ChatGPT

AnimationKit OpenAPI Specification and Swift Package Scaffold

Overview and Spec-First Approach

AnimationKit is a new microservice in the Fountain ecosystem designed for creating and managing animation timelines synchronized with MIDI 2.0 events. The API is defined in an OpenAPI 3.1 specification (YAML) as the single source of truth, following Fountain's declarative, spec-first methodology [1]. From this spec, Apple's Swift OpenAPI Generator will produce Swift types, a client API, and a server stub that runs on SwiftNIO (without Vapor). This ensures a consistent contract and codegen-driven implementation [2] [3]. The server will integrate with existing Fountain components: it will use the **midi2** library for Universal MIDI Protocol (UMP) event handling, **Fountain-Store** for persistence, and **swift-secretstore** for secure authentication.

Below we detail key aspects of the design, then provide the OpenAPI spec and a scaffold of the generated Swift package.

Timeline Model and UMP Integration

Timeline Representation: Animation timelines are represented as JSON objects containing an ordered list of **events** aligned with the Universal MIDI Packet (UMP) format. Each event includes a timestamp and a MIDI 2.0 message. The JSON schema for events is derived from Fountain's MIDI2 infrastructure, meaning it matches the structure of MIDI 2.0 UMP messages [4]. In practice, each event is essentially a UMP packet (32, 64, or 128 bits) with an associated time. For example, a Note-On event in JSON carries the same fields (note number, 16-bit velocity, etc.) as defined in the MIDI2 schema [5] [6]. This alignment ensures that timeline data can be directly fed into the Engraver-centric MIDI2 engine for playback, with minimal translation.

Timestamping: Timeline events use a timebase (e.g. seconds or milliseconds) to schedule when each UMP message should occur. Internally, the playback engine may leverage MIDI 2.0 *Jitter Reduction Timestamps* (JR Timestamp UMP packets) to achieve precise scheduling, but clients of this API can simply provide human-readable timestamps. The system will convert these to the proper UMP timing messages as needed. This design leverages the existing MIDI2 timeline execution engine in Fountain – as seen in the Teatro player that uses MIDI 2.0 note durations to time visual frame playback [7] – ensuring that animations and audio stay perfectly in sync.

Schema: In the OpenAPI spec below, we define a `Timeline` schema with fields like `id`, `name`, and an array of `events`. Each event is defined as an `UmpEvent` object containing a `time` (in seconds) and a `message` which conforms to the MIDI 2.0 UMP structure. For brevity, the spec treats the `message` as a oneOf of `Ump32`, `Ump64`, `Ump128` packet formats [8], each of which in turn contains the appropriate fields (group, channel, status, and data bytes) per the MIDI 2.0 spec. This means a timeline event can represent any MIDI 2.0 message – note on/off, CC, program change, etc. – exactly as defined in the MIDI2 library, ensuring **full fidelity with the UMP standard**. The spec is kept in sync with the MIDI2 JSON schema so that any future updates to UMP definitions can be adopted easily.

Endpoints for OTIO Interoperability

To support editorial interoperability, AnimationKit provides endpoints to **import and export timelines in OTIO (OpenTimelineIO)** format. OpenTimelineIO is an interchange format (serialized as JSON) widely used for timeline data in film/video editing [9]. The service can thus bridge between Fountain's MIDI-driven timelines and external editing tools.

- **Import OTIO:** Clients can `POST /timelines/importOTIO` with an OTIO JSON payload to create a new timeline. The server will translate the OTIO timeline (clips, tracks, time ranges) into the equivalent AnimationKit timeline (with UMP events). For example, an OTIO **Timeline** with tracks and clips will be mapped to an AnimationKit timeline with events corresponding to those clips' start/end times or cues. The OpenAPI spec defines this request as accepting `application/json` OTIO data (we treat it opaquely via a generic `OTIO` schema). On success, the API responds with `201 Created` and the newly created `Timeline` object in AnimationKit's native format.

- **Export OTIO:** Clients can retrieve a timeline in OTIO format by calling `GET /timelines/{id}/otio`. This will return an OTIO JSON representation of the specified timeline, allowing the timeline to be loaded into video editing or compositing tools that understand OTIO. Under the hood, the server converts the sequence of UMP-timed events into an OTIO **Timeline** with Tracks and Clips. (For instance, MIDI note events might translate into OTIO clips or markers along a track representing an instrument or animation layer.) The response uses `application/json` content (the OTIO JSON structure), and includes the appropriate OTIO top-level keys like `"OTIO_SCHEMA": "Timeline.1"`, `"tracks"`, `"children"`, etc., per the OTIO specification [10] [11].

These OTIO endpoints make it possible to **round-trip timelines** between Fountain's animation system and other timeline-based tools. A user could import a timeline from a video editor to drive synchronized MIDI/animation playback, or export a procedurally generated MIDI timeline for further editing in an NLE (non-linear editor).

RESTful API for Timeline CRUD and Playback

AnimationKit's API follows RESTful design principles, providing endpoints to create, read, update, delete, list, and control timelines. All endpoints are documented in the OpenAPI spec (see the **Paths** in the YAML below). Key endpoints include:

- **List Timelines:** `GET /timelines` – Returns a collection of timeline records. This can include basic metadata for each timeline (e.g., id, name, duration) and support pagination if needed (though not required in this initial spec). Clients use this to discover available timelines.

