Cresta Client SDK Developer Guide

Introduction

This user guide documents the Cresta client SDK (APIs and packages) and usage snippets for external developers building their own contact center platform or application and integrating with Cresta conversation (chat and audio) services.

SDK packages

The Cresta SDK suite consists of the following packages:

1. **@cresta/client**: This is the core Vanilla JS framework agnostic client SDK with custom HTML elements for UI components.
   - Latest version: 0.21.0
2. **@cresta/client-sdk-sample**: This is the quickstart sample app using the Vanilla JS client SDK. In order to sign in with the app with popup, the app origin needs to be authorized. The default origins used by the sample app http://127.0.0.1:1234 and http://localhost:1234 should be authorized. Reach out to the Cresta team to do so. If you are using BYOID for authentication, there is no need to add authorized origins.
   - Latest version: 0.18.0
3. **@cresta/react-client**: This is the Cresta react components SDK for React developers. It provides react component wrappers for the Cresta web components. It is meant to be used with the **@cresta/client** package.
   - Latest version: 0.10.0
4. **@cresta/client-sdk-react-sample:** This is the quickstart sample app using the React client SDK for React developers. In order to sign in with the app with popup, the app origin needs to be authorized. The default origins used by the sample app http://127.0.0.1:1234 and http://localhost:1234 should be authorized. Reach out to the Cresta team to do so. If you are using BYOID for authentication, there is no need to add authorized origins.
   - Latest version: 0.22.0

5. **@cresta/client-sdk-emulator**: This is the Cresta emulator package. It is used to facilitate development using a mock Cresta AI server. It is useful for hermetic testing, local development or integration tests in an environment where outgoing network requests are not allowed, or when the AI model is not ready. It can be used with the provided sample apps (**@cresta/client-sdk-react-sample** and **@cresta/client-sdk-sample**). The emulator should run in the background and the sample app configuration service endpoint will need to be updated to point to the emulator server URL. Learn more about the emulator.
    - Latest version: 0.14.0
6. **@cresta/client-sdk-auth-sample**: This is the Cresta sample OIDC auth server. It is used to facilitate generation of RS256-signed OIDC ID tokens which can be used an external tokens for sign-in with Cresta (click here to learn more about this feature). It can be used with the provided sample apps (**@cresta/client-sdk-react-sample** and **@cresta/client-sdk-sample**). The server should run in the background and the sample apps configuration file **client-sdk-sample-config.json** will need to be updated to point to this URL. Learn more about the sample auth server. If you are planning to use BYOID to authenticate with Cresta, you don't have to use this server and you can use your own or a 3rd party OIDC service. This is just a sample and a packaged mechanism to readily generate these spec-compliant ID tokens.
    - Latest version: 0.3.0
7. **@cresta/client-sdk-utils**: This package provides additional customer-related utilities that can be used with the Cresta SDK. This includes support for HMAC client authentication that may be used when calling a proxy server to ensure data integrity (messages not tampered with) and to verify the authenticity of the request (coming from the expected sender). See below for more information.
    - Latest version: 0.1.0

Before you begin

In order to install any Cresta package, you will need to configure the Cresta npm registry URL in your npm or yarn package manager configuration.

For customers using npm package manager, add the Cresta npm registry URL for the **@cresta** scoped packages in the **.npmrc** file.

```
@cresta:registry=https://npm.cresta.com
```

For customers using yarn package manager, add the Cresta npm registry URL for the **@cresta** scoped packages in the **.yarnrc** file.

```
npmScopes:
  cresta:
    npmRegistryServer: "https://npm.cresta.com"
```

Now you can start install Cresta packages:

```
npm install --save @cresta/client
```

You will also need the following from the Cresta team:

- **Add your origin as authorized origin:** This is the origin where your production or development app will run. Reach out to Cresta to provide your origins. For the sample apps, the http://127.0.0.1:1234 and http://localhost:1234 are used.
  - This is an important security measure to only allowlist certain origins. Initially this will be open to all origins. If you are using BYOID for authentication, there is no need to add authorized origins and this step can be skipped.
- **client-sdk-sample-config.json**: The configuration file used to initialize the sample app. This is also used by the @cresta/client SDK in general. Reach out to Cresta to get the file.

If you are using `signInWithPopup`, you will also need Cresta account credentials to use for testing. Use the **director UI > User management** to create one if you don't already have one. You can also configure your SAML IdP for sign-in. Reach out to Cresta to help configure your IdP for SSO (Single Sign On).

If you are planning to use popup sign-in instead of BYOID, you can skip all the above requirements by using the emulator during development and testing. The emulator allows you to test and develop without the real AI model, customer ID, test account or IdP (SAML) setup for authentication, authorized origin registration, etc. So you can skip the add authorized origin step and the need for real account credentials. For the configuration file, you can use random values (except for the `serviceEndpoint` which needs to point to the emulator server, e.g. http://localhost:8081).

Why use the SDK?

We encourage using the Cresta Client SDK for integrating with Cresta service for various technical reasons over using the REST API directly for various technical reasons:

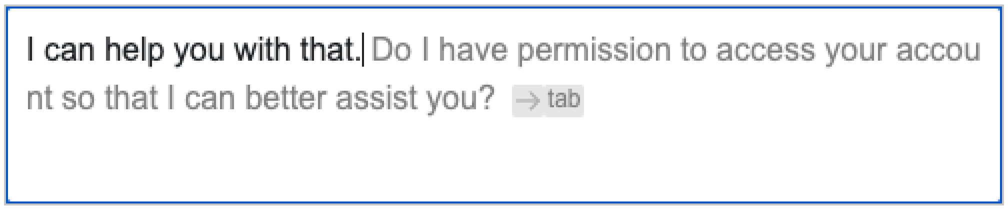
- The client SDK is simple and easy to use. It will abstract the underlying complexities of connecting and integrating with the Cresta backend. The underlying REST APIs and ClientSubscription WebSocket service can be quite complex to set up and wire correctly.
- The client SDK streamlines authentication via popup flow or BYOID (Bring Your Own ID) token exchange which follows the RFC 8693 spec.
- The client SDK facilitates collection (in the background) of various client analytics related to Cresta feature usage that are needed by the Cresta Director and Cresta Insights.

- The client SDK relies on a feedback loop to continuously improve its suggestions and data (e.g. notifies the backend when a sent message is picked from a Cresta suggestion even if it is modified).
- The SDK has a proven record of reliability (already used by the majority of Cresta Platform integration). It's very efficient in terms of computational power required. It was designed with low power machines in mind. It has a minimal footprint on the rest of the application in terms of processing and network bandwidth.The SDK even offers the ability to run all API calls to the Cresta server in a dedicated web worker thread to free up the main UI thread.
- The Cresta SDK is fail-safe: If the Cresta server is inaccessible, the SDK will run in no-op mode. For example, smart compose will continue to work but no hints will be presented.
- With the SDK, customers get the latest updates, bug fixes and new features for free with no additional work.
- The SDK offers multiple Cresta UI components that can be customized and restyled. Some components will mimic the underlying component's style and will have no visible footprint (e.g. the Smart Compose Overlay element which mimics the underlying input box style to show text hints as you type).

Cresta UI Components

The Cresta SDK provides built-in modular UI components for agent assist integrations for chat and voice (Custom HTML Elements or React components) that can be directly mounted into your application via the SDK. Additional advanced lower level APIs are also provided to build your own components. These are covered in more details in the later sections below.

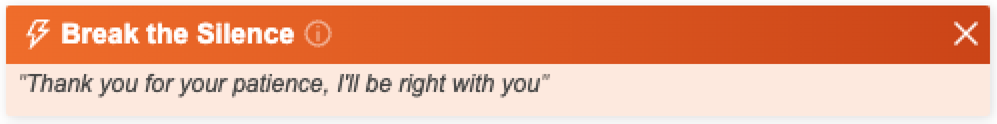
- Smart Compose:
  The Smart Compose overlay and standalone components provide auto-complete hints as the agent types into the conversation input box. It will display a grayed hint next to the input text. The hint will mimic the input text style but will be displayed in a gray color. By default, clicking the right arrow will select the next word in the hint (the selected word will mimic the color of the input text and the cursor will move to the end of the accepted word). Clicking the tab key will select the whole hint. The shortcut keys (right arrow and tab arrow) can be customized, along with their chip templates (e.g. the right arrow and tab icons shown at the end of the hint0.


I can help you with that. Do I have permission to access your account so that I can better assist you?

- Hints:
  Cresta hints are used to provide behavioral hints to the agent. Examples: Offer a discount, remind the user to greet the visitor, show empathy, etc.

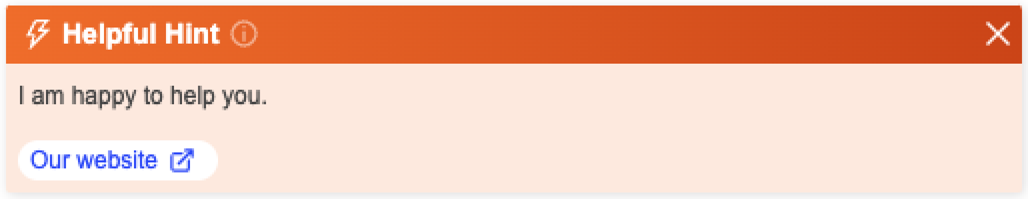
Cresta behavioral hints may have multiple subcomponents (title, tooltip description, optional text suggestion which when clicked will be populated in smart compose input box, optional links, etc.). The hint can be dismissed by clicking the **x** icon in the top right corner. Hints also are automatically dismissed after some delay (typically 30 seconds).

Example with suggestion text:



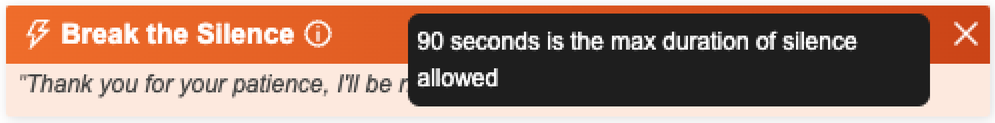
"Thank you for your patience, I'll be right with you"

Example with additional link:



I am happy to help you.

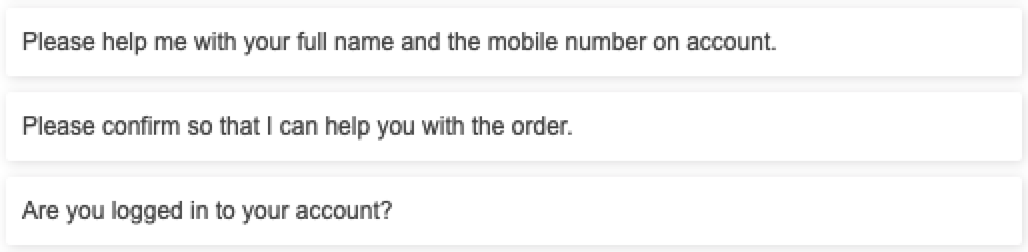
Example with the tooltip expanded showing the description of the hint.



"Thank you for your patience, I'll be r

90 seconds is the max duration of silence allowed

- Reply Suggestions
  These are the AI generated reply suggestions based on the context of the ongoing conversation. They typically can be 1 to 3 suggestions. The agent can click to select a reply suggestion and it will be auto-populated in the smart compose element where it can be further modified and then submitted.

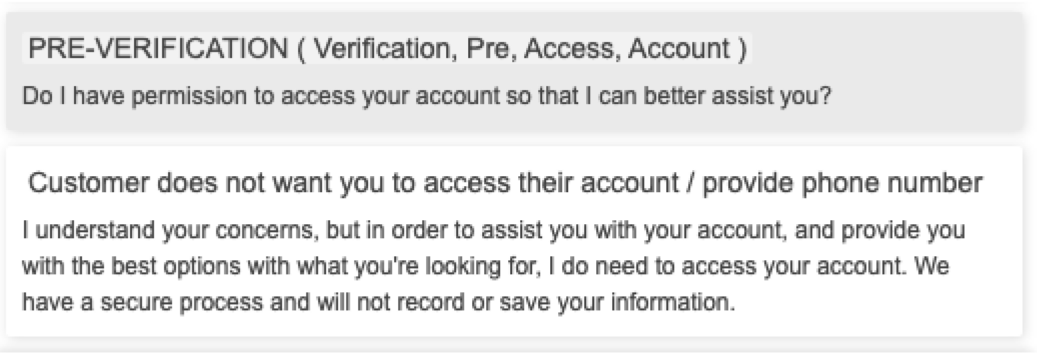


Please help me with your full name and the mobile number on account.

Please confirm so that I can help you with the order.

Are you logged in to your account?

- Canned Suggestions
  These are not AI generated and are based on preconfigured suggestions in Cresta Director and triggered based on various conversation criteria. In appearance, they are similar to AI reply suggestions but typically have a title in addition to the suggestion text. The agent can click to select a canned suggestion and it will be auto-populated in the smart compose element where it can be further modified and then submitted.


PRE-VERIFICATION (Verification, Pre, Access, Account)

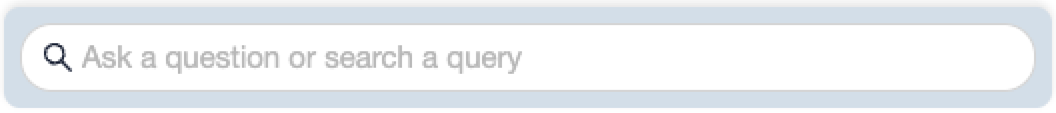
Do I have permission to access your account so that I can better assist you?

Customer does not want you to access their account / provide phone number

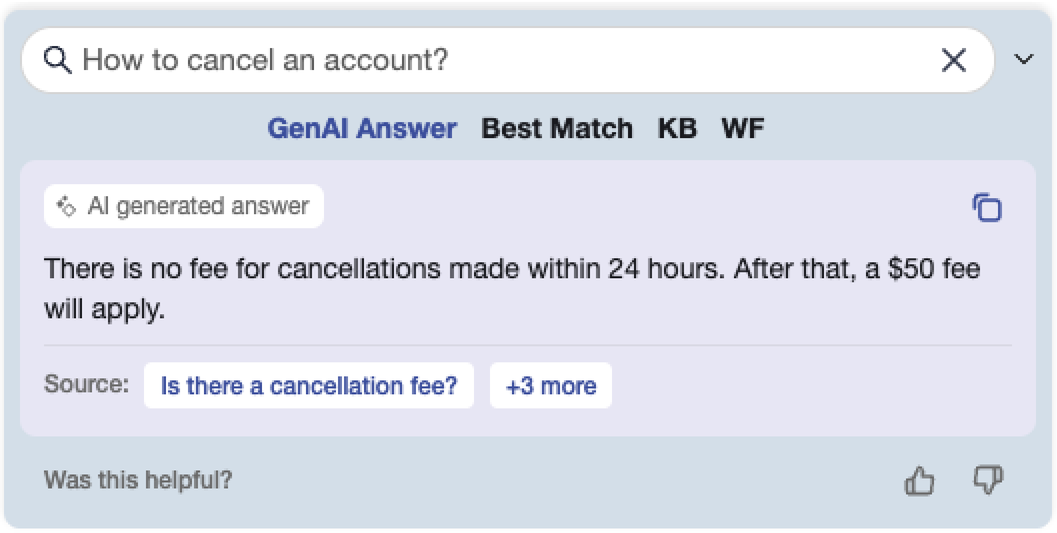
I understand your concerns, but in order to assist you with your account, and provide you with the best options with what you're looking for, I do need to access your account. We have a secure process and will not record or save your information.

- Omnisearch Component
  This includes the following features:
  - Cresta Knowledge Base search: Used to facilitate knowledge base search by entering text into the input box and getting related articles. For example, a visitor may inquire about a cancellation fee. The agent can search for that and get a summary and a link for more details on this topic.
  - Cresta Guided Workflow search: Used to facilitate guided workflow search by entering text into the input box and getting related workflows. For example, a visitor may inquire about some multi-step process (e.g. apply for a loan). The agent will use this component to search for this topic and then go through the related steps (linear or branching) while walking the visitor through them.
  - Cresta Generative Knowledge Assist: Used to facilitate generative knowledge assist search by entering text into the input box and getting AI generated reply and related articles. For example, a visitor may inquire about the return process. A retrieval question will be extracted by the backend and sent to the component. The agent can use the extracted question or enter the question manually and get the AI generated reply. **Note**: This feature is currently in beta phase and is still undergoing changes and improvements.

  The component presents an input box to query or search for various topics. When searching for various topics, multiple results may be shown depending on the enabled settings (e.g. KB, guided workflow or generative knowledge assist is enabled, etc). The zero-state of the component looks as follows:

  - Generative Knowledge Assist: When enabled the “GenAI Answer” tab is shown. Selecting this will show the AI generated answer.

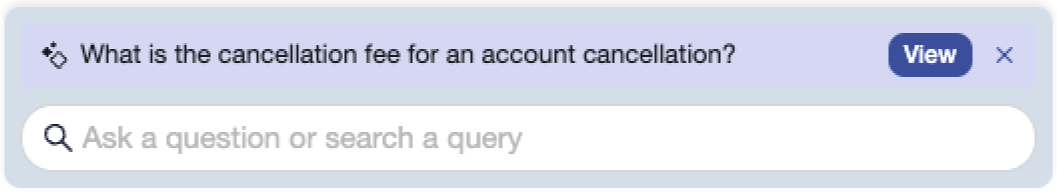


How to cancel an account?

There is no fee for cancellations made within 24 hours. After that, a $50 fee will apply.

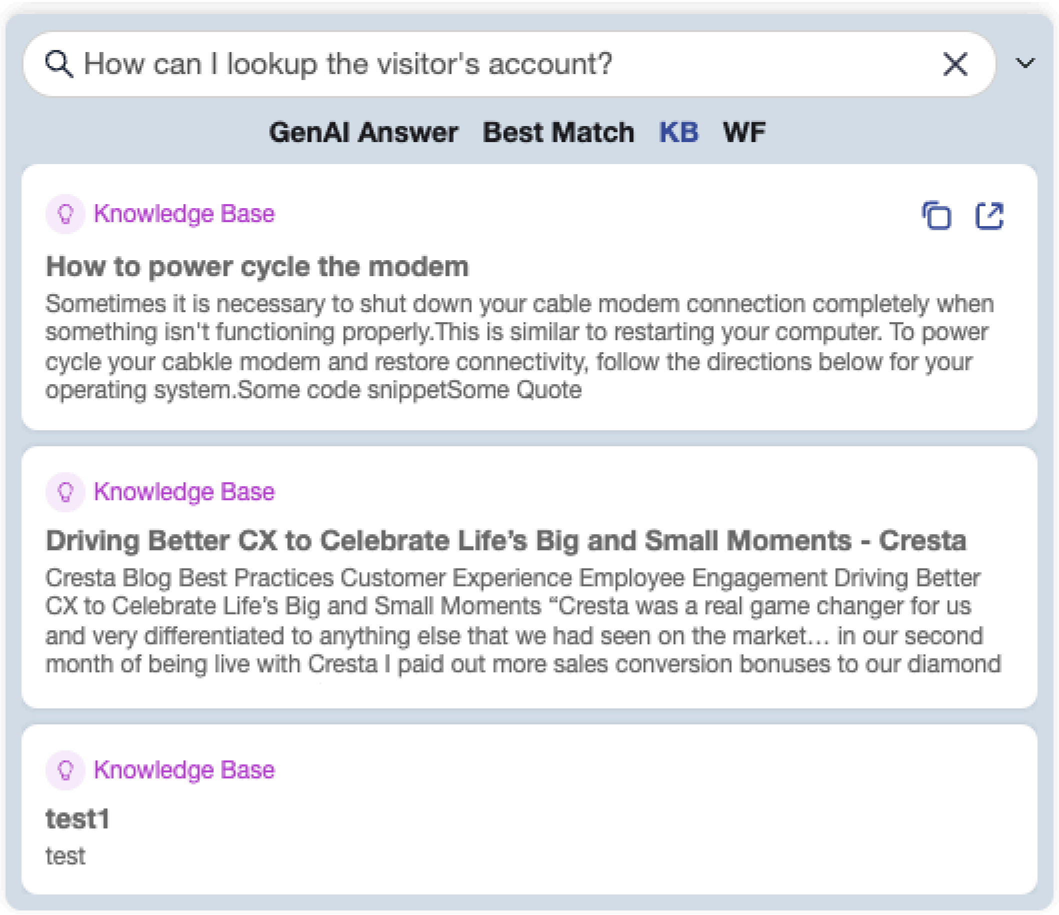
Source: Is there a cancellation fee? +3 more

Questions may also be auto-retrieved from the ongoing conversation. Clicking the “View” button will display the generated answer in the “GetAI Answer” tab.



What is the cancellation fee for an account cancellation?

- Knowledge Base: When enabled the “KB” tab is shown. Selecting this will show the knowledge base articles that match the search entry.


How can I lookup the visitor's account?

Knowledge Base

How to power cycle the modem

Sometimes it is necessary to shut down your cable modem connection completely when something isn't functioning properly.This is similar to restarting your computer. To power cycle your cabkle modem and restore connectivity, follow the directions below for your operating system.Some code snippetSome Quote

Knowledge Base

Driving Better CX to Celebrate Life's Big and Small Moments - Cresta

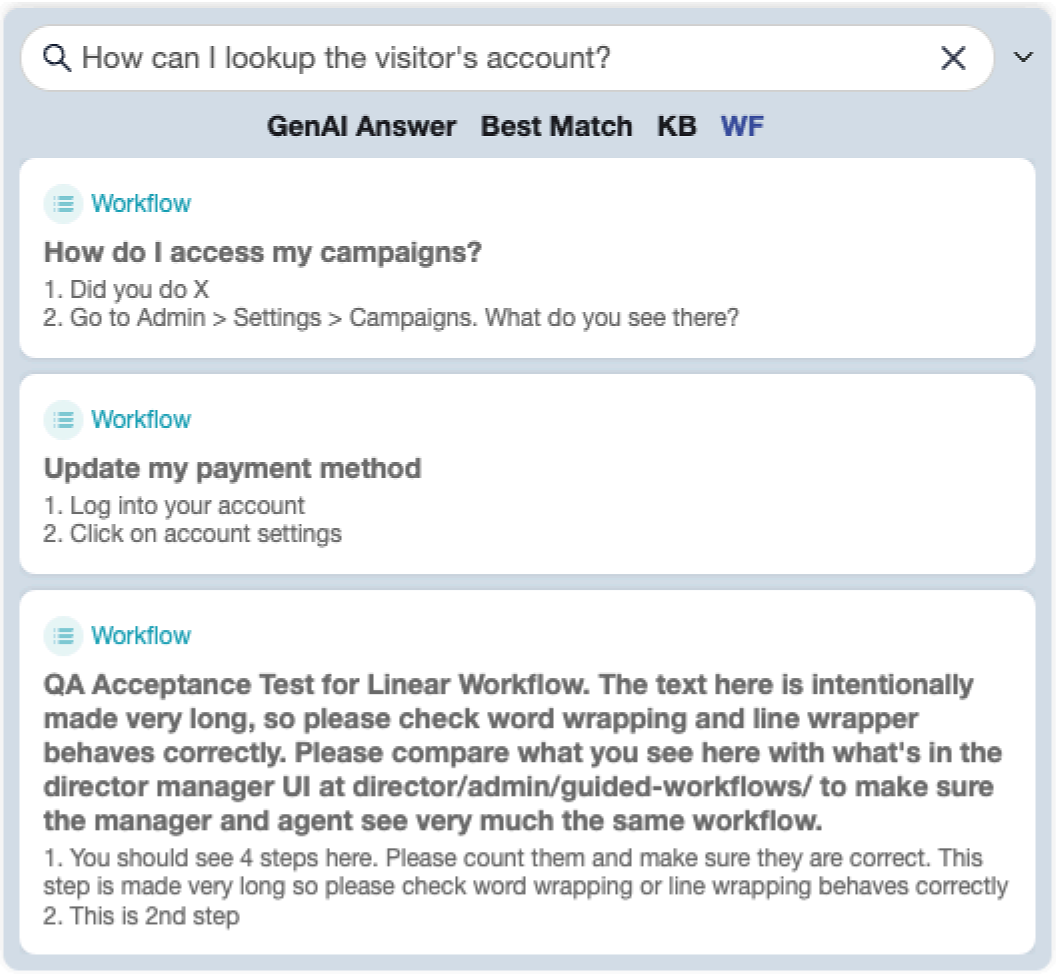
Cresta Blog Best Practices Customer Experience Employee Engagement Driving Better CX to Celebrate Life's Big and Small Moments "Cresta was a real game changer for us and very differentiated to anything else that we had seen on the market… in our second month of being live with Cresta I paid out more sales conversion bonuses to our diamond

Knowledge Base

test1

test

- Guided Workflow: When enabled the "GW" tab is shown. Selecting this will show the guided workflow answers that match the question. The guided workflow articles can be linear or branching guided workflows.



How do I access my campaigns?
1. Did you do X
2. Go to Admin > Settings > Campaigns. What do you see there?

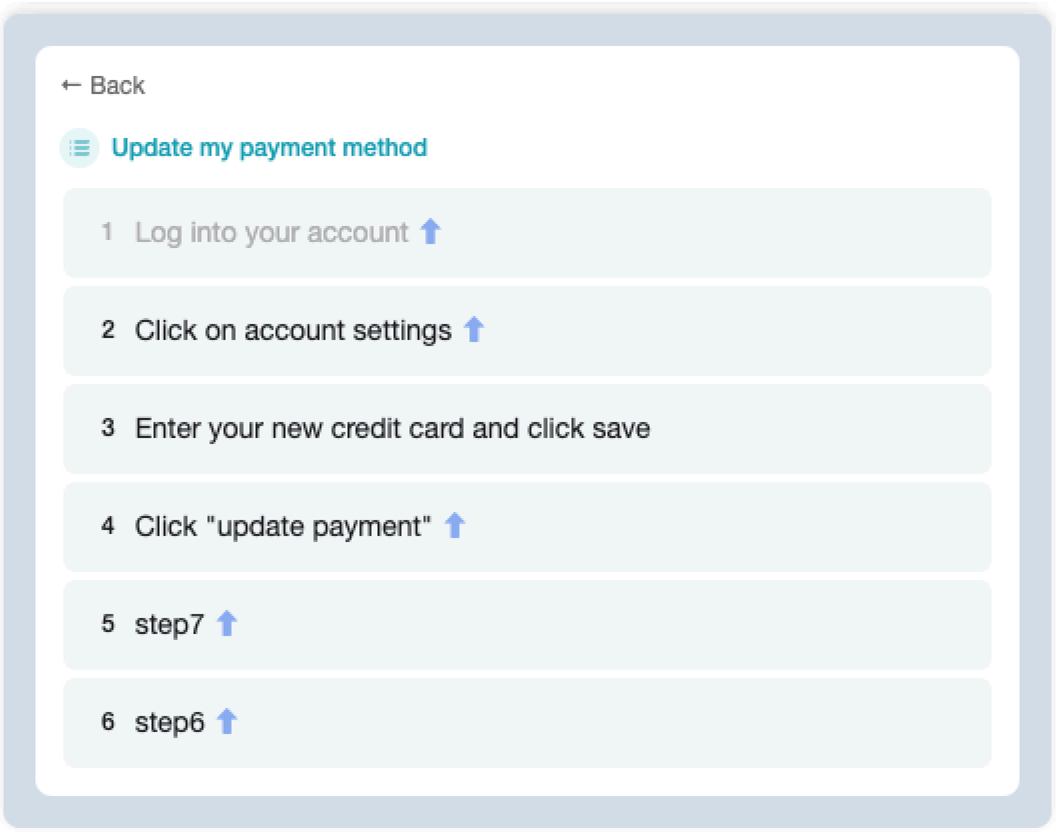
Workflow

Update my payment method
1. Log into your account
2. Click on account settings

Workflow

QA Acceptance Test for Linear Workflow. The text here is intentionally made very long, so please check word wrapping and line wrapper behaves correctly. Please compare what you see here with what's in the director manager UI at director/admin/guided-workflows/ to make sure the manager and agent see very much the same workflow.
1. You should see 4 steps here. Please count them and make sure they are correct. This step is made very long so please check word wrapping or line wrapping behaves correctly
2. This is 2nd step

Selecting a linear workflow, will display the sequential steps to follow.



