ChatGPT

Designing a Declarative Animation API for Fountain Coach

Introduction:
This report evaluates the implementation of a dedicated **declarative Animation API** for the Fountain Coach platform, supporting **2D, 3D, and audible animations**. The goal is to design an OpenAPI-defined service that can store animation definitions in Fountain-Store, be interpreted by FountainKit, and integrate with existing modules (Teatro, SDLKit, LayoutKit). The API should enable external systems to define and trigger animations, including synchronized audio (with spatial sound). Key considerations include the architectural integration, representation of animation sequences, use of standard formats (Lottie, glTF, OpenTimelineIO), real-time playback/editing feasibility in Teatro, and sound synchronization (including spatial audio). The following sections provide a detailed technical evaluation.

1. Architectural Design and Integration

Proposed Architecture: The Animation API would follow Fountain's spec-first, modular design, similar to LayoutKit and SDLKit. We envision a new **Animation Service** (e.g., an `AnimationKit` module) defined by an OpenAPI 3.1 specification. This spec would describe endpoints and schemas for creating, updating, retrieving, and playing animation sequences. Using OpenAPI aligns with existing patterns – for example, SDLKit's JSON control API is defined via an OpenAPI spec and code-generated for Swift servers/clients [1]. The Animation API would likewise have a YAML spec (e.g., `animation.yaml`), with Swift types and server stubs generated by Apple's OpenAPI generator (ensuring consistency and evolvability). This service could run as a microservice (HTTP+JSON) or be embedded in-process like other Fountain modules.

Integration with Existing Repositories:
- **Fountain-Store:** Animations would be stored as records in Fountain-Store (the persistence engine). Each animation could be a document in a dedicated collection (e.g., "animations"), containing a declarative JSON payload describing the timeline, scenes, and assets. By making the API declarative, entire animation specs can be versioned and persisted easily. Fountain-Store's role is to provide ACID storage and query for these records [2] [3]. The Animation API layer would use Fountain-Store's client or HTTP API to save and retrieve animation documents. This ensures animations are shareable and persistent across sessions.
- **FountainKit:** Acting as the orchestrator, FountainKit would include the logic to interpret the stored animation definitions and interface with runtime components. FountainKit's modular workspace could host the Animation service implementation (likely as a new `FountainServiceKit-Animation` package). The FountainSpecCuration component would manage the OpenAPI spec for the Animation API alongside other service specs [4] [5], enabling the fountain toolchain to generate clients and documentation.
- **Teatro (Front Theater):** Teatro is the front-end visualization and playback environment (with CLI/GUI roles [6]). It would integrate the Animation API in two ways: (1) **Design-time** – possibly providing an editor UI or command interface for creating animation sequences (e.g. a timeline editor or DSL usage) which then calls the Animation API to store definitions; (2) **Runtime** – loading an animation spec (via FountainKit) and playing it back in sync with other content. Teatro already hosts a plugin system for renderers [6], so an Animation plugin can be added to load an animation document and use the appropriate renderer (2D canvas or 3D engine) for playback. Teatro's existing `TeatroPlayerView`

supports playing a series of frames with controls (play/pause/reset) [7]; this could be extended or complemented with a new player that handles continuous animations and keyframes (including 3D scenes and audio).
- **SDLKit:** SDLKit provides cross-platform windowing, a rendering canvas for 2D, and audio output [8]. It can serve as the **runtime rendering engine** for animations. For 2D animations, SDLKit's immediate drawing API and texture support can render frames or sprite sequences. For 3D, SDLKit can hand off a native rendering context (e.g. a Metal layer or OpenGL context) to specialized rendering logic – SDLKit can fetch *CAMetalLayer* on macOS or a Vulkan/GL surface on other platforms [9]. This means the Animation module could create a 3D rendering pipeline (using SceneKit, RealityKit, or a custom OpenGL/Metal engine) and embed it into the SDLKit window. SDLKit's audio playback (already used by TeatroSampler) would handle sound output, ensuring that any audio tracks in the animation stay in sync with the visuals. Integrating through SDLKit keeps the playback cross-platform and consistent with how LayoutKit renders to screen [10] [11].
- **LayoutKit:** For 2D vector-based animations or UI scene transitions, LayoutKit can be leveraged to generate vector scenes (`Scene` objects) from high-level specifications [12]. For example, if an animation involves transitioning between UI layouts or pages, LayoutKit can compute each state as a vector scene. The Animation API could accept high-level descriptions (e.g. "transition this UI from state A to B") and delegate to LayoutKit to get the initial and final scenes. Then, the Animation engine can interpolate or cross-fade between those scenes. In this way, LayoutKit provides the *static layout snapshots* while the Animation system handles the temporal interpolation. The output of LayoutKit (a `Scene` display list) can be rendered via SDLKit's canvas or converted to SVG/PNG for export [13]. This separation ensures clarity: LayoutKit for layout, AnimationKit for time-based changes.

