deep, we assume we can craft a prompt or message like:

```
let lastSceneContent = ... // extract last scene from screenplayText
await askVM?.ask(question: "Continue the following scene:\n\
(lastSceneContent)\n###\nContinue:")
```

And the LLM might continue after the ### marker (just a prompt design choice).

Visualizing Streaming: If we wanted the streaming effect in a fancy way, we could integrate it with Teatro's SSE-over-MIDI. Teatro is capable of receiving SSE events embedded in MIDI (they call it *Fountain SSE Envelope*) [5] [28]. Essentially, tokens can be packaged as MIDI SysEx events and TeatroPlayerView will display them as they come, possibly even in sync with a beat. That's beyond our current scope, but

it's great to know the framework supports those advanced visualizations (like showing words appearing in time, perhaps for an AI-driven character speech with a controllable pace).

In summary, our AI integration in this tutorial is basic: it fetches a suggestion and appends it. But we've touched on how the *token-based streaming* works and how we could show partial progress. The key part is that **FountainAI's GUI and architecture are built to handle streaming** – token flows can be visualized and even coordinated with MIDI for precise playback [36]. If implementing fully: - Use `URLSession` to connect to FountainAI's LLM Gateway SSE endpoint (which likely is something like `/v1/chat/completions` with stream). - As events arrive, parse JSON to get the token text, and update the screenplay text state. - Meanwhile, because our UI is built on SwiftUI and the text editor's bound to `screenplayText`, the new text appears live.

Fallback logic is handled by choosing which `LLMService` to use (OpenAI vs local). If a request fails mid-way, we could also implement a catch where we fallback to a non-streaming completion: e.g., if streaming fails, call a normal completion endpoint to get the rest. Or simply notify the user and keep whatever was generated.

6. Workflows: Manual Development vs Codex-Assisted Generation

Throughout this tutorial, we've shown step-by-step how to implement each feature manually, while also noting places where **AI (Codex)** can help generate code or content. Let's summarize the two workflows:

- **Manual Workflow:** You, the developer, write the Swift code, design the UI, and integrate frameworks through careful reading of docs and iterative testing. This gives you full control and understanding of the system. We manually wrote the UI layout, persistence calls, storyboard logic, and some dummy parsing. Each component was informed by documentation (like using Teatro's API as per the examples [1] or FountainStore usage from its blueprint [40]). Manual coding can be time-consuming, but it ensures you know exactly how data flows and can optimize or debug as needed.

- **Codex-Generated Workflow:** You leverage AI to generate boilerplate or even complex logic by describing your intent in natural language. For instance, we could have asked Codex to "create a function to parse custom SOUND cues from fountain text" and likely gotten the `extractAudioCues` logic (maybe with a different regex style). We also could generate SwiftUI code for the layout (as we discussed) in seconds, rather than writing from scratch. The **ideal approach** is a combination: use Codex to get a starting point, then refine it manually. This is exactly how FountainAI's tools are meant to be used – the platform emphasizes a synergy where AI can draft code and the developer supervises and tweaks it.

Example: The *Storyboard DSL with CodexPreviewer* is a great illustration of AI assistance. By printing a textual representation of the storyboard frames [14], you give the AI an understanding of the UI flow. The AI can then decide to change a transition or adjust frame content without needing to run the code. In practice, you might do:

```
let prompt = CodexStoryboardPreviewer.prompt(storyboard)
print(prompt)
```

This outputs something like:

```
Frame 0: [Intro Scene – "Welcome"]
Frame 1: [Transition crossfade]
Frame 2: [End Scene – "Goodbye"]
```

You feed this to GPT-4 and say “make the transition 20 frames instead of 10, and change 'Goodbye' to 'The End' in the last frame.” The model can output the modified code for the storyboard. This is **round-trip design with AI**.