← Back

Update my payment method

1 Log into your account

2 Click on account settings

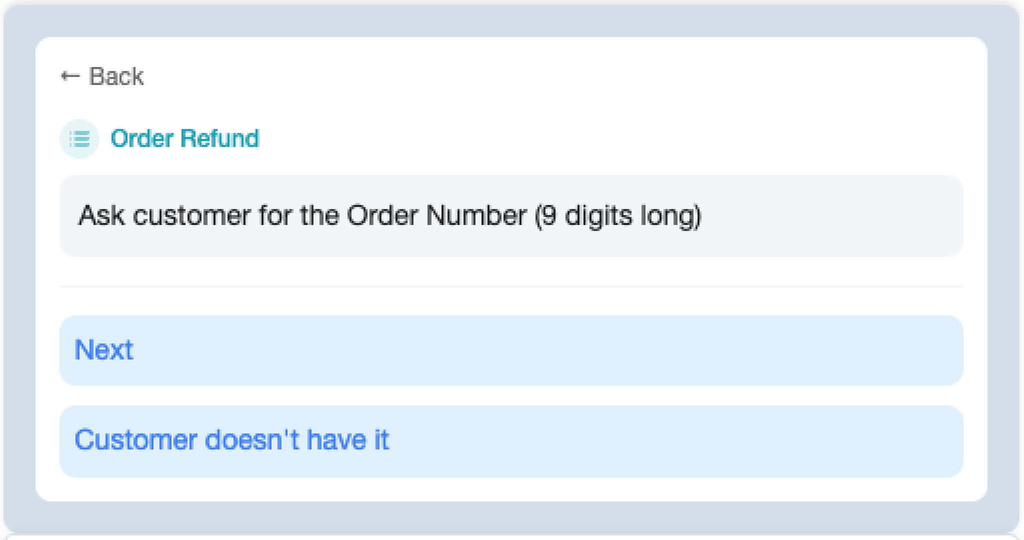
3 Enter your new credit card and click save

4 Click "update payment"

5 step7

6 step6

Selecting a branching workflow, will display the tree like branching steps to follow.



← Back

Order Refund

Ask customer for the Order Number (9 digits long)

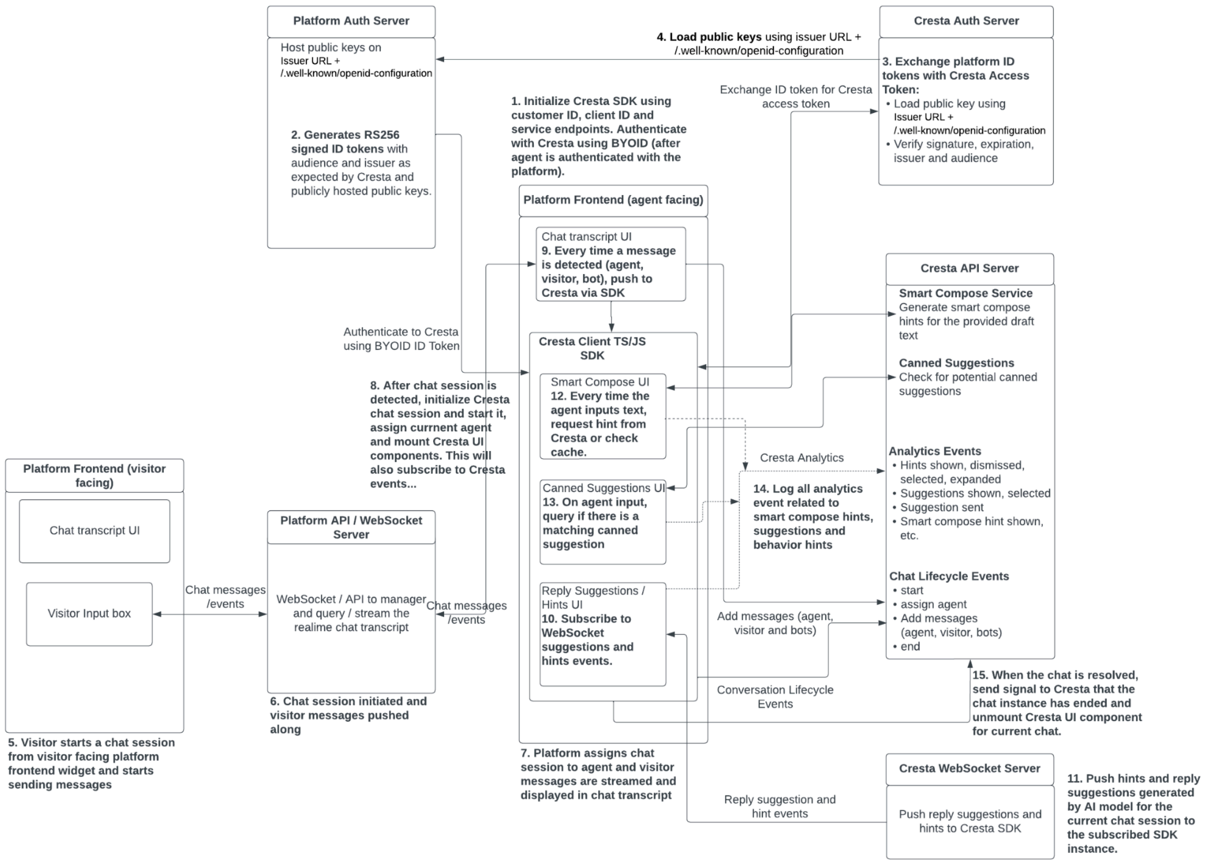
Next

Customer doesn't have it

Note that a “Best Match” tab is also available in the results section. This can contain a combination of best matches (e.g. KB and GWF results together)

Cresta Client SDK Diagram

This flow diagram illustrates SDK usage with BYOID authentication for a chat integration.


Platform Auth Server
Host public keys on Issuer URL + /.well-known/openid-configuration
2. Generates RS256 signed ID tokens with audience and issuer as expected by Cresta and publicly hosted public keys.
4. Load public keys using issuer URL + /.well-known/openid-configuration
Cresta Auth Server
3. Exchange platform ID tokens with Cresta Access Token:
• Load public key using Issuer URL + /.well-known/openid-configuration
• Verify signature, expiration, issuer and audience
Exchange ID token for Cresta access token
1. Initialize Cresta SDK using customer ID, client ID and service endpoints. Authenticate with Cresta using BYOID (after agent is authenticated with the platform).
Platform Frontend (agent facing)
Chat transcript UI
9. Every time a message is detected (agent, visitor, bot), push to Cresta via SDK
Authenticate to Cresta using BYOID ID Token
Cresta Client TS/JS SDK
8. After chat session is detected, initialize Cresta chat session and start it, assign current agent and mount Cresta UI components. This will also subscribe to Cresta events...
Smart Compose UI
12. Every time the agent inputs text, request hint from Cresta or check cache.
Canned Suggestions UI
13. On agent input, query if there is a matching canned suggestion
Reply Suggestions / Hints UI
10. Subscribe to WebSocket suggestions and hints events.
Cresta API Server
Smart Compose Service
Generate smart compose hints for the provided draft text
Canned Suggestions
Check for potential canned suggestions
Cresta Analytics
14. Log all analytics event related to smart compose hints, suggestions and behavior hints
Analytics Events
• Hints shown, dismissed, selected, expanded
• Suggestions shown, selected
• Suggestion sent
• Smart compose hint shown, etc.
Chat Lifecycle Events
• start
• assign agent
• Add messages (agent, visitor, bots)
• end
Add messages (agent, visitor and bots)
Conversation Lifecycle Events
15. When the chat is resolved, send signal to Cresta that the chat instance has ended and unmount Cresta UI component for current chat.
Platform Frontend (visitor facing)
Chat transcript UI
Visitor Input box
Chat messages /events
Platform API / WebSocket Server
WebSocket / API to manager and query / stream the realime chat transcript
Chat messages /events
5. Visitor starts a chat session from visitor facing platform frontend widget and starts sending messages
6. Chat session initiated and visitor messages pushed along
7. Platform assigns chat session to agent and visitor messages are streamed and displayed in chat transcript
Cresta WebSocket Server
Push reply suggestions and hints to Cresta SDK
Reply suggestion and hint events
11. Push hints and reply suggestions generated by AI model for the current chat session to the subscribed SDK instance.

Core (Vanilla JS) SDK (@cresta/client)

The **@cresta/client** package provides a Vanilla JS framework agnostic version of the SDK. It offers all the functionalities and UI components needed to integrate with Cresta chat and voice services.

For React developers, refer to the **@cresta/react-client** package.

The Cresta client SDK facilitates integrations with the following Cresta chat components:

- Cresta smart compose (standalone)
- Cresta behavioral suggestions / hints
- Cresta reply and canned suggestions
- Cresta Knowledge Base search
- Cresta Guided Workflow
- Cresta Generative Knowledge Assist (beta)

The SDK also supports the chat smart compose overlay element. Unlike the standalone counterpart (chat smart compose element), this element is used in conjunction with the platform chat input box. It is useful when the chat input box supports additional capabilities not supported in the standalone smart compose element. This includes rich text support, ability to upload files or images, etc. From the agent's perspective, the overlay will not be visible and the platform input box will continue to operate as expected, but with additional features (e.g. display and acceptance of smart compose hints).

The Cresta client SDK also facilitates integrations with the following Cresta voice components:

- Cresta behavioral suggestions / hints
- Cresta Knowledge Base search
- Cresta Guided Workflow
- Cresta Generative Knowledge Assist (beta)

Note: You can view the `@cresta/client` documentation and API references via browser by installing the package and then running a local web server and pointing it to serve the static HTML files at `./node_modules/@cresta/client/docs/html`. A quick way to do that is to install `http-server` module and use that to serve the HTML site:

```
npm install -g http-server
http-server ./node_modules/@cresta/client/docs/html
```

You can then navigate to the launched website.

Detailed API

Refer to: index.d.ts typings file in **@cresta/client** package.

Prerequisites for @cresta/client

The following requirements are needed:

- Installation of Node.js version `14.17.6` or greater.
- Installation of npm version `6.14.15` or greater.
- Already set up the project namespace and AI model with the Cresta team.
- Currently, seamless token exchange support is not supported by Cresta. You will need to either configure your SAML IdP with Cresta or create usernames/password in the Cresta Director UI for your real or test users.
- The Cresta client SDK configuration file. This should look like:

```
{
  "customerId": "CUSTOMER_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com"
}
```

  This allows initialization of Cresta clients:

```
const client = cresta.createClient({
  customerId: 'CUSTOMER_ID',
  // This is used for the authentication URL.
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
  /**
   * The optional OAuth client ID, required when using external credentials
   * for authentication (BYOID). This should match the external ID token
   * audience field. This should be registered with Cresta.
   */
  clientId: "ID_TOKEN_AUDIENCE_FIELD",
  // You can optionally specify how auth state is persisted in storage.
  // authStorageType: 'indexeddb',
});
```

- When using a proxy with external credentials for authentication (BYOID), the **authEndpoint** also has to be specified.

```
const client = cresta.createClient({
  customerId: "CUSTOMER_ID",
  // This is used for the authentication popup URL.
  customerOrigin: "https://CUSTOMER_ID.cresta.com",
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: "https://proxy.example.com",
  // The authEndpoint doesn't need to be specified unless a proxy endpoint is
  // used. The default endpoint is https://auth.cresta.com
  authEndpoint: "https://proxy.example.com",
  /**
   * The optional OAuth client ID, required when using external credentials
   * for authentication (BYOID). This should match the external ID token
   * audience field.
   */
  clientId: "ID_TOKEN_AUDIENCE_FIELD",
});
```

- Optionally, when using the SDK with the Cresta SDK emulator, the following configuration field should be provided:

```
{
  "customerId": "CUSTOMER_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "http://localhost:8081",
  "emulatorMode": true,
  "emulatorUser": {
    "userId": 1234567890,
    "email": "jane.doe@cresta.ai",
    "displayName": "Jane Doe",
    "userRoles": ["AGENT"],
    "username": "jane.doe"
  }
}
```

Where `serviceEndpoint` should point to the emulator URL and `emulatorMode` should be set to `true` to automate sign-in when `signInWithPopup` is triggered. An optional `emulatorUser` can be provided to customize the profile used by the mocked credentials. Otherwise a fixed static profile is used.

- A Cresta profile ID is needed to initialize a chat manager before a chat session can be initialized, or a voice manager before an incoming voice session can be detected.

```
// Initialize a chat instance.
const chat = client.chatManager('PROFILE_ID').chat('CHAT_ID_1');
```

```
// Initialize a voice instance.
const voiceManager = client.voiceManager("PROFILE_ID");

// Listen to incoming or ongoing voice sessions.
voiceManager.onVoiceSession((session) => {
  console.log(
      `Voice session change detected, ${session.id}, ${session.status}`
  );
});
```

- Adding the authorized origin for your application to the list of authorized origins (contact the Cresta team and provide them with these origins). For security reasons, this is needed for popup sign-in to succeed and return tokens to your app. This is not needed when using the emulator with `emulatorMode` set to `true`. This step is not needed if using external credentials for login (BYOID).

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save @cresta/client@latest
```

In order to install the client SDK with your package, the **rxjs** peer dependency needs to be satisfied in the **package.json**:

```
 "dependencies": {
   "@cresta/client": "latest",
   "rxjs": "^7.3.1",
   ...
 },
```

Usage

Initialize the Cresta client

The Cresta client can be initialized using the Cresta configuration:

```
import {cresta} from '@cresta/client';

// On app initialization.
const client = cresta.createClient({
  customerId: 'CUSTOMER_ID',
  // This is used for the authentication URL.
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
  /**
   * The optional OAuth client ID, required when using external credentials
   * for authentication (BYOID). This should match the external ID token
   * audience field. This should be registered with Cresta.
   */
  clientId: "ID_TOKEN_AUDIENCE_FIELD",
});
```

When using a proxy or API gateway, substitute the proxy endpoint in the `serviceEndpoint` and the optional `authEndpoint` fields.

You can also initialize multiple `CrestaClient` instances by specifying a unique `name` when calling `createClient()`.

```
import { cresta } from "@cresta/client";

// On app initialization.
const client1 = cresta.createClient(
  {
    customerId: "CUSTOMER_ID1",
    customerOrigin: "https://CUSTOMER_ID1.cresta.com",
    serviceEndpoint: "https://api-CUSTOMER_ID1.cresta.com",
  },
  "customer1"
);

const client2 = cresta.createClient(
  {
    customerId: "CUSTOMER_ID2",
    customerOrigin: "https://CUSTOMER_ID2.cresta.com",
    serviceEndpoint: "https://api-CUSTOMER_ID2.cresta.com",
  },
  "customer2"
);
```

When specifying multiple instances, each instance will require separate authentication.

Common reasons for using multiple instances:

- Multiple Cresta customers are using the same application.
- An agent may have multiple Cresta agent profiles for the same customer. In that case, the agent may be able to switch between accounts without signing out. The same customer ID is used for both instances but authentication for each instance will yield a different agent account.

You can specify how the auth state is persisted in storage by passing the `authStorageType` type in the client configuration.

```
import {cresta} from '@cresta/client';

// On app initialization.
const client = cresta.createClient({
  customerId: 'CUSTOMER_ID',
  // This is used for the authentication URL.
  customerOrigin: 'https://login.cresta.com',
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
  // Use indexedDB to store auth state.
  authStorageType: 'indexeddb',
});
```

This can have the following values:

- `local`: Web `localStorage`. This is used by default for non-Chrome extension environments.
- `session`: Web `sessionStorage`.
- `chrome-extension-local`: Chrome extension `chrome.storage.local`. This is used by default for Chrome extension environments.
- `none`: No persistence is used. In memory storage only.
- `indexeddb`: Web `indexedDB`.

Authentication & BYOID (Bring Your Own IDentity)

In order to integrate with Cresta services, an agent has to be authenticated with Cresta. This enables API calls to the Cresta backend to be authenticated (via Bearer access tokens).

Cresta SDK supports authentication via popup. Popup flow supports customer SAML IdPs but can also support Cresta test username/password accounts.
Test accounts can be created via the Cresta Director **User Management** UI.

```
async function loginWithCresta() {
  try {
    await client.auth.signInWithPopup();
  } catch (e) {
    console.log(e.code);
  }
}

document.getElementById('sign-in-with-cresta-button-id').addEventListener('click', (e) => {
  loginWithCresta();

  e.preventDefault();
});
```

When using the emulator (with `emulatorMode` set to `true`), authentication is done automatically when `signInWithPopup` is called. Mock credentials are automatically provisioned.

Note that you can also directly sign in with Cresta username and password directly via the SDK without going through a popup. This is not recommended. Only use this if necessary (e.g. if an HTTP interceptor is to be used and you cannot use the BYOID solution). It is preferred that `signInWithPopup()` is used instead.

```
// Only use this if necessary (if an HTTP request interceptor is to be used).
// It is prefered that `signInWithPopup()` is used instead.
async function loginWithCresta(username: string, password: string) {
  try {
    await client.auth.signInWithUsernameAndPassword(username, password);
  } catch (e) {
    console.log(e.code);
  }
}
```

However, as agents have to also be separately authenticated to the platform, prompting the agent to sign in again to Cresta may not be convenient and may add unnecessary friction. As a result, Cresta SDK offers a seamless mechanism for identity federation based on the OAuth 2.0 Token Exchange rfc8693 standard, where the Cresta SDK will exchange a platform OIDC ID token with a Cresta access token and take care of authorizing API calls with that token.

This capability is recommended over the popup flow due to its superior user experience. This flow also offers security benefits by keeping the Cresta session in-sync (no Cresta refresh token is issued) with the platform session as Cresta will issue an access token with similar expiration as the provided ID token (within reasonable range) and will request a new ID token on expiration. In addition, the user role (agent, manager, admin, etc) can be kept up to date as this is copied and updated from the original ID token claims on each refresh (this is currently not supported and the "agent" role is always used).

External Token Requirements

As of now, only OIDC ID tokens are supported. Customers need to mint these tokens on demand after platform sign-in and provide them to the Cresta client SDK. The client SDK will exchange these external tokens for Cresta access tokens and cache these tokens until their natural lifetime expires and will minimize the calls to generate new ID tokens unless the cached access token is expired or invalidated.

Payload

The following standard fields will be required:

- iss: The URL of the issuer of the token. This should be registered with Cresta. It will also be used to retrieve the platform public keys to verify the token signature.
- sub: The subject identifier. This is the locally unique identifier within the issuer of the end-user.
- aud: The audience that the token is intended to. This should match the Cresta client ID.
- exp: The expiration time represented in the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.
- iat: The issue time of the token represented in the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

Optional fields:

- nbf: current date/time MUST be after or equal to the not-before date/time listed in the "nbf" claim.
- email_verified: Whether the user's email address is verified or not.
- nonce: String value used to associate a client session with an ID token, and to mitigate replay attacks.
- picture: the URL of the user's profile picture.
- given_name: user given name or first name.
- family_name: user last name or surname.

For Cresta's purposes, the following token fields should be provided:

- sub: The subject identifier. This is the locally unique identifier within the issuer of the end-user.
- email: The email address representing the end user.
- name: the user's full name.

Sample payload:

```
{
  "sub": "10193942248242",
  "iss": "https://openid.c2id.com",
  "aud": "cresta-client-12345",
  "iat": 1647884459,
  "exp": 1647888059,
  "nbf": 1647884459,
  "name": "John Smith",
  "email": "john.smith@example.com",
  "email_verified": true
}
```

The **iss** and **aud** fields will need to be registered with Cresta.

Header/ Signing algorithms

The initially supported JWT signing algorithms will be limited to the asymmetric RS256. This makes retrieval of the public key, as well as public key rotations a lot easier. The key ID in the token header will be used to determine the public key used to verify the token signature.

This is a sample header of a token using RS256. Notice the **kid** field used to identify the corresponding key which was used to generate the signature.

```
{
  "alg": "RS256",
  "kid": "b06a119158e8b282171418a67dea83740b5ee77e",
  "typ": "JWT"
}
```

Sample ID token with RS256

```
eyJhbGciOiJSUzI1NiIsImtpZCI6InlKdHBSOGtzRjNRVk84M3FaMVBkajNUY0M0SndlWCIsInR5cCI6IkpX
VCJ9.eyJzdWIiOiI5ODc2NTQzMjEwIiwiaXNzIjoiaHR0cHM6Ly9kMGUxLTI2MDEtNjQ2LTlkMDEtYTVhMC1
mYzcyLTQwODktODBlNS00YTQ4Lm5ncm9rLmlvL2NyZXN0YSIsImF1ZCI6InByb2ZpbGUtY2hhdC1kZW1vIiw
iaWF0IjoxNjY0OTU0Mjc5LCJuYmYiOjE2NjQ5NTQyNzksImV4cCI6MTY2NDk1NDQ1OSwibmFtZSI6IkphbmU
gRG9lIiwiZW1haWwiOiJqYW5lLmRvZUBjcmVzdGEuY29tIiwiZW1haWxfdmVyaWZpZWQiOnRydWUsIm5vbmN
lIjoiamF4c3Y1cDdwYW4ifQ.XWGZRHZ5BxOZSQGZmRtykXE5RPCBG5OYPWoCzD3rXezuXAEB33lP_PA_mWs-
9Lgx_JNKr9vPrCRHk0ITw9IjOBWBPyRJPXVTMgLt4UUjtqTOhhpWfogwp92W7yreMpjMhkjK-YdijSebF-F7
lA_gQOumPGjzJ034VYZU42f27vqx70hw2VuEgcajThN1JVscCly-f1DLJ3qZPwIAxxtnXyAMKLY5gnsf6ClG
-emU9Bikr_AbHJ3nAtq0zH008xOQsefq2pPmSbJKNC73SFheaC0BmedtA6IxeSKR_QLJm4qDEaro_p-wIkYA
tlWrr1j8DbBw9gFrt8SmTqYMDMumvA
```

Cresta auth service will obtain the public key via the OIDC metadata endpoint constructed using: Issuer URL + `/.well-known/openid-configuration`

Example:

https://d0e1-2601-646-9d01-a5a0-fc72-4089-80e5-4a48.ngrok.io/cresta/.well-known/openid-configuration

The public keys will be cached on the Cresta server after they are fetched to minimize the need to re-fetch them each time.

```
{
  "issuer": "https://d0e1-2601-646-9d01-a5a0-fc72-4089-80e5-4a48.ngrok.io/cresta",
  "jwks_uri":
"https://d0e1-2601-646-9d01-a5a0-fc72-4089-80e5-4a48.ngrok.io/cresta/.well-known/jwks.json",
  "response_types_supported": ["id_token"],
  "subject_types_supported": ["public"],
  "id_token_signing_alg_values_supported": ["RS256"]
}
```

Cresta would then need to follow the **jwks_uri** to retrieve the public keys.
Following the **jwks_uri**:

```
{
  "keys": [
    {
      "kty": "RSA",
      "n":
"zX3PbsP0haVPZIQ8rz5M44Ig3xj8hW1lfWqXE-YgnJ1uZLg6z2sBzxjAWsXLKYe3bHFtOjNGesdFgC4eO4bzbwqdZSN
lV6l1Bt14XBi_hgrUfzJFEFtGS9hD3IYLoxgw9pUnQNdqHHFEUQAXXf4e8XyFODr5qRVXLRID27plE2bJ7ihoY3gCKMq
pNfx_TIEsWuWbGA8DIwGLVqMK-_6k0spQQGn51ZlsXX-qPm54XGzntRQLVMBiH6b0R80kZp6AWHlJaYQd2Sg6ebCD4Ym
nm9ZdFvS0WxUO3xnmwZCTetkLgGEdbc8qNauS09S1Kpo4ZBEKyYBGgDvRpMQ_fkYpiw",
      "e": "AQAB",
      "alg": "RS256",
      "use": "sig",
      "kid": "yJtpR8ksF3QVO83qZ1Pdj3TcC4JweX"
    }
  ]
}
```

The key ID in the token header can be used to find the corresponding key in the discovery doc.

User ID requirements

Cresta will rely on the ID token `Subject` field (`sub`) to uniquely identify the user. The user ID should be unique per customer. If you are using SAML IdP to also sign in users (either to the application integrating with Cresta via the Cresta SDK or via Cresta Director), the SAML response `NameID` field should match the ID token `sub` field. This ensures that the same user is returned for both sign-ins. Otherwise, if the 2 fields do not match, 2 user accounts will be created (one for the SAML sign-in and another for the OIDC token exchange sign-in).

Prerequisites

Since Cresta only requires the RS256 signing algorithm, customers only need to register the ID token issuer URL and the ID token audience field (**iss** and **aud** fields in the token) with Cresta so Cresta can securely verify the provided tokens before minting a Cresta access token.

The audience field would then need to be provided during initialization of the Cresta client instance:

```
import { cresta } from "@cresta/client";

// On app initialization.
const client = cresta.createClient({
  customerId: "CUSTOMER_ID",
  // This is used for the popup authentication URL.
  customerOrigin: "https://CUSTOMER_ID.cresta.com",
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: "https://api-CUSTOMER_ID.cresta.com",
  /**
   * The OAuth client ID, required when using external credentials
   * for authentication. This should match the external ID token audience
   * field.
   */
  clientId: "ID_TOKEN_AUDIENCE_FIELD",
});
```

Client SDK Usage

There are 2 mechanisms to sign in using BYOID in the client SDK:

- You can either provide an implementation of the `ExternalCredentials` interface to provide the SDK with external ID tokens on demand.

```
/**
 * The platform token type. As of now, only id_token is supported.
 * https://datatracker.ietf.org/doc/html/rfc8693#section-3
 */
type TokenType = "urn:ietf:params:oauth:token-type:id_token";

/** Defines the platform token object returned by the platform developer. */
interface AuthToken {
  /** The platform token itself, in string format. */
  token: string;
  /** The platform token type. */
  tokenType: TokenType;
  /** The token expiration time in milliseconds since the epoch time. */
  expirationTimeMs: number;
}

/**
 * A credentials class used to provide a platform token on demand to Cresta.
 * Cresta Auth service will exchange it for a Cresta access token.
 */
interface ExternalCredentials {
  /**
   * The external credentials session identifier. This helps match different
   * instances of external credentials synchronously without having to request
   * an external token (e.g. ID token) and exchange it for a Cresta access
   * tokens to determine the corresponding user ID. The external user ID can
   * also be used for this field.
   */
  readonly sessionId: string;

  /**
   * Returns a platform token when called. Developers need to implement this
   * method.
   *
   * @param forceRefresh - Developers can return unexpired cached tokens
   *     unless this field is set to true. By default this should be false.
   * @returns A promise that resolves with the platform token.
   */
  getToken(forceRefresh: boolean): Promise<AuthToken>;
}
```

- Or you can provide SharedWorker options `SharedWorkerExternalCredentialsOptions` used to instruct the SDK how to postMessage a SharedWorker for these tokens when it needs one.

```
interface SharedWorkerExternalCredentialsOptions {
  /**
   * The SharedWorker absolute URL. This will be messaged by the SDK for
   * platform tokens.
   */
  url: string;
  /** The optional worker options. */
  options?: WorkerOptions | string;
  /**
   * The external credentials session identifier. This helps match different
   * instances of external credentials synchronously without having to request
   * an external token (e.g. ID token) and exchange it for a Cresta access
   * tokens to determine the corresponding user ID. The external platform user
   * ID can also be used for this field.
   */
  sessionId: string;
}
```

If `ExternalCredentials` are provided, these will only be persisted in the current tab, and each tab will have to call `signInWithExternalCredentials`, as well as on the current page reload. If a SharedWorker is used, the credentials will be persisted and shared across tabs and `signInWithExternalCredentials` only needs to be called once, typically after platform sign-in.

The external credentials or SharedWorker options can be passed to the Cresta client SDK.