Component Interaction: The diagram below summarizes the integration (components and their roles):

Component	Role in Animation System
Animation API Service	Defines the OpenAPI contract (JSON schema for animations). Handles external requests to create/update/play animations. Uses Fountain-Store for persistence.
Fountain-Store	Stores animation documents (JSON). Provides transactional updates and queries for animations [2] [3].
FountainKit	Core logic layer: implements Animation service handlers (parsing JSON into internal models), and coordinates with Teatro/SDLKit. Bridges stored specs to runtime objects.
Teatro (Front-End)	Provides UI/CLI to define animations (possibly via a DSL or timeline editor) and to preview/play them. Uses FountainKit to load animation data and either passes frames to `TeatroPlayerView` or initiates a custom player for 3D. Ensures integration with the user's session or AI agent workflow.
LayoutKit	(Optional) Generates vector scene snapshots for 2D states; assists in interpolation for UI/layout animations [14].
SDLKit	Executes the rendering and audio output: opens windows/canvases, draws 2D frames, or hosts 3D contexts [8]. Also plays audio streams for soundtracks.

This architecture is **extensible** and aligns with existing design patterns. By using an OpenAPI spec as the source of truth, the Animation API can be iteratively expanded while clients (and AI agents) rely on a stable contract. It fits into the microservice layout of FountainKit (similar to how there are planner, tool, and persist services [15]). The Animation service could be invoked by AI planning modules to

automatically generate animations (for instance, the AI could decide on a storyboard and call the API to materialize it), or by external creative tools for content creation.

2. Declarative Animation Sequence Representation

A declarative approach is crucial – animation sequences will be described as **data** (JSON documents) rather than code. Fountain's existing **Storyboard DSL** provides a starting point for thinking declaratively: it allows defining sequences of scenes and transitions between them [16]. In the DSL, a `Storyboard` is composed of named `Scene` states and `Transition` steps (e.g. crossfades or tweens with given duration in frames) [14]. This yields a timeline of frames that can be expanded and played back [17]. However, the current storyboard model is relatively simple (supporting only crossfade and basic tweening between static scenes). The new Animation API must represent more complex sequences, including fine-grained keyframes, 3D transformations, and synchronized audio cues.

Proposed Representation Model: We recommend structuring the animation spec around a **timeline with tracks and keyframes**, drawing inspiration from industry standards:

- **Timeline and Tracks:** The animation JSON could have a top-level timeline duration and a set of tracks (or layers). Each track could represent a distinct animated entity or channel – for example, one track for a 2D UI scene, another for a 3D model animation, another for audio. This is similar to how OpenTimelineIO models timelines with multiple tracks (video, audio, etc.) and transitions between clips [18]. An "Animation" record might contain an array of `tracks`, each with a type (2D, 3D, audio) and a sequence of keyframe segments or references to asset files. For simplicity, tracks can be time-synchronized (all using the same time base, e.g., milliseconds or frames).

- **Keyframes and Transitions:** Within each track, animations can be specified by keyframes (for continuous property animations) or clips (for media segments). For example, a 2D track might say: at time 0 show Scene A, at 2s transition to Scene B by crossfade over 0.5s. A 3D track might specify: load Model X, at t=1s start animation "Run" on that model (which plays for 3s), etc. We can support both explicit keyframes (numerical values for properties at specific times) and high-level transitions (like "crossfade" which implicitly affects opacity over a duration). The DSL's concept of `Transition(style, frames)` would be generalized to support more styles (move, rotate, scale, etc., especially for 3D) and tied to real timeline durations (frames or seconds) [16].

- **Scenes and Assets:** The spec should allow referencing **scenes or assets** by ID. For 2D, a "scene" might be a Fountain UI view hierarchy (which could be stored as a separate artifact or included inline – possibly reusing the format of LayoutKit's `Scene` or the Storyboard DSL syntax). For 3D, an asset reference might point to a 3D model file (like a glTF model) stored in Fountain-Store or a URL. The animation track can then refer to animations defined within that model (e.g., play animation clip named "walk" from the glTF). We might define an `Asset` type in the API with fields like `type: 2D|3D|audio`, `uri` or `data`, etc., which could be uploaded or linked.

- **Declarative Sound Cues:** Audio tracks would contain references to audio files or sequences (e.g., a WAV/MP3 file, or a MIDI sequence for music). Instead of controlling audio imperatively, the timeline can declare: at 0s, play this audio clip; or attach this audio to a 3D position. This way, sound is just another track in the JSON spec, which the runtime will handle.

