ChatGPT

Integrating ChatKit-JS into FountainKit Gateway and Publishing Frontend

Overview: ChatKit-JS and FountainKit Gateway

ChatKit-JS is OpenAI's **"batteries-included" chat interface framework** – a drop-in JavaScript component that delivers a complete chat UI out of the box [1]. Its key features include rich support for **streaming responses, tool usage visualization, interactive widgets, file uploads, thread management, and source annotations** [2]. In practice, developers can simply embed the ChatKit component, provide it with a client token, and get an advanced chat UI **without reinventing the wheel** [3].

FountainKit is a Swift-based AI orchestration platform. The **FountainKit Gateway** serves as the central entry point, authenticating requests and routing them to AI services and tools [4]. Notably, the gateway includes a **"publishing frontend"** – a static web frontend that can host assets like HTML/JS for user-facing interfaces [5]. The gateway's built-in `PublishingFrontendPlugin` will serve files from a `Public/` directory (e.g. an `index.html`) whenever a request doesn't match an API route [6]. This means we can host a ChatKit-powered web app **directly from the FountainKit Gateway**, providing users with a chat interface served by the gateway itself.

Goal: Deeply integrate ChatKit-JS into this FountainKit Gateway frontend and backend. We want to enable **FountainKit-powered chat apps** that leverage ChatKit's UI on the front-end and FountainKit's full AI reasoning capabilities on the back-end. In the sections below, we'll outline integration strategies, step-by-step setup, and how to maximize the use of ChatKit alongside FountainKit (and related Fountain Coach components like Fountain-Store and swift-secretstore). The aim is to **"tutorialize" the process** for both internal Fountain developers and external app developers, showcasing the entire Fountain Coach ecosystem in action.

Publishing Frontend Setup for ChatKit

First, we prepare the **publishing frontend** to host the ChatKit interface. The FountainKit Gateway will serve any static files placed in its `Public` directory (by default) – for example, if a user accesses the gateway's base URL, the plugin will return `Public/index.html` [6]. We can leverage this to deliver an HTML page running ChatKit.

Steps to set up the frontend:

1. **Include ChatKit assets:** Download or reference the ChatKit JS bundle and any required CSS. For instance, include the ChatKit script via OpenAI's CDN or your fork's build. In `index.html` (served by the gateway), add:

```
<script src="https://cdn.platform.openai.com/deployments/chatkit/
chatkit.js" async></script>
```

If using the forked ChatKit (with potential modifications), host the script file in `Public/` and include it accordingly (e.g. `<script src="/chatkit.js"></script>`).

2. **Initialize the ChatKit component:** The simplest approach is using the provided web component or React bindings. For a minimal setup, you can use the web component in HTML:

```
<openai-chatkit style="height: 600px; width: 100%;"></openai-chatkit>
```

Alternatively, if using React in the frontend, render `<ChatKit>` via `@openai/chatkit-react` as in the ChatKit docs.

3. **Configure ChatKit to obtain a client token:** ChatKit requires a **client secret/token** to connect to a chat session. In the default OpenAI flow, this token is obtained by calling a server endpoint (e.g. `/api/chatkit/session`) which creates a ChatKit session via OpenAI's API [7] [8]. For our integration, we'll implement a similar endpoint in FountainKit's gateway (details in the next section). The front-end should be set to call this endpoint. For example, using the ChatKit React hook:

```
useChatKit({
  api: {
    async getClientSecret(existing) {
      // (Optional: handle token refresh if `existing` provided)
      const res = await fetch('/api/chatkit/session', { method:
'POST' });
      const { client_secret } = await res.json();
      return client_secret;
    }
  }
});
```

This ensures the ChatKit UI can retrieve a valid token to start the chat session.

4. **Verify static serving:** Once `index.html` (with the ChatKit integration) and any asset files are in the `Public` directory, run the FountainKit gateway (e.g. via `swift run gateway-server`). Navigate to the gateway's URL in a browser – the ChatKit UI should load, showing the chat interface. Thanks to the gateway's publishing plugin, no separate web server is needed; the gateway itself hosts this interface [5].