```
async function signInWithCresta(
  client: CrestaClient,
  credentialsOrOptions:
    | ExternalCredentials
    | SharedWorkerExternalCredentialsOptions
) {
  await client.auth.signInWithExternalCredentials(credentialsOrOptions);
}
```

By using a ShareWorker, it makes it easier to synchronize the external credentials state across tabs/windows and for multiple tabs integrating with Cresta to get and share tokens on demand from a central location. However for this approach to work, you will need to host a SharedWorker and listen to **cresta#getToken** message events and respond to them with fresh OIDC tokens.

The **postMessage** request from the Cresta SDK to the SharedWorker will have the form:

```
interface GetTokenRequest {
  type: "cresta#getToken";
  // Whether to force refresh of the token regardless if an unexpired
  // token is cached.
  forceRefresh: boolean;
  // The Cresta request audience. This has the form:
  // customers/CUSTOMER_ID/clients/CLIENT_ID.
  audience: string;
  // The request event ID. This should be returned in the response.
  eventId: string;
}
```

The expected **postMessage** response from the SharedWorker on success will have the form:

```
interface GetTokenResponseSuccess {
  type: "cresta#getToken";
  // The Cresta request audience. This has the form:
  // customers/CUSTOMER_ID/clients/CLIENT_ID.
  audience: string;
  // This is the same event ID sent along the request. It should be
  // returned in the response.
  eventId: string;
  status: "success";
  // The requested tokens.
  token: AuthToken;
}
```

The expected **postMessage** response from the SharedWorker on error will have the form:

```
interface GetTokenResponseFailure {
  type: "cresta#getToken";
  audience: string;
  // This is the same event ID sent along the request. It should be
  // returned in the response.
  eventId: string;
  status: "error";
  // A message providing details about the error.
  error: string;
}
```

Sample **SharedWorker** implementation:

```
let pendingTokenRequestPromise: Promise<AuthToken> | null = null;

(self as unknown as SharedWorkerGlobalScope).addEventListener(
  "connect",
  (e) => {
    const port = e.ports[0];

    port?.addEventListener("message", async (event) => {
      // This event is required to be handled in the SharedWorker. This
      // message will be sent by the Cresta SDK to get a valid unexpired
      // OIDC ID token.
      if (event.data.type === "cresta#getToken") {
        const { forceRefresh, eventId, audience } = event.data;

        try {
          // Cache pending request for tokens to avoid multiple token requests
          // running at the same time in parallel.
          if (!pendingTokenRequestPromise) {
            pendingTokenRequestPromise = getTokens(forceRefresh);
          }

          const token = await pendingTokenRequestPromise;

          pendingTokenRequestPromise = null;
          // Return the updated tokens to the caller. Pass the audience and
          // eventId back too.
          port.postMessage({
            audience,
            type: "cresta#getToken",
            status: "success",
            token,
            eventId,
          });
        } catch (error) {
          pendingTokenRequestPromise = null;
          // Notify caller of the token error. Pass the audience and eventId
          // back too.
          port.postMessage({
            audience,
            eventId,
            type: "cresta#getToken",
            status: "error",
            error: error.message,
          });
        }
      }
    });

    // Start sending message queued on the port.
    port?.start();
  }
);
```

You would then need to pass the **SharedWorker** info to the Cresta SDK after the user signs in with the OIDC provider. This only needs to be done once and the Cresta SDK will cache this information and persist it across tabs and page reloads:

```
async function signInWithCresta(client: CrestaClient, creds: OidcCredentials) {
  await client.auth.signInWithExternalCredentials({
    url: sharedWorkerUrl,
    sessionId: oidcCredentials.sessionId,
    options: { type: "module" },
  });
}
```

When signing out for either approach:

```
async function signOut(client: CrestaClient) {
  // Make sure to revoke the original OIDC credentials if revocable.
  // await creds.revoke();
  // Sign out from Cresta. This will also ensure Cresta stops trying to get
  // new tokens from the provided ExternalCredentials.
  await client.auth.signOut();
}
```

A sample OIDC auth server is provided to facilitate generation and testing BYOID with Cresta. Refer to the @cresta/client-sdk-auth-sample documentation below.

Integration in Twilio Flex

When integrating with Cresta in the Twilio Flex platform, you can sign in to Cresta directly without any user interaction, e.g. without prompting the user to sign in with popup, SAML IdP or Cresta username/password.
Customers also do not need to implement Bring Your Own ID (BYOID) using OIDC ID tokens. In fact, BYOID sign-in can readily be achieved by passing the Twilio `FlexManager` instance as Cresta supports directly exchanging Twilio Flex tokens for Cresta Access tokens. The Cresta SDK package will be able to retrieve Twilio Flex tokens from the `FlexManager` instance on demand whenever it needs to generate a Cresta access token to call Cresta APIs.

Before you can use this feature, you will need to provide the following to the Cresta team:

- Twilio Flex `AccountSID`: The Twilio Account SID
- Twilio Flex `AuthToken`: The Twilio Flex `authToken` secret.
- Cresta client ID: The Cresta client ID which will be used as the audience field when calling the Cresta STS endpoint. This is a unique identifier that should be configured with Cresta so it is able to lookup the IdP configuration containing the `AccountSID` and `AuthToken` to securely verify Twilio Flex tokens.

The following example illustrates how to initialize the Cresta SDK and authenticate in a Twilio Flex environment:

```
import { cresta } from "@cresta/client";
//...
export default class CrestaFlexPlugin extends FlexPlugin {
  constructor() {
    super(PLUGIN_NAME);
  }
  /**
   * This code is run when your plugin is being started
   * Use this to modify any UI components or attach to the actions framework
   *
   * @param flex { typeof Flex }
   * @param manager { Flex.Manager }
   */
  async init(flex: typeof Flex, manager: Flex.Manager): Promise<void> {
    // Initialize Cresta client instance..
    const client = cresta.createClient({
      customerId: "CUSTOMER_ID",
      customerOrigin: "https://CUSTOMER_ID.cresta.com",
      serviceEndpoint: "https://api-CUSTOMER_ID.cresta.com",
      // Needed for the token exchange feature.
      clientId: "AUDIENCE_FIELD",
    });
    // Sign in with Twilio Flex. Only the manager instance needs to be provided.
    await client.auth.signInWithExternalCredentials({
      tokenType: "urn:ietf:params:oauth:token-type:twilio_token",
      flexManager: manager,
    });
    // ...
  }
  // ...
}
```

`signInWithExternalCredentials()` should always be called on page initialization as these type of credentials are not persisted since the `FlexManager` instance cannot be serialized.

Existing Twilio Flex customers migrating to this solution should ensure that the agent's email is propagated to Twilio Flex. This is needed so the agent's Cresta account and record (e.g. Cresta Director record) is persisted post-migration. Otherwise, a new account for the same agent may be created on Cresta's side resulting into 2 accounts for the same agent.

Cresta integration with Twilio Flex has been tested with `@twilio/flex-ui` version `1.32.0`. It should be compatible with `@twilio/flex-ui` semver version `^1`.

Authentication with Cresta username and password

Important: For customers using sign-in with Cresta username and password, we highly recommend using the sign-in with Cresta popup flow instead of inlining sign-in.

```
async function loginWithCresta() {
  try {
    await client.auth.signInWithPopup();
  } catch (e) {
    console.log(e.code);
  }
}
```

However, if there is a need to inline the sign-in flow due to some special restrictions or limitations (e.g. the need to proxy the Cresta API calls from behind a firewall or needing to modify the requests when proxying these calls, etc.), the SDK provides a mechanism to sign in with username and password directly from your application without going through the Cresta login popup.

When implementing your own UI to sign in with Cresta username and password, a reCAPTCHA token is required for security reasons to be provided along the request. Cresta password authentication integrates with reCAPTCHA Enterprise to prevent automated brute force attacks and apply domain verification to mitigate against phishing attacks or password replay attacks, etc.

Cresta SDK abstracts this integration for you. You don't even need to include the reCAPTCHA library separately. The Cresta SDK will do that for you and wrapper utilities/helpers are provided to handle the rendering of the reCAPTCHA and the generation of the reCAPTCHA token and how it is passed along the authentication request to Cresta.

Before you begin implementation, reach out to the Cresta team for the customer-specific reCAPTCHA site keys. You will need 2 keys:

- Site key for the background score-based reCAPTCHA.
- Site key for the interactive "I'm not a robot" checkbox reCAPTCHA.

You will also need to provide Cresta a list of domains where the login UI will be hosted on. You also have the option to allowlist usernames for testing purposes. This will exempt these users from any reCAPTCHA requirement. This is useful if you are using a dedicated account for automated end-to-end testing.

When implementing the flow, always use a score-based reCAPTCHA first. This is completely frictionless and invisible to the end user. Cresta will create a reCAPTCHA assessment based on that token. If it yields a high score (low risk), then the sign-in request is processed. Otherwise if the request yields a low score (high risk), then an error is returned. When a high risk sign-in attempt is detected, the user can recover by solving a checkbox reCAPTCHA challenge. This should affect only a small percentage of legitimate authentication attempts.

You will first need to render the score-based reCAPTCHA using the score-based reCAPTCHA site key:

```
import { RecaptchaVerifier, ScoreBasedRecaptchaVerifier } from "@cresta/client";
let recaptchaVerifier: RecaptchaVerifier = new ScoreBasedRecaptchaVerifier(
  // Site key for the background score-based reCAPTCHA.
  RECAPTCHA_SITE_KEY
);
recaptchaVerifier.render();
```

On initial sign-in, pass the `RecaptchaVerifier` in the API call:

```
try {
  await client.auth.signInWithUsernameAndPassword(
    username,
    password,
    recaptchaVerifier
  );
} catch (e) {
  // ...
}
```

For most users this should be sufficient to pass the score-based token and complete sign-in. However, for some users, additional verification via checkbox reCAPTCHA is needed. You can catch this error during sign-in and render a checkbox reCAPTCHA using the checkbox reCAPTCHA site key:

```
import { InteractiveRecaptchaVerifier } from "@cresta/client";
// ...
try {
  // ...
} catch (e) {
  if (e.message.includes("recaptcha high risk")) {
    recaptchaVerifier = new InteractiveRecaptchaVerifier(
      // Site key for the interactive "I'm not a robot" checkbox reCAPTCHA.
      RECAPTCHA_CHECKBOX_SITE_KEY,
      // Container of the checkbox reCAPTCHA. This can be the element ID or
      // the HTMLElement where the checkbox reCAPTCHA will be rendered.
      recaptchaContainerIdOrElement
    );
    recaptchaVerifier.render();
  }
}
```

You will have to pass the `RecaptchaVerifier` in the API call again. However, you will need to verify the reCAPTCHA challenge is resolved before doing so:

```
if (!recaptchaVerifier.resolved) {
  // This happens when the reCAPTCHA checkbox is not checked yet.
  showError('Please click the "I\'m not a robot checkbox"');
  return;
}
try {
  await client.auth.signInWithUsernameAndPassword(
    username,
    password,
    recaptchaVerifier
  );
} catch (e) {
  // ...
}
```

Full snippet:

```
import {
  InteractiveRecaptchaVerifier,
  RecaptchaVerifier,
  ScoreBasedRecaptchaVerifier,
} from "@cresta/client";
// ...
let recaptchaVerifier: RecaptchaVerifier = new ScoreBasedRecaptchaVerifier(
  // Site key for the background score-based reCAPTCHA.
  RECAPTCHA_SITE_KEY
);
recaptchaVerifier.render();

async function onSignInButtonClick(username: string, password: string) {
  if (!recaptchaVerifier.resolved) {
    // This happens when the reCAPTCHA checkbox is not checked yet.
    showError('Please click the "I\'m not a robot checkbox"');
    return;
  }
  try {
    await client.auth.signInWithUsernameAndPassword(
      username,
      password,
      recaptchaVerifier
    );
  } catch (e) {
    // Possible high-risk attempt, switch to using a checkbox reCAPTCHA.
    if (e.message.includes("recaptcha high risk")) {
      recaptchaVerifier = new InteractiveRecaptchaVerifier(
        // Site key for the interactive "I'm not a robot" checkbox reCAPTCHA.
        RECAPTCHA_CHECKBOX_SITE_KEY,
        recaptchaContainerIdOrElement
      );
      recaptchaVerifier.render();
    } else {
      showError(e.message);
    }
  }
}
```

You can also always use the interactive checkbox reCAPTCHA for all sign-ins but we don't recommend that as the score-based reCAPTCHA is frictionless and for the majority of users, it is sufficient for verifying the legitimacy of the request. The checkbox reCAPTCHA is used only as a fallback when the legitimacy of the operation needs to be verified.

Subscribe to error manager

Note: this feature is still experimental.

You can subscribe to background auth related errors (e.g. when the user refresh token is revoked or expired), API errors, network related errors and UI component errors by providing an `onError` callback in the `ErrorManager`.

This provides a central location to subscribe to Cresta operations in the background that emit errors. Currently, errors that directly emit errors (e.g. `signInWithPopup`) in the foreground will not emit errors via the error manager. These will need to be caught on the spot (via a `catch` block).

This capability is very useful to inform the user when there are network connection issues or prompt the user to sign in when tokens expire or are revoked.

Known issues include overzealous emission of WebSocket offline and network errors from the `ClientSubscription` service. This service is responsible for pulling chat suggestions and hints from the Cresta server. The service may occasionally close after a period of inactivity and then a new connection is reopened. The WebSocket will recover on its own. Related network errors should either be double checked via `navigator.onLine`, window `offline` event or simply ignored.

If no error handlers are subscribed to the error manager, all generated errors will be logged to the browser console as unhandled errors.

```
const unsubscribe = client.errorManager.onError((error) => {
  console.warn(`${error.code}: ${error.message}`);
});

// To unsubscribe:
// unsubscribe();
```

Chat Integrations

Cresta agent assist chat integrations typically require that the conversation events and messages are sent to Cresta via the client SDK. So when a new conversation or incoming messages are detected (commonly via a Websocket connection), the events and messages are sent to Cresta on the frontend via the Cresta client SDK. This is unlike Cresta voice agent assist integrations, where the conversation audio stream and related events are streamed separately via server to server API calls.

So in addition to providing the chat messages and events to the Cresta server, the client side integration is also responsible for displaying and rendering Cresta conversation AI features (e.g. Cresta smart compose auto-complete, hints, suggestions, KB-search, guided workflows, generative knowledge assist, etc.) for the agent to interact with.

Initialize and manage a chat session

A chat session can be initialized using the chat profile ID via the chat manager:

```
// Initialize the chat manager.
const chatManager = client.chatManager(profileId);

// Anytime a new chat session needs to be referenced:
const chat1 = chatManager.chat(chatId1);
const chat2 = chatManager.chat(chatId2);
```

The profile ID will be provided by the Cresta team.

Before a chat session can connect to Cresta AI services, it has to be started.
To start a chat session, pass the required visitor information and the optional historical chat messages.

Note: The visitor display name should be passed when a conversation is started, if the name is determinable. If the name is provided on chat start, it will be automatically used in Cresta suggestions. You should assume this will be used verbatim. However, Cresta will use some heuristics to filter out strings that are clearly not names, but this will be on a best-effort basis.
If you do not provide this field for a chat, Cresta will serve suggestions that do not contain visitor names.

```
await chat1.start({
  config: {
    // Set to true if the current agent is assigned to the conversation.
    // You can still change or assign later by calling
    // chat1.assignToCurrentAgent(...)
    //
    // assignToCurrentAgent: true,
    //
    // The visitor information.
    visitor: {
      platformSpeakerId: 'VISITOR_ID',
      displayName: 'John Smith',
    },
  },
  messages: [/* Chat session historical messages */],
});
```

The chat session then needs to be assigned to the current agent. This is needed to establish ownership of the chat sessions. Some contact center platforms allow other agents to monitor and participate in the same session. This makes it clear who the assigned agent is.

```
await chat1.assignToCurrentAgent({
  // Agent display name.
  displayName: 'Helpful Agent',
});
```

You can skip the above step if you set `assignToCurrentAgent` on `start()` call and the assigned agent hasn't changed since then.

It is recommended to explicitly set chat visibility when switching between chat sessions for performance reasons. By setting a chat as invisible, any underlying polling will be paused and resumed once the chat is made visible again.

The default visibility when a chat session is created is `visible`.

```
// When the agent switches from chat1 to chat2.
chat1.visibility = 'hidden';
chat2.visibility = 'visible';

// When the agent switches back to chat1.
chat1.visibility = 'visible';
chat2.visibility = 'hidden';
```

This is not necessary when server-sent events (SSE) are used instead of polling for long lived connections to the Cresta server.

To update the language code used for message tests in the conversation:

```
await chat1.updateLanguageCode('es-ES');
```

The language code has to be a valid IETF BCP-47 language code, such as en-US or de-CH. Cresta falls back to the profile's default language when this is not provided or if the provided value is not supported for the parent profile.

To send a message to Cresta to receive suggestions and hints, call the `addMessage()` API. This example illustrates a message sent by the visitor.

```
chat1.addMessage({
  messageId: 'MESSAGE_ID',
  speakerRole: 'visitor', // This can be 'agent' or 'system'
  text: 'I would like to learn more about your product pricing.',
  speakerUserId: 'VISITOR_ID',
  fromCurrentAgent: false,
  creationTimeMs: new Date().getTime(),
});
```

For agents sending a message:

```
chat1.addMessage({
  messageId: 'MESSAGE_ID',
  speakerRole: 'agent', // This can be 'agent' or 'system'
  text: 'Let me look up you account.',
  speakerUserId: 'AGENT_ID',
  // Whether the agent sending the message is the current agent signed in with
  // Cresta.
  fromCurrentAgent: true,
  creationTimeMs: new Date().getTime(),
});
```

To end a session after it is resolved (this will mark the chat as completed).

```
await chat1.end();
```

This will also destroy the session.

To destroy a session without ending it:

```
await chat1.destroy();
```

The same chat can still be reinitialized and restarted after it is destroyed.

To mount Cresta UI components, use the `components` reference on the chat instance. This will provide APIs to get the Cresta web components. These can then be appended to the DOM.

```
const chatSmartComposeEle = chat1.components.chatSmartComposeElement;
const chatHintsEle = chat1.components.chatHintsElement;
const chatSuggestionsEle = chat1.components.chatSuggestions;
const chatOmniSearchEle = chat1.components.chatOmniSearchElement;

containerElement.appendChild(chatSmartComposeEle);
containerElement.appendChild(chatHintsEle);
containerElement.appendChild(chatSuggestionsEle);
containerElement.appendChild(chatOmniSearchEle);
```

Or if using the smart compose overlay component:

```
// Get the smart compose overlay and attach it to the input content editable.
const contentEditable = document.getElementById('content-editable-box');
const chatSmartComposeOverlayEle =
    chat.components.getChatSmartComposeOverlayElement({
      targetElement: contentEditable
    });
const chatHintsEle = chat1.components.chatHintsElement;
const chatSuggestionsEle = chat1.components.chatSuggestions;
const chatOmniSearchEle = chat1.components.chatOmniSearchElement;

// Insert the overlay as a previous sibling of the input box.
contentEditable.parentNode.insertBefore(
    chatSmartComposeOverlayEle, contentEditable);
containerElement.appendChild(chatHintsEle);
containerElement.appendChild(chatSuggestionsEle);
containerElement.appendChild(chatOmniSearchEle);
```

Full details and snippets are available below on how to use the standalone smart compose element and the smart compose overlay element.

Chat Metadata

You can update metadata (e.g. conversation or visitor information) associated with a chat.

These can be used in the following ways:

- In Opera rules which can then change the type of Hint contents shown.
- As payload for webhooks from Cresta to external integrator automated workflows.
- To impact model output for Suggestions.
- Post call scenarios: real time coaching, assistance, QA/scorecard related actions.
- To filter or search for conversations in Cresta Director using various metadata filters.

Reach out to the Cresta team to learn more.

You can't currently self-manage metadata fields in Cresta Director. You will need to reach out to the Cresta team when adding new metadata fields and provide the following details:

- Field Name: the user friendly name used to filter chats by in the conversations list.
- Taxonomy: This is the unique key identifier to use. This is the same key used when setting the entry via the SDK. Note that for custom metadata fields, the `custom.` field will need to be prepended to the key.
- Metadata type: This can be a generic text, number or string value.
- Description: User friendly description of this metadata field.
- Whether the field represents a conversation outcome and limits the set of allowed values.

Note that the conversation has to be closed before it can be included in the filter results.

Setting the metadata fields on a specific chat session.

```
chat1.updateMetadata({
  'custom.chat_type': ['live'],
  'custom.internal_chat_ids': [23536536333, 37548067406],
  'custom.chat_settings': ['support', 'troubleshooting'],
  'custom.subscription_status': ['premium'],
});
```

Updating a new metadata object will not result in existing metadata values being removed if the new emission/update does not contain the existing metadata values' keys. As a result, you can either update the whole metadata snapshot or a diff.

Mount Cresta UI Components

Cresta provides built-in components that can be readily mounted in an application's UI.

This includes:

- Cresta Behavioral Hints Component: Used to provide behavioral hints to the agent. Examples: Offer a discount, remind the user to greet the visitor, show empathy, etc.
- Cresta Reply Suggestions Component: Used to display multiple reply and canned suggestions based on the context of the conversation. The agent can click to select a reply or canned suggestion and it will be auto-populated in the smart compose element where it can be further modified and then submitted.
- Cresta Omni Search Component: This includes the following features:
  - Cresta Knowledge Base search: Used to facilitate knowledge base search by entering text into the input box and getting related articles. For example, a visitor may inquire about a cancellation fee. The agent can search for that and get a summary and a link for more details on this topic.
  - Cresta Guided Workflow search: Used to facilitate guided workflow search by entering text into the input box and getting related workflows. For example, a visitor may inquire about some multi-step process (e.g. apply for a loan). The agent will use this component to search for this topic and then go through the related steps (linear or branching) while walking the visitor through them.
  - Cresta Generative Knowledge Assist: Used to facilitate generative knowledge assist search by entering text into the input box and getting AI generated reply and related articles. For example, a visitor may inquire about the return process. A retrieval question will be extracted by the backend and sent to the component. The agent can use the extracted question or enter the question manually and get the AI generated reply. **Note**: This feature is currently in beta phase and is still undergoing changes and improvements.

All components are available at the chat session level as HTML custom elements.

To get the reply / canned suggestions HTML element and append it or remove it from the DOM:

```
// Get the hints element.
const chatHintsEle = chat1.components.chatHintsElement;

// Append element to DOM.
containerElement.appendChild(chatHintsEle);

// Remove element from the DOM.
chatHintsEle.remove();
```

To get the reply suggestions HTML element and append it or remove it from the DOM:

```
// Get the reply suggestions element.
const chatSuggestionsEle = chat1.components.chatSuggestions;

// Append element to DOM.
containerElement.appendChild(chatSuggestionsEle);

// Remove element from the DOM.
chatSuggestionsEle.remove();
```

To enable canned suggestions in the reply suggestion component, set the `enable-canned-suggestions` attribute to `true`. The default is for this to be disabled.

```
chatSuggestionsEle.setAttribute("enable-canned-suggestions", "true");
```

To get the knowledge base / guided workflow search HTML element and append it or remove it from the DOM:

```
// Get the knowledge base search HTML element.
const chatOmniSearchEle = chat1.components.chatOmniSearchElement;

// To show only knowledge base search:
// chatOmniSearchEle.setAttribute('disable-guided-workflow', '');
// To show only guided workflow search:
// chatOmniSearchEle.setAttribute('disable-knowledge-base', '');
// To enable the generative knowledge assist:
// chatOmniSearchEle.setAttribute('enable-generative-knowledge-assist', '');

// Append element to DOM.
containerElement.appendChild(chatOmniSearchEle);

// Remove element from the DOM.
chatOmniSearchEle.remove();
```

To enable generative knowledge assist in the Omni Search component, set the `enable-generative-knowledge-assist` attribute to `true`. The default is `false` for this to be disabled.

```
chatOmniSearchEle.setAttribute("enable-generative-knowledge-assist", "true");
```

This will show the `GenAI Answer` tab when the component is expanded.

Generative knowledge assist is currently in beta phase and is still undergoing changes and improvements.

When knowledge base / guided workflow search is run, the component will expand and take up more space in the UI. To detect when the component is opened or closed, events will be triggered and can be listened to.

```
chatOmniSearchEle.addEventListener("omniSearchOpen", () => {
  // Omni search component opened.
});

chatOmniSearchEle.addEventListener("omniSearchClose", () => {
  // Omni search component closed.
});
```

When using guided workflow or knowledge base type hints with the Omni-search component, clicking the hint in the hints component will result in a guided workflow or knowledge base result being populated in the omni-search component. This may require scrolling to that component to bring it in view. To detect this hint click event, use the `omniSearchResultOpened` event that is triggered on the omni-search HTML element.

```
chatOmniSearchEle.addEventListener("omniSearchResultOpened", () => {
  // Omni search component result opened.
  chatOmniSearchEle.scrollIntoView();
});
```

For more details on how to customize the component styles, refer to the [UI components CSS Overrides]() section.

Usage snippet with smart compose standalone element

The Cresta Chat Smart Compose (Standalone) can be used if you don't want to build your own agent input box and just want to use the Cresta provided one.

It provides auto-complete hints and integrates with other Cresta components, e.g. reply suggestions.

This example illustrates a chat integration using the standalone smart compose component.