Example JSON Sketch: (illustrative format)

```
{
  "id": "anim001",
  "duration": 10000,  // in milliseconds
  "tracks": [
    {
      "id": "track1",
      "type": "2D",
      "keyframes": [
        { "time": 0, "scene": "SceneA" },
        { "time": 2000, "transition": "crossfade", "duration": 500,
"toScene": "SceneB" },
        { "time": 2500, "scene": "SceneB" }
      ]
    },
    {
      "id": "track2",
      "type": "3D",
      "asset": "model_dragon",
      "actions": [
        { "time": 1000, "playAnimation": "TakeOff", "duration": 3000 },
        { "time": 4000, "playAnimation": "FlyLoop", "loop": true }
      ]
    },
    {
      "id": "track3",
      "type": "audio",
      "clips": [
        { "time": 0, "asset": "music_intro", "volume": 0.8 },
        { "time": 3000, "asset": "sfx_roar", "spatial": true, "position": [0,
0,0] }
      ]
    }
  ],
  "assets": [
    { "id": "SceneA", "type": "2DScene", "data": { ... UI layout ... } },
    { "id": "SceneB", "type": "2DScene", "data": { ... UI layout ... } },
    { "id": "model_dragon", "type": "3DModel", "uri": "fountain://store/
models/dragon.glb" },
    { "id": "music_intro", "type": "audio", "uri": "fountain://store/audio/
intro.mp3" },
    { "id": "sfx_roar", "type": "audio", "uri": "fountain://store/audio/
roar.wav" }
  ]
}
```

In the above hypothetical structure, we have: two 2D scenes (A and B) with a crossfade transition, a 3D model ("dragon") on which two animations are played in sequence, and audio tracks for background music and a sound effect. This illustrates how a single declarative spec can encompass 2D, 3D, and audio in a unified timeline. The Animation API's job is to define and validate this schema (ensuring references and timing make sense), and to provide endpoints like `POST /animations` (to create or update such a spec in the store) and `GET /animations/{id}` (to retrieve or maybe stream the sequence).

Interpretation by FountainKit: FountainKit (and Teatro runtime) would interpret this JSON by constructing the necessary runtime objects: - Build `Renderable` views or LayoutKit Scenes for the 2D parts (using existing DSL parsing for the “data” of scenes A and B) [19] . - Load the glTF model for the 3D part (likely using an engine or framework to parse `.glb`). - Prepare audio players for the audio clips (using TeatroSampler or SDL audio API). - Then schedule these elements on a timeline scheduler. Since Fountain already uses MIDI timing for sync, one approach is to map the timeline to a MIDI sequence under the hood (e.g., each keyframe or clip corresponds to a MIDI note or timestamp) – indeed the existing `TeatroPlayerView` uses a `MIDISequence` of MIDI 2.0 notes to drive frame durations [7] . Alternatively, use a simple clock/loop to advance time. The key is the **declarative spec drives the creation of the timeline**, rather than imperative code.

Comparison to OpenTimelineIO: OpenTimelineIO (OTIO) could serve as a conceptual model for our timeline. OTIO is an interchange format that supports clips, tracks, transitions, markers, etc., but it doesn’t embed the actual media [18] . That aligns well with our needs: the animation spec references media (scenes, models, audio), which are stored separately (or in sub-records), similar to how OTIO keeps video/audio media external. We could even choose to make our JSON **compatible with OTIO’s schema**, allowing export/import from professional video tools. For instance, an OTIO timeline could represent our animation’s edit structure, where 2D/3D scenes might be treated as clips (perhaps via custom OTIO schemas for interactive content). However, OTIO by itself doesn’t describe *how to animate within a clip* (e.g., moving an object); it’s more about sequencing media. Therefore, our spec might borrow OTIO’s concepts for the high-level timeline (tracks, clips, transitions), while we handle the in-clip animations (like keyframe data or referencing internal animations in glTF). Embracing OTIO could pay off in long-term interoperability but might be overkill for initial implementation.

3. Leveraging Standard Formats (Lottie, glTF, OpenTimelineIO)

To maximize compatibility and avoid reinventing wheels, the Animation API should incorporate existing **formats/standards** for various aspects of animation:

- **Lottie (2D Vector Animations):** *Lottie* is a JSON-based vector animation format from Airbnb. It encodes After Effects animations (shapes, layers, keyframes) in a highly efficient, cross-platform way [20] . Lottie is resolution-independent (vector graphics) and much lighter than GIFs [21] . We recommend supporting Lottie for complex 2D animations like illustrations or icon animations. For example, if a user/AI wants to include a fancy animated graphic (spinning logo, animated chart, etc.), they could upload a Lottie JSON as an asset. The Animation API could either **store Lottie JSON** directly as an asset type or even allow embedding it in the timeline spec. Fountain’s runtime can then use a Lottie player/renderer to render it – possibly by integrating a library (there are C++ or Swift implementations for Lottie). Another approach is converting Lottie to our internal `Renderable` frames: since Lottie essentially describes vector drawing primitives over time, an advanced integration could parse Lottie JSON and translate it into Frame sequences or SDL draw calls. However, leveraging an existing player (like `rlottie` or `lottie-ios`) might be faster. **Pros:** Widespread format, many design tools export to Lottie; great for high-quality 2D animations (UI, illustrations) [22] . **Cons:** Focused on vector graphics – no direct 3D or audio support, and the JSON is not very human-readable (short keys for size) [23] . Still, given Lottie’s popularity, supporting it would allow designers to bring their work into FountainCoach easily.