With these steps, the FountainKit Gateway's publishing frontend will be running the ChatKit UI. Next, we need to hook up the backend logic so that ChatKit's chat interface actually talks to FountainKit's AI backend instead of OpenAI's servers.

Integration Strategies: Hosted vs. Self-Hosted Backend

There are two main integration approaches to consider, each with its trade-offs:

Option 1: OpenAI-Hosted Agent Backend (Quick Start)

Leverage OpenAI's Agent Builder and backend initially, while using FountainKit primarily as a static host. In this setup, the ChatKit front-end would connect to an OpenAI-managed agent (defined in their platform). FountainKit's role would be mostly to serve the UI, and perhaps handle auth. This option requires minimal backend development – you use OpenAI's existing ChatKit support as-is. However, **it relies on OpenAI's cloud** (their client token service and agent execution), meaning you don't yet tap into FountainKit's own LLM orchestration.

Use case: This could be a stepping stone or a quick demo – e.g. to get a chat interface up swiftly for internal testing. You would generate the `client_secret` by calling OpenAI's API (using your API key) inside a simple FountainKit gateway endpoint. In practice, your gateway's `/api/chatkit/session` route could call `openai.chatkit.sessions.create()` (using OpenAI's Python/Node SDK) as shown in OpenAI's example [7]. The returned client secret is passed back to the browser, which then uses it to communicate with the OpenAI agent.

Pros: Easiest initial path; you can get a working chat quickly.
Cons: Doesn't utilize FountainKit's power – the AI logic runs on OpenAI's side, and you're subject to their platform limitations and costs.

Option 2: Self-Hosted Chat Backend with FountainKit (Deep Integration)

Fully integrate ChatKit's front-end with FountainKit's **own AI backend** – making FountainKit the brain behind the chat. In this approach, the ChatKit UI will connect to *FountainKit's Gateway APIs* rather than OpenAI's. FountainKit will handle the conversation (utilizing its LLM Gateway, tools, persona orchestration, etc.), essentially acting as a **custom ChatKit backend**. This requires implementing endpoints and possibly token logic in the FountainKit Gateway, but yields a **powerful, self-contained system** under your control.

In this self-hosted scenario, we will **fork or configure ChatKit** to point to our FountainKit instance. Notably, OpenAI's ChatKit supports a custom API base URL via the `CHATKIT_API_BASE` setting [9]. We can use this to redirect ChatKit to our gateway's API endpoints rather than OpenAI's. Our fork of ChatKit-js may already include modifications to facilitate this (e.g. defaulting to FountainKit endpoints or adapting the session flow).

Pros: Complete control over the AI logic (no external dependency), ability to use FountainKit's unique features (tool use, custom models, on-prem deployment, etc.), and potentially better data governance. This is the **ultimate goal** for a deep integration – users interact with FountainKit's AI directly through a slick UI.
Cons: Requires development effort to implement the ChatKit-compatible APIs in FountainKit. We must ensure feature parity or compatibility so the ChatKit front-end functions correctly.

Given Fountain Coach's objectives, **Option 2 (deep integration)** is the target. Below, we focus on how to implement this self-hosted integration in a step-by-step, tutorial-like manner.

Deep Integration: ChatKit-JS with FountainKit Backend

In a deep integration, FountainKit will manage chat sessions and message processing, while ChatKit serves as the user interface. Achieving this involves work on both the **gateway backend** (to handle chat session and message APIs) and the **ChatKit front-end configuration**. Let's break down the key tasks:

1. Implementing a Chat Session Endpoint in FountainKit Gateway

We need an endpoint (e.g. `/api/chatkit/session`) in the gateway that issues a "client token" for ChatKit. In OpenAI's model, this token (client_secret) encapsulates the permissions and identity to join a chat session hosted on OpenAI. In our case, **we can define a token that our own backend will understand**. There are a few approaches:

- **Stateless token (JWT):** The gateway can generate a signed JWT as client_secret. This token might include a session ID or user info, and the front-end will use it to authenticate subsequent chat connections. The gateway would verify it on incoming requests. Using Swift's crypto libraries, you can sign a token that ChatKit will carry. (This is custom since ChatKit expects a client_secret string; our fork can be made to accept and use it).
- **Opaque session ID:** Alternatively, the gateway can generate a random session ID (and store any state server-side keyed by it). The client_secret returned could just be this session ID. ChatKit would pass it back on chat requests, and the gateway could look up the session context. This is simpler if we maintain conversation state on the server.