```
import {cresta, CrestaChatMessage, CrestaChat} from '@cresta/client';

// On app initialization.
const client = cresta.createClient({
  customerId: 'CUSTOMER_ID',
  // This is used for the authentication URL.
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This is the Cresta service endpoint. Replace with proxy origin if using proxy.
  serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
});

async function loginWithCresta() {
  try {
    await client.auth.signInWithPopup();
  } catch (e) {
    console.log(e.code);
  }
}

...
// After user logs in with platform, login with Cresta.
loginWithCresta();
```

```
// Listen to auth stage changes and handle them appropriately.
client.auth.onAuthStateChange((user) => {
  if (user) {
    // Show Cresta components.
  } else {
    // Hide Cresta components.
    // Prompt user to sign in again.
  }
});

class ChatUI {
  // ...

  async function initChatId(
      chatId: string,
      profileId: string,
      sendMessage: (text: string) => void
      draftMessage?: string,
  ): Promise<void> {
    this.chat = client.chatManager(profileId).chat(chatId);

    await this.chat.start({
      config: {
        visitor: {
          platformSpeakerId: 'VISITOR_ID',
          displayName: 'John Smith',
        },
      },
      messages: [/* initial messages */],
    });
    await this.chat.assignToCurrentAgent({
      // Agent display name.
      displayName: "Helpful Agent",
    });

    const chatSmartComposeEle = this.chat.components.chatSmartComposeElement;
    const chatHintsEle = this.chat.components.chatHintsElement;
    const chatSuggestionsEle = this.chat.components.chatSuggestions;
    const chatOmniSearchEle = this.chat.components.chatOmniSearchElement;

    this.containerElement.appendChild(chatSmartComposeEle);
    this.containerElement.appendChild(chatHintsEle);
    this.containerElement.appendChild(chatSuggestionsEle);
    this.containerElement.appendChild(chatOmniSearchEle);

    // Listen to smart compose submitted events.
    chatSmartComposeEle.addEventListener('submitted', (e) => {
      // Pass the sent message to the platform server to relay to the visitor.
      // Text message is in e.target.value.
      await sendMessage(...);
    });
    // Set any draft message in the smart compose element.
    chatSmartComposeEle.setAttribute('value', draftMessage || '');
  }

  // Triggered on a new message. Chat is already initialized.
  async function onNewMessage(message: CrestaChatMessage) {
    // Pass the new message to the chat service to update hints and suggestions.
    await chat.addMessage(message);
  }
}
```

Usage snippet with smart compose overlay element

The Cresta chat smart compose element has limitations for customers that have complex chat input boxes with advanced capabilities such as rich text support, ability to upload images, files, etc which are not supported in the standalone element. The Cresta chat smart compose overlay is designed to solve this issue where the customer can continue to use their chat input box while still receiving hints and suggestions from the overlay without changing the UX experience.

The main technical difference between the standalone and overlay element is that the latter requires a target element to be attached to and that the overlay needs to be mounted as a previous sibling to the target element `contenteditable`. The overlay also is not distinguishable from the target element it is attached to. It will mimic the exact look and style of that element.

The overlay works with `contenteditable` and `textarea` target elements. It won't work with an HTML `input` element.

The changes can be summarized in the following snippets.

The first snippet demonstrates usage with a `contenteditable` target element:

```
// Get the smart compose overlay and attach it to the input content editable.
const contentEditable = document.getElementById('content-editable-box');
const chatSmartComposeOverlayEle =
    chat.components.getChatSmartComposeOverlayElement({targetElement: contentEditable});

// Insert the overlay as a previous sibling of the input box.
contentEditable.parentNode.insertBefore(chatSmartComposeOverlayEle, contentEditable);

// Listen to smart compose overlay events.
chatSmartComposeOverlayEle.addEventListener('appendText', (e) => {
  // Append text to the contenteditable box.
  appendTextToContentEditable(contentEditable, e.detail.text);
});

chatSmartComposeOverlayEle.addEventListener('replaceText', (e) => {
  // Replace text in the contenteditable box.
  replaceTextInContentEditable(contentEditable, e.detail.text);
});
```

More complete sample:

```
import {cresta, CrestaChatMessage, CrestaChat} from '@cresta/client';
/**
 * Attempts to move the cursor to the end of the text in the element.
 *
 * @param element - The element where the cursor will be moved.
 */
function maybeMoveCursorToEnd(element: HTMLElement) {
  // Move cursor to end of text.
  element.focus();

  const selection = window.getSelection();

  if (selection?.rangeCount && selection.rangeCount > 0) {
    const range = selection.getRangeAt(0);

    range.selectNodeContents(element);
    range.collapse();
  }
}

// On app initialization.
const client = cresta.createClient({
  customerId: 'CUSTOMER_ID',
  // This is used for the authentication URL.
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This is the Cresta service endpoint. Replace with proxy origin if using proxy.
  serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
});

async function loginWithCresta() {
  try {
    await client.auth.signInWithPopup();
  } catch (e) {
    console.log(e.code);
  }
}

...
// After user logs in with platform, login with Cresta.
loginWithCresta();

// Listen to auth stage changes and handle them appropriately.
client.auth.onAuthStateChange((user) => {
  if (user) {
    // Show Cresta components.
  } else {
    // Hide Cresta components.
    // Prompt user to sign in again.
  }
});

class ChatUI {
```

```
// ...

async function initChatId(
    chatId: string,
    profileId: string,
    sendMessage: (text: string) => void
    draftMessage?: string,
): Promise<void> {
  this.chat = client.chatManager(profileId).chat(chatId);

  await this.chat.start({
    config: {
      visitor: {
        platformSpeakerId: 'VISITOR_ID',
        displayName: 'John Smith',
      },
    },
    messages: [/* initial messages */],
  });
  await this.chat.assignToCurrentAgent({
    // Agent display name.
    displayName: 'Helpful Agent',
  });

  // Get the smart compose overlay and attach it to the input content editable.
  const contentEditable = document.getElementById('content-editable-box');
  const chatSmartComposeOverlayEle =
      this.chat.components.getChatSmartComposeOverlayElement(
          {targetElement: contentEditable});

  const chatHintsEle = this.chat.components.chatHintsElement;
  const chatSuggestionsEle = this.chat.components.chatSuggestions;
  const chatOmniSearchEle = this.chat.components.chatOmniSearchElement;

  // Insert the overlay as a previous sibling of the input box.
  contentEditable.parentNode.insertBefore(chatSmartComposeOverlayEle, contentEditable);
  this.containerElement.appendChild(chatHintsEle);
  this.containerElement.appendChild(chatSuggestionsEle);
  this.containerElement.appendChild(chatOmniSearchEle);

  // Listen to smart compose overlay events.
  chatSmartComposeOverlayEle.addEventListener('appendText', (e) => {
    // Append text to the contenteditable box.
    appendTextToContentEditable(contentEditable, e.detail.text);
    maybeMoveCursorToEnd(contentEditable);
  });

  chatSmartComposeOverlayEle.addEventListener('replaceText', (e) => {
    // Replace text in the contenteditable box.
    replaceTextInContentEditable(contentEditable, e.detail.text);
    maybeMoveCursorToEnd(contentEditable);
  });
  }

  // Triggered on a new message. Chat is already initialized.
  async function onNewMessage(message: CrestaMessage) {
    // Pass the new message to the chat service to update hints and suggestions.
    await chat.addMessage(message);
  }
}
```

The second snippet demonstrates the differences when using with a `textarea` target element:

```
#smart-compose-overlay-container {
  position: relative;
}

.overlay-target {
  border: 1px solid #cccccc;
  height: 100px;
  width: 500px;
}
```

```
<div id="smart-compose-overlay-container">
  <textarea
    id="textarea-input-box"
    class="overlay-target"
    name="textarea"
  ></textarea>
</div>
```

```
// ...

class ChatUI {
  // ...

  async function initChatId(
      chatId: string,
      profileId: string,
      sendMessage: (text: string) => void
      draftMessage?: string,
  ): Promise<void> {
    this.chat = client.chatManager(profileId).chat(chatId);

    await this.chat.start({
      config: {
        visitor: {
          platformSpeakerId: 'VISITOR_ID',
          displayName: 'John Smith',
        },
      },
      messages: [/* initial messages */],
    });
    await this.chat.assignToCurrentAgent({
      // Agent display name.
      displayName: "Helpful Agent",
    });

    // Get the smart compose overlay and attach it to the textarea input
    const textArea = document.getElementById('textarea-input-box');
    const chatSmartComposeOverlayEle =
        this.chat.components.getChatSmartComposeOverlayElement(
            {targetElement: textArea});

    const chatHintsEle = this.chat.components.chatHintsElement;
    const chatSuggestionsEle = this.chat.components.chatSuggestions;
    const chatOmniSearchEle = this.chat.components.chatOmniSearchElement;
    // Insert the overlay as a previous sibling of the input box.
    // When using absolute position, you will need to insert the overlay
    // after the textarea instead.
    textArea.parentNode.insertBefore(chatSmartComposeOverlayEle, textArea);
    this.containerElement.appendChild(chatHintsEle);
    this.containerElement.appendChild(chatSuggestionsEle);
    this.containerElement.appendChild(chatOmniSearchEle);

    // Listen to smart compose overlay events.
    chatSmartComposeOverlayEle.addEventListener('appendText', (e) => {
      // Append selected hint text to the textarea.
      textArea.value = `${textArea.value}${e.detail.text}`;
      // Trigger input event to notify smart compose overlay of the state
      // change.
      contentEditable.dispatchEvent(new Event('input'));
    });

    chatSmartComposeOverlayEle.addEventListener('replaceText', (e) => {
      // Replace text in the textarea.
      textArea.value = e.detail.text;
      // Trigger input event to notify smart compose overlay of the state
      // change.
      contentEditable.dispatchEvent(new Event('input'));
    });
  }

  // ...
}
```

By default the Smart Compose Overlay uses the following shortcut keys:

- Right Arrow to select the next word in the provided hint.
- Tab to select the entire hint.

To customize the shortcut keys, a `keyboardShortcutConfig` needs to be specified. In the following configuration, right arrow + SHIFT is used for the single word selection and right arrow alone is used for the full hint selection.

```
chatSmartComposeOverlayEle.keyboardShortcutConfig = {
  // Use arrow right + Shift key for single word selection.
  insertSingleWordKeyboardShortcut: {
    enabled: true,
    keyboardShortcut: {
      key: "ArrowRight",
      shiftKey: true,
    },
    keyboardShortcutChipTemplate:
      (document.getElementById(
        "word-by-word-key-chip"
      ) as HTMLTemplateElement) ?? undefined,
  },
  // Use arrow right for full hint selection
  insertFullHintKeyboardShortcut: {
    enabled: true,
    keyboardShortcut: {
      key: "ArrowRight",
    },
    keyboardShortcutChipTemplate:
      (document.getElementById("full-hint-key-chip") as HTMLTemplateElement) ??
      undefined,
  },
};
```

A keyboard shortcut key chip (for each key) can also be specified. This is the chip that is shown at the right side of the hint indicating the shortcut key used for the single word or full hint selection. When using a custom config, an `HTMLTemplateElement` for the chip for each key can be specified. This is typically unicode characters, SVG or PNG. By specifying them as `HTMLTemplateElement`, these won't be rendered in the DOM.

```
<template id="word-by-word-key-chip">⇧→</template>
<template id="full-hint-key-chip">
  <svg
    width="24"
    height="24"
    xmlns="http://www.w3.org/2000/svg"
    fill-rule="evenodd"
    clip-rule="evenodd"
  >
    <path
      d="M21.883 12l-7.527 6.235.644.765 9-7.521-9-7.479-.645.764 7.529
6.236h-21.884v1h21.883z"
    />
  </svg>
```

```
</template>
```

Build your own Components (Headless mode)

As part of the client SDK, built-in UI components (Cresta smart compose, reply suggestions, behavioral suggestions, etc) are readily provided to facilitate integration and reduce onboarding time significantly. The fixed and stable CSS structure of these components and the CSS class names are documented to allow developers the ability to customize the UI by overriding these CSS classes.

However, if the level of customization provided is not sufficient to achieve the desired user experience and styling requirements, the SDK provides event-based APIs to enable developers the ability to build their own custom UI components that integrate with Cresta AI services.

The event-based APIs provide the same level of service as the built-in UI components. They provide a mechanism to collect related analytics, signals and feedback on par with the built-in UI components. They are also compatible with the existing UI components (developers can use both types of components at the same time or use a custom-built component, e.g. smart compose with another built-in component, e.g. suggestions).

Unlike the built-in components, building your own component will require more effort to implement correctly. Care must be taken to ensure that analytics signals continue to be triggered to provide Cresta feedback and the signals necessary to maintain the high quality AI service. We always recommend starting with the built-in UI components first and trying to customize those. If the level of customization is insufficient or limiting, then resort to building your own components.

Smart Compose

The client SDK provides event-based APIs to facilitate building your own custom Smart Compose component. The API exposes the following functionality:

- Ability to generate hint candidates for provided draft text.
- Built-in Smart Compose cache to optimize when a new request for a hint candidate needs to be requested and when cached responses can be used instead.
- Built-in throttling for performance purposes.
- Ability to detect incoming text (e.g. from reply or behavioral suggestions being selected) that needs to be populated in the Smart Compose component.
- Analytics signals triggers for various actions (hint shown in UI, hint partially or fully selected, etc.)

The event-based APIs are available from the `CrestaChat` via `chat.events.smartCompose`.

Whenever the agent types text into the custom Smart Compose input box, the `updateDraftText()` API needs to be called:

```
// As the user types, update the selected text so far. This should be throttled.
chat1.events.smartCompose.updateDraftText(draftText);
```

You can then subscribe to hint candidates that match the draft text via `onHintCandidate()` or `hintCandidate$` observable.

```
// Listen to new smart compose hint candidates.
// Subscribe using RxJS observable.
// chat1.events.smartCompose.hintCandidate$.subscribe((candidate) => {...});
// Subscribe using callback function.
chat1.events.smartCompose.onHintCandidate((candidate) => {
  // Remove previous hint.
  // ...
  // Show new hint in the DOM (in gray state). Then trigger analytics event.
  // ...
  // Unique ID identifying the candidate hint.
  console.warn(candidate.id);
  // The hint text.
  console.warn(candidate.hintText);
  // Trigger analytics event when the candidate is shown to the end user.
  candidate.show();
});
```

To listen to incoming text that needs to be populated in the Smart Compose component when a reply suggestion or behavioral suggestion is clicked, you can subscribe via `onText()` or `text$` observable.

```
// Listen to new incoming text. This may happen if a reply suggestion or
// behavior suggestion is clicked. The text content should be populated in the
// Smart Compose input element.
// Subscribe using RxJS observable.
// chat1.events.smartCompose.text$.subscribe((text) => {...});
// Subscribe using callback function.
chat1.events.smartCompose.onText((text) => {
  // Show text in the Smart Compose input element.
  // ...
});
```

Every time a hint candidate is generated, a unique ID is provided on the candidate and a mechanism to lookup the hint candidate by ID is available (as long as the hint candidate is not stale/destroyed).

```
const hintCandidate = chat1.events.smartCompose.get(candidateId);
```

When the hint is shown (appended to the DOM), the `show()` method needs to be called.

```
chat1.events.smartCompose.get(candidateId)?.show();
```

When the hint is selected, either partially via single word selection or fully by selecting the entire hint, the `insert()` method needs to be called.

```
// Get a reference to the candidate trigger insert analytics event.
chat1.events.smartCompose.get(candidateId)?.insert(
  // If partially inserted (e.g. Agent selects a single word from the hint).
  false
  // If the full hint is inserted (e.g. Agent selects the full hint).
  // true
);
```

Fully functioning illustrative sample code is available in the Vanilla JS sample app `@cresta/client-sdk-sample` and the React JS sample app `@cresta/client-sdk-react-sample`.

The Cresta emulator provides a mode to run diagnostics to analyze whether a custom-built Smart Compose component is compliant with Cresta best practices and requirements.

To take advantage of this mode, start the emulator with the `--run-compliance-tests` flag.

```
npx cresta-emulator --run-compliance-tests
```

You will need to point your application Cresta config `serviceEndpoint` to the emulator (the default URL is `http://localhost:8081`).

The emulator will print out instructions in the terminal to follow on actions to run in the application as it records usage and then when the emulator process is stopped, a report will be printed out in the terminal with recommendations on how to fix potential issues.

Sample output report:

```
──────────────────────────── Running compliance tests ────────────────────────────
Generating report for Smart Compose...
- Out of 4 generated hints, no events were triggered to
  indicate the hint candidate was shown.
  "candidate.show()" needs to be called on a hint candidate
  when it is appended to DOM.
- No partially selected (word by word) smart compose hint
  was detected.
  "candidate.insert(false)" needs to be called when a hint
  candidate is partially selected, e.g. right arrow or a
  custom key is clicked and the next word from the hint is
  inserted.
- No fully selected smart compose hint was detected.
  "candidate.insert(true)" needs to be called when the hint
  candidate is fully selected, e.g. tab or a custom key is
  clicked and the full hint is inserted or the user types
  the full hint, etc.
```

Reply and Canned Suggestions

The client SDK provides event-based APIs to facilitate building your own custom Reply and Canned Suggestions component. The API exposes the following functionality:

- Ability to detect the latest reply suggestions for a specified conversation.
- Ability to detect the latest canned suggestions based on text typed into the smart compose input box.
- Analytics signals triggers for various actions (suggestions shown in UI, suggestion is selected, etc.)

The event-based APIs are available from the `CrestaChat` via `chat.events.replySuggestions`.

Normally build your own suggestions are not used with built-in suggestions component. You would use one or the other.

You can then subscribe to incoming Cresta reply suggestions for a conversation via `onSuggestions()` or `suggestions$` observable.

```
// Listen to reply suggestions change.
// Subscribe using RxJS observable.
// chat1.events.replySuggestions.suggestions$.subscribe((suggestions) => {...});
// Subscribe using callback function.
chat1.events.replySuggestions.onSuggestions((suggestions) => {
  // Remove all previous suggestions.
  // ...
  suggestions.forEach((suggestion) => {
    // Show new suggestion in DOM. Then trigger analytics event.
    // ...
    // Unique ID identifying the reply suggestions.
    console.warn(suggestion.id);
    // The reply suggestion text.
    console.warn(suggestion.text);
    // Trigger analytics event when the suggestion is shown to the end user.
    suggestion.show();
  });
});
```

Similarly you can subscribe to incoming Cresta canned suggestions based on agent input via `onCannedSuggestions()` or `cannedSuggestions$` observable.

However, canned suggestions must be enabled first. They are disabled by default:

```
chat1.events.replySuggestions.enableCannedSuggestions = true;
```

Note that canned suggestions are similar to reply suggestions but come with an additional title to be displayed along with the content.

```
// Listen to canned suggestions change.
// Subscribe using RxJS observable.
// chat1.events.replySuggestions.cannedSuggestions$
//     .subscribe((suggestions) => {...});
// Subscribe using callback function.
chat1.events.replySuggestions.onCannedSuggestions((suggestions) => {
  // Remove all previous suggestions.
  // ...
  suggestions.forEach((suggestion) => {
    // Show new suggestion in DOM. Then trigger analytics event.
    // ...
    // Unique ID identifying the canned suggestions.
    console.warn(suggestion.id);
    // The canned suggestion title.
    console.warn(suggestion.title);
    // The canned suggestion text.
    console.warn(suggestion.text);
    // Trigger analytics event when the suggestion is shown to the end user.
    suggestion.show();
  });
});
```

Every time a reply or canned suggestion is generated, a unique ID is provided on the suggestion and a mechanism to lookup the suggestion by ID is available (as long as the suggestion is not stale/destroyed).

```
const suggestions = chat1.events.replySuggestions.get(candidateId);
```

When the suggestion is shown (appended to the DOM), the `show()` method needs to be called.

```
chat1.events.replySuggestions.get(candidateId)?.show();
```

When the suggestion is selected (via user click), the `select()` method needs to be called.

```
// Get a reference to the suggestion and trigger select analytics event.
chat1.events.replySuggestions.get(candidateId)?.select();
```

More comprehensive snippet:

```
// Listen to reply suggestions change.
chat1.events.replySuggestions.onSuggestions((suggestions) => {
  // Remove all previous suggestions.
  const suggestionContainer = document.getElementById("suggestions-container");
  suggestionContainer.innerHTML = "";
  // Insert new suggestions.
  suggestions.forEach((suggestion) => {
    // Show new suggestion in DOM.
    const row = document.createElement("div");
    const text = document.createTextNode(suggestion.text);
    row.appendChild(text);
    suggestionContainer.appendChild(row);
    // Add suggestion ID as a data attribute.
    row.setAttribute("data-suggestion-id", suggestion.id);
    // On element click, select the suggestion.
    row.addEventListener("click", (e) => {
      // Get the suggestion ID.
      const suggestionId = e.target.getAttribute("data-suggestion-id");
      // Lookup the suggestion associated with the suggestion ID>
      const suggestionRef = chat1.events.replySuggestions.get(suggestionId);
      // Select the suggestion.
      suggestionRef?.select();
    });
    // Mark suggestion as shown.
    suggestion.show();
  });
});

// Listen to canned suggestions change.
chat1.events.replySuggestions.onCannedSuggestions((suggestions) => {
  // Remove all previous suggestions.
  const suggestionContainer = document.getElementById(
    "canned-suggestions-container"
  );
  suggestionContainer.innerHTML = "";
  // Insert new suggestions.
  suggestions.forEach((suggestion) => {
    // Show new suggestion in DOM.
    const row = document.createElement("div");
    const title = document.createElement("h6");
    const text = document.createElement("span");
    text.appendChild(document.createTextNode(suggestion.content));
    title.appendChild(document.createTextNode(suggestion.title));
    row.appendChild(title);
    row.appendChild(text);
    suggestionContainer.appendChild(row);
    // Add suggestion ID as a data attribute.
    row.setAttribute("data-suggestion-id", suggestion.id);
    // On element click, select the suggestion.
    row.addEventListener("click", (e) => {
      // Get the suggestion ID.
      const suggestionId =
        (e.target as HTMLDivElement).getAttribute("data-suggestion-id") ??
        (
          (e.target as HTMLDivElement).parentNode as HTMLDivElement
        )?.getAttribute("data-suggestion-id");
      // Lookup the suggestion associated with the suggestion ID.
      const suggestionRef = chat1.events.replySuggestions.get(suggestionId);
      // Select the suggestion.
      suggestionRef?.select();
    });
    // Mark suggestion as shown.
    suggestion.show();
  });
});
```

Fully functioning illustrative sample code is available in the Vanilla JS sample app `@cresta/client-sdk-sample` and the React JS sample app `@cresta/client-sdk-react-sample`.

The Cresta emulator provides a mode to run diagnostics to analyze whether a custom-built Reply Suggestions component is compliant with Cresta best practices and requirements. To take advantage of this mode, start the emulator with the `--run-compliance-tests` flag.

```
npx cresta-emulator --run-compliance-tests
```

You will need to point your application Cresta config `serviceEndpoint` to the emulator (the default URL is `http://localhost:8081`).

The emulator will print out instructions in the terminal to follow on actions to run in the application as it records usage and then when the emulator process is stopped, a report will be printed out in the terminal with recommendations on how to fix potential issues.

Sample output report:

```
──────────────────────────────────── Running compliance tests
────────────────────────────────────

Generating report for Reply Suggestions...
- Out of 3 generated suggestions, no events were triggered to indicate
  any reply suggestions were shown.
  "suggestion.show()" needs to be called on a reply suggestion when it
  is appended to the DOM.
- No selected reply suggestion was detected.
  "candidate.select()" needs to be called when the reply suggestion is
  selected via click, etc.
- No submitted reply suggestion was detected.
  There could be an issue in the population of the suggestion in Smart
  Compose. To detect incoming suggestion, use
  "chat1.events.smartCompose.onText()" or
  "chat1.events.smartCompose.text$" to detect the incoming suggestion
  text.

Generating report for Canned Suggestions...
- No selected canned suggestion was detected.
  "suggestion.select()" needs to be called when the canned suggestion
  is selected via click, etc.
- No submitted canned suggestion was detected.
  There could be an issue in the population of the suggestion in
  Smart Compose. To detect incoming suggestions, use
  "chat1.events.smartCompose.onText()" or
  "chat1.events.smartCompose.text$" to detect the incoming suggestion
  text.
─────────────────────────────────────────────────────────────────────
───────────────────────────
```

Hints

The client SDK provides event-based APIs to facilitate building your own custom Hints component. The API exposes the following functionality:

- Ability to detect the latest hints for a specified conversation. This includes the title, suggestion text, description, how long the hint is displayed, etc.
- Analytics signals triggers for various actions (hints shown in UI, dismissed by the user, description expanded or the hint suggestion is selected, etc.)

The event-based APIs are available from the `CrestaChat` via `chat.events.hints`.

You can then subscribe to incoming Cresta reply suggestions for a conversation via `onHints()` or `hints$` observable.

```
// Listen to hints change.
// Subscribe using RxJS observable.
// chat1.events.hints.hints$.subscribe((hints) => {...});
// Subscribe using callback function.
chat1.events.hints.onHints((hints) => {
  // Remove all previous hints.
  // ...
  hints.forEach((hint) => {
    // Show new hint in DOM. Then trigger analytics event.
    // ...
    // Unique ID identifying the hints.
    console.warn(hint.id);
    // The hint type.
    console.warn(hint.type);
    // The hint title.
    console.warn(hint.title);
    // The optional hint description.
    console.warn(hint.description);
    // The optional hint suggestion.
    console.warn(hint.suggestionText);
    // The duration to wait in milliseconds before dismissing the hint.
    console.warn(hint.durationMs);
    // Trigger analytics event when the hint is shown to the end user.
    suggestion.show();
  });
});
```

There are currently 4 types of hints that are supported. You can determine the hint type by inspecting the `hint.type` property value:

- `standardHint`: This is a standard hint. Selecting a suggestion in this hint should populate the suggestion text in Smart Compose for chat conversations.
- `knowledgeBaseHint`: This is a knowledge-base article hint. Selecting this hint should select the corresponding article in the Omni-search component.
- `guidedWorkflowHint`: This is a guided workflow hint. Selecting this hint should select the corresponding workflow in the Omni-search component.
- `complianceHint`: This is a compliance hint.

Every time a hint is generated, a unique ID is provided on the hint and a mechanism to lookup the hint by ID is available (as long as the hint is not stale/destroyed).

```
const hint = chat1.events.hints.get(hintId);
```

When the hint is shown (appended to the DOM), the `show()` method needs to be called.

```
chat1.events.hints.get(hintId)?.show();
```

When the hint suggestion is selected (via user click), the `select()` method needs to be called. This will do the following depending on the type of hint:

- Populate the suggestion text in the smart compose input box for a standard hint.
- Open the KB article or the guided workflow in the Omni-search component for a KB article hint or guided workflow hint.

```
// Get a reference to the hint and trigger select event.
chat1.events.hints.get(hintId)?.select();
```

When the hint is dismissed by the user (via a dismiss button click), the `dismiss()` method needs to be called.

```
// Get a reference to the hint and trigger dismiss event.
```

```
chat1.events.hints.get(hintId)?.dismiss();
```

The hint should typically be dismissed automatically after some duration `hint.dismissMs`:

```
if (hint.durationMs) {
  setTimeout(() => {
    if (!hint.dismissed && !hint.destroyed) {
      hint.dismiss();
    }
  }, hint.durationMs);
}
```

An optional description is provided. If the description is minimized and then expanded via some tooltip, etc. the `expand()` method needs to be called:

```
if (hint.description) {
  // The tooltip is not immediately shown. Ideally, we should trigger
  // this if the duration between mouseenter and mouseleave exceeds some
  // threshold, e.g. > 500ms.
  title.addEventListener("mouseover", (e) => {
    hint.expand();
  });
}
```

More comprehensive snippet:

```
// Listen to hints change.
chat1.events.hints.onHints((hints) => {
  // Remove all previous hints.
  const hintsContainer = document.getElementById("hints-container");
  hintsContainer.innerHTML = "";

  // Insert new hints.
  hints.forEach((hint) => {
    // Show new hint in DOM.
    // Hint container.
    const row = document.createElement("div");
    // Hint title element.
    const title = document.createElement("div");
    // Hint suggestion text element.
    const suggestion = document.createElement("div");
    // Dismiss hint button.
    const dismissButton = document.createElement("button");
    if (hint.title) {
      title.appendChild(document.createTextNode(hint.title));
      // Set description as tooltip.
      title.setAttribute("title", hint.description ?? hint.title);
      row.appendChild(title);
      // Expand hint when description tooltip is shown. If this is always
      // visible, trigger it immediately.
      if (hint.description) {
        // The tooltip is not immediately shown. Ideally, we should trigger
        // this if the duration between mouseenter and mouseleave exceeds some
        // threshold, e.g. > 500ms.
        title.addEventListener("mouseover", (e) => {
          // Get the hint ID.
          const hintId = e.target.parentElement.getAttribute("data-hint-id");
          // Lookup the hint associated with the hint ID.
          const hintRef = chat1.events.hints.get(hintId);
          // Expand the hint.
          hintRef?.expand();
        });
      }
    }
    // Append suggestion text if available.
    if (hint.suggestion) {
      suggestion.appendChild(document.createTextNode(hint.suggestion));
      row.appendChild(suggestion);
      // On element click, select the suggestion.
      suggestion.addEventListener("click", (e) => {
        // Get the hint ID.
        const hintId = e.target.parentElement.getAttribute("data-hint-id");
        // Lookup the hint associated with the hint ID.
        const hintRef = chat1.events.hints.get(hintId);
        // Select the hint.
        hintRef?.select();
      });
    }
    // Append dismiss button.
    row.appendChild(dismissButton);
    // On dismiss button click, trigger hint.dismiss().
    dismissButton.addEventListener("click", (e) => {
      // Get the hint ID.
      const hintId = e.target.parentElement.getAttribute("data-hint-id");
      // Lookup the hint associated with the hint ID.
      const hintRef = chat1.events.hints.get(hintId);
      // Dismiss the hint.
      hintRef?.dismiss();
    });
    // If hint durationMs is available, dismiss the hint after that duration.
    if (hint.durationMs) {
      setTimeout(() => {
        if (!hint.dismissed && !hint.destroyed) {
          dismissButton.click();
        }
      }, hint.durationMs);
    }
    // Append hint to hints container.
    hintsContainer.appendChild(row);
    // Add hint ID as a data attribute.
    row.setAttribute("data-hint-id", hint.id);
    // Mark hint as shown.
    hint.show();
  });
});
```

Fully functioning illustrative sample code is available in the Vanilla JS sample app `@cresta/client-sdk-sample` and the React JS sample app `@cresta/client-sdk-react-sample`.

The Cresta emulator provides a mode to run diagnostics to analyze whether a custom-built Hints component is compliant with Cresta best practices and requirements.
To take advantage of this mode, start the emulator with the `--run-compliance-tests` flag.

```
npx cresta-emulator --run-compliance-tests
```

You will need to point your application Cresta config `serviceEndpoint` to the emulator (the default URL is `http://localhost:8081`).

The emulator will print out instructions in the terminal to follow on actions to run in the application as it records usage and then when the emulator process is stopped, a report will be printed out in the terminal with recommendations on how to fix potential issues.

Sample output report:

```
──────────────────────────────────── Running compliance tests
────────────────────────────────────

Generating report for Hints...
- Out of 2 generated hints, no events were triggered to indicate any
  hints were shown.
  "hint.show()" needs to be called on a hint when it is appended to
  the DOM.
- No expanded hint description was detected.
  "hint.expand()" needs to be called when a hint description is
  expanded.
- No dismissed hint was detected.
  "hint.dismiss()" needs to be called when a hint is dismissed via
  click or after the "hint.durationMs" expires.
──────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────
```

Voice Integrations

Unlike chat integrations, agent assist voice integrations with Cresta typically require that audio streams and audio conversation events are streamed or sent to Cresta servers via server to server API calls. This is done separately from the frontend integration via the Cresta client SDK. The client side integration is mainly responsible for detecting these conversations and consuming/displaying their related Cresta AI features (e.g. Cresta hints, transcription services, auto-summarization, etc.) for the agent to interact with

Initialize and manage a voice session

To detect an incoming voice conversation, add a voice session listener on the voice manager. When an incoming voice session is detected, you can detect various conversation data (conversation ID, status, start and end time).