- **Create Timeline:** `POST /timelines` – Creates a new timeline with a client-provided JSON body (in the AnimationKit timeline format). The request body includes optional metadata (like a name) and the list of UMP events. On success, returns `201 Created` with the full timeline object (including its assigned `id`). If the client omits an `id`, the server will generate a unique one.

- **Get Timeline:** `GET /timelines/{id}` – Retrieves a single timeline by its identifier. Returns the timeline in JSON format (UMP-aligned), including all events. This allows clients to fetch and inspect timeline contents.

- **Update Timeline:** `PUT /timelines/{id}` – Replaces an existing timeline with a new definition. Clients supply the full timeline JSON in the request. This is useful for editing a timeline (e.g., adding/removing events or changing metadata). A successful update returns the updated timeline. (The spec could also allow partial updates via PATCH, but for simplicity we use PUT with the full resource.) Versioning or concurrency control could be handled via etags or a version field in the future, but is out of scope for now.

- **Delete Timeline:** `DELETE /timelines/{id}` – Deletes a timeline record. This removes the timeline from the store (or marks it as deleted). Returns 204 No Content on success. Deleting a timeline that is currently playing (if any) would first stop playback.

- **Play Timeline:** `POST /timelines/{id}/play` – Triggers playback of the specified timeline. This is a **control operation** rather than a resource manipulation. Calling this endpoint will instruct the server to send the timeline's MIDI events to the appropriate output (for example, a synthesizer or animation engine) in real-time. The server responds immediately (e.g., 202 Accepted or 204 No Content) once playback has started, while the actual playback runs asynchronously. If the timeline cannot be played (e.g., invalid format or missing output device), an error response is returned. This endpoint could be extended in the future with query parameters or request body to specify playback options (like tempo, loop, etc.), but by default it plays once from start. *(Note: Pause/stop controls are not explicitly defined in this spec, but could be added as* `PUT /timelines/{id}/play` *with a state payload, or separate* `/pause` *and* `/stop` *endpoints.)*

All endpoints use standard HTTP methods and status codes, and standard JSON request/response bodies. Errors are reported using appropriate HTTP error codes (400 for bad input, 404 for not found, etc.), potentially with a problem+json body detailing the issue. The OpenAPI spec defines response schemas for the success cases and references common error schemas for failure cases.

The API path structure is nested under `/timelines` to clearly scope all operations to timeline resources. The **operationIds** in the spec (e.g., `listTimelines`, `createTimeline`, `playTimeline`) will map to method names in the generated Swift `APIProtocol`. This clean separation of concerns and the **RESTful design** align with Fountain's broader architecture of well-defined microservice APIs.

Data Persistence via Fountain-Store

Under the hood, AnimationKit will use **Fountain-Store** as its persistence layer. Fountain-Store is a pure-Swift embedded ACID database engine [12], which means all timeline records are stored locally on disk with full transactional safety. When the AnimationKit server starts, it will initialize a Fountain-Store database (or connect to an existing one) and use a dedicated collection (e.g., named `"timelines"`) to save timeline records.

Each timeline is stored as a record consisting of an `id` (the key) and the timeline data (the JSON structure with events) as the value. Thanks to Fountain-Store's schema-less design, the timeline JSON can be stored without impedance mismatch, and the engine's MVCC and snapshotting ensure consistency (so a timeline can be safely updated or read concurrently). The server can use Fountain-

Store's secondary indexing capability to index certain fields (for example, if we want to query timelines by name or by certain event content in the future).

Integration Details: The AnimationKit service will likely wrap Fountain-Store's Swift API to perform CRUD operations in each handler: - For **Create**, on `POST /timelines`, the server will generate a new ID (if not provided), then call `put` or `upsert` on the Fountain-Store collection to save the timeline data. Fountain-Store will write it to the WAL and SSTable, returning success if persisted. - **List** (`GET /timelines`) can leverage Fountain-Store's iteration or indexing to retrieve all timeline keys and perhaps some summary data. Pagination support can be added using Fountain-Store's opaque pagination tokens [13] [14] to handle large data sets, though for now we can return all or a fixed-size page of timelines. - **Get** reads the record by ID using a simple `get` operation. - **Update** uses `put` with the same key to overwrite the existing record (leveraging Fountain-Store's MVCC versioning under the hood for consistency). - **Delete** uses Fountain-Store's deletion functionality (which could either hard-delete or mark tombstone depending on configuration; Fountain-Store supports a `deleted` flag in responses [15], so we might opt for a soft-delete where `deleted=true` is set, or simply remove the record). - **Play** doesn't directly involve the database, except that the timeline data to play is first retrieved from the store (to get the latest version). Once retrieved, the server streams the events to the MIDI engine.

By using Fountain-Store, the AnimationKit service benefits from a robust, tested storage layer that aligns with the rest of FountainAI's stack. It also means the entire service can run with zero external dependencies (just like Fountain-Store itself). The openAPI spec for Fountain-Store serves as a reference for how the HTTP interface might look [1], although in AnimationKit, we embed the store rather than expose it directly (the AnimationKit API is higher-level). In summary, **timeline records are persisted reliably** and can even survive server restarts (the store will reload state on startup).