For simplicity, start with **opaque session ID**. Implement a POST route `/api/chatkit/session` in the gateway that does: - Authenticate the caller (e.g. ensure a valid API key if required by gateway – FountainKit uses an `X-API-Key` header for protected calls [10]). For internal usage, this might not be needed; for external, it is. - Create a new session record (if maintaining state) or at least generate a unique ID/token. - Return `{ "client_secret": "<the-session-id>" }` to the requester (the ChatKit UI).

This will allow the ChatKit front-end to obtain a `client_secret` that is meaningful to FountainKit.

Tip: You can use **swift-secretstore** for any secret keys needed (e.g. signing keys, API keys). FountainKit can safely retrieve these from the SecretStore (which supports Keychain, secret-service, or encrypted file storage on different platforms [11] [12]). This ensures no plaintext secrets (like OpenAI API keys, if FountainKit calls OpenAI under the hood, or JWT signing secrets) are hardcoded in code. Instead, they can be securely stored and accessed at runtime.

2. Routing Chat Messages to FountainKit's LLM Gateway

With a session established, ChatKit will need to send user messages and receive AI responses via FountainKit. How does ChatKit actually communicate during the chat? Under the hood, ChatKit likely uses a combination of HTTP and streaming (possibly WebSockets or Server-Sent Events) to transmit messages and receive the AI's responses in real-time. Our integration must plug into FountainKit's equivalent functionality:

- **Design an API contract:** We should expose an endpoint (or set of endpoints) that ChatKit will use for the chat conversation. If mimicking OpenAI's API, we might implement endpoints like:
- `POST /api/chatkit/message` (client sends a new user message along with the session token).
- `GET /api/chatkit/events` or WebSocket at `/api/chatkit/stream` (client listens for streaming responses/events for that session). Alternatively, if our fork of ChatKit expects a specific pattern (maybe a single endpoint that upgrades to an event stream), we follow that.

FountainKit already has an **LLM Gateway API** that could be repurposed: for instance, a `POST /chat` endpoint to handle chat requests [13] [14]. That OpenAPI spec indicates the gateway can receive a chat request with a model, messages array, and optional function calls [15] [16]. We can utilize this by having

our ChatKit endpoint call into the same logic: essentially, when a user message comes in, FountainKit's planner/orchestrator will take the conversation (messages so far) and produce the assistant's reply (possibly via a configured LLM like GPT-4, with FountainKit's policy and tool plugins in effect).

- **Streaming responses:** ChatKit supports streaming, so users see the AI typing word-by-word [17]. To enable this, FountainKit's response should be sent as a stream. In Swift NIO or Vapor (depending on how FountainKit is built), you might implement this via a chunked HTTP response or upgrading to an SSE/WS. For example, the gateway could offer a **Server-Sent Events (SSE)** endpoint that pushes tokens as they're generated. The ChatKit UI would connect to this stream after sending a message. If our ChatKit fork is configured properly, it will connect to FountainKit's SSE channel instead of OpenAI's.

If SSE/WebSockets are not immediately available, a simpler fallback is to have the ChatKit front-end poll or do long-polling. However, since ChatKit is built for real-time, investing in streaming is worth it. (Check if FountainKit already has streaming support in its LLM Gateway; if not, consider adding an SSE route on top of the chat completion logic).

- **Connecting the dots:** Modify the ChatKit front-end configuration to use our custom base URL. As noted, setting the environment variable `CHATKIT_API_BASE` (or equivalent in config) to our gateway's URL will make the ChatKit JS send requests to our backend [9]. For example, if our gateway runs at `https://myfountain.example.com`, we ensure ChatKit targets `https://myfountain.example.com/api/` for its calls (instead of the default OpenAI endpoints). Our fork of ChatKit-JS likely has this base URL adjusted already for FountainCoach usage. Ensure the front-end knows the correct Workflow or Agent identifier if needed (in OpenAI's case, a `workflow_id` was used [18]; for FountainKit, we might not need that, or we could repurpose it to select a persona).