```
const voiceManager = client.voiceManager("PROFILE_ID");

const unsubscribeVoiceSession = voiceManager.onVoiceSession((voiceSession) => {
  console.log(
    `Voice session change detected, ${voiceSession.id}, ${voiceSession.status}`
  );

  // Voice conversation ID.
  console.log(voiceSession.id);
  // Voice conversation status ("started", "ended", "on-hold", "unknown").
  console.log(voiceSession.status);
  // Voice conversation start time (ISO date string format).
  console.log(voiceSession.startTime);
  // Voice conversation end time (ISO date string format), null if unavailable.
  console.log(voiceSession.endTime);
});
```

To stop listening to voice sessions.

```
unsubscribeVoiceSession();
```

Voice sessions will be detected for a specific agent. When the logged in agent changes or is signed out, the previous voice sessions should be removed from the DOM and a new listener should be subscribed for the new user:

```
client.auth.onAuthStateChange((user) => {
  if (user) {
    // New sign in or a change of user.
    // Remove all voice session components and unsubscribe from existing voice
    // session listeners.
    unsubscribeVoiceSession();
    // Clear UI.
    // ...
    // Re-subscribe to new voice sessions.
    voiceManager.onVoiceSession((voiceSession) => {
      // ...
    });
  } else {
    // No user logged in.
    // Remove all voice session components and unsubscribe from existing voice
    // session listeners.
    unsubscribeVoiceSession();
    // Clear UI.
    // ...
  }
});
```

You can get the list of all active voice sessions at any time. Note that only one voice session can be active at any time:

```
const voiceSessions = voiceManager.getVoiceSessions();
```

To listen to status changes in a specific voice session:

```
const unsubscribeVoiceSessionStatus = voiceSession.onStatusChange(() => {
  // Status, start and end time may change as a result.
  // Voice conversation status ("started", "ended", "on-hold", "unknown").
  console.log(voiceSession.status);
  // Voice conversation start time (ISO date string format).
  console.log(voiceSession.startTime);
  // Voice conversation end time (ISO date string format), null if unavailable.
  console.log(voiceSession.endTime);

  // If a session ends, it is recommended that the session is detroyed and no
  // longer referenced to free up memory and resources.
  if (voiceSession.status === "ended") {
    console.log(
      `Voice session change detected, ${session.id}, ${session.status}`
    );

    // Destroy and remove from DOM after some delay.
    setTimeout(() => {
      session.destroy();
      sessionContainerEle.remove();
    }, 5000);
  }
});
```

To stop listening to a voice session status changes.

```
unsubscribeVoiceSessionStatus();
```

Mount Cresta UI Components

Cresta provides built-in components that can be readily mounted in an application's UI.

This includes:

- Cresta Behavioral Hints Component: Used to provide behavioral hints to the agent. Examples: Offer a discount, remind the user to greet the visitor, show empathy, etc.
- Cresta Omni Search Component: This includes the following features:
    - Cresta Knowledge Base search: Used to facilitate knowledge base search by entering text into the input box and getting related articles. For example, a visitor may inquire about a cancellation fee. The agent can search for that and get a summary and a link for more details on this topic.

- Cresta Guided Workflow search: Used to facilitate guided workflow search by entering text into the input box and getting related workflows. For example, a visitor may inquire about some multi-step process (e.g. apply for a loan). The agent will use this component to search for this topic and then go through the related steps (linear or branching) while walking the visitor through them.
- Cresta Generative Knowledge Assist: Used to facilitate generative knowledge assist search by entering text into the input box and getting AI generated reply and related articles. For example, a visitor may inquire about the return process. A retrieval question will be extracted by the backend and sent to the component. The agent can use the extracted question or enter the question manually and get the AI generated reply. **Note**: This feature is currently in beta phase and is still undergoing changes and improvements.

All components are available at the voice session level as HTML custom elements.

To get the hints HTML element and append it or remove it from the DOM:

```
// Get the hints element.
const voiceHintsEle = voiceSession.components.voiceHintsElement;

// Append element to DOM.
containerElement.appendChild(voiceHintsEle);

// Remove element from the DOM.
voiceHintsEle.remove();
```

To get the knowledge base / guided workflow search HTML element and append it or remove it from the DOM:

```
// Get the knowledge base search HTML element.
const voiceOmniSearchEle = voiceSession.components.voiceOmniSearchElement;

// To show only knowledge base search:
// voiceOmniSearchEle.setAttribute('disable-guided-workflow', '');
// To show only guided workflow search:
// voiceOmniSearchEle.setAttribute('disable-knowledge-base', '');
// To enable the generative knowledge assist:
// voiceOmniSearchEle.setAttribute('enable-generative-knowledge-assist', '');

// Append element to DOM.
containerElement.appendChild(voiceOmniSearchEle);

// Remove element from the DOM.
voiceOmniSearchEle.remove();
```

To enable generative knowledge assist in the Omni Search component, set the `enable-generative-knowledge-assist` attribute to `true`. The default is `false` for this to be disabled.

```
voiceOmniSearchEle.setAttribute("enable-generative-knowledge-assist", "true");
```

This will show the `GenAI Answer` tab when the component is expanded.

When knowledge base / guided workflow search is run, the component will expand and take up more space in the UI. To detect when the component is opened or closed, events will be triggered and can be listened to.

```
voiceOmniSearchEle.addEventListener("omniSearchOpen", () => {
  // Omni search component opened.
});

voiceOmniSearchEle.addEventListener("omniSearchClose", () => {
  // Omni search component closed.
});
```

When using guided workflow or knowledge base type hints with the Omni-search component, clicking the hint in the hints component will result in a guided workflow or knowledge base result being populated in the omni-search component. This may require scrolling to that component to bring it in view. To detect this hint click event, use the `omniSearchResultOpened` event that is triggered on the omni-search HTML element.

```
voiceOmniSearchEle.addEventListener("omniSearchResultOpened", () => {
  // Omni search component result opened.
  chatOmniSearchEle.scrollIntoView();
});
```

For more details on how to customize the component styles, refer to the UI components CSS Overrides section.

Build your own Components (Headless mode)

As part of the client SDK, built-in UI components (behavioral suggestions/hints, etc) are readily provided to facilitate integration and reduce onboarding time significantly. The fixed and stable CSS structure of these components and the CSS class names are documented to allow developers the ability to customize the UI by overriding these CSS classes.

However, if the level of customization provided is not sufficient to achieve the desired user experience and styling requirements, the SDK provides event-based APIs to enable developers the ability to build their own custom UI components that integrate with Cresta AI services.

The event-based APIs provide the same level of service as the built-in UI components. They provide a mechanism to collect related analytics, signals and feedback on par with the built-in UI components. They are also compatible with the existing UI components (developers can use both types of components at the same time or use a custom-built component with another built-in component).

Unlike the built-in components, building your own component will require more effort to implement correctly. Care must be taken to ensure that analytics signals continue to be triggered to provide Cresta feedback and the signals necessary to maintain the high quality AI service. We always recommend starting with the built-in UI components first and trying to customize those. If the level of customization is insufficient or limiting, then resort to building your own components.

Hints

The client SDK provides event-based APIs to facilitate building your own custom Hints component. The API exposes the following functionality:

- Ability to detect the latest hints for a specified conversation. This includes the title, suggestion text, description, how long the hint is displayed, etc.
- Analytics signals triggers for various actions (hints shown in UI, dismissed by the user, description expanded or the hint suggestion is selected, etc.)

The event-based APIs are available from the `CrestaVoice` via `voiceSession.events.hints`.

You can then subscribe to incoming Cresta hints for a conversation via `onHints()` or `hints$` observable.

```
// Listen to hints change.
// Subscribe using RxJS observable.
// voiceSession.events.hints.hints$.subscribe((hints) => {...});
// Subscribe using callback function.
voiceSession.events.hints.onHints((hints) => {
  // Remove all previous hints.
  // ...
  hints.forEach((hint) => {
    // Show new hint in DOM. Then trigger analytics event.
    // ...
    // Unique ID identifying the hints.
    console.warn(hint.id);
    // The hint type.
    console.warn(hint.type);
    // The hint title.
    console.warn(hint.title);
    // The optional hint description.
    console.warn(hint.description);
    // The optional hint suggestion.
    console.warn(hint.suggestionText);
    // The duration to wait in milliseconds before dismissing the hint.
    console.warn(hint.durationMs);
    // Trigger analytics event when the hint is shown to the end user.
    suggestion.show();
  });
});
```

There are currently 4 types of hints that are supported. You can determine the hint type by inspecting the `hint.type` property value:

- `standardHint`: This is a standard hint. Selecting a suggestion in this hint should populate the suggestion text in Smart Compose for chat conversations.
- `knowledgeBaseHint`: This is a knowledge-base article hint. Selecting this hint should select the corresponding article in the Omni-search component.
- `guidedWorkflowHint`: This is a guided workflow hint. Selecting this hint should select the corresponding workflow in the Omni-search component.
- `complianceHint`: This is a compliance hint.

Every time a hint is generated, a unique ID is provided on the hint and a mechanism to lookup the hint by ID is available (as long as the hint is not stale/destroyed).

```
const hint = voiceSession.events.hints.get(hintId);
```

When the hint is shown (appended to the DOM), the `show()` method needs to be called.

```
voiceSession.events.hints.get(hintId)?.show();
```

When the hint suggestion is selected (via user click), the `select()` method needs to be called. This will do the following depending on the type of hint:

- Populate the suggestion text in the smart compose input box for a standard hint for chat conversations.
- Open the KB article or the guided workflow in the Omni-search component for a KB article hint or guided workflow hint.

```
// Get a reference to the hint and trigger select event.
voiceSession.events.hints.get(hintId)?.select();
```

When the hint is dismissed by the user (via a dismiss button click), the `dismiss()` method needs to be called.

```
// Get a reference to the hint and trigger dismiss event.
voiceSession.events.hints.get(hintId)?.dismiss();
```

The hint should typically be dismissed automatically after some duration `hint.dismissMs`:

```
if (hint.durationMs) {
  setTimeout(() => {
    if (!hint.dismissed && !hint.destroyed) {
      hint.dismiss();
    }
  }, hint.durationMs);
}
```

An optional description is provided. If the description is minimized and then expanded via some tooltip, etc. the `expand()` method needs to be called:

```
if (hint.description) {
  // The tooltip is not immediately shown. Ideally, we should trigger
  // this if the duration between mouseenter and mouseleave exceeds some
  // threshold, e.g. > 500ms.
  title.addEventListener("mouseover", (e) => {
    hint.expand();
  });
}
```

More comprehensive snippet:

```
// Remove all previous hints.
const hintsContainer = document.getElementById("hints-container");

// Listen to hints change.
voiceSession.events.hints.onHints((hints) => {
  hintsContainer.innerHTML = "";
  hintsContainer.className = "hints-container";

  // Insert new hints.
  hints.forEach((hint) => {
    // Show new hint in DOM.
    // Hint container.
    const row = document.createElement("div");
    // Hint title element.
    const title = document.createElement("div");
    // Hint suggestion text element.
    const suggestion = document.createElement("div");
    // Dismiss hint button.
    const dismissButton = document.createElement("button");
    if (hint.title) {
      title.appendChild(document.createTextNode(hint.title));
      // Set description as tooltip.
      title.setAttribute("title", hint.description ?? hint.title);
      row.appendChild(title);

      // Expand hint when description tooltip is shown. If this is always
      // visible, trigger it immediately.
      if (hint.description) {
        // The tooltip is not immediately shown. Ideally, we should trigger
        // this if the duration between mouseenter and mouseleave exceeds some
        // threshold, e.g. > 500ms.
        title.addEventListener("mouseover", (e) => {
          // Get the hint ID.
          const hintId = e.target.parentElement.getAttribute("data-hint-id");
          // Lookup the hint associated with the hint ID.
          const hintRef = voiceSession.events.hints.get(hintId);
          // Expand the hint.
          hintRef?.expand();
        });
      }
    }

    // Append suggestion text if available.
    if (hint.suggestionText) {
      suggestion.appendChild(document.createTextNode(hint.suggestionText));
      row.appendChild(suggestion);

      // On element click, select the suggestion.
      suggestion.addEventListener("click", (e) => {
        // Get the hint ID.
        const hintId = e.target.parentElement.getAttribute("data-hint-id");
        // Lookup the hint associated with the hint ID.
        const hintRef = voiceSession.events.hints.get(hintId);
        // Select the hint.
        hintRef?.select();
      });
    }

    // Append dismiss button.
    row.appendChild(dismissButton);

    // On dismiss button click, trigger hint.dismiss().
    dismissButton.addEventListener("click", (e) => {
      // Get the hint ID.
      const hintId = e.target.parentElement.getAttribute("data-hint-id");
      // Lookup the hint associated with the hint ID.
      const hintRef = voiceSession.events.hints.get(hintId);
      // Dismiss the hint.
      hintRef?.dismiss();
    });

    // If hint durationMs is available, dismiss the hint after that duration.
    if (hint.durationMs) {
      setTimeout(() => {
        if (!hint.dismissed && !hint.destroyed) {
          dismissButton.click();
        }
      }, hint.durationMs);
    }

    // Append hint to hints container.
    hintsContainer.appendChild(row);
    // Add hint ID as a data attribute.
    row.setAttribute("data-hint-id", hint.id);
    // Mark hint as shown.
    hint.show();
  });
});
```

Fully functioning illustrative sample code is available in the Vanilla JS sample app `@cresta/client-sdk-sample` and the React JS sample app `@cresta/client-sdk-react-sample`.

Controlling network settings

You can control the Cresta API call network timeout and retry delay via the `cresta` app singleton. This will apply to all Cresta client instances:

```
import { cresta } from "@cresta/client";

cresta.configure({
  network: {
    // Timeout network requests after 30 seconds.
    // When not specified, the browser default value is used.
    timeoutMs: 30000,
    // Delay before retrying after a failed network requests.
    // To stop retry, set to negative number. Setting to 0 will use the default
    // 2s.
    retryDelayMs: 45000,
  },
});
```

You can also disable usage of WebSockets to communicate with the Cresta `ClientSubscription` service which subscribes to events used by Cresta features, e.g. reply and behavioral suggestions. You have the option to use server-sent events or polling when WebSockets are disabled. It is recommended not to disable this unless necessary. When disabling WebSockets, server-sent events are recommended over polling as using polling to pull this data from Cresta is a far more expensive operation.

For chat, this setting has to be set before the first conversation is created.
For voice, this setting has to be set before instantiating the voice manager.

```
import { cresta } from '@cresta/client';
cresta.configure({
  network: {
    // Timeout network requests after 30 seconds.
    // When not specified, the browser default value is used.
    timeoutMs: 30000,
    // Delay before retrying after a failed network requests.
    // To stop retry, set to negative number. Setting to 0 will use the default
    // 2s.
    retryDelayMs: 45000,
    // Disable WebSockets and use polling. This is false by default.
    // For chat, this has to be set before the first conversation is created.
    // For voice, this setting has to be set before instantiating the voice
    // manager, e.g. client.voiceManager(profileId)
    // It cannot be changed afterwards.
    disableWebSockets: true,
    // Disable WebSockets and set the polling interval, in milliseconds, to
    // query the Cresta server for suggestions and other data. The default
    // value is 2 seconds. Care must be taken when updating this value as long
    // durations may result in suggestions being delayed.
    // disableWebSockets: {pollingIntervalMs: 3000},
    // It is recommended to use server-sent events (SSE) instead of polling when
    // disabling WebSockets by setting:
    // disableWebSockets: {enableEventSource: true},
  },
});
```

When using polling with chat integrations, it is recommended to explicitly set chat visibility when switching between chat sessions for performance reasons. By setting a chat as invisible, any underlying polling will be paused and resumed once the chat is made visible again.

```
// When the agent switches from chat1 to chat2.
chat1.visibility = 'hidden';
chat2.visibility = 'visible';

// When the agent switches back to chat1.
chat1.visibility = 'visible';
chat2.visibility = 'hidden';
```

For voice integrations, the polling cannot be turned off after it is enabled but the polling interval can be dynamically changed (increased or decreased) as needed.

You can customize the polling interval via the `pollingIntervalMs` setting. The default value is 2 seconds. This is useful when network connections are slow or when conversations are not very active or do not require immediate real-time interactions. This can be adjusted on-demand (increased when conversations are not active, decreased when conversations are active). It is preferable to use the `visibility` feature over this.

Updating the chat visibility is not needed when using server-sent events (SSE) for long lived connections to the Cresta server.

Cresta SDK exposes the ability to intercept outgoing HTTP requests to the Cresta server. This may be useful when using a proxy server between the app and the Cresta server. For example, the proxy server may be using a different path for each proxied API and may be applying additional authentication, e.g. HMAC auth header authentication, etc:

In addition, Cresta SDK exposes the ability to intercept incoming HTTP responses from the Cresta server and allows for modification of these responses before the SDK processes them.

```
cresta.configure({
  network: {
    // The callback is triggered right before the HTTP request is processed.
    httpRequestInterceptors: [
      // Multiple callbacks can be passed. They will be executed sequentially
      // with the output HttpRequest of one callback feeding as an input into
      // the next callback.
      async (request: HttpRequest) => {
        // URL path: request.urlPath
        // Headers: request.headers
        // Raw HTTP request body: request.body
        // HTTP method: request.method
        // Example: add addititonal authentication header:
        request.headers["Other-Authentication"] = await generateSignature(
          request
        );

        // Return the modified HTTP request.
        return {
          ...request,
          // Modify the URL path to match the proxy's expected one.
          // The request origin should use the one defined in the CrestaConfig,
          // e.g. serviceEndpoint. Modifying that should be done in the
          // CrestaConfig.
          urlPath: `/proxy/path/cresta/${request.urlPath}`,
        };
      },
    ],
    // The callback is triggered right after the HTTP response is received and
    // before it is processed by the SDK.
    httpResponseInterceptors: [
      // Multiple callbacks can be passed. They will be executed sequentially
      // with the output HttpResponse of one callback feeding as an input into
      // the next callback.
      async (response: HttpResponse) => {
        // The API identifier of the response: response.kind.
        // The parsed JSON response body: response.body.
        if (
          response.kind ===
          "cresta.v1.smartcompose.GenerateSmartComposeCandidatesResponse"
        ) {
          // Log the draft text and the generated smart compose hint.
          console.log(`Draft text: ${response.body.draftText}`);
          console.log(
            `Generated hint: ${response.body.smartComposeCandidates[0]?.text}`
          );
          return { ...response };
        }
        // The response that will be received and processed by the SDK.
        return response;
      },
    ],
  },
});
```

React Client SDK (@cresta/react-client)

The Cresta React Client SDK `@cresta/react-client` provides Cresta UI components for react applications. Used in tandem with the `@cresta/client` package, they can offer all the functionalities and UI components needed to integrate with Cresta chat and voice agent assist services.

Note: You can view the `@cresta/react-client` documentation and API references via browser by installing the package and then running a local web server and pointing it to serve the static HTML files at `./node_modules/@cresta/react-client/docs/html`. A quick way to do that is to install `http-server` module and use that to serve the HTML site:

```
npm install -g http-server
http-server ./node_modules/@cresta/react-client/docs/html
```

You can then navigate to the launched website.

Detailed API

Refer to: index.d.ts typings file in **@cresta/react-client** package.

Prerequisites for @cresta/react-client

As `@cresta/react-client` has a peer dependency on `@cresta/client`, the same prerequisites are needed.

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save @cresta/react-client@latest
```

Other than `@cresta/client`, the `rxjs` and `react` peer dependencies also need to be satisfied in the `package.json`:

```
"dependencies": {
  "@cresta/react-client": "latest",
  "@cresta/client": "latest",
  "react": "^16.8.0",
  "rxjs": "^7.3.1",
  ...
}
```

Usage

Integration with chat services

Mount UI Components

Instead of using DOM elements provided by the `@cresta/client` package, `@cresta/react-client` provides wrapper react components for use.

```
import React, {useState, useEffect} from 'react';
import {CrestaChat, CrestaUser} from '@cresta/client';
import {CrestaChatSmartCompose, CrestaChatHints, CrestaChatSuggestions,
CrestaChatOmniSearch} from '@cresta/react-client';
const CrestaWidgetWithSmartCompose = (props: {
  chat: CrestaChat;
  user: CrestaUser;
}) => {
  const onSubmitted = (message: string) => {
    // Handle agent sent message
    const message = {
      messageId: randomString(5),
      speakerRole: 'agent',
      text,
      speakerUserId: props.user.userId,
      fromCurrentAgent: true,
      creationTimeMs: new Date().getTime(),
    } as CrestaChatMessage;
    props.chat.addMessage(message);
  };
  const smartComposeButtonRef = useRef(null);
  return <>
    <div>
      <CrestaChatSmartCompose
        chatKey={props.chat.key}
        onSubmitted={onSubmitted}
        sendButtonRef={smartComposeButtonRef} />
      <button ref={smartComposeButtonRef}>Send</button>
    </div>
    <CrestaChatHints chatKey={props.chat.key} />
    <CrestaChatSuggestions
      chatKey={props.chat.key}
      enableCannedSuggestions={
        // Set to true to enable canned suggestions.
        // Enable canned suggestions. Normally build your own suggestions are
        // not used with built-in suggestions component. You would use one or
        // the other.
        // For demonstration purposes we are using both in this sample app.
        true
      }
    />
    <CrestaChatOmniSearch chatKey={props.chat.key} />
  </>;
};
```

Or if using the smart compose overlay component:

```
import React, {useState, useEffect} from 'react';
import {CrestaChat, CrestaUser} from '@cresta/client';
import {CrestaChatSmartComposeOverlay, CrestaChatHints, CrestaChatSuggestions,
CrestaChatOmniSearch} from '@cresta/react-client';
const CrestaWidgetWithSmartComposeOverlay = (props: {
  chat: CrestaChat;
  user: CrestaUser;
}) => {
  const [targetElement, setTargetElement] = useState(null);
  const inputBox = useRef(null);
  useEffect(() => {
    if (inputBox.current !== null) {
      setTargetElement(inputBox.current);
    }
  }, [inputBox.current]);
  const onReplaceText = (e: CustomEvent) => {
    // handling insert selected suggestions or hints.
    if (targetElement) {
      targetElement.innerText = e.detail.text;
      // other handling like moving cursor to the end of the text, focus etc.
    }
  };
  const onAppendText = (e: CustomEvent) => {
    // handling insert accepted smart compose hint.
    if (targetElement) {
      targetElement.innerText += e.detail.text;
      // other handling like moving cursor to the end of the text, focus etc.
    }
  };
  const submit = () => {
    // handling send button click
     if (targetElement) {
      const text = targetElement.innerText || '';
      const message = {
        messageId: randomString(5),
        speakerRole: 'agent',
        text,
        speakerUserId: props.user.userId,
        fromCurrentAgent: true,
        creationTimeMs: new Date().getTime(),
      } as CrestaChatMessage;
      props.chat.addMessage(message);
      targetElement.innerText = '';
      targetElement.focus();
    }
  };
  return <>
    {targetElement &&
      <CrestaChatSmartComposeOverlay
        chatKey={props.chat.key}
        targetElement={targetElement}
        onReplaceText={onReplaceText}
        onAppendText={onAppendText} />}
    <div ref={inputBox} contentEditable />
    <button onClick={submit}>Send</button>
    <CrestaChatHints chatKey={props.chat.key} />
    <CrestaChatSuggestions
      chatKey={props.chat.key}
      enableCannedSuggestions={
        // Set to true to enable canned suggestions.
        // Enable canned suggestions. Normally build your own suggestions are
        // not used with built-in suggestions component. You would use one or
        // the other.
        // For demonstration purposes we are using both in this sample app.
        true
      }
    />
    <CrestaChatOmniSearch chatKey={props.chat.key} />
  </>;
};
```

By default the Smart Compose Overlay uses the following shortcut keys:

- Right Arrow to select the next word in the provided hint.
- Tab to select the entire hint.

To customize the shortcut keys, a `keyboardShortcutConfig` needs to be specified. In the following configuration, right arrow + SHIFT is used for the single word selection and right arrow alone is used for the full hint selection.

```
const CrestaWidgetWithSmartComposeOverlay = (props: {
  chat: CrestaChat;
  user: CrestaUser;
}) => {
  // ...
  return (
    <>
      {targetElement && (
        <CrestaChatSmartComposeOverlay
          chatKey={props.chat.key}
          targetElement={targetElement}
          onReplaceText={onReplaceText}
          onAppendText={onAppendText}
          keyboardShortcutConfig={{
            // Use arrow right + Shift key for single word selection.
            insertSingleWordKeyboardShortcut: {
              enabled: true as const,
              keyboardShortcut: {
                key: "ArrowRight",
                shiftKey: true,
              },
              KeyboardShortcutChipComponent: SingleWordChipComponent,
            },
            // Use arrow right for full hint selection
            insertFullHintKeyboardShortcut: {
              enabled: true as const,
              keyboardShortcut: {
                key: "ArrowRight",
              },
              KeyboardShortcutChipComponent: FullHintChipComponent,
            },
          }}
        />
      )}
      <div ref={inputBox} contentEditable />
      <button onClick={submit}>Send</button>
      <CrestaChatHints chatKey={props.chat.key} />
      <CrestaChatSuggestions
        chatKey={props.chat.key}
        enableCannedSuggestions={
          // Set to true to enable canned suggestions.
          // Enable canned suggestions. Normally build your own suggestions are
          // not used with built-in suggestions component. You would use one or
          // the other.
          // For demonstration purposes we are using both in this sample app.
          true
        }
      />
      <CrestaChatOmniSearch chatKey={props.chat.key} />
    </>
  );
};
const SingleWordChipComponent = () => <>{"\u21E7\u2192"}</>;
const FullHintChipComponent = () => (
  <svg
    width="24"
    height="24"
    xmlns="http://www.w3.org/2000/svg"
    fillRule="evenodd"
    clipRule="evenodd"
  >
    <path d="M21.883 12l-7.527 6.235.644.765 9-7.521-9-7.479-.645.764 7.529
6.236h-21.884v1h21.883z" />
  </svg>
);
```

A keyboard shortcut key chip (for each key) can also be specified. This is the chip that is shown at the right side of the hint indicating the shortcut key used for the single word or full hint selection. When using a custom config, a React functional or class Component for the chip for each key can be specified.