Authentication and Access Control (swift-secretstore)

Security is critical since timeline data and control could be sensitive (and you don't want unauthorized triggering of animations). AnimationKit will enforce authentication using a **Bearer token / API key model**, integrated with the `swift-secretstore` library. The approach follows the pattern used in Fountain-Store's optional HTTP server [16]:

- **Secret Storage:** The server expects an API key or token to be provisioned in a secure store. This can be provided via environment (e.g., an env var like `ANIMATIONKIT_API_KEY` for quick setups) or via `swift-secretstore` for more secure deployments. Swift-SecretStore allows storing and retrieving secrets in a platform-appropriate way (Keychain on Apple platforms, Secret Service on Linux desktops, or encrypted file for headless Linux) [17] [18]. At startup, AnimationKit will attempt to retrieve a secret (e.g., using a key like `"animationkit-api-key"`) from the secret store. If found, authentication is enabled; if not, the service may refuse to start or run with auth disabled for local testing (configurable).

- **Bearer/Auth Verification:** When auth is enabled, each incoming request must include a valid token. The service will accept either an HTTP header `Authorization: Bearer <token>` or `x-api-key: <token>` [16]. In the request handler (likely a middleware or the first part of request processing), AnimationKit will compare the provided token with the expected secret loaded from SecretStore. If it matches, the request proceeds; if not, the server responds with `401 Unauthorized`. This is exactly how Fountain-Store's HTTP server secures its endpoints when an API key is set [16]. We will leverage the same logic – possibly even abstracted into a

small helper since the pattern is identical (the Fountain-Store HTTPServer code demonstrates checking the `Authorization` header for a "Bearer " prefix and matching the secret [19]).

- **OpenAPI Spec Security Scheme:** In the OpenAPI YAML, we define a security scheme `bearerAuth` of type HTTP Bearer. All endpoints are marked as secured by this scheme (using OpenAPI's `security` field). This signals to clients that they must supply a Bearer token. The spec's descriptions will note that the token should be obtained out-of-band (perhaps configured per deployment) and that SecretStore manages it server-side. For example, the spec might include a snippet in the description: "This API uses a Bearer token for authentication. Set the `Authorization: Bearer <token>` header. The token is configured in the server's SecretStore (e.g., Keychain or file) and must match for access [16]."

Using `swift-secretstore` for managing the token means we **never hard-code secrets** in the code or spec. The token can be rotated or stored safely, and the same code works across macOS and Linux. This lightweight auth mechanism is appropriate for a service likely running in a controlled environment (perhaps on-device or local network for the coaching application). It's not full OAuth (not needed here), but provides a simple secure gate.

SwiftNIO Server Implementation (Spec-Driven)

The server is implemented using **SwiftNIO** directly (without Vapor), taking advantage of Apple's Swift OpenAPI Generator to create the server stubs. The OpenAPI spec will be packaged in the Swift project, and the SwiftPM plugin will generate code for both the client and server.

Swift Package Structure: The project is a Swift Package named `AnimationKit`. We include the OpenAPI YAML (e.g., `Sources/AnimationKit/openapi.yaml`) and an OpenAPI generator config file. The Package manifest adds dependencies on the Swift OpenAPI runtime libraries and the generator plugin (similar to how `RulesKit` is set up [20] [21]). For example, `Package.swift` will declare something like:

```
.package(url: "https://github.com/apple/swift-openapi-generator", from:
"1.5.0"),
.package(url: "https://github.com/apple/swift-openapi-runtime", from:
"1.0.0"),
.package(url: "https://github.com/apple/swift-openapi-urlsession", from:
"1.0.0"), // for client
// (We might also include OpenAPIHummingbird if using Hummingbird for server
transport)
```

and for the target:

```
.target(
    name: "AnimationKit",
    dependencies: [
        .product(name: "OpenAPIRuntime", package: "swift-openapi-runtime"),
        .product(name: "OpenAPIURLSession", package: "swift-openapi-
urlsession"),
        // possibly .product(name: "OpenAPIHummingbird", package: "swift-
```

```
openapi-generator") if exists
        "FountainStore", "SecretStore", "MIDI2" // our internal deps
    ],
    resources: [
        .copy("openapi.yaml")
    ],
    plugins: [
        .plugin(name: "OpenAPIGenerator", package: "swift-openapi-generator")
    ]
)
```

This configuration instructs SwiftPM to run the OpenAPI generator plugin at build time. The plugin reads our `openapi.yaml` and produces Swift sources for: - `AnimationKit.APIProtocol` – a protocol with one method per operation (e.g., `func listTimelines(...)`, `func createTimeline(...)`, etc.), using strongly-typed input/output models. - **Request/Response Models** – Swift structs or enums for the request parameters and responses for each operation, and codable struct definitions for schemas like `Timeline`, `TimelineEvent`, etc. - `AnimationKit.Client` – a client struct implementing APIProtocol, for convenient use by other components or even CLI tools, allowing them to call the service with type-safe code [22] [23]. - **Server Stubs** – the generator can also produce server-side code such as a default implementation of each API call that simply returns a "not implemented" or calls into a provided handler. In practice, we will implement the `APIProtocol` ourselves to connect the routes to the Fountain-Store and MIDI playback logic.