At this point, when the user types a message in the ChatKit UI, the following should occur: the message is sent via our new API endpoint to FountainKit; FountainKit's gateway receives it, passes it to the reasoning system (applying whatever persona or tools are configured), and begins streaming back the AI's response; ChatKit displays the response in real-time. We have essentially **replaced OpenAI's agent backend with FountainKit**, achieving deep integration.

3. Utilizing Fountain-Store for Persistence (Optional)

To "squeeze as much as we can out of it," consider leveraging **Fountain-Store** for storing chat data or other artifacts. Fountain-Store is a versatile persistence layer that supports local, cloud, or Git-backed storage for JSON documents [19]. In a chat app context, Fountain-Store could be used to log conversation history, store long-term memories, or save user-provided files. FountainKit can interact with Fountain-Store through a simple client API or via defined endpoints. In fact, some FountainKit agents (e.g. a sample **SamplerAgent** for audio) already **use Fountain-Store by default** for saving data [20].

For example, you might implement a feature where after a chat session, the full transcript is saved as a JSON (or Markdown) in Fountain-Store for auditing or later retrieval. Since Fountain-Store comes with an OpenAPI spec and client, you can invoke it from within FountainKit's flows to persist or fetch data as needed. Persisting chat history would benefit external developers (to build features like conversation continuity or analytics) and internal needs (debugging or fine-tuning data collection).

4. Security and Access Control

Because FountainKit is likely to be used both internally and by external developers/users, security is crucial. By integrating ChatKit, we potentially expose a public-facing chat UI, so we must enforce proper authentication and usage limits: - Leverage FountainKit Gateway's **API key mechanism** for external requests [10]. An external app embedding ChatKit would need to supply an API key (perhaps via a custom header or as part of the client token request). We can issue API keys to trusted developers or users. Internally, for a public demo, you might allow an open session but still monitor usage. - Use FountainKit's **policy plugins**. The gateway can enforce rate limiting, request budgets, and safety filters via its plugin system [4]. Ensure those (like rate-limiter, destructive-guardian, etc.) are configured when opening the system to external use. This prevents abuse and keeps the AI's outputs in check. - Keep secrets safe via **swift-secretstore** as mentioned, and avoid exposing any sensitive config in the front-end. All API keys (for FountainKit itself or for any upstream LLMs it calls) should reside server-side, fetched from secure storage by FountainKit.

With the front-end and back-end now working together, we have a functional integration. Next, let's consider how to **maximize ChatKit's rich features** with FountainKit's capabilities.

Maximizing ChatKit's Features with FountainKit

One major benefit of ChatKit is the rich, dynamic chat experience it offers. A deep integration should strive to **fully utilize these features**, providing a cutting-edge AI assistant app. Here's how FountainKit can complement each feature:

- **Streaming, real-time responses:** FountainKit's LLM Gateway should stream tokens as the model generates them, which ChatKit will display incrementally [17]. Using streaming not only improves UX but also allows FountainKit's intermediate reasoning steps to possibly interleave (e.g., showing that the assistant is "thinking" or calling a tool). Ensure the integration uses streaming APIs of the underlying model (OpenAI API supports it; if using open-source models, consider libraries that stream).

- **Tool use and agent actions visualization:** FountainKit is built for agent orchestration – it can integrate tools (via function calling or external services) and has a Planner/Curation system for reasoning. ChatKit can visualize **agentic actions and chain-of-thought** steps [21]. To leverage this, FountainKit should emit events or messages when the agent uses a tool or takes an intermediate step. For example, if the AI calls a "SearchTool" or a custom `SamplerTool`, we can send a special message (perhaps with a role like `system` or a ChatKit-specific event payload) that the ChatKit UI renders in the conversation (e.g., " Agent is searching for information..." or a structured display of the tool's result). Our ChatKit fork might already handle certain event types for actions. By exposing FountainKit's internal reasoning (within safe limits) to ChatKit, developers and users get transparency into the AI's decision process – fulfilling that chain-of-thought visualization promise.