The other available Cresta React components can be mounted as follows:

- Cresta Behavioral Hints Component: Used to provide behavioral hints to the agent. Examples: Offer a discount, remind the user to greet the visitor, show empathy, etc.

  ```
  <CrestaChatHints chatKey={props.chat.key} />
  ```

- Cresta Reply Suggestions Component: Used to display multiple reply and canned suggestions based on the context of the conversation. The agent can click to select a reply or canned suggestion and it will be auto-populated in the smart compose element where it can be further modified and then submitted.

  ```
  <CrestaChatSuggestions chatKey={props.chat.key} />
  ```

  To enable canned suggestions in the reply suggestion component, set the `enableCannedSuggestions` property to `true`. The default is for this to be disabled.

  ```
  <CrestaChatSuggestions
    chatKey={props.chat.key}
    enableCannedSuggestions={true}
  ```

```
/>
```

- Cresta Omni Search Component: This includes the following features:
    - Cresta Knowledge Base search: Used to facilitate knowledge base search by entering text into the input box and getting related articles. For example, a visitor may inquire about a cancellation fee. The agent can search for that and get a summary and a link for more details on this topic.
    - Cresta Guided Workflow search: Used to facilitate guided workflow search by entering text into the input box and getting related workflows. For example, a visitor may inquire about some multi-step process (e.g. apply for a loan). The agent will use this component to search for this topic and then go through the related steps (linear or branching) while walking the visitor through them.
    - Cresta Generative Knowledge Assist: Used to facilitate generative knowledge assist search by entering text into the input box and getting AI generated reply and related articles. For example, a visitor may inquire about the return process. A retrieval question will be extracted by the backend and sent to the component. The agent can use the extracted question or enter the question manually and get the AI generated reply. **Note**: This feature is currently in beta phase and is still undergoing changes and improvements.

```
<div ref={omniSearchComponent}>
   <CrestaChatOmniSearch
     chatKey={props.chat.key}
     disableGuidedWorkflow={false}
     disableKnowledgeBase={false}
     onSearchOpen={() => {
       console.log("Omni search component opened.");
     }}
     onSearchClose={() => {
       console.log("Omni search component closed.");
     }}
     onSearchResultOpened={() => {
       // This may happen when a guided workflow or knowledge base type hint is
       // clicked in the hint component. This will result in a guided workflow
       // or knowledge base result being populated in the omni-search
       // component. This may require scrolling to that component to bring it
       // in view.
       consoles.log("Omni search component result opened.");
       omniSearchComponent.current?.scrollIntoView();
     }}
   />
 </div>
```

To enable Generative Knowledge Assist in the Omni Search component, set the `enableGenerativeKnowledgeAssist` property to `true`. The default is `false` for this to be disabled.

```
<div ref={omniSearchComponent}>
  <CrestaChatOmniSearch
    chatKey={props.chat.key}
    enableGenerativeKnowledgeAssist={true}
    onSearchOpen={() => {
      console.log("Omni search component opened.");
    }}
    onSearchClose={() => {
      console.log("Omni search component closed.");
    }}
    onSearchResultOpened={() => {
      // This may happen when a guided workflow or knowledge base type hint is
      // clicked in the hint component. This will result in a guided workflow
      // or knowledge base result being populated in the omni-search
      // component. This may require scrolling to that component to bring it
      // in view.
      console.log("Omni search component result opened.");
      omniSearchComponent.current?.scrollIntoView();
    }}
  />
</div>
```

Usage snippet to initialize the chat

```
import React, {useEffect, useRef, useState} from 'react';
import {cresta, CrestaChatMessage, CrestaChat} from '@cresta/client';
// Use this as an event handler to login.
async function loginWithCresta(client: CrestaClient) {
  try {
    await client.auth.signInWithPopup();
  } catch (e) {
    console.log(e.code);
  }
}
...
const ChatUI = () => {
  const [client, setClient] = useState<CrestaClient|null>(null);
  const [user, setUser] = useState<CrestaUser|null>(null);
  const [chat, setChat] = useState<CrestaChat|null>(null);
  useEffect(() => {
    // On app initialization.
    const client = cresta.createClient({
      customerId: 'CUSTOMER_ID',
      // This is used for the authentication URL.
      customerOrigin: 'https://CUSTOMER_ID.cresta.com',
      // This is the Cresta service endpoint. Replace with proxy origin if using
      // proxy.
      serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
    });
    client.auth.onAuthStateChange((user) => {
      setUser(user);
    });
    setClient(client);
  }, []);
  const chatIdInput = useRef(null);
  const startChat = (chatId: string) => {
    if (client) {
      const newChat = client
        .chatManager(config.profileId).chat(chatId);
      newChat.start({
        config: {
          visitor: {
            platformSpeakerId: `visitor-${randomString(20)}`,
            displayName: 'John Smith',
          },
        },
      }).then(() => {
        addLog(`Started: chat ID ${newChat.id}`);
        return newChat.assignToCurrentAgent({
          // Agent display name.
          displayName: 'Helpful Agent',
        });
      }).then(() => {
        addLog(`Assigned chat ID ${newChat.id} to current agent.`);
        setChat(newChat);
      });
    }
  };
  if (chat) {
    // Or use <CrestaWidgetWithSmartComposeOverlay />
    return <CrestaWidgetWithSmartCompose
      chat={chat}
      user={user} >
  } else {
    // Provide your own logic here to get a chat id  or pass in a chat id from
    // other places to this component. This is just an example to demo how to
    // initialize a chat.
    return <>
      <input placeholder='enter a chat id' type='text' ref={chatIdInput} />
      <button onClick={() => {
        if (chatIdInput.current) {
          startChat(chatIdInput.current.value);
        }
      }}>Start chat</button>
    </>;
  }
};
```

Integration with voice services

Mount UI Components

Instead of using DOM elements provided by the `@cresta/client` package, `@cresta/react-client` provides their react components for use.

```
import React, { useState, useEffect } from "react";
import { CrestaClient, CrestaUser, CrestaVoice, CrestaVoiceManager } from "@cresta/client";
import { CrestaVoiceHints, CrestaVoiceOmniSearch } from "@cresta/react-client";

export const App = () => {
  const [client, setClient] = useState<CrestaClient|null>(null);
  const [user, setUser] = useState<CrestaUser|null>(null);

  useEffect(() => {
    // On app initialization.
    const client = cresta.createClient({
      customerId: 'CUSTOMER_ID',
      // This is used for the authentication URL.
      customerOrigin: 'https://CUSTOMER_ID.cresta.com',
      // This is the Cresta service endpoint. Replace with proxy origin if using
      // proxy.
      serviceEndpoint: 'https://api-CUSTOMER_ID.cresta.com',
    });
    client.auth.onAuthStateChange((user) => {
      setUser(user);
    });
    setClient(client);
  }, []);

  return (
    <>
      {client && <VoicePanel
        client={client}
        voiceManager={client.voiceManager(PROFILE_ID)}
        user={user} />}
    </>
  );
}

const VoicePanel = (props: {
  client: CrestaClient;
  voiceManager: CrestaVoiceManager;
  user: CrestaUser|null;
}) => {
  const [voiceSessions, setVoiceSessions] = useState<CrestaVoice[]>(
      [...props.voiceManager.getVoiceSessions()]);

  useEffect(() => props.voiceManager.onVoiceSession(() => {
    setVoiceSessions([...props.voiceManager.getVoiceSessions()]);
  }), []);

  return <div className='voice-session-container'>
    {voiceSessions.map((voiceSession) => (
      <VoiceSession
          key={voiceSession.key}
          voiceSession={voiceSession}
          setVoiceSessions={setVoiceSessions}/>
    ))}
  </div>;
}

const VoiceSession = (props: {
  voiceSession: CrestaVoice;
  setVoiceSessions: ((voiceSessions: CrestaVoice[]) => void);
}) => {
  const [status, setStatus] = useState<string>(props.voiceSession.status);
  const [startTime, setStartTime] = useState<string|undefined>(
      props.voiceSession.startTime);
  const [endTime, setEndTime] = useState<string|undefined>(
      props.voiceSession.endTime);

  useEffect(() => props.voiceSession.onStatusChange(
      (newStatus) => {
        setStatus(newStatus);
        setStartTime(props.voiceSession.startTime);
        setEndTime(props.voiceSession.endTime);

        if (newStatus === 'ended') {
          console.log(
              `Voice session change detected, ${props.voiceSession.id}, ${
                  newStatus}`);
          setTimeout(() => {
            props.voiceSession.destroy();
            props.setVoiceSessions(
                [...props.voiceSession.voiceManager.getVoiceSessions()]);
          }, 5000);
        }
      }), []
  );

  return <div className='voice-session'>
    <h6 className='voice-session-title'>
      Voice call: {props.voiceSession.id}
    </h6>
    <ul className='voice-session-info'>
      <li>Status: {status}</li>
      <li>Start time: {startTime ? new Date(startTime).toString(): 'N/A'}</li>
      <li>End time: {endTime ? new Date(endTime).toString(): 'ongoing'}</li>
    </ul>
    <CrestaVoiceHints voiceKey={props.voiceSession.key} />
    <div ref={omniSearchComponent}>
```

```
      <CrestaVoiceOmniSearch
        voiceKey={props.voiceSession.key}
        disableGuidedWorkflow={false}
        disableKnowledgeBase={false}
        enableGenerativeKnowledgeAssist={true}
        onSearchOpen={() => {
          props.addLog('Omni search component opened.');
        }}
        onSearchClose={() => {
          props.addLog('Omni search component closed.');
        }}
        onSearchResultOpened={() => {
          // This may happen when a guided workflow or knowledge base type hint
          // is clicked in the hint component. This will result in a guided
          // workflow or knowledge base result being populated in the
          // omni-search component. This may require scrolling to that component
          // to bring it in view.
          props.addLog('Omni search component result opened.');
          omniSearchComponent.current?.scrollIntoView();
        }}
      />
    </div>
  </div>;
}
```

Cresta React components can be mounted as follows:

- Cresta Behavioral Hints Component: Used to provide behavioral hints to the agent. Examples: Offer a discount, remind the user to greet the visitor, show empathy, etc.

  ```
  <CrestaVoiceHints voiceKey={props.voiceSession.key} />
  ```

- Cresta Omni Search Component: This includes the following features:
  - Cresta Knowledge Base search: Used to facilitate knowledge base search by entering text into the input box and getting related articles. For example, a visitor may inquire about a cancellation fee. The agent can search for that and get a summary and a link for more details on this topic.
  - Cresta Guided Workflow search: Used to facilitate guided workflow search by entering text into the input box and getting related workflows. For example, a visitor may inquire about some multi-step process (e.g. apply for a loan). The agent will use this component to search for this topic and then go through the related steps (linear or branching) while walking the visitor through them.
  - Cresta Generative Knowledge Assist: Used to facilitate generative knowledge assist search by entering text into the input box and getting AI generated reply and related articles. For example, a visitor may inquire about the return process. A retrieval question will be extracted by the backend and sent to the component. The agent can use the extracted question or enter the question manually and get the AI generated reply. **Note**: This feature is currently in beta phase and is still undergoing changes and improvements.

```
<div ref={omniSearchComponent}>
   <CrestaVoiceOmniSearch
     voiceKey={props.voiceSession.key}
     disableGuidedWorkflow={false}
     disableKnowledgeBase={false}
     onSearchOpen={() => {
       console.log("Omni search component opened.");
     }}
     onSearchClose={() => {
       console.log("Omni search component closed.");
     }}
     onSearchResultOpened={() => {
       // This may happen when a guided workflow or knowledge base type hint is
       // clicked in the hint component. This will result in a guided workflow
       // or knowledge base result being populated in the omni-search
       // component. This may require scrolling to that component to bring it
       // in view.
       console.log("Omni search component result opened.");
       omniSearchComponent.current?.scrollIntoView();
     }}
   />
 </div>
```

To enable Generative Knowledge Assist in the Omni Search component, set the `enableGenerativeKnowledgeAssist` property to `true`. The default is `false` for this to be disabled.

```
<div ref={omniSearchComponent}>
   <CrestaVoiceOmniSearch
     voiceKey={props.voiceSession.key}
     enableGenerativeKnowledgeAssist={true}
     onSearchOpen={() => {
       console.log("Omni search component opened.");
     }}
     onSearchClose={() => {
       console.log("Omni search component closed.");
     }}
     onSearchResultOpened={() => {
       // This may happen when a guided workflow or knowledge base type hint is
       // clicked in the hint component. This will result in a guided workflow
       // or knowledge base result being populated in the omni-search
       // component. This may require scrolling to that component to bring it
       // in view.
       console.log("Omni search component result opened.");
       omniSearchComponent.current?.scrollIntoView();
     }}
   />
 </div>
```

UI components CSS Overrides

Customization can be achieved by overriding the Cresta component CSS classes with custom CSS styles.

Smart compose
The standalone smart compose component is a wrapper of the smart compose overlay and a div contenteditable. To edit styles, stick to modifying the nested `div[role="textbox"]` content editable.

Classes (from outer to inner, indentations illustrate the nested tree structure):

```
cresta-smart-compose
    div.smart-compose
        content-editable-overlay
            .smart-compose__hint
            .smart-compose__chips-wrapper
                .smart-compose__tab
                .smart-compose__right-arrow
    div[role="textbox"] .smart-compose-content-editable
```

Chat suggestions
Classes (from outer to inner, indentations illustrate the nested tree structure):

```
.cresta-suggestion-cards__wrapper
    .cresta-suggestion-cards
        .cresta-suggestion-card
            .cresta-suggestion-card__header
            .cresta-suggestion-card__content
```

Note that `.cresta-suggestion-card__header` will show up for suggestions with titles (typical for canned suggestions).

Chat hints
Classes (from outer to inner, indentations illustrate the nested tree structure):

```
.hint__wrapper
      .hint__box
          .hint__content
              .hint__card
                  .hint__card_panel
                      .hint__card_panel_header
                          .hint__card_panel_title
                              .hint__card_panel_content flex_row
                                  .hint__card_panel_title_icon
                                      .hint__card_panel_title_text
                              .hint__card_panel_x_button
                          .hint__card_panel_content
```

Cresta Omni Search Component
Classes (from outer to inner, indentations illustrate the nested tree structure):

Closed view:

```
cresta-omni-search
  lib-omni-seach
    lib-omni-seach-search-panel
      .omni-search__panel .omni-search__panel_closed
        lib-omni-search-search-panel-search-header
          .omni-search__search_header_container
            .omni-search__search_header_input_container
              lib-omni-search-input.omni-search__search_header_input
                .omni-search__search_input_container
                  .omni-search__search_input_prefix
                    .omni-search__search_input_icon
                      lib-omni-search-search-icon
                  input.omni-search__search_input
                  .omni-search__search_input_suffix
```

Expanded autocomplete:

```
cresta-omni-search
  lib-omni-seach
    lib-omni-seach-search-panel
      .omni-search__panel .omni-search__panel_closed
        lib-omni-search-search-panel-search-header
          .omni-search__search_header_container
            .omni-search__search_header_input_container
              lib-omni-search-input.omni-search__search_header_input
                .omni-search__search_input_container
                  .omni-search__search_input_prefix
                    .omni-search__search_input_icon
                      lib-omni-search-search-icon
                  input.omni-search__search_input
                  .omni-search__search_input_suffix
                  .omni-search__search_autocomplete_container
                    ul.search__search_autocomplete_list
                      li.omni-search__search_autocomplete_list_item_active
                        lib-omni-search-search-icon
                        ...
                      li.omni-search__search_autocomplete_list_item
                        lib-omni-search-search-icon
                        ...
                      li.omni-search__search_autocomplete_list_item
                        lib-omni-search-search-icon
                        ...
```

Expanded view with search results with knowledge base and workflow results:

```
cresta-omni-search
```

```
lib-omni-seach
  lib-omni-seach-search-panel
    .omni-search__panel .omni-search__panel_opened
      lib-omni-search-search-panel-search-header
        .omni-search__search_header_container
          .omni-search__search_header_input_container
            lib-omni-search-input.omni-search__search_header_input
              .omni-search__search_input_container
                .omni-search__search_input_prefix
                  .omni-search__search_input_icon
                    lib-omni-search-search-icon
                input.omni-search__search_input
                .omni-search__search_input_suffix
                  .omni-search__search_input_icon .omni-search__search_input_clear_icon
                    lib-omni-search-clear-icon
            .omni-search__search_header_close_button_container
              .omni-search__close_search_panel_button_icon

      .omni-search__panel_content_container
        .omni-search__panel_content_results
          lib-omni-search-search-result-tab-container
            .omni-search__search_result_tab_bar
              .omni-search__search_result_tab_bar_item
                ...
              .omni-search__search_result_tab_bar_item
                ...
              .omni-search__search_result_tab_bar_item
                ...
            .omni-search__search_result_tab_content
              lib-omni-search-search-result-card
                .omni-search__search_result_card
                  .omni-search__search_result_header
                    .omni-search__search_result_snippet_header
                      .omni-search__workflow-snippet_header
                        .omni-search__search_result_snippet_icon
                          .omni-search__workflow_icon
                        .omni-search__search_result_snippet_icon_text
                  .omni-search__search_result_title
                  .omni-search__search_result_passage
                    .omni-search__search_result_passage_workflow_preview_text
                      ...
                    .omni-search__search_result_passage_workflow_preview_text
                      ...

              lib-omni-search-search-result-card
                .omni-search__search_result_card
                  .omni-search__search_result_header
                    .omni-search__search_result_snippet_header
                      .omni-search__knowledge-base-snippet_header
                        .omni-search__search_result_snippet_icon
                          .omni-search__knowledge_base_icon
                        .omni-search__search_result_snippet_icon_text
                      .omni-search__search_result_snippet_header_action_button
```

```
                        .omni-search__copy_button_icon
                      .omni-search__search_result_snippet_header_action_button
                        .omni-search__new_tab_button_icon
                  .omni-search__search_result_title
                  .omni-search__search_result_passage
                    lib-knowledge-base-article-body
                      .kb-article__body
                        .kb-article__body_wrapper

              lib-omni-search-search-result-card
                ...
              lib-omni-search-search-result-card
                ...
              lib-omni-search-search-result-card
                ...
```

Selected single knowledge base article view:

```
cresta-omni-search
  lib-omni-seach
    lib-omni-seach-search-panel
      .omni-search__panel .omni-search__panel_opened
        lib-omni-search-active-result
          .omni-search__active_result_container
            .omni-search__active_knowledge_base_result
              lib-knowledge-base-article
                lib-knowledge-base-article-panel
                  .kb-article__panel
                    lib-knowledge-base-article-top-bar
                      .kb-article__top_bar
                        .kb-article__back_button_text
                    lib-knowledge-base-article-header
                      .kb-article__header
                        .kb-article__header_title
                        .kb-article__action_button
                          .kb-article__copy_button_icon
                        .kb-article__action_button
                          .kb-article__new_tab_button_icon
                    lib-knowledge-base-article-body
                      .kb-article__body
                        .kb-article__body_wrapper
```

Selected single branching workflow view:

```
cresta-omni-search
  lib-omni-seach
    lib-omni-seach-search-panel
      .omni-search__panel .omni-search__panel_opened
        lib-omni-search-active-result
          .omni-search__active_result_container
            .omni-search__active_knowledge_base_result
              .omni-search__active_guided_workflow_result
        lib-guided-workflow
          lib-guided-workflow-active-workflow
            .guided-workflow__active_workflow_view
              lib-guided-workflow-top-bar
                .guided-workflow__top_bar
                  .guided-workflow__back_button_text
              div
                .guided-workflow__icon_container
                  .guided-workflow__guided_workflow_icon
                  .guided-workflow__icon_text
                lib-guided-workflow-branching-workflow
                  .guided-workflow__branching_workflow_view
                    lib-guided-workflow-workflow-step
                      .guided-workflow__workflow_step_card_container
                        .guided-workflow__branching_workflow_step_card
                          .guided-workflow__branching_workflow_step_text_container
                            .guided-workflow__branching_workflow_step_text
                              .guided-workflow__step_text
                              .guided-workflow__step_text_insert_icon
                        .guided-workflow__branching_workflow_divider
                        .guided-workflow__step_next_group
                          lib-guided-workflow-workflow-next
                            .guided-workflow__workflow_next_card
                          lib-guided-workflow-workflow-next
                            ...
                          lib-guided-workflow-workflow-next
                            ...
```

Selected single linear workflow view:

```
cresta-omni-search
  lib-omni-seach
    lib-omni-seach-search-panel
      .omni-search__panel .omni-search__panel_opened
        lib-omni-search-active-result
          .omni-search__active_result_container
            .omni-search__active_knowledge_base_result
              .omni-search__active_guided_workflow_result
                lib-guided-workflow
                  lib-guided-workflow-active-workflow
                    .guided-workflow__active_workflow_view
                      lib-guided-workflow-top-bar
                        .guided-workflow__top_bar
                          .guided-workflow__back_button_text
                      div
                        .guided-workflow__icon_container
                          .guided-workflow__guided_workflow_icon
                          .guided-workflow__icon_text
                        lib-guided-workflow-linear-workflow
                          .guided-workflow__linear_workflow_view
                            lib-guided-workflow-workflow-step
                              .guided-workflow__workflow_step_card_container
                                .guided-workflow__linear_workflow_step_card
```

```
                                    .guided-workflow__workflow_step_number
                                    .guided-workflow__linear_step_text
                                      .guided-workflow__step_text
                                      .guided-workflow__step_text_insert_icon
                              lib-guided-workflow-workflow-step
                                ...
                              lib-guided-workflow-workflow-step
                                ...
```

Error codes

Many errors are translated from the canonical backend API errors. API errors will also contain additional details surfacing the full backend response.

Client code	Description
Mainly API related errors but also shared with client side errors	
cresta/invalid-argument	Client specified an invalid error argument. Check the error message and details for more information
cresta/unauthenticated	Request not authenticated due to missing, invalid or expired Cresta access tokens.
cresta/resource-exhausted	Either out of resource quota or reaching rate limiting.
cresta/unknown	Unknown server error.
cresta/internal	Internal server error.
cresta/unavailable	Service unavailable, typically due to server being down.
creata/deadline-exceeded	Request deadline exceeded.
cresta/already-exists	The resource being created already exists.
cresta/not-found	A specified resource is not found.
cresta/permission-denied	Client does not have sufficient permission to access a specified resource.
cresta/cancelled	Request cancelled by the client.
cresta/failed-precondition	Request can not be executed in the current system state.
Client related errors	
cresta/network-error	Request failed due to network related connection issues.
cresta/websocket-error	WebSocket streaming error detected.

cresta/shared-worker-unsupported	SharedWorker is not supported in this environment.
cresta/popup-closed	Popup operation failed due to a user closing the popup prematurely.
cresta/popup-blocked	Popup failed to open due to browser blocking popup.
cresta/web-storage-unsupported	Unable to access web storage. This may happen in some browsers in private mode or when a user disabled cookies and site data.
cresta/timeout	Operation failed due to a client side timeout.
cresta/object-destroyed	Running an operation on a destroyed object.
cresta/token-revoked	Thrown when the current Cresta tokens are revoked.
cresta/operation-not-supported	The developer is calling some operation in an environment that is not supported (e.g. web worker, service worker, etc.)

Known Issues

- The sample app and the Cresta client SDK is currently only tested in the latest version of the Chrome browser.
- The "agent" role is always set on the generated Cresta access token for BYOID (Bring Your Own IDentity) authentication via `signInWithExternalCredentials`.
- Voice agent assist on-hold events are currently not detected when polling is enabled (WebSockets are disabled). These are detected when server-sent events are used instead of WebSockets.

Experimental features

The Cresta SDK is designed to have no impact on the main functionality of an application. It is designed with the following principles:

- Minimal footprint on the rest of the application in terms of processing and network bandwidth.
- Fail-safe: If the Cresta server is inaccessible, the SDK will run in no-op mode. For example, smart compose will continue to work but no hints will be presented.
- Self recoverable: If network errors occur while calling Cresta servers, the app goes offline or Cresta tokens are revoked/expired, once the connection is re-established and the user is re-authenticated, the Cresta SDK will recover automatically and suggestions / hints will start streaming again.

The Cresta SDK also offers the ability to proxy all Cresta network operations to a separate dedicated inline web worker thread instead of the main UI thread. This will only proxy Cresta network operations while the rest of the app network operations will continue to run in the main UI thread.

This can be done by configuring the network settings on the `cresta` singleton.

```
import { cresta } from '@cresta/client';

cresta.configure({
  network: {
    // Enable web worker proxy.
    experimentalWorkerProxy: true,
  },
});
```

Disclaimer: This feature is still in **experimental** phase and care should be taken when using it.

SDK Integration Test Cases for Quality Control

This section enumerates the different test cases to go through for quality assurance of agent assist integrations with Cresta using the SDK.

When testing using the Cresta production server, it is highly recommended if possible (developer/integrator has access to Cresta Director) to use Director to inspect the conversation being tested to confirm the expected behavior, e.g. all messages are being received by Cresta and the suggestions and hints shown in Director match with the ones shown on the integration side. More details are provided below.

Agent Test Cases for Chat

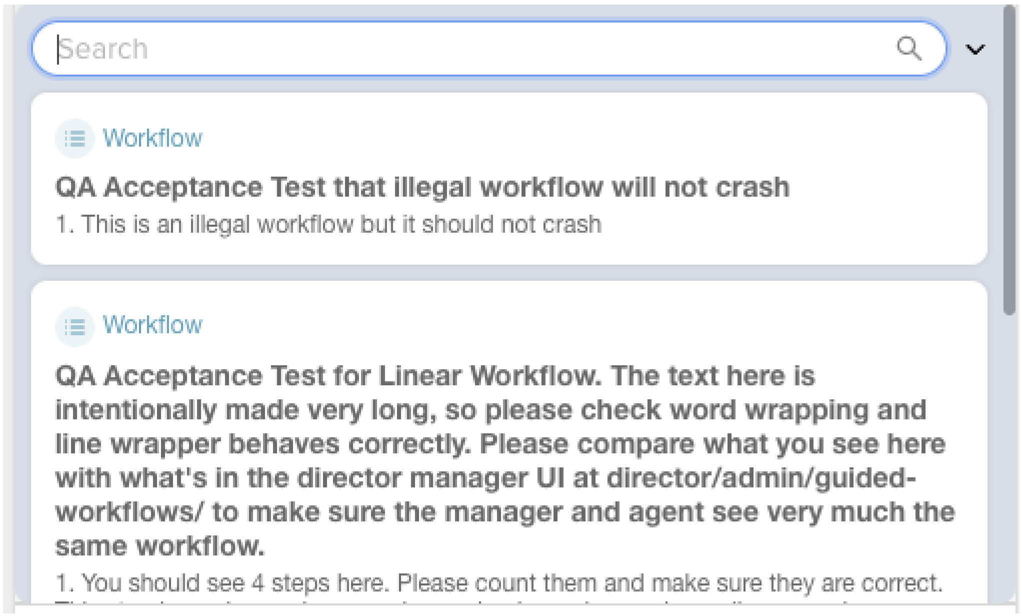
1. Basic
    a. Smart compose, hints, and reply suggestions should be mounted properly for a new chat.
    b. Messages should be visible on Director (including agent, visitor and any bot/system messages)
    c. Director should show the correct chat owner for the agent
2. Suggestions
    a. Hints and reply suggestions should be pulled up automatically.
    b. Close a hint by clicking the “close” button.
    c. Click on a reply suggestion to populate it into the smart compose input box. The smart compose input box send functionality should be enabled automatically

(some input boxes may disable sending if no text is inputted, this check will confirm the suggestion inserted text is detected).

   d. Click the "Send" button (or the equivalent button or type Enter) to send the message.
   e. In Director the message should be decorated with "Suggestions (used)" underneath it. Clicking on that should show the suggestions with the used one highlighted.

3. Smart compose
   a. Type some text into smart compose to trigger a smart compose hint (the grayed text).
   b. Press the right arrow (or the customized shortcut key) to select one word from the hint.
   c. Press tab (or the customized shortcut key) to auto-complete (select the remaining of the hint) with the shown hint.
   d. Hit enter (or click the equivalent send button) to send the message.
4. Switch chat (for platforms that allow an agent to handle multiple chats simultaneously)
   a. Create multiple active chats via multiple visitor sessions.
   b. Type in some different draft messages in each chat, e.g. "chat1", "chat2".
   c. Switch between these chats, observe that
      i. The draft message should be preserved (this is also dependent on the platform and the integrator's preferences. Some platforms may not want to preserve the draft text)
      ii. The corresponding hints/suggestions should be shown instantly without a delay.
   d. Confirm that messages sent in the different chats are sent to the correct conversation (confirm via Director).
5. Transfer chat (for platforms that allow chat transfers from one agent to another). Access to multiple agent accounts is needed for this test.
   a. Log in to the platform with another agent (agent B) where the current agent is agent A.
   b. Transfer a chat from agent A to agent B.
   c. Observe that agent B should automatically accept the chat (this can be platform dependent, if the platform requires the agent to manually accept, have agent B accept the chat) and smart compose, hints, and reply suggestions should be mounted properly for the transferred chat.
   d. Confirm in Director that the chat is marked as assigned to agent B.
6. Close chat
   a. Close/resolve a chat by clicking the equivalent "End chat" button from the agent side.
   b. Observe that Cresta smart compose, hints, suggestions should all unmount, and it restores to the platform's default UI without Cresta components.
   c. Confirm in Director that the chat is moved from Live chats to Closed chats.