In our editor app scenario, one could envision a future feature where the user can ask the app (through an AI interface) to **rearrange scenes**, **find inconsistencies**, or **suggest dialogue changes**. Under the hood, those requests would be handled by FountainAI’s services (like a Planner or Semantic Analyzer) and possibly by generating new Teatro views or updated screenplay content. We have the building blocks in place: FountainStore holds the data, FountainAI services can be queried, and Teatro can render any new content visually.

Testing and Verification: Regardless of manual or AI-assisted development, always test each component: - Use unit tests for functions like `extractAudioCues` (e.g., feed a sample string and assert the output is correct). - Use snapshot tests for the storyboard rendering if possible (TeatroRenderAPI’s deterministic output allows generating an SVG and comparing to a reference [41]). - Test persistence by saving and reloading data, verifying nothing is lost and indices work (e.g., search for a keyword and see if FTS returns the correct script ID). - Check multimedia sync by adding obvious cues (like a text overlay counting seconds) to ensure timing is as expected.

Both workflows benefit from FountainAI’s design principles – because the system is OpenAPI-driven and modular [42] , even our GUI app only interacts with documented APIs or libraries. This keeps things reliable and easier to debug. For instance, our use of FountainStore is through its stable API (put/get), and our use of Teatro is via the Render API and Player which are meant to be consistent across versions.

Finally, to **run everything together locally**: 1. Ensure any required background services are running. For basic usage, we might not need FountainAI’s servers, but if you want the full AI integration, you should run the FountainAI Gateway and associated servers (Auth, LLM Gateway, Persist server etc.). This could be done via Docker or `swift run` for each service if you have the monorepo. (Detailed setup of those is beyond this tutorial, but FountainAI’s README and docs provide guidance on starting the gateway and persist services; for example, you might run `swift run PersistServer` to serve FountainStore over HTTP, etc.) 2. Launch the ScreenplayEditor app (either via `open .app` or `swift run`). On first launch, if using the scaffolded approach, complete the onboarding by entering your OpenAI API key (it stores it securely) [43] . 3. Test creating a screenplay, saving it, reloading it (maybe restart the app to ensure it persists). Try the play function to see the animation and hear audio. Then test the AI “continue” feature – if you have configured the API key and internet, it should stream some completion. If no key, ensure the fallback (mock service) returns something (in our mock, it just echoes the prompt’s last user message, which might not be very useful, but at least it shows the plumbing works).

You should now have a fully functional macOS application that goes well beyond a simple text editor – it’s a **multimodal screenplay IDE**: - Write and save scripts in a standardized format. - Leverage an AI assistant to help write or analyze the script, with streaming feedback. - Visualize the script as an animated storyboard, complete with timing and audio, which can be invaluable for understanding the

rhythm of scenes or for presentations. - All running locally, keeping your creative data private (FountainStore ensures data is local and ACID-safe [25] [3]).

Conclusion

In this tutorial, we integrated **Teatro, Codex, FountainStore, FountainAI, and MIDI2** to build a unique screenplay editor. We covered setup, UI construction, persistence, multimedia synchronization, and AI streaming. This demonstrates the powerful synergy of these components – each designed to be open, scriptable, and developer-friendly – resulting in a feature-rich application without needing to reinvent the wheel for rendering or data storage.

Moving forward, you can extend this foundation with many enhancements: - A more sophisticated text editor for Fountain (syntax highlighting, auto-complete of character names, etc.). - Using the **Fountain parser** in Teatro to display a formatted script view side-by-side with raw text. - Improved **media integration**, e.g., support images or video clips as storyboard elements (Teatro's view system could likely embed images). - Collaboration features, saving versions or using FountainStore's snapshots to implement "undo" across sessions. - Additional AI capabilities: e.g., a character dialog coach, or using FountainAI's function-calling system to let the AI modify the script via defined operations.