- **glTF (3D Models & Animations):** *glTF (GL Transmission Format)* is an open standard from Khronos for 3D scenes and models. It supports 3D geometry, materials, textures, **scene hierarchy and animations (skeletal or morph target animations)** [24]. glTF is designed to be efficient for runtime delivery and is often called the "JPEG of 3D" [24]. For the Animation API, glTF is the ideal way to handle 3D content. We can allow users to upload glTF models (and related assets like textures) to Fountain-Store or reference external URLs. The animation spec can then refer to these models and specify which animation clips (if the glTF contains multiple animations) to play, or directly manipulate the scene graph (e.g., move a node along a path). By using glTF, we tap into a huge ecosystem: many tools (Blender, Maya, etc.) export glTF, and engines (Three.js, Babylon.js, Unity via import) can read it. **Integration:** On Apple platforms, we could load glTF via ModelIO or SceneKit (SceneKit has built-in support for loading Collada and potentially glTF with some converters, or use third-party Swift libraries for glTF 2.0). Alternatively, since SDLKit can provide a Vulkan/Metal surface, we might integrate a lightweight renderer that reads glTF (there are C++ libs like tinygltf plus a rendering codepath). Initially, using SceneKit/RealityKit may be quickest: e.g., feed the glTF into RealityKit (by converting to USDZ if needed – Apple's RealityKit prefers USDZ, but there are converters from glTF). Another option is embedding a WebGL engine via WebView for rendering glTF with Three.js, but that adds complexity and isn't as native. **Pros:** glTF is an efficient, interoperable 3D format supporting animations and PBR materials [24]. It would allow Fountain to present rich 3D content (e.g., characters, environments) with minimal custom format work. **Cons:** glTF doesn't include audio, and mixing glTF animation with our timeline requires us to manage the timing (glTF's animations have their own keyframes that we need to trigger at the right global time). Also, to use glTF fully we need a rendering engine that handles 3D drawing, lighting, etc. – using SceneKit/RealityKit ties us to Apple platforms, while a custom engine is non-trivial. A pragmatic path is to use SceneKit on Apple and maybe embed an OpenGL renderer on other platforms (since SDLKit can get us an OpenGL context). Over time, we could abstract this so that the AnimationKit chooses the appropriate 3D backend based on environment (e.g., SceneKit if available, else fallback to a minimal GL renderer with fixed function or custom shader like the Metal sample triangle we have).

- **OpenTimelineIO (Timeline Editing):** *OpenTimelineIO (OTIO)* is a format and library from Pixar for editorial timelines – essentially an advanced form of an Edit Decision List (EDL) that includes tracks, clips, transitions, markers, and so on [18]. While OTIO does not carry the actual media, it is excellent for describing the structure of a timeline (order of shots, layering of audio, etc.). Incorporating OTIO could be done in two ways: **internally** or **externally**. Internally, we might use OTIO's data model (or a subset) to structure our Animation timeline. For example, we could map our JSON spec to an OTIO `Timeline` object (with `Track` for each track, `Clip` for each media piece, and `Transition` for crossfades). This could be beneficial if we leverage OTIO's existing library to parse/format timelines and perhaps use its API for editing operations (like trimming, swapping clips). Externally, we could allow **import/export** of OTIO files: e.g., export a Fountain animation as an `.otio` file so that a video editor could load it (though that editor wouldn't know how to handle a "3D model clip," we might represent 3D content as filler with metadata). Another external integration: if the user has a complex sequence prepared in a video editing tool, they might import it via OTIO to get the timing and basic structure, then attach actual interactive content in Fountain. **Pros:** OTIO is a proven standard for timeline data, encouraging interoperability across tools. It cleanly separates content from timing, which fits our need to reference external assets [18]. **Cons:** It doesn't intrinsically support interactive content (it's video-focused), so we'd have to extend or misuse some fields to represent 3D scenes or interactive states. Also, OTIO is relatively heavy-weight (and mainly a Python/C++ library); using it might introduce a dependency that we need to manage. For the first iteration of the Animation API, we can borrow ideas from OTIO (like how to structure tracks and transitions) without fully implementing OTIO support. We should, however, design our JSON with an eye towards future OTIO compatibility, as that could open doors to integration with editing software.