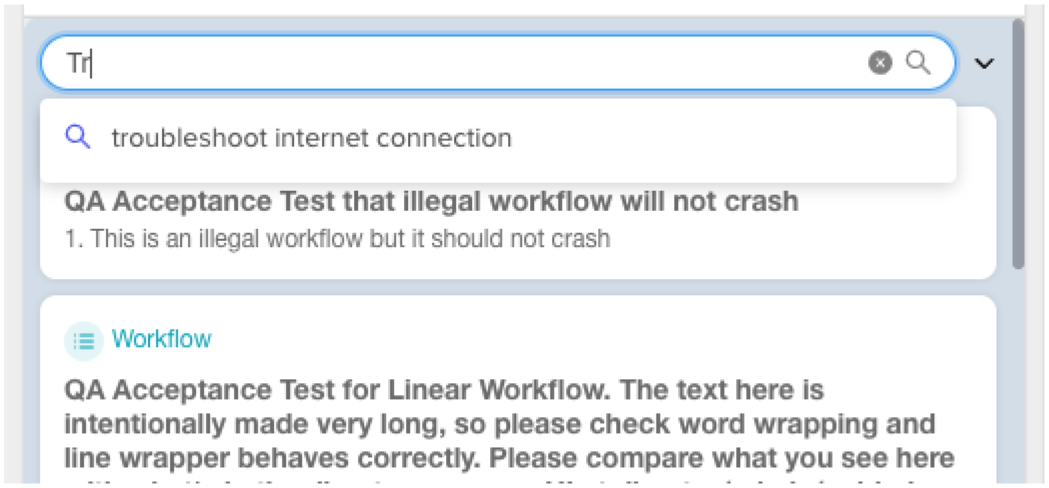
7. Monitor chat (for platforms that allow an agent to monitor or participate in chats not assigned to them). This tests a combination of agents accessing the same chat where one is logged in to Cresta and the other is not.
    a. Create a chat and assign to agent A (not logged in to Cresta)
    b. Login as agent B with Cresta. Join the chat.
    c. Send messages as visitor, agent A and agent B.
    d. Confirm in Director that all messages are recorded (including the ones belonging to agent A not logged in to Cresta).
8. Backfill chat (this helps test when historical messages are sent before an agent logged in to Cresta receives them, that the messages will be sent to Cresta when the agent logs in to Cresta and accesses the same chat)
    a. Access a chat without logging in to the platform. Create a new chat with messages from the visitor and the agent.
    b. Login with Cresta as the same agent this time. Join the previous chat.
    c. Confirm that in Director the chat shows up with the correct assignee and all the messages (including the earlier ones when the agent was not logged in to Cresta).
9. Reopen a closed chat (for platforms that allow closed chats to be reopened)
    a. Reopen a closed chat.
    b. Confirm that smart compose and suggestion is mounted.
10. Omni search (KB search / Guided Workflow / Generative Knowledge Assist)
    a. Click on Omni search entry input —> should open Omni search panel and display "zero search" results with suggested guided workflows. The screenshot will look differently per customer.



QA Acceptance Test that illegal workflow will not crash
1. This is an illegal workflow but it should not crash

QA Acceptance Test for Linear Workflow. The text here is intentionally made very long, so please check word wrapping and line wrapper behaves correctly. Please compare what you see here with what's in the director manager UI at director/admin/guided-workflows/ to make sure the manager and agent see very much the same workflow.
1. You should see 4 steps here. Please count them and make sure they are correct.

    b. Type in search queries (e.g. `tr` for troubleshooting) into the search bar and validate search autocomplete results are fetched/shown. The keyword and result screenshots will differ from one customer to another.


Tr

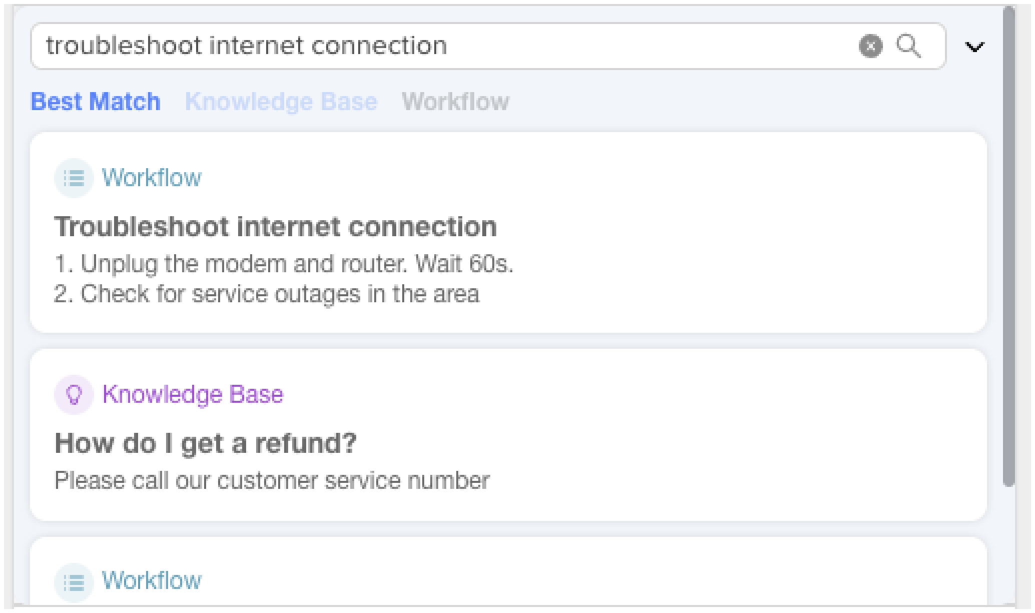
troubleshoot internet connection

QA Acceptance Test that illegal workflow will not crash

1. This is an illegal workflow but it should not crash

QA Acceptance Test for Linear Workflow. The text here is intentionally made very long, so please check word wrapping and line wrapper behaves correctly. Please compare what you see here

c. Click on an autocomplete result and verify that a search query is submitted


troubleshoot internet connection

Troubleshoot internet connection

1. Unplug the modem and router. Wait 60s.
2. Check for service outages in the area

How do I get a refund?

Please call our customer service number

d. Enter a search query and type enter or click the magnifying glass icon in the search bar to verify that a search query is submitted

e. Switch search result tabs (e.g. Best Results → Knowledge Base) and verify that a search query is submitted

f. Open a KB article result (KB result should have a URL for this test) — verify that when clicking the copy button, the URL is inserted to the text input (e.g. smart compose); verify that when clicking the open in new tab button, the article is opened in a new browser tab

g. Open a GWF result - verify that when a step is clicked, that the text is inserted into the text input (e.g. smart compose). For branching workflows, you will see that clicking on the step will insert text into the text input and then display the next step options. Clicking on the next step will advance to the next workflow step but will not insert any text into the text input.
h. Click the ← button on the top left corner to close the GWF result.
i. Switch the tab to `GenAI Answer`.
j. Start a conversation and send at least one message.
k. Type a question and press `Enter` to submit it, should see the generated reply.
l. Send a message that will trigger the question retrieval, you can try to ask the question that's related to the KB/GWF content.
m. When there is an extracted question, you should see a blue bar appear near the top of the Omni Search component. Click the `View` button to auto switch to the `GenAI Answer` tab and show the generated reply.

n. Click the toggle arrow on the top right of the Omni search panel to close/collapse the Omni search component

11. Network timeouts
    a. Simulate slow network connections. Cresta hints and suggestions may be slow to show but the UI should still be smooth and functioning.
    b. Simulate offline mode. Cresta hints and suggestions will stop being served but the rest of the UI should not be affected. Restore the network connection and Cresta should work again (hints and suggestions should start showing). Confirm that all messages sent during network issues are received by Cresta after the network connection is restored by checking the conversation in Director.
12. Cresta Auth session expiration or revocation
    a. In the middle of a conversation, simulate Cresta tokens being expired or revoked by going into *localStorage* and clearing the entry with key *customers/CUSTOMER_ID:creds*.
    b. Confirm that the state change was detected in the integration as a signout operation and that the Cresta sign-in button is shown for the user to sign in again to Cresta (e.g. via popup sign-in).
    c. After sign-in, confirm hints and suggestions are working again.
    d. Confirm the conversation state matches Director conversation state.

Quickstart Sample app (Vanilla JS)

Refer to the sample app README for instructions.

The Cresta quickstart sample provides an easy way to see the Cresta client SDK for chat and voice agent assist services in action.

It provides examples using the different APIs provided by the client SDK. This includes the following Cresta UI components for chat:

- Cresta Chat Smart Compose Standalone Component
- Cresta Chat Smart Compose Overlay Component
- Cresta Behavioral Hints Component
- Cresta Reply Suggestions Component (for reply and canned suggestions)
- Cresta Omni Search Component
    - Knowledge base
    - Guided workflows
    - Generative knowledge assist (beta)

This also includes the following Cresta UI components for voice:

- Cresta Behavioral Hints Component
- Cresta Omni Search Component
    - Knowledge base
    - Guided workflows
    - Generative knowledge assist (beta)

Prerequisites

The following requirements are needed:

- Installation of Node.js version 14.17.6 or greater.
- Installation of npm version 6.14.15 or greater.
- Already set up the project namespace and AI model with the Cresta team.
- Currently, seamless token exchange support is not supported by Cresta. You will need to either configure your SAML IdP with Cresta or create usernames/password in the Cresta Director UI for your real or test users.
- The Cresta client SDK configuration file. This should look like:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com"
}
```

- Optionally, when using the sample app with the Cresta SDK emulator, the following configuration field should be provided:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "http://localhost:8081",
  "emulatorMode": true,
  "emulatorUser": {
    "userId": 1234567890,
    "email": "jane.doe@cresta.ai",
    "displayName": "Jane Doe",
    "userRoles": ["AGENT"],
    "username": "jane.doe"
  }
}
```

Where `serviceEndpoint` should point to the emulator URL and `emulatorMode` should be set to `true` to automate sign-in when `signInWithPopup` is triggered. An optional `emulatorUser` can be provided to customize the profile used by the mocked credentials. Otherwise a fixed static profile is used.

- Adding the authorized origin for your application to the list of authorized origins (contact the Cresta team and provide them with these origins). For security reasons, this is needed for popup sign-in to succeed and return tokens to your app. For the sample app, you will need to add http://localhost:1234 as your authorized origin. This is not needed when using the emulator with `emulatorMode` set to `true`.

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save-dev @cresta/client-sdk-sample
```

Run quickstart app

To launch the sample app:

- Save the configuration file in the root folder of your package under the file name `client-sdk-sample-config.json`.
- To build and run the app, a bin script is provided:

```
npx cresta-sdk-sample
# You can also run it by calling the script directly:
# ./node_modules/.bin/cresta-sdk-sample
```

- Go to `http://localhost:1234/` in your browser and you can start testing with the Cresta quickstart.

Using quickstart app

The quickstart provides 5 tabs:

- Chat tab for all chat specific functionality. This includes starting chats, adding agent and visitor messages, ending and destroying chats, etc.
- Voice tab for all voice specific functionality. This includes detecting active voice sessions and interacting with the related Cresta AI services.

- General tab for non-conversation specific functionality. This includes authentication functionality and setting conversation skills.
- Logs tab to view app logging of events from the main window instead of the browser console.
- Errors tab to view general errors. This currently displays Auth errors related to session expiration or revocation. It also displays Auth network errors on session refresh.

Managing a chat session

To manage a chat session:

- Enter an identifier to uniquely identify the session. Click "Start" to start receiving intelligence data from Cresta. Before starting the chat session, you can also choose to use a `textarea` or a `contenteditable` target element for the smart compose overlay (this can't be changed after the session is started). `contenteditable` is the default.
- Click "End & Destroy" to end the session and stop receiving data.
- Click "Destroy" to just destroy the session without ending it.
- To change the session, click "End & Destroy" or "Destroy" first.

Chat sessions are not persisted in the quickstart sample. To continue the same session after a page reload, make sure to enter the same unique chat session ID previously used and then click "Start". You can also generate a random unique ID by clicking the "UUID" button and then clicking "Start".

After starting a chat, you can simulate a visitor sending a message by entering a message in the visitor message input box and clicking "Submit as visitor". You can also simulate a system message by clicking "Submit as system".

A chat transcript UI is available to display the progression of the chat between the visitor and agent.

You will need to be authenticated via the "Generals" tab in order to receive Cresta intelligence data.

An agent can send a message via the **Cresta Chat Smart Compose (Standalone)** or **Cresta Chat Smart Compose (Overlay)** elements. You can click enter or click "Submit" button to send massages. These are wired to auto-complete with Cresta smart compose hints.

Cresta behavioral hints, reply/canned suggestions and omni search (knowledge base, guided workflow and generative knowledge assist) are also mounted to the UI in the **Cresta Chat Hints, Suggestions and Search** section after the chat is started and the user is authenticated. Clicking the reply or canned suggestions will populate the suggestion in both smart compose elements. Entering text in the omni search component will return related results.

Normally, you would use one of either (and not both):

- Cresta Chat Smart Compose (Standalone) if you don't want to build your own agent input box and just want to use the Cresta provided one.
- Cresta Chat Smart Compose (Overlay) if you have your own content editable with rich text support and other advanced customizations. The overlay would be invisibly inserted on top of your element without affecting its style and behavior.

Managing a voice session

Cresta voice conversations are automatically detected in the sample app `Voice` tab as long as there is an active voice conversation. Voice integrations with Cresta typically require that audio streams and audio conversation events are streamed or sent to Cresta servers via server to server API calls. This is done separately from the frontend integration via the Cresta client SDK. The client side integration is mainly responsible for detecting these conversations and consuming/displaying their related Cresta AI features (e.g. Cresta hints, transcription services, etc.) for the agent to interact with.

To test Cresta voice features in the sample app, start a voice conversation between a visitor (customer) and the logged in Cresta agent. Note this will not be done via the sample app. It will be initiated the standard way a voice conversation is done for the platform in use (e.g. Twilio Flex, Amazon Connect, etc). You can also test voice integrations using the Cresta Emulator. This facilitates the simulation of voice conversations programmatically bypassing the need to pre-configure and hookup the real audio streams to the real Cresta server. Refer to the `@cresta/client-sdk-emulator` documentation or README for more details.
After the voice session is initiated and provided the server to server integration is successful with Cresta, the sample app will detect the conversation in the voice panel and display metadata associated with the session. Cresta behavioral hints and the omni-search component (knowledge base, guided workflow and generative knowledge assist) are mounted to the UI for the active voice session after the session is detected and the user is authenticated.

Voice sessions are persisted in the quickstart sample. An ongoing voice session will be detected and populated in the voice tab after a page reload. This capability comes automatically with the Cresta client SDK. No additional work is needed to persist the session.

You will need to be authenticated via the "Generals" tab in order to receive Cresta intelligence data.

Using BYOID (Bring Your Own ID)

The Cresta SDK and the sample app supports sign in using an OIDC ID token signed using RS256 algorithm. The benefit of using this feature is that it provides a seamless mechanism for identity federation where the Cresta SDK will exchange a platform OIDC ID token with a Cresta access token and take care of authorizing API calls with that token. This can be done under the hood without prompting the agent to sign in again to Cresta.

We provide a sample OIDC server `@cresta/client-sdk-auth-sample` that can be integrated with the sample app to issue ID tokens. However, you can use your own mechanims for generating ID tokens, whether you are using some 3rd party solution (e.g. Okta, Auth0, Firebase Auth, etc.) or building your own OIDC IdP, or token endpoint that verifies session cookies and returns ID tokens, etc.

Testing using the Emulator and the sample OIDC server

To enable testing of this feature in the sample app using the Cresta Emulator and the sample OIDC server. You will need to do the following:

- Start the Cresta Emulator server. This is available in the `@cresta/client-sdk-emulator` package. The server runs by default on `http://localhost:8081`.
- Start the Cresta Auth sample server which is used to issue OIDC ID tokens. This is available in the `@cresta/client-sdk-auth-sample` package. The server runs by default on `http://localhost:5555`.
- Update the `client-sdk-sample-config.json` file in the project root folder:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "authEndpoint": "http://localhost:8081",
  "serviceEndpoint": "http://localhost:8081",
  "clientId": "ai",
  "emulatorMode": true,
  "oidcServerOrigin": "http://localhost:5555",
  "oidcClientId": "cresta"
}
```

  Use the Emulator URL for the **authEndpoint** and **serviceEndpoint**. Use the Sample auth server URL for the **oidcServerOrigin**. For the **client-sdk-sample-config.json** / **oidcClientId**, use the **clientId** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package. For the **client-sdk-sample-config.json** / **clientId**, use the **audience** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package.

- Start the sample app by running `npx cresta-sdk-sample`. This will launch the sample app at http://localhost:1234.
- In the **General** tab under **Cresta BYOID**, enter a username and password that corresponds to the clientId/audience selected from **data/data.json** file in `@cresta/client-sdk-auth-sample` package. Click the **Sign in** button.
- This will sign in via the OIDC provider and then sign in to Cresta. By default, the Cresta SDK will use a **SharedWorker** to refresh tokens and share them across tabs of the

same app. This is the case unless the "Use SharedWorker" checkbox is unchecked. In that case, the credentials are not persisted.
- To sign out from the OIDC provider and Cresta, click the **Sign out** button in the **General** tab under **Cresta BYOID** section.

Testing using Cresta production and the sample OIDC server

To enable testing of this feature in the sample app using the sample OIDC server and the Cresta production service endpoint. You will need to do the following:

- Start the Cresta Auth sample server which is used to issue OIDC ID tokens. This is available in the `@cresta/client-sdk-auth-sample` package. The server runs by default on `http://localhost:5555` but you will need to host the server so it is accessible to Cresta which needs to retrieve the public key so it can verify the token signature. You can use ngrok to facilitate that.

  Run: `ngrok http 5555`
  This will output an HTTPS ngrok URL. This will be used to access the Auth server.

```
https://4ae6-2601-646-xyz.ngrok.io -> http://localhost:5555
```

- Update the **client-sdk-sample-config.json** file in the root project folder:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com",
  // This doesn't need to be specified unless you are using a proxy.
  // "authEndpoint": "https://auth.cresta.com",
  // This should match the ID token audience which should match the Cresta
  // client ID. This should match the audience in data/data.json
  // in `@cresta/client-sdk-auth-sample` package.
  "clientId": "profile-chat-demo",
  // The OIDC server origin. This should match the ngrok URL.
  "oidcServerOrigin": "https://4ae6-2601-646-xyz.ngrok.io",
  // The OIDC client ID. This can be retrieved or modified in data/data.json
  // in `@cresta/client-sdk-auth-sample` package.
  // This can used to construct the issuer URL:
  // https://4ae6-2601-646-xyz.ngrok.io/cresta
  "oidcClientId": "cresta"
}
```

- Use the Sample auth server URL (ngrok URL proxy) for the **oidcServerOrigin**. For the **client-sdk-sample-config.json** / **oidcClientId**, use the **clientId** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package. For the

client-sdk-sample-config.json / **clientId**, use the **audience** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package.

- You will also need to register the issuer URL of the OIDC server and the ID token audience with Cresta since you will calling production endpoints.
  For the above example:
  - Issuer URL: https://4ae6-2601-646-xyz.ngrok.io/cresta
  - Audience: **profile-chat-demo**

- Start the sample app by running `npx cresta-sdk-sample`. This will launch the sample app at http://localhost:1234.
- In the **General** tab under **Cresta BYOID**, enter a username and password that corresponds to the clientId/audience selected from **data/data.json** file in `@cresta/client-sdk-auth-sample` package. Click the **Sign in** button.
- This will sign in via the OIDC provider and then sign in to Cresta. By default, the Cresta SDK will use a **SharedWorker** to refresh tokens and share them across tabs of the same app. This is the case unless the "Use SharedWorker" checkbox is unchecked. In that case, the credentials are not persisted.
- To sign out from the OIDC provider and Cresta, click the **Sign out** button in the **General** tab under **Cresta BYOID** section.

Using reCAPTCHA with inline username and password sign-in

You will need to add the 2 site keys to the configuration JSON file (reach out to the Cresta team to determine these values):

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com",
  "recaptchaSiteKey": "...",
  "recaptchaCheckboxSiteKey": "..."
}
```

When these 2 values are provided, a switch is displayed next to the username and password to toggle between score-based and interactive checkbox reCAPTCHA.

The sample app demonstrates how to implement both flows and toggle between them but in production, a score-based reCAPTCHA should always be used first and on high risk error, an interactive checkbox reCAPTCHA should be used to recover.

Check the `@cresta/client` documentation on how to implement sign-in with reCAPTCHA.

Quickstart Sample app (React)

The Cresta quickstart sample provides an easy way to see the Cresta react client SDK for chat and voice agent assist services in action.

It provides examples using the different APIs provided by the react client SDK. This includes the following Cresta UI components for chat:

- Cresta Chat Smart Compose Standalone Component
- Cresta Chat Smart Compose Overlay Component
- Cresta Behavioral Hints Component
- Cresta Reply Suggestions Component (for reply and canned suggestions)
- Cresta Omni Search Component
    - Knowledge base
    - Guided workflows
    - Generative knowledge assist (beta)

This also includes the following Cresta UI components for voice:

- Cresta Behavioral Hints Component
- Cresta Omni Search Component
    - Knowledge base
    - Guided workflows
    - Generative knowledge assist (beta)

This project was bootstrapped with Create React App.

Prerequisites

The following requirements are needed:

- Installation of Node.js version 14.17.6 or greater.
- Installation of npm version 6.14.15 or greater.
- Installation of react version `16.8` or greater.
- Already set up the project namespace and AI model with the Cresta team.
- Currently, seamless token exchange support is not supported by Cresta. You will need to either configure your SAML IdP with Cresta or create usernames/password in the Cresta Director UI for your real or test users.
- The Cresta client SDK configuration file. This should look like:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com"
}
```

- Optionally, when using the sample app with the Cresta SDK emulator, the following configuration field should be provided:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "http://localhost:8081",
  "emulatorMode": true,
  "emulatorUser": {
    "userId": 1234567890,
    "email": "jane.doe@cresta.ai",
    "displayName": "Jane Doe",
    "userRoles": ["AGENT"],
    "username": "jane.doe"
  }
}
```

  Where `serviceEndpoint` should point to the emulator URL and `emulatorMode` should be set to `true` to automate sign-in when `signInWithPopup` is triggered. An optional `emulatorUser` can be provided to customize the profile used by the mocked credentials. Otherwise a fixed static profile is used.

- Adding the authorized origin for your application to the list of authorized origins (contact the Cresta team and provide them with these origins). For security reasons, this is needed for popup sign-in to succeed and return tokens to your app. For the sample app, you will need to add http://localhost:1234 as your authorized origin.This is not needed when using the emulator with `emulatorMode` set to `true`.

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save-dev @cresta/client-sdk-react-sample
```

Run quickstart app

To launch the sample app:

- Save the configuration file in the root folder of your package under the file name `client-sdk-sample-config.json`.
- To build and run the app, a bin script is provided:

```
npx cresta-sdk-react-sample
# You can also run it by calling the script directly:
# ./node_modules/.bin/cresta-sdk-react-sample
```

- Go to `http://localhost:1234/` in your browser and you can start testing with the Cresta quickstart.

Using quickstart app

The quickstart provides 5 tabs:

- Chat tab for all chat specific functionality. This includes starting chats, adding agent and visitor messages, ending and destroying chats, etc.
- Voice tab for all voice specific functionality. This includes detecting active voice sessions and interacting with the related Cresta AI services.
- General tab for non-conversation specific functionality. This includes authentication functionality and setting conversation skills.
- Logs tab to view app logging of events from the main window instead of the browser console.
- Errors tab to view general errors. This currently displays Auth errors related to session expiration or revocation. It also displays Auth network errors on session refresh.

Managing a chat session

To manage a chat session:

- Enter an identifier to uniquely identify the session. Click "Start" to start receiving intelligence data from Cresta. Before starting the chat session, you can also choose to use a `textarea` or a `contenteditable` target element for the smart compose overlay (this can't be changed after the session is started). `contenteditable` is the default.
- Click "End & Destroy" to end the session and stop receiving data.
- Click "Destroy" to just destroy the session without ending it.
- To change the session, click "End & Destroy" or "Destroy" first.

Chat sessions are not persisted in the quickstart sample. To continue the same session after a page reload, make sure to enter the same unique chat session ID previously used and then click "Start". You can also generate a random unique ID by clicking the "UUID" button and then clicking "Start".

After starting a chat, you can simulate a visitor sending a message by entering a message in the visitor message input box and clicking "Submit as visitor". You can also simulate a system message by clicking "Submit as system".

A chat transcript UI is available to display the progression of the chat between the visitor and agent.

You will need to be authenticated via the "Generals" tab in order to receive Cresta intelligence data.

An agent can send a message via the **Cresta Chat Smart Compose (Standalone)** or **Cresta Chat Smart Compose (Overlay)** elements. You can click enter or click "Submit" button to send massages. These are wired to auto-complete with Cresta smart compose hints.

Cresta behavioral hints, reply/canned suggestions and omni search (knowledge base, guided workflow and generative knowledge assist) are also mounted to the UI in the **Cresta Chat Hints, Suggestions and Search** section after the chat is started and the user is authenticated. Clicking the reply or canned suggestions will populate the suggestion in both smart compose elements. Entering text in the omni search component will return related results.

Normally, you would use one of either (and not both):

- Cresta Chat Smart Compose (Standalone) if you don't want to build your own agent input box and just want to use the Cresta provided one.
- Cresta Chat Smart Compose (Overlay) if you have your own content editable with rich text support and other advanced customizations. The overlay would be invisibly inserted on top of your element without affecting its style and behavior.

Managing a voice session

Cresta voice conversations are automatically detected in the sample app `Voice` tab as long as there is an active voice conversation. Voice integrations with Cresta typically require that audio streams and audio conversation events are streamed or sent to Cresta servers via server to server API calls. This is done separately from the frontend integration via the Cresta client SDK. The client side integration is mainly responsible for detecting these conversations and consuming/displaying their related Cresta AI features (e.g. Cresta hints, transcription services, etc.) for the agent to interact with.

To test Cresta voice features in the sample app, start a voice conversation between a visitor (customer) and the logged in Cresta agent. Note this will not be done via the sample app. It will be initiated the standard way a voice conversation is done for the platform in use (e.g. Twilio Flex, Amazon Connect, etc). You can also test voice integrations using the Cresta Emulator. This facilitates the simulation of voice conversations programmatically bypassing the need to pre-configure and hookup the real audio streams to the real Cresta server. Refer to the `@cresta/client-sdk-emulator` documentation or README for more details.
After the voice session is initiated and provided the server to server integration is successful with Cresta, the sample app will detect the conversation in the voice panel and display metadata associated with the session. Cresta behavioral hints and the omni-search component (knowledge base, guided workflow and generative knowledge assist) are mounted to the UI for the active voice session after the session is detected and the user is authenticated.

Voice sessions are persisted in the quickstart sample. An ongoing voice session will be detected and populated in the voice tab after a page reload. This capability comes automatically with the Cresta client SDK. No additional work is needed to persist the session.

You will need to be authenticated via the "Generals" tab in order to receive Cresta intelligence data.

Using BYOID (Bring Your Own ID)

The Cresta SDK and the sample app supports sign in using an OIDC ID token signed using RS256 algorithm. The benefit of using this feature is that it provides a seamless mechanism for identity federation where the Cresta SDK will exchange a platform OIDC ID token with a Cresta access token and take care of authorizing API calls with that token. This can be done under the hood without prompting the agent to sign in again to Cresta.

We provide a sample OIDC server `@cresta/client-sdk-auth-sample` that can be integrated with the sample app to issue ID tokens. However, you can use your own mechanims for generating ID tokens, whether you are using some 3rd party solution (e.g. Okta, Auth0, Firebase Auth, etc.) or building your own OIDC IdP, or token endpoint that verifies session cookies and returns ID tokens, etc.

Testing using the Emulator and the sample OIDC server

To enable testing of this feature in the sample app using the Cresta Emulator and the sample OIDC server. You will need to do the following:

- Start the Cresta Emulator server. This is available in the `@cresta/client-sdk-emulator` package. The server runs by default on `http://localhost:8081`.
- Start the Cresta Auth sample server which is used to issue OIDC ID tokens. This is available in the `@cresta/client-sdk-auth-sample` package. The server runs by default on http://localhost:5555.
- Update the `client-sdk-sample-config.json` file in the root project folder:

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "authEndpoint": "http://localhost:8081",
  "serviceEndpoint": "http://localhost:8081",
  "clientId": "ai",
  "emulatorMode": true,
  "oidcServerOrigin": "http://localhost:5555",
  "oidcClientId": "cresta"
}
```

  Use the Emulator URL for the **authEndpoint** and **serviceEndpoint**. Use the Sample auth server URL for the **oidcServerOrigin**. For the **client-sdk-sample-config.json / oidcClientId**, use the **clientId** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package. For the **client-sdk-sample-config.json /**

clientId, use the **audience** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package.

