Sierra Web SDK

Overview

The Web SDK enables you to display a Sierra agent chat window (The Sierra chat UI) in a web application. It is a simple JavaScript API with a number of customization options to enable your Sierra agent to look and feel like the rest of your web experience.

Every Sierra agent has a globally unique **agent token** that identifies your agent. That token is included directly in the Web SDK `<script>` tag URL.

Agent token

Your agent token is:

```
mS6nPAgBlE2009YBlE21KNq6Ntoxx_ftrix3K0nhCHY
```

Setup

Script tag

Embed this tag anywhere on your page, e.g., within `<head>`.

```
<script type="module" src="https://sierra.chat/agent/mS6nPAgBlE2009YBlE21KNq6Ntox
```

Content Security Policy

If your site has a Content Security Policy (CSP), you will need to add the follow root URL:

```
https://sierra.chat
```

to your CSP's `img-src`, `script-src`, `style-src`, and `connect-src` to enable the `<script>` tag and the agent to load properly. For example:

```
Content-Security-Policy:
    img-src https://sierra.chat/ blob:;
    script-src https://sierra.chat/;
    style-src https://sierra.chat/;
    connect-src https://sierra.chat/;
```

Invocation

HTML

Once the embed script has been included on the page, you can launch the agent by adding the `data-sierra-chat` attribute to a link:

```
<a data-sierra-chat="modal" href="https://sierra.chat/agent/mS6nPAgBlE2009YBlE21K
    Launch agent
</a>
```

It is a best practice to include the `href` attribute as we did above so that the link falls back to the full-screen chat experience if the customer clicks the link before the `<script>` tag has loaded.

JavaScript

You can also invoke the agent programmatically. Once the `<script>` tag has loaded, a `sierra` JavaScript object is available in the global namespace, allowing programmatic access to the agent. The `SierraAPI` reference page describes the available methods.

Configuration options

Defining `sierraConfig`

To customize the behavior and appearance of the Sierra agent chat window, define `sierraConfig` in the global scope:

```
<script>
var sierraConfig = {
    display: "corner"
};
</script>
```

The `SierraAPIConfig` reference page describes the available configuration options. It supports a rich set of look and feel options, control over agent behavior, as well as callbacks that can be registered to handle conversation events.

By default, the agent is initialized as soon as the script tag is loaded, using the value of the `sierraConfig` object in the global scope. If you wish to do an on-demand initialization (e.g. after loading additional configuration information), set the initial `sierraConfig` to a `{waitForInit: true}` placeholder object. When you are ready to initialize, call `sierra.init(config)` with the populated `SierraAPIConfig` object.

```
var sierraConfig = { waitForInit: true };

// ...
async function initSierra() {
    const userToken = await fetchUserToken();
    sierra.init({
        display: "corner",
        variables: {
            "USER_TOKEN": userToken,
        },
    });
}
```

Display customization

The `SierraAPIConfig` reference page describes the built-in customization options (the `colors` and `typography` options, among others). If you need additional control over the Sierra chat UI, it is possible to use CSS, but you must be aware of the following:

The Sierra chat UI is displayed using the **shadow DOM** to minimize the risk of interference between it and the hosting page. Content within the shadow DOM is not affected by the host page CSS unless it is exposed via **CSS shadow parts**. The following parts are available to allow additional styling options:

- `launcher`: The chat launcher button (available when using the `launcher` option)
- `banner`: The banner displayed above the launched button (available when using the `proactiveBanner` banner option).

When using `unstyled` option, additional parts are available for styling. Their hierarchy is as follows:

- `chat-dialog-container` or `chat-fullscreen-container`, `chat-corner-container`: The overall chat container
  - `chat`: The root chat element.
    - `chat-title-bar`: The title bar
      - `chat-title-bar-logo`: The logo in the title bar
      - `chat-title-bar-text`: The text in the title bar
      - `chat-title-bar-button`: Buttons in the title bar (the two specific buttons are `chat-title-bar-button-actions` and `chat-title-bar-button-close`)
        - `chat-title-bar-button-icon`: Icons for buttons in the title bar
    - `chat-body`: The main chat content area, shown under the title bar
      - `chat-messages`: The list of chat messages
        - `chat-disclosure`: The disclosure text, if enabled
          - `chat-disclosure-link`: Links within the disclosure text, if any
        - `chat-bubble`: Chat message bubbles (messages of specific types are `chat-bubble-user`, `chat-bubble-assistant` and `chat-bubble-human_agent`)
          - `chat-assistant-typing-indicator-icon`: The typing indicator icon that appears when agent is typing
        - `chat-message-footer`: The text that displays below the chat message bubble
          - `chat-agent-name`: The agent name and icon portion of the footer
          - `chat-timestamp`: The timestamp portion of the footer
        - `chat-status-message`: The status text that displays between message bubbles (e.g. "Agent connected")
        - `chat-ended-message`: The status text that displays when a conversation is ended
        - `chat-transfer-message`: The message that displays once the chat is transferred