Server Setup: Without Vapor, we have a couple of options. We can use SwiftNIO's HTTP bootstrap to create an HTTP server channel and route requests (similar to how Fountain-Store's HTTPServer does with `ChannelInboundHandler` for HTTP1 [24] [25]). However, to simplify integration of the generated handlers, we may leverage a minimal framework like **Hummingbird** with the OpenAPI Hummingbird adapter (as demonstrated in community examples [26] [27]). Hummingbird is built on NIO and would let us register the generated routes easily via `OpenAPIHummingbird` (which provides a `HummingbirdTransport`). The end result is still a lightweight, pure SwiftNIO server, just without having to manually write decoding logic for each path.

For example, we might have a `main.swift` (in an executable target, say `AnimationKitServer`) like:

```
import AnimationKit
import Hummingbird
import OpenAPIHummingbird
import OpenAPIRuntime

struct Handler: AnimationKit.APIProtocol {
    // Implement each operation
    func listTimelines(_ input: Operations.ListTimelines.Input) async throws
-> Operations.ListTimelines.Output {
        let all = try store.listAllTimelines()  // pseudo-code for fetching
from Fountain-Store
        return .ok(.init(body: .json(all)))
    }
    // ... other methods (create, get, update, delete, play) implemented
```

```
similarly ...
    func playTimeline(_ input: Operations.PlayTimeline.Input) async throws -
> Operations.PlayTimeline.Output {
        let id = input.path.id
        guard let timeline = try store.getTimeline(id: id) else {
            return .notFound
        }
        midiEngine.play(timeline)  // use Engraver/MIDI2 engine to schedule
events
        return .accepted  // or .noContent
    }
}
@main
struct AnimationKitServer {
    static func main() async throws {
        // Setup Fountain-Store and SecretStore
        let store = try FountainStore(path: "/var/lib/animationkit")  //
initialize DB
        let secret = try loadSecretToken()  // use SecretStore to get token
(or env)
        // Setup NIO server (here using Hummingbird for brevity)
        let app =
HBApplication(configuration: .init(address: .hostname(port: 8080)))
        let transport = HummingbirdTransport(app: app, enableRequestLogging:
false)
        // Register handlers
        let handler = Handler(store: store, midiEngine: MIDIEngine.default,
authToken: secret)
        try handler.registerHandlers(on: transport, serverURL: URL(string:
"/")!)
        // Middleware for auth: check Authorization header against `secret`
(if present)
        if let token = secret {
            app.middleware.add(HBMiddleware() { req, next in
                if let reqToken = req.headers.bearerToken(), reqToken ==
token {
                    return try await next.handle(req)
                }
                return req.failure(.unauthorized)
            })
        }
        try app.start()
        app.wait()  // keep running
    }
}
```

The above pseudo-code illustrates how the generated `APIProtocol` is used – we implement it in a `Handler` struct, then register the handler with a `ServerTransport`. If not using Hummingbird, we would instead manually wire NIO channels to the `OpenAPIRuntime` router, but using an existing adapter is simpler. In either case, **no Vapor dependency is involved**, fulfilling the requirement to use SwiftNIO directly for the HTTP server.

Client usage: The generated client (`AnimationKit.Client`) can be used by other Swift components to call the AnimationKit service. For example, another service or an iOS app could use `AnimationKit.Client(serverURL: ..., transport: URLSessionTransport())` and call `client.createTimeline(...)` or `client.playTimeline(...)` with compile-time checked models, rather than hand-crafting HTTP requests. This is a huge productivity and safety win, as demonstrated by Apple's examples [28] [29].

Finally, we include rich **documentation comments** in the OpenAPI spec (using Markdown in descriptions) to explain the UMP timeline model, how OTIO import/export works, and the authentication scheme. These will carry over into the generated Swift code as API documentation. For instance, the spec notes that the `events` array is in UMP format and references the MIDI2 spec, and that the `Authorization: Bearer` header is required (with a link to how the secret is managed) [16].

In summary, **AnimationKit** is designed with a modern spec-first API, integrating tightly with Fountain's MIDI2 and storage systems, and using Apple's OpenAPI tools to generate a robust Swift package. Below is the proposed OpenAPI 3.1 YAML specification for AnimationKit, followed by an outline of the generated code structure.