Summary of Formats: The table below summarizes these recommended standards and their relevance:

Standard	Purpose in Fountain Animation	Strengths	Limitations
Lottie (JSON) [20]	2D vector animations (UI, illustrations). Can embed designer-created animations (from After Effects via Bodymovin).	Rich, resolution-independent vector graphics; widely supported on web and mobile; lightweight alternative to GIF [21] .	No native 3D or audio; complex animations might tax CPU if not hardware-accelerated; requires integrating a Lottie renderer.
glTF (GL Transmission Format) [24]	3D models and animations (characters, 3D scenes). Used for meshes, materials, and skeletal animations.	Open standard for 3D scenes; supports meshes, materials, *scene graph and keyframe animations* [24] ; efficient binary form (.glb) for runtime; many export/import tools.	No audio; needs a 3D rendering engine to display; may require conversion on Apple platforms (to SceneKit's format); handling high-poly models or complex lighting might be beyond Fountain's scope without external engine help.
OpenTimelineIO (OTIO) [18]	Timeline structure and editing interchange. Represents sequence of clips/ tracks with transitions, for combining media in time.	Well-suited for editing workflows; supports multi-track, transitions, markers, etc.; good for synchronizing multiple media types on a timeline.	Doesn't carry media content (only references) [25] ; primarily video/film oriented (our "interactive" content would be custom); library might be heavy to integrate for real-time use; not needed if custom timeline is simple.

By leveraging these standards, the Animation API can avoid starting from scratch. In practice, we might implement support incrementally: **Phase 1** could focus on core JSON schema and basic 2D/3D keyframe features, **Phase 2** add import of Lottie/glTF assets in the pipeline, and **Phase 3** consider OTIO import/ export for pro editing integration. Wherever possible, we keep the system open: for instance, if an animation is essentially a Lottie file with music, we might just store the Lottie (which covers the visual part) and attach an audio track. If it's a glTF, we store the glTF and an OTIO timeline for edits, etc. The open API ensures external tools (and even AI agents) could feed in these formats. For example, an AI could fetch a Lottie JSON from LottieFiles and POST it to our API to include it in an animation; or a user could upload a .glb model to add a 3D element.

4. Runtime Playback and Editing in Teatro

Playback Feasibility: Running the animations at runtime within Teatro is feasible with careful coordination of the aforementioned components. Teatro already demonstrates *real-time playback* of frame sequences via `TeatroPlayerView`, which takes an array of rendered frames and a MIDI timing sequence to play them with audio sync [7]. Extending this to a more dynamic animation timeline involves a few changes: - Instead of pre-rendered frames only, the player needs to handle continuous updates of scene state (especially for smooth 3D or 2D property animations). We may implement an **AnimationPlayerView** that advances time on a ticker (e.g., 60 FPS display link or driven by MIDI clock ticks) and at each step computes the next frame from the declarative model. For 2D UI transitions like crossfades, this means blending the two `Renderable` scenes per the fraction of progress. For keyframe animations (e.g., move an element along a path), it means interpolating properties (position, opacity, etc.) between keyframes. In 3D, it means advancing the glTF animation or manipulating the scene graph nodes for the current time. This is more complex than the frame-by-frame approach, but it's doable with either a small animation engine or leveraging existing ones (SceneKit's `SCNTransaction` or `SCNAction` could animate properties if we feed it keyframes, or we update manually each frame).

- **Teatro & SDLKit UI integration:** Since Teatro is likely a macOS/iOS app (with SwiftUI or similar for UI) plus SDL for drawing, the AnimationPlayerView can be implemented as an SDLKit canvas embedded in the app UI. SDLKit can already create a window or view for drawing; on macOS it can integrate with NSView via Metal layer. Teatro's GUI should allow an area where the animation is previewed (for instance, a "stage" view). The user (or AI agent through the UI) can hit Play to start the animation. The API provides the data, and the player takes over to execute it. We can reuse the controls from `TeatroPlayerView` – play, pause, reset – and extend them to also scrub the timeline if needed. Because we have a known timeline duration, implementing a scrubber (time slider) is possible, which is helpful for editing.

- **Editing capabilities:** In initial versions, editing might be text-based or via the DSL (users writing scenes and transitions code, or an AI agent generating them). However, an ultimate goal could be a **visual timeline editor** in Teatro. With the declarative spec as the backbone, Teatro could present a timeline UI: tracks stacked vertically, time on the horizontal axis, clips/keys drawn accordingly (similar to video editing or Keynote's animation timeline). The user could drag scenes or adjust durations, and under the hood Teatro would update the JSON spec (via the API or directly if local) and the changes would reflect in playback. This is a more advanced UI/UX task, but the data model supports it. Even without a full GUI editor, users can edit animation JSON or use the DSL and re-run playback. Also, because the spec is OpenAPI-defined, external editing tools could be built (for example, a web-based timeline editor that calls the API to save changes, which Teatro then syncs from Fountain-Store).