- `chat-transfer-message-icon`: The icon that's displayed within the transfer message

- `chat-input-root`: `div` that wraps all chat input elements, including pending file uploads, the message input box, and the send/upload/new buttons
- `chat-input-hidden`: Conditionally appears on the root `div` when the chat input is hidden by a custom attachment
  - `chat-input-container`: `div` containing the message input box and the send/upload/new buttons
    - `chat-input`: Textarea input for customer messages (removed when chat is ended)
    - `chat-send-button`: Send button for customer messages (may be disabled if sending is not possible)
    - `chat-upload-button`: Upload button for customer messages (only shown if conversation is active and file upload enabled)
      - `chat-upload-button-icon`: Icon for the upload button
    - `chat-new-button`: Button to start a new conversation, shown when chat is ended (removed when textarea is shown)

- `chat-actions`: The root element of the chat actions menu (when using the `unstyledChatActions` option`)
  - `chat-actions-print-transcript`: Menu item to print a transcript
  - `chat-actions-end-conversation`: Menu item to end the conversation
- `chat-confirm-end-conversation-title`: Title of the end conversation confirmation dialog
- `chat-confirm-end-conversation-body`: Body of the end conversation confirmation dialog
- `chat-confirm-end-conversation-footer`: Footer of the end conversation confirmation dialog
  - `chat-confirm-end-conversation-confirm-button`: Confirm button in the footer of the end conversation confirmation dialog
  - `chat-confirm-end-conversation-cancel-button`: Cancel button in the footer of the end conversation confirmation dialog

As an example, the `corner` chat container could be given a custom drop shadow via:

```
[data-sierra-chat-container]::part(chat-corner-container) {
    box-shadow: 0 1px 5px rgba(0, 0, 0, 0.8);


```

```
        border-radius: 20px;
    }
```

Agent state and personalization

To initialize your agent with state, e.g., a JSON web token for an authenticated customer, you can pass *variables* and *secrets* to your agent:

```
var sierraConfig: {
    // Initial values for conversation variables and secrets.
    // Variables and secret values should always be strings.
    variables: {
        "VAR_NAME": "value",
    },
    secrets: {
        "SECRET_NAME": "value",
    },
};
```

Secrets passed through the our SDKs are encrypted with a conversation-specific key. Secrets cannot be logged or made visible in Agent Studio.

Secret expiry handling

The `onSecretExpiry` callback enables automatic refresh of expired secrets during a conversation. When a secret expires, Sierra calls your callback to request a fresh value, maintaining the conversation flow without interruption.

Configuration

Implement the `onSecretExpiry` callback in your `sierraConfig`:

```
var sierraConfig = {
    // ... other configuration options ...

    onSecretExpiry(secretName) {
        // Return a fresh value for the secret, or null if the secret cannot be
        switch (secretName) {
            case "SESSION_TOKEN":
```

```
                return refreshSessionToken();
            default:
                console.warn(`Secret refresh not supported for: ${secretName}`);
                return null;
        }
    },
};
```

The callback accepts a secret name and returns either the refreshed value or null if the secret cannot be refreshed. You can return a value directly or a Promise that resolves to the value.

How it works

1. When your agent requests a secret refresh, Sierra detects the expired secret

`onSecretExpiry`

3. Your callback fetches a fresh token from your backend
4. Sierra updates the secret value and continues the conversation

The refresh happens asynchronously in the background, so the conversation continues smoothly without visible delays to the customer.

For implementation details and best practices, see how to handle token expiration in **Authentication**.

Custom embed script

In addition to the standard Sierra embed script, it is possible to define a custom embed script in your agent codebase. This allows you to:

- Expose a simpler or more bespoke API to the embedding website
- Hide Sierra-specific information like variables and secrets
- Coordinate releases of your agent with its embed script

To set up a custom embed script:

1. Add a **web bundle** if your agent does not already have one.
2. Add a `web/embed.ts` file. It can contain any custom logic that you want to run, and should eventually call **`sierra.init()`** with the appropriate configuration.
3. Build and release an agent as usual (**hot-swap** may be useful for iterating during development).

You can optionally also add a `web/embed.css` file to your agent to load additional CSS for your script. This is especially useful when using the `unstyled` option and `part` selectors described **above**. Note that this CSS is loaded in the context of the host page, not the Shadow DOM that the agent is rendered in.