OpenAPI 3.1 Specification (AnimationKit API)

```
openapi: "3.1.0"
info:
  title: "AnimationKit API"
  version: "0.1.0"
  description: |
    AnimationKit service for managing and playing animation timelines.
    A timeline is a sequence of timed MIDI 2.0 events (Universal MIDI
Packets)
    that can drive animations or musical sequences. This API allows clients
to
    create, query, update, delete, and play timelines. It also supports
importing
    and exporting timelines in **OpenTimelineIO (OTIO)** format for
interoperability.

    **UMP Timeline Model:** Timeline events are represented as JSON objects
matching
    the MIDI 2.0 Universal MIDI Packet structure (aligned with Fountain
midi2).
    Each event has a timestamp (seconds) and a MIDI message (e.g. Note On,
Note Off, etc.).
    The message fields (group, channel, note, velocity, etc.) conform to the
MIDI 2.0 spec.

    **OTIO Integration:** OTIO is an open interchange format for editorial
timelines [9].
    Use the `/timelines/importOTIO` endpoint to import an OTIO timeline
(JSON) – it will be converted
    into a UMP-based timeline. Use `GET /timelines/{id}/otio` to export any
timeline to OTIO (JSON)
```

```
    for use in external tools.

    **Authentication:** All endpoints require a Bearer token. Include
`Authorization: Bearer <token>`
    or `x-api-key: <token>` in requests. The token is configured on the
server (e.g., via SecretStore) [16].
servers:
  - url: "http://localhost:8080"
    description: "Local development server"
security:
  - bearerAuth: []   # Global security: all endpoints require auth
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
# using a simple bearer token (not necessarily JWT, but format is a opaque
token)
  schemas:
    Timeline:
      type: object
      required: [id, events]
      properties:
        id:
          type: string
          description: Unique identifier of the timeline record.
        name:
          type: string
          description: An optional human-friendly name for the timeline.
        description:
          type: string
          description: Optional longer description of the timeline.
        events:
          type: array
          description: "Ordered list of timeline events. Each event has a
time (s) and a MIDI 2.0 UMP message."
          items:
            $ref: "#/components/schemas/UmpEvent"
        duration:
          type: number
          format: float
          description: "Total duration of the timeline in seconds.
(Optional, can be derived from events.)"
      example:
        id: "timeline-123"
        name: "Demo Timeline"
        description: "Example timeline with two note events"
        events:
          - time: 0.0
            message:
```

```
              messageType: 4
              group: 0
              statusNibble: 9    # Note On (0x9)
              channel: 0
              body:
                noteNumber: 60
                velocity16: 100
                attributeType: 0
                attributeData16: 0
          - time: 1.0
            message:
              messageType: 4
              group: 0
              statusNibble: 8    # Note Off (0x8)
              channel: 0
              body:
                noteNumber: 60
                velocity16: 0
                attributeType: 0
                attributeData16: 0
    UmpEvent:
      type: object
      required: [time, message]
      properties:
        time:
          type: number
          format: float
          description: "Timestamp in seconds from the start of the timeline
when this event occurs."
        message:
          description: "MIDI 2.0 UMP message object. Format varies based on
messageType."
          oneOf:
            - $ref: "#/components/schemas/UmpPacket32"
            - $ref: "#/components/schemas/UmpPacket64"
            - $ref: "#/components/schemas/UmpPacket128"
    UmpPacket32:
      type: object
      description: "32-bit UMP (Utility or System message)."
      properties:
        messageType:
          type: integer
          enum: [0, 1, 2, 3, 15]   # Utility=0, System=1, MIDI1 Channel
Voice=2, SysEx7=3, Stream=15
        group:
          type: integer
          minimum: 0
          maximum: 15
          description: MIDI group (0-15). For Utility and Stream messages in
v1.1+, use 0.
        statusByte:
```

```
          type: integer
          minimum: 0
          maximum: 255
          description: "Status byte or further classification of the message
(meaning depends on messageType)."
        data:
          type: array
          items: { type: integer, minimum: 0, maximum: 255 }
          description: "Up to 2 data bytes for 32-bit messages (e.g., System
Common messages)."
      required: [messageType]
    UmpPacket64:
      type: object
      description: "64-bit UMP (MIDI 2.0 Channel Voice message) 30 31 ."
      properties:
        messageType:
          type: integer
          enum: [4]   # MIDI2 Channel Voice messages have type 4
        group:
          $ref: "#/components/schemas/Group"
        statusNibble:
          $ref: "#/components/schemas/Midi2StatusNibble"
        channel:
          $ref: "#/components/schemas/Uint4"

# The body of a 64-bit message depends on statusNibble (which defines the
specific message)
        body:
          oneOf:
            - $ref: "#/components/schemas/Midi2.NoteOn"
            - $ref: "#/components/schemas/Midi2.NoteOff"
            - $ref: "#/components/schemas/Midi2.PolyPressure"
            - $ref: "#/components/schemas/Midi2.ControlChange"
            - $ref: "#/components/schemas/Midi2.ProgramChange"
            - $ref: "#/components/schemas/Midi2.ChannelPressure"
            - $ref: "#/components/schemas/Midi2.PitchBend"
            # (Other MIDI2 channel voice messages could be added, e.g. RPN/
NRPN, but omitted for brevity)
      required: [messageType, group, statusNibble, channel, body]
    UmpPacket128:
      type: object
      description: "128-bit UMP (Data messages: SysEx8/MIDI2 Stream, or Flex
Data) 32 33 ."
      properties:
        messageType:
          type: integer
          enum: [5, 13]   # 5 = SysEx8/MDS, 13 = Flex Data
        group:
          $ref: "#/components/schemas/Group"
        data:
          type: array
```

```
          items: { type: integer, minimum: 0, maximum: 255 }
          description: Up to 14 bytes of data (for 128-bit messages).
      required: [messageType, data]
    # Scalar and MIDI-specific types:
    Group:
      type: integer
      minimum: 0
      maximum: 15
      description: MIDI Group number (0-15).
    Uint4:
      type: integer
      minimum: 0
      maximum: 15
    Uint7:
      type: integer
      minimum: 0
      maximum: 127
    Uint14:
      type: integer
      minimum: 0
      maximum: 16383
    Uint16:
      type: integer
      minimum: 0
      maximum: 65535
    Uint32:
      type: integer
      minimum: 0
      maximum: 4294967295
    Midi2StatusNibble:
      type: integer
      minimum: 8
      maximum: 15
      description: "Low 4 bits of Status Byte indicating the MIDI 2.0
Channel Voice message type (0x8 = NoteOff, 0x9 = NoteOn, 0xA = Poly
Pressure, 0xB = CC, 0xC = Program, 0xD = Channel Pressure, 0xE = Pitch
Bend)."
    NoteAttributeType:
      type: integer
      minimum: 0
      maximum: 127
      description: "Note attribute type (for future use, per MIDI2 spec; 0 =
none/default)."
    # MIDI2 Channel Voice message bodies:
    "Midi2.NoteOn":
      type: object
      description: "Status 0x9 – Note On (with 16-bit velocity) [5] [6]."
      properties:
        noteNumber:
          $ref: "#/components/schemas/Uint7"
        velocity16:
```

```
          $ref: "#/components/schemas/Uint16"
        attributeType:
          $ref: "#/components/schemas/NoteAttributeType"
        attributeData16:
          $ref: "#/components/schemas/Uint16"
      required: [noteNumber, velocity16]
    "Midi2.NoteOff":
      type: object
      description: "Status 0x8 – Note Off (with 16-bit velocity) 34 35 ."
      properties:
        noteNumber:
          $ref: "#/components/schemas/Uint7"
        velocity16:
          $ref: "#/components/schemas/Uint16"
        attributeType:
          $ref: "#/components/schemas/NoteAttributeType"
        attributeData16:
          $ref: "#/components/schemas/Uint16"
      required: [noteNumber, velocity16]
    "Midi2.PolyPressure":
      type: object