- **Performance considerations:** Real-time playback of complex animations (especially 3D) means we must be mindful of performance. Using native frameworks like SceneKit/RealityKit for 3D leverages hardware acceleration for rendering and animation; those frameworks are optimized for real-time (SceneKit can handle moderately complex scenes at 60 fps). If using a custom GL/Metal path, we might need to implement frustum culling, optimize draw calls, etc., which is a heavy lift. So leaning on existing engines initially is wise. For 2D, SDLKit's renderer is hardware-accelerated (SDL uses GPU for rendering textures, lines, etc., when possible) and should easily handle typical UI animations or Lottie rasterization if done properly. We might also consider simplifying the rendering when the animation is driven by AI generation (e.g., if frames are for GPT's reasoning, they can be low-res or simplified visuals). Since Teatro has been used for GPT planning visuals (as mentioned, frame-by-frame reasoning steps) [26], adding continuous animations should not break that – rather it enriches the output.

- **Concurrency:** To avoid stalling the UI, the animation player should run in a loop that doesn't block user interaction. SDLKit is main-thread oriented for drawing (MainActor for SDLCore) [27], but audio playback and other tasks can be separate threads. We might use the tick from the audio/MIDI as the driver: e.g., each MIDI tick triggers a frame advance (which is how TeatroPlayerView syncs with music). If no MIDI, we use a display link or timer. Ensuring the timeline remains in sync (no drift between audio and video) is crucial – more on this in the next section.

Editing Workflow in Teatro: In an interactive setting, an author could follow this flow: create a new Animation via a GUI form or code – e.g., define scenes and keyframes with helper tools – and save it (which calls the Animation API to persist). Then use Teatro's preview to play it. If adjustments are needed, edit and re-run. Because the system is declarative, small changes (like duration of a transition, or swapping an asset) only require changing the data, not rewriting large code. We can also imagine **AI assistance**: the AI agent (via Codex) could propose an animation sequence, output it as JSON or DSL text, and the user can preview it immediately. The Storyboard DSL is already designed for AI to reason about sequences [28]; the Animation API takes it further by supporting richer media. The synergy between an AI agent and a declarative animation spec is powerful: the agent could fill in an AnimationSpec (perhaps using a natural language to JSON tool) and the system would render it. Teatro's role would then be to facilitate this loop, showing previews and allowing corrections.

Runtime Example: Suppose we have an animation that crossfades between two app UI screens while a 3D mascot character waves, and a sound plays. The JSON is stored in Fountain-Store. The user hits "Play" in Teatro. Under the hood: 1. FountainKit retrieves the animation JSON (either it was already loaded or via an API call). 2. It loads necessary assets: e.g., prepares two UI `Scene` objects for the screens (with LayoutKit or by constructing the SwiftUI-like views via the DSL), loads the 3D model (e.g., a .glb for the mascot) into a SceneKit scene, and loads the audio file (AVAudioPlayer or SDL audio buffer). 3. The AnimationPlayerView starts a timeline of, say, 5 seconds total. At t=0, it renders Scene A (2D) via SDLKit's canvas on the window, and also shows the 3D model (through a Metal layer possibly overlaid or composited – or the 3D is drawn into the same SDL window if we integrate at low-level). Audio starts playing. 4. As time advances, by t=2s it begins crossfade: the player mixes Scene A and Scene B visuals (this could be done by gradually increasing Scene B's opacity and decreasing Scene A's, or rendering them to textures and blending – SDLKit can handle texture alpha blending). Simultaneously, if the 3D character has an animation (wave) starting at 1s, the player instructs SceneKit to start that animation at the correct time. SceneKit handles the interpolation of the character's skeleton. Our player just ensures it starts at t=1s and perhaps stops or loops it accordingly. 5. The audio track might have, say, a roar sound at 3s attached to the character. At 3s, the player triggers that sound. If spatial audio is on, and if using SceneKit, we could attach an SCNAudioPlayer to the character node with the audio – SceneKit will then handle panning/attenuation as the character moves (if it moves). If using SDL audio, we might simulate stereo panning (if needed, see next section). 6. The timeline completes at 5s; the player either stops or loops as defined.

Throughout, Teatro could display the time or allow jumping to a time (which means setting all elements to the state corresponding to that time – which is doable if we keep references to initial scenes and apply animations up to that time).

This runtime description shows it's quite feasible, leveraging a mix of our infrastructure and existing frameworks. The **feasibility is high** for playback, given that much of the needed functionality exists (rendering 2D frames, playing audio, syncing via MIDI). The main new challenge is integrating a 3D engine; fortunately, SDLKit's design anticipates external rendering (providing native handles [9]). We might render the 3D content offscreen and composite into SDL's renderer as a texture for simplicity, or create a dedicated viewport. Both approaches are technically possible.