- **Rich interactive widgets:** ChatKit supports embedding interactive widgets or rich content in the chat stream [22]. FountainKit can take advantage of this by returning specially formatted outputs. For instance, if the AI's answer includes a chart, diagram, or audio player, ChatKit could render it as an interactive element rather than just text. How to do this? One way is using Markdown or HTML in the assistant's response (ChatKit likely sanitizes/handles certain HTML or provides a widget API). Another is if ChatKit defines a protocol for widgets (perhaps via a function call or a block of JSON that the UI knows how to present). Investigate ChatKit docs for "widgets" –

possibly the fork or samples show how, e.g., an answer could include a `<!-- widget: {type: "table", data: {...}} -->` comment or similar. With FountainKit, implementing a widget might simply be formatting the tool's output appropriately. **Example:** If FountainKit's agent queries a database and gets a result set, it could return it in a JSON that ChatKit then renders as an interactive table in the chat. The forked ChatKit might have extended widget support for custom needs.

- **File and image attachments:** ChatKit can handle uploads [23] – you could allow users to send files (images, PDFs, etc.) to the AI. FountainKit would need to accept those uploads (the ChatKit component likely sends them to a backend endpoint). We can integrate this with Fountain-Store: for instance, when a user uploads a file via ChatKit UI, have the gateway store it (maybe in a configured S3 bucket or local storage via Fountain-Store) and then provide the file path or content to the FountainKit agent. This opens up use cases like, "Analyze this document" or "Describe this image" handled by FountainKit tools. Ensure the endpoint for uploads is in place (e.g. `POST /api/chatkit/upload`) and that the agent knows how to retrieve the file (the session context could carry a reference to the stored file). This way, attachments become part of the conversational workflow.

- **Thread management and history:** ChatKit supports multiple threads and organizing conversations [24]. FountainKit can support this by allowing separate session IDs or conversation objects for each thread. The client token/session we implemented already serves this purpose: each new session (client_secret) corresponds to a fresh conversation. We should ensure the front-end can start a new session when the user wants a new chat (perhaps by calling the session API again). On the backend, if we want to persist threads across page reloads for a user, we could tie session IDs to user accounts or store them in Fountain-Store. But even without a login system, ChatKit's UI might just keep in memory threads during one visit. For now, supporting multiple simultaneous sessions is mostly on the UI side – the backend just needs to handle any valid session ID references it sees.

- **Source citations and references:** If FountainKit provides source citation data (say the agent uses a retrieval-augmented generation that includes source URLs or document titles), ChatKit can display those as annotation markers [25]. We should format FountainKit's responses to include citations in the way ChatKit expects. This could be as simple as appending footnote-style text ("[1]") and providing a list of sources at the end of the answer. Or ChatKit may have a structured way to pass metadata (perhaps via a special function call result that the UI renders as citations). In any case, **widen the usage** of FountainKit's ecosystem by integrating, for example, **Fountain-Store as a vector DB** for retrieving documents, and then show the sources in ChatKit UI. FountainKit's modular design (with OpenAPI-described tools) makes it feasible to plug in a knowledge base tool, use it in the agent's reasoning, and surface the results in ChatKit.

In summary, by thoughtfully connecting FountainKit's outputs to ChatKit's display capabilities, we ensure that **users get the richest experience**: real-time answers with transparency and interactivity, powered by FountainKit's robust back-end logic. This maximizes the utility of both ChatKit and FountainKit.

Developer Experience: Internal vs. External Users

One challenge is to make this integration beneficial and approachable for both **internal developers** (the Fountain team itself, building the platform and apps) and **external app developers** (third parties

or clients using the Fountain platform to build their own AI applications). Let's differentiate the perspectives:

- **Internal Developers (FountainKit maintainers & app builders):** For the Fountain team, integrating ChatKit means you have a ready-made UI to pair with your back-end. Internally, you can create new "FountainKit Apps" rapidly by focusing on back-end logic (personas, tools, models) while relying on ChatKit for front-end. For example, if you develop a new agent that performs specific tasks (say, an AI audio engineer using the SamplerAgent from earlier), you can spin up a new ChatKit frontend for it with minimal effort. The internal developer workflow might look like: *Define a new agent persona or capability in FountainKit (perhaps add a new plugin or OpenAPI spec) → Configure the gateway to load it → Drop a new index.html (or extend the existing one) that maybe selects that persona/workflow → Voilà, a new AI tool with UI is live.* FountainKit's open, **OpenAPI-first architecture** means adding capabilities is straightforward – e.g., simply adding a new OpenAPI spec can make new tools available to the reasoning system [26]. No front-end coding is needed beyond maybe adjusting the theme or text, because ChatKit can handle the UI. This allows the internal team to **swiftly iterate** on AI app ideas. (Truly "swiftly" – both in terms of speed and using Swift code!)

- **External Developers (Using Fountain as a Platform):** Once the ChatKit + FountainKit integration is stable, you can offer it as a platform feature. External developers could build their own applications on FountainKit by either embedding the ChatKit component in their product or by using the hosted UI. Two modes are possible:
  (a) *Embedding ChatKit in external apps:* For example, a third-party web app could include the `<openai-chatkit>` web component (or use the ChatKit React library) and point it to your FountainKit deployment. They would use an API key you provide and perhaps a workflow ID or session endpoint that targets a specific agent persona. With minimal JavaScript, they get a chat interface in their own app that talks to FountainKit's AI. To support this, you'd publish documentation (similar to OpenAI's) instructing how to set `CHATKIT_API_BASE` to your endpoint, how to obtain client tokens (maybe the external developer calls your `/api/chatkit/session` with their credentials), and any customization options. Essentially, FountainKit could become a **self-hosted alternative to OpenAI's Agent platform** for others, and ChatKit makes integration easy for them.

(b) *Using Fountain's hosted frontend:* Alternatively, external users who are less technical might just use a FountainKit-provided UI (the publishing frontend we set up). For instance, if you host a public URL for your AI assistant, they can visit it and use the chat. This is more like offering a product (e.g. a support chatbot or a creative assistant) directly to end-users. In this scenario, external *developers* might not be involved at all – it's more about external *users*. Still, having a polished UI via ChatKit helps drive adoption.

In both cases, **tutorial-style documentation is key**. We should create guides for: - *"How to Build a FountainKit App with ChatKit"* – showing how to define a new agent (perhaps by writing an OpenAPI tool spec or configuring a prompt/persona), and how to stand up a new chat interface for it. This guide benefits internal devs or power users of the platform. - *"Using the FountainKit Chat API (for Developers)"* – explaining the endpoints, auth, and how to embed ChatKit externally. This is akin to OpenAI's ChatKit integration guide but tailored to our platform. We'd include code snippets, e.g., how to fetch a client token from our service and initialize the ChatKit component (very much like the openAI guide, but pointing to our URLs) [8] [9].

By catering to both audiences, we **widen the usage perspective across the entire Fountain Coach ecosystem**. Internal devs will see how they can quickly compose new solutions by mixing ChatKit UI with FountainKit's back-end modules (LLM gateway, Fountain-Store, custom tools in Teatro or others), and external devs will appreciate that they can leverage a proven UI and robust backend without starting from scratch.

Swift Implementation and Next Steps

A huge advantage of this integration is that **FountainKit's backend is written in Swift**, which is a fast and safe systems language – so we can extend it swiftly (pun intended) to support ChatKit. The Gateway's modular design (plugins and OpenAPI-defined services) means adding new routes for ChatKit or tweaking behavior can often be done with a few lines of Swift code. For example, writing the session endpoint or an upload handler can piggyback on FountainKit's existing HTTP server framework. As long as we conform to ChatKit's expected API shape, the heavy lifting (AI reasoning, etc.) is handled by FountainKit's core logic.