      description: "Status 0xA – Polyphonic Key Pressure (32-bit pressure)."
      properties:
        noteNumber: $ref: "#/components/schemas/Uint7"
        polyPressure32: $ref: "#/components/schemas/Uint32"
      required: [noteNumber, polyPressure32]
    "Midi2.ControlChange":
      type: object
      description: "Status 0xB – Control Change (32-bit value)."
      properties:
        control: $ref: "#/components/schemas/Uint7"
        controlValue32: $ref: "#/components/schemas/Uint32"
      required: [control, controlValue32]
    "Midi2.ProgramChange":
      type: object
      description: "Status 0xC – Program Change (with optional Bank)."
      properties:
        program: $ref: "#/components/schemas/Uint7"
        bankMsb: $ref: "#/components/schemas/Uint7"
        bankLsb: $ref: "#/components/schemas/Uint7"
        bankValid:
          type: boolean
          description: "If true, bankMsb/Lsb are present; if false, no bank
change."
      required: [program]
    "Midi2.ChannelPressure":
      type: object
      description: "Status 0xD – Channel Pressure (32-bit)."
      properties:
        channelPressure32: $ref: "#/components/schemas/Uint32"
      required: [channelPressure32]
```

```
    "Midi2.PitchBend":
      type: object
      description: "Status 0xE – Pitch Bend (32-bit)."
      properties:
        pitchBend32: $ref: "#/components/schemas/Uint32"
      required: [pitchBend32]
    OTIO:   # (Representing an OpenTimelineIO timeline as an opaque JSON
object)
      type: object
      description: |
        A JSON object in OpenTimelineIO format representing a timeline.
        This will typically have keys like "OTIO_SCHEMA": "Timeline.x",
"tracks", "children", etc.,
        per the OTIO specification 10 36 .
        The exact structure is not validated here; the server will interpret
it using an OTIO library.
paths:
  /timelines:
    get:
      summary: List all timelines
      description: |
        Retrieve a list of all animation timelines.
        This returns an array of timeline metadata and/or full objects.
      operationId: listTimelines
      security:
        - bearerAuth: []
      responses:
        "200":
          description: A list of timelines.
          content:
            application/json:
              schema:
                type: object
                properties:
                  items:
                    type: array
                    items:
                      $ref: "#/components/schemas/Timeline"

# We wrap in an object with "items" to allow future expansion (pagination
tokens, etc).
        default:
          description: Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"   # (Assume a generic
error schema defined elsewhere)
    post:
      summary: Create a new timeline
      description: |
```

```
      Create a new animation timeline by providing the timeline JSON.
      The request body should include the timeline's events in UMP format.
      The server will assign an `id` if not provided.
    operationId: createTimeline
    security:
      - bearerAuth: []
    requestBody:
      required: true
      content:
        application/json:
          schema: $ref: "#/components/schemas/Timeline"
    responses:
      "201":
        description: Timeline created successfully.
        content:
          application/json:
            schema: $ref: "#/components/schemas/Timeline"
        headers:
          Location:
            description: URL of the created timeline resource.
            schema:
              type: string
      "400":
        description: Invalid input (malformed timeline).
      default:
        description: Error
        content:
          application/json:
            schema: $ref: "#/components/schemas/Error"
/timelines/{id}:
  parameters:
    - name: id
      in: path
      required: true
      schema:
        type: string
      description: The timeline's unique identifier.
  get:
    summary: Get a timeline by ID
    operationId: getTimeline
    security:
      - bearerAuth: []
    responses:
      "200":
        description: The timeline data.
        content:
          application/json:
            schema: $ref: "#/components/schemas/Timeline"
      "404":
        description: Timeline not found.
      default:
```

```
          description: Error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
    put:
      summary: Update/replace a timeline
      operationId: updateTimeline
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema: $ref: "#/components/schemas/Timeline"
      responses:
        "200":
          description: Timeline updated successfully.
          content:
            application/json:
              schema: $ref: "#/components/schemas/Timeline"
        "201":
          description: Timeline created (if it did not exist and was thus
newly created by PUT).
          content:
            application/json:
              schema: $ref: "#/components/schemas/Timeline"
        "400":
          description: Invalid timeline data.
        "404":
          description: Timeline not found (if expecting existing).
        default:
          description: Error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
    delete:
      summary: Delete a timeline
      operationId: deleteTimeline
      security:
        - bearerAuth: []
      responses:
        "204":
          description: Timeline deleted successfully.
        "404":
          description: Timeline not found.
        default:
          description: Error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
  /timelines/{id}/play:
```

```
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: string
        description: Timeline ID to play.
    post:
      summary: Play a timeline
      description: |
        Begin playback of the specified timeline. The server will stream the
timeline's MIDI events
        to the output (synth/animation engine). This returns immediately
after scheduling playback.

        **Note:** Ensure the server is connected to an appropriate MIDI
output or animation subsystem.
        This endpoint just triggers playback; it does not return the played
data.
      operationId: playTimeline
      security:
        - bearerAuth: []
      responses:
        "202":
          description: Playback started (accepted). The timeline is now
playing.
        "404":
          description: Timeline not found.
        default:
          description: Error or playback failure
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
  /timelines/importOTIO:
    post:
      summary: Import a timeline from OTIO
      description: |
        Create a new timeline by importing an OpenTimelineIO file.

The request should contain OTIO JSON data; the server will convert it into a
Timeline (UMP events).

        This returns the created timeline in AnimationKit's native format.
      operationId: importTimelineOTIO
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema: $ref: "#/components/schemas/OTIO"
```

```
            example:
              OTIO_SCHEMA: "Timeline.1"
              name: "Imported Timeline"
              tracks: { OTIO_SCHEMA: "Stack.1", children: [] }
              # ... (an example minimal OTIO structure) ...
      responses:
        "201":
          description: Timeline imported successfully.
          content:
            application/json:
              schema: $ref: "#/components/schemas/Timeline"
        "400":
          description: OTIO parse or conversion error.
        default:
          description: Error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
  /timelines/{id}/otio:
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: string
        description: The timeline ID to export.
    get:
      summary: Export a timeline to OTIO
      description: |
        Retrieve the timeline in OpenTimelineIO (OTIO) format. The response
is a JSON representation
        of the timeline as an OTIO Timeline, which can be used in editorial
tools.
      operationId: exportTimelineOTIO
      security:
        - bearerAuth: []
      responses:
        "200":
          description: OTIO timeline returned.
          content:
            application/json:
              schema: $ref: "#/components/schemas/OTIO"
        "404":
          description: Timeline not found.
        default:
          description: Error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
```