Feel free to explore the documentation of each component for deeper dives: - Teatro's docs for advanced rendering, custom animations, and SSE token streams [44] [28] . - FountainStore's docs for performance tuning, using secondary indexes, and data backup strategies [40] [45] . - FountainAI's guides for connecting to various LLM providers and maximizing the use of personas and tools in text generation [46] . - MIDI2's examples for complex MIDI messages, MPE, or using the Core MIDI bridge to interface with external devices [47] [48] .

By combining these, you have the ingredients to build not just a screenplay editor, but any interactive storytelling application that blends text, AI, and multimedia. Happy coding, and happy writing!

Sources:

- Teatro README and Docs – for usage of the Render API and Storyboard DSL [1] [2] .
- FountainStore README – for understanding the persistence engine's role and setup [4] .
- FountainAI README and Q&A – for GUI scaffolding and integration of AI services [11] [8] .
- MIDI2 README and Examples – for constructing MIDI sequences and using the Apple bridge [29] [49] .
- FountainAI GUI Roadmap – for context on streaming token visualization and capabilities in the GUI [50] .

---

1 26 41 RenderAPI.md
https://github.com/Fountain-Coach/Teatro/blob/2ca7bf9bf39244a6e77a6025f5bf6761a5416857/Docs/RenderAPI.md

2 14 15 16 27 34 10_StoryboardDSL.md
https://github.com/Fountain-Coach/Teatro/blob/2ca7bf9bf39244a6e77a6025f5bf6761a5416857/Docs/Chapters/10_StoryboardDSL.md

3 11 README.md
https://github.com/Fountain-Coach/the-fountainai/blob/7b9e624cfbea218c70994e5dc55319c1a234c9f5/README.md

4 17 25 40 45 README.md
https://github.com/Fountain-Coach/Fountain-Store/blob/c182b976f872d11b2e89bcc5187776fb2dea0ce3/README.md

5 32 44 README.md
https://github.com/Fountain-Coach/Teatro/blob/2ca7bf9bf39244a6e77a6025f5bf6761a5416857/README.md

6 README.md
https://github.com/Fountain-Coach/the-fountainai/blob/7b9e624cfbea218c70994e5dc55319c1a234c9f5/Examples/HelloFountainAITeatro/README.md

7 8 12 13 43 46 GUI_App_Scaffolding_QA.md
https://github.com/Fountain-Coach/the-fountainai/blob/7b9e624cfbea218c70994e5dc55319c1a234c9f5/docs/Q&A/GUI_App_Scaffolding_QA.md

9 10 37 38 39 new-gui-app.sh
https://github.com/Fountain-Coach/the-fountainai/blob/7b9e624cfbea218c70994e5dc55319c1a234c9f5/Scripts/new-gui-app.sh

18 19 20 23 24 Store.swift
https://github.com/Fountain-Coach/Fountain-Store/blob/c182b976f872d11b2e89bcc5187776fb2dea0ce3/Sources/FountainStore/Store.swift

21 22 FTS.swift
https://github.com/Fountain-Coach/Fountain-Store/blob/c182b976f872d11b2e89bcc5187776fb2dea0ce3/Sources/FountainFTS/FTS.swift

28 30 31 33 35 49 11_TeatroPlayer.md
https://github.com/Fountain-Coach/Teatro/blob/2ca7bf9bf39244a6e77a6025f5bf6761a5416857/Docs/Chapters/11_TeatroPlayer.md

29 main.swift
https://github.com/Fountain-Coach/midi2/blob/fec93dae9efbb4f1d56984eb0889983962a1dbab/Examples/SequenceExportDemo/Sources/SequenceExportDemo/main.swift

36 42 50 GUI_Development_Roadmap.md
https://github.com/Fountain-Coach/the-fountainai/blob/7b9e624cfbea218c70994e5dc55319c1a234c9f5/docs/GUI_Development_Roadmap.md

47 48 README.md
https://github.com/Fountain-Coach/midi2/blob/fec93dae9efbb4f1d56984eb0889983962a1dbab/README.md