5. Sound Synchronization and Spatial Audio Handling

Synchronizing audio with visuals has been a core part of Teatro's design (using MIDI 2.0 timelines) [7]. The new Animation API should continue to ensure tight AV sync, even as capabilities expand.

Timeline-Based Sync: The declarative spec ties events to timeline timestamps, so the master timeline serves as the source of truth for synchronization. Using a single clock for both audio and video prevents drift. One method is to derive a MIDI sequence from the timeline: for example, treat each distinct time event as a MIDI note (with the note's duration matching the length of a visual event). The existing `TeatroPlayerView` takes a `[Renderable]` frames list and a `MIDISequence` of notes (with durations) to time them [7] – that concept can be generalized. If we can map our timeline to a sequence of ticks (e.g., 480 ticks per beat, and some BPM such that 1 tick = 1ms or similar), we could feed a MIDI clock. However, using MIDI might constrain to musical time; instead, we could manage directly in milliseconds or seconds since our content isn't necessarily musical (though if it is, MIDI helps, but we can allow arbitrary time for general animations).

Alternatively, we run a **scheduler loop** where audio playback time is the reference (since audio hardware usually has a stable clock). For instance, when playing a sound or music track, we can regularly query the playback time (many audio APIs allow this) and use it to advance the animation. This way, if the audio lags or speeds up, visuals stay locked. If no continuous audio, we can use system clock or display refresh. In summary, **the approach is to have a single timeline driving both domains**. We must also consider start delays: e.g., ensure that when the user hits play, the audio and visuals truly start together.

Spatial Audio: Spatial (3D) audio means sound sources have position and the listener (camera) position affects volume and panning. If we use a 3D engine like SceneKit or RealityKit, we get spatial audio largely for free. **SceneKit example:** We attach an `SCNAudioSource` to a node (with positional = true), SceneKit will apply distance attenuation and panning based on the camera node's position. SceneKit's audio even supports basic reverb regions, etc. RealityKit similarly can place audio in AR space. Using those frameworks would handle spatial sound elegantly. The Animation API spec would just need to allow specifying a sound with a position (and maybe orientation/spread).

If we are not in an Apple environment or not using those frameworks, we might consider SDL's audio capabilities. SDL itself doesn't have built-in spatial audio, but we can implement a simplified model: for example, if we restrict to stereo output, compute pan (left-right volume) based on the 2D screen position of a sound source or the angle relative to listener in 3D. Volume can be adjusted by distance (simple 1/r falloff or so). For more advanced HRTF (head-related transfer function) or surround sound, external libraries or platform APIs are needed (OpenAL, or Microsoft's spatial sound APIs on Windows, etc.). Given Fountain's scope, it might suffice to do basic stereo panning for spatial cues if not using an engine that supports full 3D audio.

Audio Format and Sync: We should support at least two kinds of audio in the animations: - **Music/ Background tracks:** Possibly longer music that spans the timeline (could be handled via streaming playback from file, e.g., AVAudioEngine or SDL's audio callback). These often define the overall timing (especially if the animation is like a music video or a scored scene). - **Sound effects:** Shorter clips triggered at specific points (like the “roar” or a UI click). These can often be preloaded and played with low latency at the right moment.

The Animation API’s declarative format might include a field for *sync* on audio events, e.g., a flag to indicate that a particular sound should drive the tempo. For instance, if an animation is created for a song, we might want to ensure the timeline’s BPM matches the song’s BPM. We could allow specifying beats or tempo, but this may be too detailed – simply using absolute time likely suffices.

One area to consider is **latency**: If using MIDI, note events and audio might have slight delays. If using AVFoundation, starting playback of an audio file can have slight latency. We can mitigate this by starting audio a tick early or by using precise scheduling (AVAudioEngine allows scheduling buffers at specific times). The system might also query audio playback position frequently and adjust (e.g., drop or duplicate a frame if needed to resync – though ideally we avoid that).

Ensuring Cross-Platform Sync: TeatroSampler was designed to keep audio and visuals in sync across platforms [29]. It uses an actor-based MIDI 2.0 sampler to produce sound with precise note durations. For arbitrary audio files, we might not have such granular control, but we can still take advantage of the fact Fountain-Store and the services run locally (so low-latency). We could even convert an audio file’s timeline into a MIDI representation (e.g., a beat track) if needed, but that’s likely unnecessary. Instead, in a cross-platform manner, we rely on SDLKit’s audio if on non-Apple (which uses SDL3’s audio stream API, likely offering callback-based continuous output [30]). On Apple, AVAudioEngine or AudioQueue can be used. In all cases, driving the visual timeline from the audio callback thread (or a high-priority timer) might yield better sync than an event loop on the main thread.