(Note: The spec above omits a generic `Error` *schema for brevity, but in practice we would define a proper error object or use* `application/problem+json` *for error responses.)*

Generated Swift Package Scaffold (Outline)

After running the Swift OpenAPI Generator plugin on the spec, we get a Swift package structure like:

```
AnimationKit/
├─ Sources/AnimationKit/openapi.yaml          (the spec file, included as
resource)
├─ Sources/AnimationKit/Models.swift          (generated models for schemas:
Timeline, UmpEvent, etc.)
├─ Sources/AnimationKit/API.swift             (generated APIProtocol with all
operations)
├─ Sources/AnimationKit/Client.swift          (generated Client
implementation)
├─ Sources/AnimationKit/Servers+Stubs.swift   (generated server stubs/
middleware if any)
├─ Sources/AnimationKitServer/main.swift      (our custom main to run the
server, if we include an exec target)
├─ Package.swift                              (SwiftPM manifest with
OpenAPIGenerator plugin, etc.)
└─ ...
```

Key elements from the generated code:

- **Models:** The `Timeline` schema becomes a Swift struct, for example:

  ```swift
  public struct Timeline: Codable {
      public var id: String
      public var name: String?
      public var description: String?
      public var events: [UmpEvent]
      public var duration: Double?
      // CodingKeys and init... (generated automatically)
  }
  ```

  Similarly, each `UmpEvent`, `UmpPacket64`, and MIDI2 message body (Midi2.NoteOn, etc.) is a Swift struct or enum conforming to `Codable`. The generator will handle the oneOf structures by creating enums. For instance, `UmpEvent.message` might be represented as an enum with cases `.umpPacket32(UmpPacket32)`, `.umpPacket64(UmpPacket64)`, etc., or as a protocol with polymorphic types – the exact representation depends on the generator's strategy for oneOf (likely an enum with associated values for each variant).

- **APIProtocol:** The spec's operations yield a protocol like:

```
protocol APIProtocol {
    func listTimelines(_ input: Operations.ListTimelines.Input) async
throws -> Operations.ListTimelines.Output
    func createTimeline(_ input: Operations.CreateTimeline.Input) async
throws -> Operations.CreateTimeline.Output
    // ... and so on for getTimeline, updateTimeline, deleteTimeline,
playTimeline, importTimelineOTIO, exportTimelineOTIO ...
}
```

Each operation's Input/Output are types that include parsed parameters. For example, `Operations.GetTimeline.Input` will have a `path` property with `id`, and `Operations.GetTimeline.Output` will be an enum for the possible responses (e.g., `.ok(Timeline)`, `.notFound`, etc.).

