A2UI blog post

Introduction

A2UI (Agent-to-UI) is a protocol designed by Google to standardize how AI agents communicate with user interfaces. Instead of tightly coupling agents to specific frontends, A2UI defines a clear contract for intent, state, and actions - making it easier to build interactive, agent-driven experiences that are portable, composable, and UI-agnostic. As AI systems move from passive responses to active collaboration, A2UI provides the missing layer between intelligence and interaction.

In simple words is - instead of always receiving text responses, using A2UI you can have an interactive UI rendered across devices, allowing the users, for example, to tap on buttons to perform actions, instead of typing.

The specification for this protocol is still not final, but sure enough, at Stream we already have an integration with our Chat SDK.

In this post, we will see how we can integrate the Stream SwiftUI SDK with an A2UI compatible rendering SDK called GenUI. The code in this article was written with the 0.8 version of the specification.

What are we building?

We are going to take our AI assistant iOS implementation and extend it to support a custom attachment that will render responses compatible with A2UI.

The A2UI responses would be about listing top 5 restaurants, and making a (fake) reservation to one of those. It's the same Restaurant Finder demo which can be found in the official a2ui repo, but optimized for our chat integration.

You can find a video of how the end result will look like here:

attachment:653b4f8b-8834-4a61-b664-2923c7a18fb1:demo-stream.mov

As you will see, adding support for A2UI in existing products is possible, and it can work well with the rest of your product.

You can find a working version of this project, on iOS and NodeJS.

A2UI 101

The A2UI v0.8 specification defines a streaming, JSONL-based protocol that enables AI agents to declaratively describe user interfaces as a sequence of abstract UI messages. It was designed for progressive rendering and LLM friendliness: components are represented in a flat list with string IDs, and messages like `surfaceUpdate`, `dataModelUpdate`, and `beginRendering` allow a client to render and update UI incrementally without waiting for a complete payload.

Interaction in v0.8 flows through defined message types and a shared component catalog: the agent streams component definitions and data model changes over Server-Sent Events (or equivalent transport), and the client maps abstract component types (e.g., `Button`, `TextField`, `Row`) from its catalog to native views. The UI's reactive behavior is driven by separate client-to-server event messages when users interact with the interface; these events feed back into the agent's logic and may result in further UI messages.

In contrast, v0.9 (draft) represents a philosophical and architectural shift toward a prompt-first design. Rather than optimizing solely for structured output (like strict JSONL that LLMs emit via function calling), v0.9 embeds JSON schemas directly into the model's system prompt so that the model generates UI messages in context.

Now that we know the basics of A2UI, let's see how we can support it, both server-side and on a mobile app.

Server-side integration

We will take an existing server-side SDK (Stream Chat AI SDK), that integrates Stream Chat with Vercel's AI SDK. This integration allows us to easily switch between LLMs, while keeping a unified Stream Chat interface.

As an addition, we want UI interactions (button clicks, form submits, etc.) to drive the assistant

the same way typed text does. That means we would need to:

1. Detect the A2UI payload on incoming messages.
2. Convert that payload into a prompt the model can understand.
3. Allow integrators to attach structured A2UI response data to the outgoing Stream Chat message without rewriting agent logic.

Step 1: Add hooks for input and output customizations

We're introducing two optional hooks:

- `userMessageHandler`: lets you intercept an incoming message. If you return `true`, the agent stops and you handle the message yourself.
- `finalMessageAugmentor`: lets you add structured fields to the final message update that is sent to Stream Chat.

Below is an example of how you can pass these hooks when you create an `AgentManager` (available in the Stream Chat AI SDK):

```
const agentManager = new AgentManager({
  serverToolsFactory: () => createDefaultTools(),
  agentIdResolver: buildAgentUserId,
  finalMessageAugmentor: restaurantA2uiAugmentor,
  userMessageHandler: createA2uiInteractionHandler(),
});
```

Step 2: Convert an A2UI interaction into a prompt

Incoming messages can include an `a2ui_interaction` payload. When we see one, we:

1. Read `a2ui_interaction` from `message.extraData` (or top-level fields).
2. Parse it as JSON.
3. Extract `userAction` and its `context`.
4. Produce a natural-language description, for example:

- `USER_WANTS_TO_BOOK: {restaurant}`
- `User submitted a booking for {restaurant} ...`

5. Replace the raw `text` with this description before sending it to the model.

This makes A2UI events feel like typed input, so the model can reason about them without special casing.

```
const base = (message as { extraData?: Record<string, unknown
> }).extraData;
const raw =
  (base?.a2ui_interaction ??
    (message as unknown as Record<string, unknown>)['a2ui_int
eraction']) as
    | string
    | undefined;
const payload = raw ? (JSON.parse(raw) as InteractionPayload)
: undefined;
const actionName = payload?.userAction?.name;
```

```
switch (interaction.actionName) {
  case 'book_restaurant':
    await channel.sendMessage({ text: '', ai_generated: true,
a2ui: payload } as any);
    return true;
  case 'submit_booking':
    await channel.sendMessage({ text: '', ai_generated: true,
a2ui: payload } as any);
    return true;
  default:
    return false;
}
```

Step 3: Convert the model output into an A2UI response

When the model finishes, we send a final message update to Stream Chat. This is the
place to attach structured A2UI response data.