Both JavaScript and CSS files may reference images that are in the agent `web/` directory. The build process will inline them as `data:` URLs in the generated custom embed script.

Once an agent has a custom embed script, it can be loaded by using the `custom` variant of the embed script URL:

```
<script type="module" src="https://sierra.chat/agent/mS6nPAgBlE2009YBlE21KNq6Ntox
```

As as example, here is a custom embed script that uses a hypothetical backend API to configure the agent. It also configures a **launcher** with a custom icon:

```
import iconURL from "./icon.png";
import { type SierraAPIConfig } from "@sierra/web";

async function loadSierraForUser() {
    // Backend API that returns returns information used to initialize the agent
    // It exchanges the authentication cookie for a user token that can be passe
    // to the agent.
    const response = await fetch("/api/sierra-config");
    const { greeting, userToken } = await response.json();

    const config: SierraAPIConfig = {
        display: "corner",
        customGreeting: greeting,
        secrets: {
            "USER_TOKEN": userToken,
        },
        launcher: {
            icon: {
                url: iconURL,
                width: 32,
                height: 32,
                alt: "Outfitters Logo",
            },

```

```
            },
        };
        sierra.init(config);
    }

    loadSierraForUser();
```

On your website, the script would be loaded, it would then fetch the user token from the backend API, and finally it would initialize the agent. If you wished to rename the `USER_TOKEN` secret or add additional configuration options, you could in the same code change update `web/embed.ts` *and* the agent code that uses the secret. Finally, you could then rely on those modifications being deployed together as part of an agent release.

Adding CSS to web bundles

To style your web bundle components, create a `web/main.css` file alongside your `main.tsx`:

```
/* web/main.css */
.my-attachment {
    padding: 16px;
    border-radius: 8px;
    background-color: #f3f4f6;
}
```

The CSS is automatically bundled and loaded with your web bundle.

You can use any CSS approach:

- **Pre-built CSS frameworks** like Bootstrap or Bulma work via standard imports
- **CSS-in-JS libraries** like styled-components or emotion are supported
- **Tailwind CSS** is automatically processed when you include `@tailwind` directives

Framework integration

React

The Sierra agent can be embedded in a React application by creating a custom component. It uses the `useEffect` hook to configure and load the Sierra agent. Specifically, it:

- Defines the `sierraConfig` global with the **agent configuration**, including any callbacks that need to be registered.
- Adds a remote script to load the Sierra agent. The `onLoad` callback can be used to wait for the `sierra` object to be available. Once it fires, any method from the **JavaScript API** can be called on it. It is also possible to store the `sierra` object in in React **state** or **ref**.

```tsx
import { useEffect } from "react";
import { SierraAPIConfig } from "@sierra/web";

export function SierraAgent({onConversationEnd}: {onConversationEnd: () => void}
    useEffect(() => {
        const config: SierraAPIConfig = {
            display: "corner",
            canEndConversation: true,
            onConversationEnd,
        };

        window.sierraConfig = config;

        const agentScript = document.createElement("script");
        agentScript.type = "module";
        agentScript.src = "https://sierra.chat/agent/mS6nPAgBlE2009YBlE21KNq6
        agentScript.onload = () => {
            // The `sierra` object is now available.
            window.sierra?.openChatModal();
        };
        document.head.appendChild(agentScript);

        return () => {
            document.head.removeChild(agentScript);
        };
    }, []);

    return null;
}
```

Next.js

The Next.js **Script** component can be used to load and configure the Sierra agent. Two instances are commonly used:

- An inline script to define the `sierraConfig` object. The value is specified using a template literal to have actual `{` and `}` characters.
- A remote script to load the Sierra agent. The `onLoad` callback can be used to wait for the `sierra` object to be available. Once it fires, any method from the **JavaScript API** can be called on it. It is also possible to store the `sierra` object in in React **state** or **ref**.

```
import Script from 'next/script'

export function SierraAgent() {
    <Script id="sierra-config">
        {`const sierraConfig = {
            display: "corner"
        };`}
    </Script>
    <Script
        type="module"
        src="https://sierra.chat/agent/mS6nPAgBlE2009YBlE21KNq6Ntoxx_ftrix3K0nhC
        onLoad={() => {
            // The `sierra` object is now available.
            window.sierra.openChatModal()
        }}
    />
}

```

Playground

Set the configuration below using valid JSON to experiment with different approaches:

```
{
    "preload": "timeout",
    "customGreeting": "How can I help you today?",
    "initialUserMessage": "",
    "display": "corner",
    "dimensions": {
        "width": "30em",
        "height": "40em"
    }
}
```

Try it