- **Client:** The `Client` struct implements `APIProtocol`. It uses the OpenAPIRuntime to issue HTTP calls. For instance, `client.getTimeline(.init(path: .init(id: "timeline-123")))` will perform a GET request to `/timelines/timeline-123` and return a typed result. This is all handled by the generated code using the URLSession transport by default [28] [29]. The client is useful for testing or integrating the service.

- **Server stub:** The generator can produce a basic server that routes HTTP requests to an `APIProtocol` implementation. If using a transport like Vapor or Hummingbird, there's typically an extension like `handler.registerHandlers(on: transport, serverURL: URL)` as shown in Apple's example [37] [38]. In our project, since we prefer not to use Vapor, we might use Hummingbird's extension (if available) or write our own router that uses the `OpenAPIRuntime` `ServerRequestDecoder` and `ServerResponseEncoder` to hook into NIO's pipeline. Essentially, the generated code will include logic to decode incoming HTTP requests into the corresponding `Operations.X.Input` and then call our `APIProtocol` implementation.

- **Integration code (custom):** We will have to write an implementation of `APIProtocol` (e.g., `AnimationKitHandler` class or struct) that connects to FountainStore and MIDI playback as described earlier. This is where we use `FountainStore` to fetch/save data, and trigger the MIDI2 playback engine. This part is not generated (since it's the business logic), but the generator makes it easier by handling all the HTTP and serialization details. Our `main.swift` (if present) will set up the server and call `registerHandlers`. The snippet in the previous section gives an idea of how that might look, including an auth middleware using the secret token.

- **Auth hooks:** The generated code will include the `SecuritySchemes` but handling auth is up to us. Typically, one would implement a middleware in front of the generated handlers. In Vapor or Hummingbird, we add a middleware that checks the Authorization header. In a pure NIO setup, we could intercept in the ChannelInboundHandler before dispatching to the OpenAPI handler. The Fountain-Store HTTP server code provides a blueprint for manual checking in NIO [19]. We will follow that: retrieve the token from SecretStore at startup, and then compare against incoming headers for each request. If we detect a mismatch, we return an HTTP 401 before calling the API handler.

- **Documentation:** The spec's markdown descriptions will appear in the generated code as comments. For example, the doc comment for the `Timeline` struct will mention it's a sequence of UMP events. Each operation will have a comment explaining usage, including the OTIO conversion or auth requirement. This helps future developers using the API or reading the code.

In conclusion, the AnimationKit Swift package will be largely generated, ensuring consistency with the OpenAPI spec and reducing boilerplate. Developers can focus on implementing the core logic: storing timelines and playing them. The use of Fountain-Store and swift-secretstore aligns with existing Fountain architecture (a secure, local-first design), and the adherence to the MIDI2 UMP standard ensures compatibility with Fountain's Engraver/midi2 engine for actual timeline execution [7]. By exporting to and importing from OTIO [10], the system stays flexible and open to creative workflows outside the Fountain ecosystem. This spec-first, codegen-based approach will make AnimationKit maintainable and reliable as it evolves alongside Fountain's other components.

Sources:

- Fountain-Store design and API key integration [1] [16]
- Fountain Engraving (spec-first OpenAPI with Swift codegen) [2] [3]
- Teatro Animation/MIDI integration (MIDI2 timeline playback) [7]
- MIDI 2.0 UMP JSON schema (for event format alignment) [4] [5]
- OpenTimelineIO format (JSON structure for timelines) [9] [36]
- Apple Swift OpenAPI Generator usage [28] [37]

---

[1] [12] [13] [14] [15] [16] README.md
https://github.com/Fountain-Coach/Fountain-Store/blob/50654f4d799ff9b5e7bba0c1df57e03c89c56ef1/README.md

[2] [3] README.md
https://github.com/Fountain-Coach/Engraving/blob/5c7b08007cdfd8b2b40846d9ccd68f020b9ac327/README.md

[4] [5] [6] [8] [30] [31] [32] [33] [34] [35] midi2.full.openapi.json
https://github.com/Fountain-Coach/midi2/blob/fec93dae9efbb4f1d56984eb0889983962a1dbab/midi2.full.openapi.json

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

[9] [10] [11] [36] File Format Specification — OpenTimelineIO 0.13.0 documentation
https://opentimelineio.readthedocs.io/en/v0.13/tutorials/otio-file-format-specification.html

[17] [18] README.md
https://github.com/Fountain-Coach/swift-secretstore/blob/724b285759206fc87cd56066d265e70215d83d28/README.md

[19] [24] [25] main.swift
https://github.com/Fountain-Coach/Fountain-Store/blob/50654f4d799ff9b5e7bba0c1df57e03c89c56ef1/Sources/FountainStoreHTTPServer/main.swift

[20] [21] Package.swift
https://github.com/Fountain-Coach/Engraving/blob/5c7b08007cdfd8b2b40846d9ccd68f020b9ac327/codegen/swift/RulesKit-SPM/Package.swift

[22] [23] [28] [29] Introducing Swift OpenAPI Generator | Swift.org
https://www.swift.org/blog/introducing-swift-openapi-generator/

26 27 OpenAPI generator with Swift on Server | by Szabolcs Toth | Mar, 2025 | Medium
https://medium.com/@kicsipixel/openapi-generator-with-swift-on-server-ca20070cf239

37 38 GitHub - apple/swift-openapi-generator: Generate Swift client and server code from an OpenAPI document.
https://github.com/apple/swift-openapi-generator