Spatial Audio Data in API: To support spatial audio declaratively, we include spatial parameters in the spec for audio events: e.g., for an audio clip, allow an optional position (x, y, z) and perhaps a reference to which 3D object it’s attached to (or to the camera for UI sounds). Also parameters like volume, looping, etc. This data would be fed into the engine: if using SceneKit, we’d attach audio accordingly; if manual, we’d compute pan/volume. We might also include a simple flag for 2D vs 3D sound. If `spatial: true` and we have a 3D context, we treat (x,y,z); if `spatial: false`, just play normally (maybe stereo as is).

Example: In the earlier JSON example, the roar had `"spatial": true, "position": [0,0,0]` meaning it should emit from the origin of the 3D scene. If our camera is not at origin, the user would hear it offset appropriately. If the character moves and roars, ideally the sound moves with it – to achieve that, we might instead specify `attachTo: "model_dragon"` in the spec for that clip, meaning the sound follows that model’s position.

In summary, the Animation API will treat audio as first-class timeline elements to be synced with frame updates. By using consistent timing mechanisms and leveraging spatial audio capabilities of underlying engines (or implementing basic spatialization), we can ensure that audible components of an animation feel integrated. Visual frame updates and audio waveforms will progress in lockstep. The result will be animations that *look and sound cohesive* – e.g., a character’s movement matches the music beat, or a door slam sound happens exactly as the door closes in the animation.

Conclusion: The introduction of a declarative Animation OpenAPI for Fountain Coach is a **natural evolution** of the platform, bringing together 2D UI sequences, 3D content, and audio into a unified system. Architecturally, it extends existing modules (Store, Kit, Teatro, SDLKit, LayoutKit) with a focused service that manages animations as data. By adopting and integrating standard formats like Lottie for vector animations [20], glTF for 3D scenes [24], and referencing OpenTimelineIO for timeline structure [18], FountainCoach can avoid reinventing the wheel and ensure compatibility with external tools. The runtime feasibility analysis shows that with careful use of SDLKit (for rendering surfaces and audio) and possibly SceneKit/RealityKit (for 3D engine support), the platform can achieve real-time playback and even interactive editing of animations within Teatro. Audio-visual synchronization, already a strength of Teatro's design [7], will remain robust, with potential enhancements for spatial sound to increase immersion.

In implementing this Animation API, the development should prioritize **modularity** (clear separation of spec, engine, and rendering), **performance** (efficient handling of frames and assets), and **usability** (making it easy for users or AI agents to create rich animations via high-level declarations). The outcome will empower Fountain AI and human creators alike to script complex multimedia narratives – from interactive tutorials with animated characters to AI-generated theater pieces – all within the Fountain Coach ecosystem, using open standards and modern practices as the backbone.

Sources:

- Fountain Teatro Docs – *Storyboard DSL and Animation System* [14] [7]
- SDLKit Documentation – *OpenAPI approach and SDL integration* [31] [9]
- LayoutKit Documentation – *Spec-first design and Canvas integration* [32] [13]
- Wikipedia – *Lottie format (vector animation JSON)* [20]
- Wikipedia – *glTF format (3D scenes and animation)* [24]
- OpenTimelineIO Docs – *Overview of timeline editing format* [18]

---

[1] [8] [9] [27] [30] [31] README.md
https://github.com/Fountain-Coach/SDLKit/blob/9fcb3a8aa124f4d1817e96f19c92fac1f3a6fa73/README.md

[2] [3] openapi-fountainstore.yaml
https://github.com/Fountain-Coach/Fountain-Store/blob/50654f4d799ff9b5e7bba0c1df57e03c89c56ef1/docs/openapi-fountainstore.yaml

[4] [5] [15] README.md
https://github.com/Fountain-Coach/FountainKit/blob/56386fceb449d0c4344d0ce28ac7188119c6fde4/README.md

[6] [10] [11] [12] [13] [32] README.md
https://github.com/Fountain-Coach/LayoutKit/blob/12fab59022eb117ca5299905aabbfe2fd6a8a9b1/README.md

[7] [26] [29] 05_AnimationSystem.md
https://github.com/Fountain-Coach/Teatro/blob/2da76dc54ae6328bffe64087888c9f0de3c9edff/Docs/Chapters/05_AnimationSystem.md

[14] [16] [17] [19] [28] 10_StoryboardDSL.md
https://github.com/Fountain-Coach/Teatro/blob/2da76dc54ae6328bffe64087888c9f0de3c9edff/Docs/Chapters/10_StoryboardDSL.md

[18] [25] Welcome to OpenTimelineIO's documentation! — OpenTimelineIO 0.18.0.dev1 documentation
https://opentimelineio.readthedocs.io/en/latest/

[20] [21] [22] [23] Lottie (file format) - Wikipedia
https://en.wikipedia.org/wiki/Lottie_(file_format)

[24] glTF - Wikipedia
https://en.wikipedia.org/wiki/GlTF