Here is the flow:

1. The response handler builds the final `set` payload that will be sent to `partialUpdateMessage`.
2. If `finalMessageAugmentor` is provided, it receives the final text, the message, the channel, and the full model message list.
3. The augmentor returns an object with extra fields (for example, an A2UI response
   payload).
4. Those fields are merged into the final `set` payload and sent to Stream Chat.

In practice, this means your augmentor can return something like:

- `a2ui_response`: the structured response you want the client to render.
- Any other metadata your UI needs.

The agent keeps the standard text update, and the A2UI response is also available in the same Stream Chat message update.

```
const payload = buildRestaurantPayload(restaurants, resolvedT
itle, userInput);
return {
  a2ui: payload,
};
```

```
await channel.sendMessage({
  text: '',
  ai_generated: true,
  a2ui: payload,
} as any);
```

iOS integration

Next, let's see how we can add support for the A2UI protocol in an existing app that integrates Stream Chat.

The support for custom attachments in Stream Chat helps us implement this in a simple way.

At a high level, the flow looks like this:

- The server sends a message with an `a2ui` payload.
- The iOS app renders the payload as a GenUI surface.
- The user taps or submits a form on that surface.
- The iOS app sends a new Stream Chat message with `a2ui_interaction` metadata.

Step 1: Add the GenUI package

First, you need to add this unofficial A2UI package that implements the A2UI protocol on iOS and SwiftUI.

The package can be added via Swift Package Manager, which is the standard way of adding dependencies on iOS.

Step 2: Handle the `a2ui` payload

Incoming Stream Chat messages can now optionally include an `a2ui` payload. The sample parses it from the message's `extraData` into a strongly typed model for rendering, so downstream code can work with a real surface id and message array.

This is the bridge between raw JSON in Stream Chat and a renderer-friendly model. Once this is in place, the rest of the UI code can work with `A2uiPayload` instead of loosely typed dictionaries.

```
struct A2uiPayload {
    let surfaceId: String
    let messages: [A2uiMessage]
```

```
    init?(rawJSON: RawJSON?) {
        guard let rawJSON, case let .dictionary(dictionary) =
rawJSON else { return nil }
        guard let surfaceId = dictionary["surfaceId"]?.string
Value ??
              dictionary["surface_id"]?.stringValue else { re
turn nil }
        guard let rawMessages = dictionary["messages"]?.array
Value else { return nil }

        let parsed = rawMessages.compactMap { rawMessage -> A
2uiMessage? in
            guard let json = rawMessage.jsonMap else { return
nil }
            return try? A2uiMessageFactory.fromJson(json)
        }

        guard !parsed.isEmpty else { return nil }
        self.surfaceId = surfaceId
        self.messages = parsed
    }
}
```

Step 3: Render A2UI messages in the message list

In Stream Chat's SwiftUI SDK, the view customizations are done by implementing a `ViewFactory` protocol, which exposes slots that you can use to provide your own implementation of certain components.

One of those slots is for custom attachments.

```
    @ViewBuilder
    func makeCustomAttachmentViewType(
        for message: ChatMessage,
        isFirst: Bool,
        availableWidth: CGFloat,
```

```
        scrolledId: Binding<String?>
    ) -> some View {
        if let payload = A2uiPayload(rawJSON: message.extraDa
ta["a2ui"]) {
            GenUIView(payload: payload, message: message, cha
tClient: chatClient)
        } else {
            let isGenerating = message.extraData["generatin
g"]?.boolValue == true
            let displayText = message.extraData["a2ui_display
_text"]?.stringValue ?? message.text
            StreamingMessageView(
                content: displayText,
                isGenerating: isGenerating
            )
            .padding()
        }
    }
```

The view factory switches between normal streaming text and A2UI rendering based
on the presence of an `a2ui` payload.

This keeps the app behaving regularly when there is no A2UI data, and it unlocks rich UI when there is. The fallback text also means you can still show a readable summary even if a surface fails to render.

This keeps A2UI responses as UI surfaces while normal responses still render as streaming text. The optional `a2ui_display_text` keeps the fallback text aligned with the action context.

Step 4: Host GenUI surfaces and forward interactions

When a surface is on screen, we spin up a renderer with `A2uiMessageProcessor`, then listen for user actions and forward them to Stream Chat.