In terms of *"swift" next steps*: - **Finalize the ChatKit fork:** Ensure that our forked `Fountain-Coach/chatkit-js` is aligned with these integration needs. This might involve defaulting `CHATKIT_API_BASE` to our gateway, adjusting any hard-coded OpenAI endpoints, and perhaps customizing the theme to match FountainCoach branding. The fork could also expose new hooks or event handlers if we want to display additional info (e.g., logging events for debugging). - **Testing end-to-end:** Before writing extensive docs, test the full flow internally. Use a dummy persona (even just a simple echo or a GPT-3.5 wrapper) to verify that a message goes from ChatKit UI → FountainKit → LLM and back with streaming. Iron out any issues in handshake (e.g., token expiration, CORS if hosting front-end separately, etc.). This testing will inform the tutorial details. - **Documentation and tutorials:** As planned, write thorough documentation targeting the two groups mentioned. Include code snippets and perhaps screenshots of the working chat UI. For internal docs, also include how to extend the system (for instance, "to add a new tool for the agent, drop its OpenAPI spec in the right location and FountainKit can auto-discover it [26]"). This shows the full spectrum of the Fountain Coach code in action – from low-level secret management to high-level front-end integration. - **Leverage the full FountainCoach repository spectrum:** The beauty of this integration is that it can highlight many components: - *FountainKit* for orchestrating AI (with plugins for auth, curation, etc.), - *Fountain-Store* for persistence (conversation logs, knowledge base), - *swift-secretstore* for credentials (API keys for LLMs or tool access), - *Teatro* (if any multimedia output or advanced rendering is needed by the agent – not directly used in a text chat, but one could imagine an agent producing a musical snippet or visualization with Teatro and sending it as a widget).
By showing how each part can be used in the context of a ChatKit-powered app, we demonstrate a comprehensive platform. For instance, an agent could call **Teatro** to render a chart or MIDI and send back an image/audio widget, stored via Fountain-Store, displayed via ChatKit. This may be beyond the basic tutorial, but it's a compelling vision for the future.

In conclusion, integrating ChatKit-JS into FountainKit can be done *swiftly* and yields a powerful synergy. We described both quick and deep integration paths, and detailed how to implement the deep integration step by step. By **documenting the process thoroughly**, we empower internal developers to rapidly create new AI-driven "Fountain Apps" and enable external developers to harness FountainKit through an easy front-end. This fusion of ChatKit's polished UI with FountainKit's robust backend opens the entire FountainCoach ecosystem to wider usage, demonstrating its full spectrum – from secure secret management to scalable AI reasoning – in one cohesive experience. **With ChatKit and FountainKit together, developers get the best of both worlds: a first-class chat interface and a powerful, extensible AI platform driving it** [1] [3]. Enjoy building with them, and happy chatting!

Sources: ChatKit-JS documentation and OpenAI starter app [1] [9]; FountainKit Gateway and plugin docs [4] [5]; Fountain-Store and SecretStore integration notes [19] [11]; FountainCoach code examples for agent tools and OpenAPI integration [20] [26].

---

[1] [2] [3] [7] [8] [17] [21] [22] [23] [24] [25] GitHub - openai/chatkit-js
https://github.com/openai/chatkit-js

[4] [5] README.md
https://github.com/Fountain-Coach/FountainKit/blob/78c0d2d5206ca2b828ec2bbc4573aa0bee2b3ae2/Packages/FountainApps/Sources/gateway-server/README.md

[6] PublishingFrontendPlugin.swift
https://github.com/Fountain-Coach/FountainKit/blob/78c0d2d5206ca2b828ec2bbc4573aa0bee2b3ae2/Packages/FountainApps/Sources/gateway-server/PublishingFrontendPlugin.swift

[9] [18] GitHub - openai/openai-chatkit-starter-app: Starter app to build with OpenAI ChatKit + Agent Builder
https://github.com/openai/openai-chatkit-starter-app

[10] [13] [14] [15] [16] llm-gateway.yml
https://github.com/Fountain-Coach/FountainKit/blob/78c0d2d5206ca2b828ec2bbc4573aa0bee2b3ae2/Packages/FountainSpecCuration/openapi/v1/llm-gateway.yml

[11] [12] README.md
https://github.com/Fountain-Coach/swift-secretstore/blob/724b285759206fc87cd56066d265e70215d83d28/README.md

[19] [20] [26] README.md
https://github.com/Fountain-Coach/midi2sampler/blob/9c4393b145308079a9f5224cc9e9d341c06a104d/README.md