This is the "bridge" between the UI surface and the chat backend. The processor tracks the A2UI surface state, and the `onSubmit` hook is where we turn button

presses or form submissions into Stream Chat messages.

```
struct GenUIView: View {
    @StateObject private var renderer: A2uiSurfaceRenderer

    init(payload: A2uiPayload, message: ChatMessage, chatClie
nt: ChatClient) {
        _renderer = StateObject(
            wrappedValue: A2uiSurfaceRenderer(
                payload: payload,
                message: message,
                chatClient: chatClient
            )
        )
    }

    var body: some View {
        GenUiSurface(host: renderer.processor, surfaceId: ren
derer.surfaceId)
            .padding(16)
            .onDisappear { renderer.dispose() }
    }
}
```

GenUI needs a message processor to manage the surface, and this hook is where we
intercept button clicks or form submissions.

Step 5: Send A2UI interactions back to the channel

User actions are converted into a new message with extra data so the server-side agent can react. The interaction forwarder preserves the raw `a2ui_interaction` JSON and adds helpful fields for the SDK to interpret.

In the sample, `A2uiSurfaceRenderer` wires this up for you. It creates an `A2uiInteractionForwarder`, subscribes to `processor.onSubmit`, and calls `forwardInteraction` whenever a user taps a button or submits a form.

```
processor.onSubmit
    .sink { [weak interactionForwarder] interaction in
        interactionForwarder?.forwardInteraction(interaction)
    }
    .store(in: &cancellables)
```

The forwarder connects to Stream Chat by creating a `ChannelController` from the `ChatClient` and calling `createNewMessage`. This sends a regular chat message with the action metadata attached in `extraData`, so the server-side agent sees it like any other user input.

```
let controller = chatClient.channelController(for: channelId)
controller.createNewMessage(text: messageText, extraData: ext
raData)
```

By sending a plain chat message with rich metadata, we keep the transport simple and the server-side logic consistent. The SDK can extract and transform the action data without requiring any client-specific protocol.

Here is the wiring from the sample so you can see how the forwarder is used:

```
final class A2uiSurfaceRenderer: ObservableObject {
    let processor: A2uiMessageProcessor
    private var cancellables: Set<AnyCancellable> = []
    private let interactionForwarder: A2uiInteractionForwarde
r?

    init(payload: A2uiPayload, message: ChatMessage, chatClie
nt: ChatClient) {
        let catalog = CoreCatalogItems.asCatalog()
        let processor = A2uiMessageProcessor(catalogs: [catal
og])
        payload.messages.forEach { processor.handleMessage
($0) }
        self.processor = processor
```

```
        self.interactionForwarder = A2uiInteractionForwarder(
            chatClient: chatClient,
            channelId: message.cid,
            sourceMessageId: message.id,
            surfaceId: payload.surfaceId
        )

        processor.onSubmit
            .receive(on: DispatchQueue.main)
            .sink { [weak interactionForwarder] interaction i
n
                interactionForwarder?.forwardInteraction(inte
raction)
            }
            .store(in: &cancellables)
    }
}
```

`UserUiInteractionMessage` is emitted by the GenUI SDK when a user action occurs on the
surface. In other words, the `interaction` you receive in `processor.onSubmit`
is created by GenUI based on the button action or form submission data defined in the A2UI payload.

```
let metadata = A2uiActionMetadata(from: interaction.text)
let messageText = metadata.query ?? metadata.description

var extraData: [String: RawJSON] = [
    "a2ui_interaction": .string(interaction.text),
    "a2ui_surface_id": .string(surfaceId),
    "a2ui_display_text": .string(metadata.description)
]

if let actionName = metadata.actionName {
    extraData["a2ui_action_name"] = .string(actionName)
}
```

```
if let label = metadata.label, !label.isEmpty {
    extraData["a2ui_action_label"] = .string(label)
}

if let query = metadata.query {
    extraData["a2ui_query_text"] = .string(query)
}

if let rawContext = metadata.rawContext {
    extraData["a2ui_action_context"] = rawContext
}

controller.createNewMessage(text: messageText, extraData: ext
raData)
```

The SDK extracts `a2ui_interaction` from extra data and turns it into a model prompt.

Conclusion

In this sample, we have seen how to support the A2UI protocol in an existing app powered by Stream Chat.

We have seen what changes we need to do server-side, to both enrich the message responses with A2UI-compatible payload, as well as how to handle user-triggered requests from the generated UI.

On the iOS side, we have seen how to extend Stream Chat's SwiftUI SDK to support generating the A2UI-driven UI.

This integration shows that we can add rich, interactive experiences without breaking the chat model or building a separate protocol. Everything stays in the Stream Chat message flow, which keeps the system flexible, testable, and easy to extend.