- Start the sample app by running `npx cresta-sdk-react-sample`. This will launch the sample app at http://localhost:1234.
- In the **General** tab under **Cresta BYOID**, enter a username and password that corresponds to the clientId/audience selected from **data/data.json** file in `@cresta/client-sdk-auth-sample` package. Click the **Sign in** button.
- This will sign in via the OIDC provider and then sign in to Cresta. By default, the Cresta SDK will use a **SharedWorker** to refresh tokens and share them across tabs of the same app. This is the case unless the "Use SharedWorker" checkbox is unchecked. In that case, the credentials are not persisted.
- To sign out from the OIDC provider and Cresta, click the **Sign out** button in the **General** tab under **Cresta BYOID** section.

Testing using Cresta production and the sample OIDC server

To enable testing of this feature in the sample app using the sample OIDC server and the Cresta production service endpoint. You will need to do the following:

- Start the Cresta Auth sample server which is used to issue OIDC ID tokens. This is available in the `@cresta/client-sdk-auth-sample` package. The server runs by default on `http://localhost:5555` but you will need to host the server so it is accessible to Cresta which needs to retrieve the public key so it can verify the token signature. You can use ngrok to facilitate that.

  Run: `ngrok http 5555`
  This will output an HTTPS ngrok URL. This will be used to access the Auth server.
  ```
  https://4ae6-2601-646-xyz.ngrok.io -> http://localhost:5555
  ```

- Update the **client-sdk-sample-config.json** file in the root project folder:
  ```
  {
    "customerId": "CUSTOMER_ID",
    "profileId": "PROFILE_ID",
    "customerOrigin": "https://CUSTOMER_ID.cresta.com",
    "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com",
    // This doesn't need to be specified unless you are using a proxy.
    // "authEndpoint": "https://auth.cresta.com",
    // This should match the ID token audience which should match the Cresta
    // client ID. This should match the audience in data/data.json
    // in `@cresta/client-sdk-auth-sample` package.
    "clientId": "profile-chat-demo",
    // The OIDC server origin. This should match the ngrok URL.
    "oidcServerOrigin": "https://4ae6-2601-646-xyz.ngrok.io",
  ```

```
  // The OIDC client ID. This can be retrieved or modified in data/data.json
  // in `@cresta/client-sdk-auth-sample` package.
  // This can used to construct the issuer URL:
  // https://4ae6-2601-646-xyz.ngrok.io/cresta
  "oidcClientId": "cresta"
}
```

- Use the Sample auth server URL (ngrok URL proxy) for the **oidcServerOrigin**. For the **client-sdk-sample-config.json** / **oidcClientId**, use the **clientId** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package. For the **client-sdk-sample-config.json** / **clientId**, use the **audience** from the **data/data.json** file in `@cresta/client-sdk-auth-sample` package.

- You will also need to register the issuer URL of the OIDC server and the ID token audience with Cresta since you will calling production endpoints.
  For the above example:
  - Issuer URL: https://4ae6-2601-646-xyz.ngrok.io/cresta
  - Audience: **profile-chat-demo**

- Start the sample app by running `npx cresta-sdk-react-sample`. This will launch the sample app at http://localhost:1234.
- In the **General** tab under **Cresta BYOID**, enter a username and password that corresponds to the clientId/audience selected from **data/data.json** file in `@cresta/client-sdk-auth-sample` package. Click the **Sign in** button.
- This will sign in via the OIDC provider and then sign in to Cresta. By default, the Cresta SDK will use a **SharedWorker** to refresh tokens and share them across tabs of the same app. This is the case unless the "Use SharedWorker" checkbox is unchecked. In that case, the credentials are not persisted.
- To sign out from the OIDC provider and Cresta, click the **Sign out** button in the **General** tab under **Cresta BYOID** section.

Using reCAPTCHA with inline username and password sign-in

You will need to add the 2 site keys to the configuration JSON file (reach out to the Cresta team to determine these values):

```
{
  "customerId": "CUSTOMER_ID",
  "profileId": "PROFILE_ID",
  "customerOrigin": "https://CUSTOMER_ID.cresta.com",
  "serviceEndpoint": "https://api-CUSTOMER_ID.cresta.com",
  "recaptchaSiteKey": "...",
  "recaptchaCheckboxSiteKey": "..."
}
```

When these 2 values are provided, a switch is displayed next to the username and password to toggle between score-based and interactive checkbox reCAPTCHA.

The sample app demonstrates how to implement both flows and toggle between them but in production, a score-based reCAPTCHA should always be used first and on high risk error, an interactive checkbox reCAPTCHA should be used to recover.

Check the `@cresta/client` documentation on how to implement sign-in with reCAPTCHA.

Cresta SDK Emulator(@cresta/client-sdk-emulator)

The Cresta SDK emulator is a mock lightweight server that emulates the behavior of the Cresta backend server. The server can run locally and the application integrating with the client SDK would be pointing to the emulator server instead of the real Cresta server.

The server emulates the conversation APIs as well as all authentication endpoints and serves mock AI suggestions and hints.

The Cresta SDK emulator is used in conjunction with the Cresta client SDK foragent assist integrations (chat and audio).

It can provide the following benefits:

- Facilitates SDK development without having to rely on the AI model being ready. As the AI model training can take some time before it is rolled out, the emulator can allow immediate testing.
- Facilitates integration tests in an environment where outgoing network requests are disallowed or can be flaky.
- Facilitates testing voice agent assist integrations by simulating voice conversations programmatically without connecting to the real Cresta servers. This allows for development and testing without having to wait for voice conversation audio streams and events to be configured with the Cresta production servers.
- Similarly, development or staging environments may be behind firewalls and the emulator provides a quick workaround.
- Facilitates authentication in automated tests where sign-in using popup can be mocked. This makes development possible without even setting a test account or provisioning a real account.
- Allows testing for custom scenarios. As the AI model can be unpredictable, the emulator provides the ability to provide custom suggestions and hints which may be useful for certain edge case testing scenarios.
- Allows testing and development without having to pollute the database and Director UI with bad data.
- Allows testing for knowledge base search UI integration before the underlying data is ingested and made accessible.

- Allows testing for guided workflow UI integration before the underlying workflows are configured and made accessible.
- Allows testing for canned suggestions integration before the namespace is ready or the suggestions are uploaded.
- Allows testing for Generative Knowledge Assist currently in beta phase (retrieval of conversation related extracted questions and their AI generated answers).

Since v0.2.0, the emulator emulates the Cresta Client Subscription service which uses a WebSocket server instead of a polling API to serve chat suggestions. This should be paired with v0.3.0 or higher of the `@cresta/client` package. When used with an older version of the SDK, the emulator is still able to support the legacy polling API.

The emulator allows you to test and develop without the real AI model, customer ID, test account or IdP (SAML) setup for authentication, authorized origin registration, etc.

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save-dev @cresta/client-sdk-emulator@latest
```

Usage

To run the Cresta Emulator server, a bin script is provided:

```
# Start server.
npx cresta-emulator
# You can also run it by calling the script directly:
# ./node_modules/.bin/cresta-emulator
```

This will launch using the default port 8081 and the built-in mock data.

To use a different port number:

```
npx cresta-emulator --port=5000
```

Built-in mock data are already shipped with the emulator code but you can also specify your own mock data file:

```
npx cresta-emulator --data=/path/to/mock/ai/data.json
```

The server will listen to file changes and update the suggestions, hints, and retrieval questions to match those changes in real-time.

In addition, you can also update the search results and retrieval questions via the `knowledgeBase`, `guidedWorkflow`, `cannedSuggestions` and `retrievalQuestions` array entries. Knowledge base and guided workflow APIs have been supported in the emulator since `v0.8.0`.
Canned suggestions APIs have been supported in the emulator since `v0.9.0`. Built-in mock results are also shipped with the emulator package.
Generative Knowledge Assist has been supported in the emulator since `v0.14.0`. Built-in mock results are also shipped with the emulator package.

Mock data format (since v0.2.0):

```
{
  "behavioralSuggestions": [
    {
      "title": "Suggestion title 1",
      "text": "Suggestion text 1",
      "coachMadeSuggestionText": "Coach-made suggestion 1"
    },
    {
      "title": "Suggestion title 2",
      "text": "Suggestion text 2"
    },
    {
      "title": "Suggestion title 3",
      "text": "Suggestion text 3",
      "coachMadeSuggestionText": "Coach-made suggestion 3"
    },
    {
      "title": "Suggestion title 4",
      "text": "Suggestion text 4"
    }
  ],
  "replySuggestions": [
    {
      "text": "Suggestion text 1"
    },
    {
      "text": "Suggestion text 2"
    }
  ],
  "smartComposeHints": [
    {
      "text": "Full text to match. This will have the user draft text as prefix."
    },
    {
      "text": "Another full text to match. This will have the user draft text as
```

```
prefix."
    }
  ],
  "knowledgeBase": [
    {
      "title": "Article title 1",
      "text": "Explanation and information for article 1.",
      "id": "Unique no-space ID for article 1, e.g. Isthereacancellationfee",
      "url": "URL for article 1, e.g. https://example.com/article1",
      "keywords": [
        "keyword1-article1",
        "keyword2-article1",
        "keyword3-article1"
      ]
    },
    {
      "title": "Article title 2",
      "text": "Explanation and information for article 2.",
      "id": "Unique no-space ID for article 2, e.g. Waiveactivationfee",
      "url": "URL for article 2, e.g. https://example.com/article2",
      "keywords": [
        "keyword1-article2",
        "keyword2-article2",
        "keyword3-article2"
      ]
    }
  ],
  "guidedWorkflow": [
    {
      "title": "Update payment method",
      "id": "Updatepaymentmethod",
      "keywords": ["update", "payment", "method"],
      "linearSteps": [
        {
          "text": "Log into account",
          "type": "STEP_TYPE_SUGGESTION"
        },
        {
          "text": "Click on account settings",
          "type": "STEP_TYPE_SUGGESTION"
        },
        {
          "text": "Enter your new credit card information and click save",
          "type": "STEP_TYPE_SUGGESTION"
        },
        {
          "text": "Click 'update payment'",
          "type": "STEP_TYPE_SUGGESTION"
        },
        {
          "text": "Suggest the user enroll in monthly auto-payment",
          "type": "STEP_TYPE_DESCRIPTION"
        }
      ]
    },
    {
      "title": "How can I cancel my trial?",
      "id": "Howcanicancelmytrial",
      "keywords": ["cancel", "trial", "renew", "subscription"],
      "branchingSteps": [
        {
          "text": "What type of subscription do you have?",
          "type": "STEP_TYPE_SUGGESTION",
          "nextSteps": [
            {
              "stepId": 2,
              "conditionText": "Regular subscription"
            },
            {
              "stepId": 3,
              "conditionText": "Premium subscription"
            }
          ]
        },
        {
          "text": "Go to Account > Settings > Subscriptions. Then click 'Do not
renew subscription'",
          "type": "STEP_TYPE_SUGGESTION",
          "nextSteps": [
            {
              "stepId": 4,
              "conditionText": "Next"
            }
          ]
        },
        {
          "text": "Go to Account > Settings > Subscriptions. Then click 'Downgrade
account'",
          "type": "STEP_TYPE_SUGGESTION",
          "nextSteps": [
            {
              "stepId": 2,
              "conditionText": "Cancel the downgraded regular subscription account"
            }
          ]
        },
        {
          "text": "Confirm cancellation in the dialog and you should get an email
confirmation.'",
          "type": "STEP_TYPE_SUGGESTION"
        }
      ]
    }
  ],
  "retrievalQuestions": [
    {
      "question": "How do I return a defective product",
      "keywords": ["return", "defective"],
      "knowledgeSessionId": "returndefectiveproduce"
    },
    {
      "question": "How do I cancel my subscription",
      "keywords": ["cancel", "subscription"],
      "knowledgeSessionId": "cancelsubscription"
    },
    {
      "question": "Is there a cancellation fee",
      "keywords": ["cancellation", "fee"],
      "knowledgeSessionId": "cancellationfee"
    },
    {
      "question": "Can you waive the upgrade fee",
      "keywords": ["waive", "upgrade fee"],
      "knowledgeSessionId": "waiveupgradefee",
      "duration": "60s"
    },
    {
      "question": "What is the return policy",
      "keywords": ["return", "policy"],
      "knowledgeSessionId": "returnpolicy",
      "duration": "100.001s"
    },
    {
      "question": "How to add another person to my account",
      "keywords": ["add", "person", "account"],
      "knowledgeSessionId": "addpersontoaccount",
      "duration": "200s"
    }
  ],  "cannedSuggestions": [
    {
      "title": "General Greeting",
      "text": "Hey! I'm $!{operator.nickname}! I'm excited that you chose to chat
with us at ACME. May I have your name to get you started?",
      "id": "generalgreeting",
      "keywords": ["hey", "excited", "general", "greeting", "started", "name"]
    },
    {
      "title": "Greeting with Specific Question or Issue",
      "text": "Hi! This is $!{operator.nickname} with ACME. I'm happy to assist you
with ________. May I have your name, please?",
      "id": "greetingwithquestion",
      "keywords": ["greeting", "question", "issue", "assist", "name", "hi"]
    },
    {
      "title": "Canceling Premium Subscription",
      "text": "Canceling your Premium subscription is easy, just log in to your
ACME account through the ACME website to manage your subscription. Look for the
Settings icon in the top left corner and click 'Manage Subscription'.",
      "id": "cancelpremiumsubscription",
      "keywords": ["cancel", "premium", "subscription"]
    },
    {
      "title": "How do I redeem a Loyal Member offer",
      "text": "Just log in to your ACME account through the ACME website to manage
your subscription. Look for the Settings icon in the top left corner and click
'Offers'. Then click 'Loyal Member'.",
      "id": "redeemloyalmemberoffer",
      "keywords": ["redeem", "loyal", "member", "offer"]
    }
  ]
}
```

For a linear guided workflow:

```
/** Interface representing a linear guided workflow. */
export interface LinearGuidedWorkflow {
  /** The overall workflow title. */
  title: string;
  /** The workflow id (no spaces). */
  id: string;
  /** Linear steps. These will be sequentially displayed. */
  linearSteps: Array<{
    /** The text describing the workflow step. */
    text: string;
    /**
     * The type of step. STEP_TYPE_SUGGESTION can be inserted into smart
     * compose as a reply suggestion. STEP_TYPE_DESCRIPTION is just a
     * description/hint to the agent.
     */
    type: 'STEP_TYPE_SUGGESTION' | 'STEP_TYPE_DESCRIPTION';
  }>;
  /** List of related keywords. */
  keywords: string[];
}
```

For a branching guided workflow:

```
/** Interface representing a branching guided workflow. */
export interface BranchingGuidedWorkflow {
  /** The workflow title. */
  title: string;
  /** The workflow id (no spaces). */
  id: string;
  /** Branching steps. These will be sequentially displayed. */
  branchingSteps: Array<{
    /** Optional title for the current branching step. */
    title?: string;
    /** The text describing the workflow step. */
    text: string;
    /**
     * The type of step. STEP_TYPE_SUGGESTION can be inserted into smart
     * compose as a reply suggestion. STEP_TYPE_DESCRIPTION is just a
     * description/hint to the agent.
     */
    type: 'STEP_TYPE_SUGGESTION' | 'STEP_TYPE_DESCRIPTION';
    /** The optional next steps. */
    nextSteps?: Array<{
      /**
       * The ID of the next step in the branchingSteps list indexed starting
       * at 1. So if current index is 1 (1st step) and this value is 3, then
       * the next step is the 3rd entry in the `branchingSteps` array.
       * The next button that will lead to this next step will have the text
       * `conditionText`.
       * You can have multiple next steps from the current step. This is useful
       * when you have multiple options for the next step.
       */
      stepId: number;
      /** The text describing the button to go to the next step. */
      conditionText: string;
    }>;
  }>;
  /** List of related keywords. */
  keywords: string[];
}
```

Mock data format (deprecated) predating v0.2.0:

```
{
  "behavioralSuggestions": [
    {
      "title": "Suggestion title 1",
      "text": "Suggestion text 1"
    },
    {
      "title": "Suggestion title 2",
      "text": "Suggestion text 2"
    }
  ],
  "replySuggestions": [
    {
      "text": "Suggestion text 1"
    },
    {
      "text": "Suggestion text 2"
    }
  ],
  "coachMadeSuggestions": [
    {
      "title": "Suggestion title 1",
      "text": "Suggestion text 1"
    },
    {
      "title": "Suggestion title 2",
      "text": "Suggestion text 2"
    }
  ],
  "smartComposeHints": [
    {
      "text": "Full text to match. This will have the user draft text as prefix."
    },
    {
      "text": "Another full text to match. This will have the user draft text as
prefix."
    }
  ]
}
```

To see the command line manual:

```
npx cresta-emulator --help
```

Testing voice integrations

For voice agent assist integrations, the emulator provides a mechanism to facilitate testing integrations by simulating voice conversations programmatically without connecting to the real Cresta servers.

This allows for development and testing without having to wait for voice conversation audio streams and events to be configured with the Cresta production servers.

In order to do this, the emulator needs to be started with additional arguments configuring an API key so it can be used to programmatically call the conversation maker APIs. Note the API key can be any arbitrary string value with no spaces, commas or colons.

```
# You can also just use one API key if one customer ID is used:
# npx cresta-emulator --api-keys=API_KEY:customer-id1
npx cresta-emulator --api-keys=API_KEY:customer-id1,API_KEY_2:customer-id2
```

Where:

- `API_KEY` is configured as an API key for customer ID `customer-id1` (the mock customer ID used for testing).
- `API_KEY_2` is configured as an API key for customer ID `customer-id2` (the mock customer ID used for testing).

On the client side, the same customer ID should be used and the service and auth endpoints have to be configured to connect to the emulator server.

```
const config = {
  customerId: 'customer-id1',
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This should use the emulator server URL.
  serviceEndpoint: 'http://localhost:8081',
  // The authEndpoint should also use the emulator server URL.
  // This endpoint is currently only used with signInWithExternalCredentials.
  authEndpoint: 'http://localhost:8081',
  // Set to true so users can be automatically authenticated with mock
  // credentials on sign-in (e.g. via signInWithPopup).
  emulatorMode: true
};

const client = cresta.createClient(config);

// ...
```

After launching the emulator, you can then start voice conversations, send messages and close the conversations programmatically.

Using cURL commands:

- `$CUSTOMER_ID`: The customer ID, e.g. `customer-id1`. This should match the one used in the application by the SDK and the API key used to start the server.
- `$PROFILE_ID`: The profile ID, e.g. `voice-profile`. This should match the one used in the application by the SDK, e.g.
- `$CONVERSATION_ID`: a unique identifier string for the conversation. A UUID can be used.

- `$USER_ID`: The user ID for the logged in user. `963dc54c593ec4b9` is the default user ID used by emulator generated access tokens. When using real production access tokens, get this value from the access token JWT payload in the `payload.user.name` nested object.
- `$MESSAGE_ID`: a unique identifier string for every conversation message. A UUID can be used. A new one should be used for every message.
- `$TEXT`: The plain text message uttered by the visitor, agent or bot.
- `$SPEAKER_TYPE`: The message speaker type: `VISITOR`, `AGENT` or `SYSTEM_MESSAGE`.
- `$CREATE_TIME`: The message creation time. This should be in ISO format, e.g. `2024-01-03T19:54:57.215Z`.

Start a conversation with API key `API_KEY`, customer ID `customer-id1`, profile ID `voice-profile` and conversation ID `5afe29c9-c6f9-48ab-8292-236912c9d40a`:

```
# API:
curl
'http://localhost:8081/v1/customers/$CUSTOMER_ID/profiles/$PROFILE_ID/conversations/$CONVERS
ATION_ID:start' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"agent":"customers/$CUSTOMER_ID/users/$USER_ID","platformInfo":{"platformConversationId":"
$CONVERSATION_ID"},"channel":"VOICE"}' \
  --compressed

# Example:
curl
'http://localhost:8081/v1/customers/customer-id1/profiles/voice-profile/conversations/5afe29
c9-c6f9-48ab-8292-236912c9d40a:start' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"agent":"customers/customer-id1/users/963dc54c593ec4b9","platformInfo":{"platformConversat
ionId":"5afe29c9-c6f9-48ab-8292-236912c9d40a"},"channel":"VOICE"}' \
  --compressed
```

Where `963dc54c593ec4b9` is the default user ID used by emulator generated access tokens. When using real production access tokens, get this value from the access token JWT payload in the `payload.user.name` nested object.

After a voice conversation is started, the conversation will be detected in the SDK in the open state.

Send a message with text `hello` (speaker is the visitor) for that conversation with message ID `f8028c27-ca19-4392-9498-abeab593085c`. A new unique ID should be generated for each message:

```
# API:
curl
'http://localhost:8081/v1/customers/$CUSTOMER_ID/profiles/$PROFILE_ID/conversations/$CONVERS
ATION_ID/messages/$MESSAGE_ID' \
  -X 'PATCH' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"name":"customers/$CUSTOMER_ID/profiles/$PROFILE_ID/conversations/$CONVERSATION_ID/message
s/$MESSAGE_ID","speaker":"$SPEAKER_TYPE","state":"COMPLETE","createTime":"$CREATE_TIME","tex
t":"$TEXT","transcriptionInfo":{"transcriptionWords":[{"word":"something","confidence":0.1,"
startOffset":"1s","endOffset":"2s"}]}}' \
  --compressed

# Example:
curl
'http://localhost:8081/v1/customers/customer-id1/profiles/voice-profile/conversations/5afe29
c9-c6f9-48ab-8292-236912c9d40a/messages/f8028c27-ca19-4392-9498-abeab593085c' \
  -X 'PATCH' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"name":"customers/customer-id1/profiles/voice-profile/conversations/5afe29c9-c6f9-48ab-829
2-236912c9d40a/messages/f8028c27-ca19-4392-9498-abeab593085c","speaker":"VISITOR","state":"C
OMPLETE","createTime":"2024-01-03T19:54:57.215Z","text":"hello","transcriptionInfo":{"transc
riptionWords":[{"word":"something","confidence":0.1,"startOffset":"1s","endOffset":"2s"}]}}'
\
  --compressed
```

Where the speaker can be `VISITOR`, `AGENT` or `SYSTEM_MESSAGE` depending if it is uttered by a visitor, agent or bot.

After the message is sent, mock AI data may be generated for that conversation by the emulator. For example a new hint may be shown for that conversation after the message is received by the emulator.

Close the conversation:

```
# API:
curl
'http://localhost:8081/v1/customers/$CUSTOMER_ID/profiles/$PROFILE_ID/conversations/$CONVERS
ATION_ID:close' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"name":"customers/$CUSTOMER_ID/profiles/$PROFILE_ID/conversations/$CONVERSATION_ID"}' \
  --compressed

# Example:
curl
```

```
'http://localhost:8081/v1/customers/customer-id1/profiles/voice-profile/conversations/5afe29
c9-c6f9-48ab-8292-236912c9d40a:close' \
  -H 'authorization: ApiKey API_KEY' \
  -H "Content-Type: application/json" \
  --data-raw
'{"name":"customers/customer-id1/profiles/voice-profile/conversations/5afe29c9-c6f9-48ab-829
2-236912c9d40a"}' \
  --compressed
```

After a voice conversation is closed, the status update for that conversation will be detected by the SDK.

Compliance tests for build your own components

The Cresta emulator provides a mode to run diagnostics to analyze whether a custom-built Smart Compose, Reply Suggestions or Hints components are compliant with Cresta best practices and requirements. A compliant implementation is critical to ensure you always get the best AI recommendations from Cresta. To take advantage of this mode, start the emulator with the `--run-compliance-tests` flag.

```
npx cresta-emulator --run-compliance-tests
```

Start the emulator server and run compliance tests and print out results on completion after the server is stopped (CTRL + C).
This is useful when building your own Cresta UI components to validate the implementation is compliant with Cresta best practices.

- Start the emulator with this flag.
- Point the application Cresta `serviceEndpoint` to the emulator.
- Then simulate conversations being started and users interacting with the UI components. Detailed instructions on what actions to trigger in the app will be printed out in the terminal where the emulator is running. Detailed instructions will be printed in the terminal on what actions to trigger.
- After running enough simulations, stop the emulator by sending the `SIGINT` process signal (CTRL + C) and a compliance analysis of the implementation will be printed out in the terminal.

Sample output report:

```
──────────────────────────── Running compliance tests ────────────────────────────
Generating report for Smart Compose...
- Out of 4 generated hints, no events were triggered to
  indicate the hint candidate was shown.
  "candidate.show()" needs to be called on a hint candidate
  when it is appended to DOM.
```

```
- No partially selected (word by word) smart compose hint
  was detected.
  "candidate.insert(false)" needs to be called when a hint
  candidate is partially selected, e.g. right arrow or a
  custom key is clicked and the next word from the hint is
  inserted.
- No fully selected smart compose hint was detected.
  "candidate.insert(true)" needs to be called when the hint
  candidate is fully selected, e.g. tab or a custom key is
  clicked and the full hint is inserted or the user types
  the full hint, etc.

Generating report for Reply Suggestions...
- Out of 3 generated suggestions, no events were triggered
  to indicate any reply suggestions were shown.
  "suggestion.show()" needs to be called on a reply
  suggestion when it is appended to the DOM.
- No selected reply suggestion was detected.
  "suggestion.select()" needs to be called when the reply
  suggestion is selected via click, etc.
- No submitted reply suggestion was detected.
  There could be an issue in the population of the
  suggestion in Smart Compose. To detect incoming
  suggestions, use "chat1.events.smartCompose.onText()" or
  "chat1.events.smartCompose.text$" to detect the incoming
  suggestion text.

Generating report for Canned Suggestions...
- No selected canned suggestion was detected.
  "suggestion.select()" needs to be called when the canned
  suggestion is selected via click, etc.
- No submitted canned suggestion was detected.
  There could be an issue in the population of the
  suggestion in Smart Compose. To detect incoming
  suggestions, use "chat1.events.smartCompose.onText()" or
  "chat1.events.smartCompose.text$" to detect the incoming
  suggestion text.

Generating report for Hints...
- Out of 2 generated hints, no events were triggered to
  indicate any hints were shown.
  "hint.show()" needs to be called on a hint when it is
  appended to the DOM.
- No expanded hint description was detected.
  "hint.expand()" needs to be called when a hint description
  is expanded.
- No dismissed hint was detected.
  "hint.dismiss()" needs to be called when a hint is
  dismissed via click or after the "hint.durationMs" expires.
─────────────────────────────────────────────────────────────────
────────────
```

Client SDK changes

You will then need to update the Client SDK initialization config to point to the local emulator server (http://localhost:8081):

```
const config = {
  customerId: 'CUSTOMER_ID',
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This should use the emulator server URL.
  serviceEndpoint: 'http://localhost:8081',
  // The authEndpoint should also use the emulator server URL.
  // This endpoint is currently only used with signInWithExternalCredentials.
  authEndpoint: 'http://localhost:8081',
  // Set to true so users can be automatically authenticated with mock
  // credentials on sign-in (e.g. via signInWithPopup).
  emulatorMode: true
};

const client = cresta.createClient(config);

// ...
```

The following 2 configuration fields need to be updated:

- **serviceEndpoint**: This would need to point to the emulator server URL.
- **emulatorMode**: This optional field (defaults to false) should be set to true to automatically sign in users without opening a popup and prompting for credentials or redirecting to customer IdP.

You can also specify the profile of the mock user:

```
const config = {
  customerId: 'CUSTOMER_ID',
  customerOrigin: 'https://CUSTOMER_ID.cresta.com',
  // This should use the emulator server URL.
  serviceEndpoint: 'http://localhost:8081',
  // The authEndpoint should also use the emulator server URL.
  // This endpoint is currently only used with signInWithExternalCredentials.
  authEndpoint: 'http://localhost:8081',
  // Set to true so users can be automatically authenticated with mock
  // credentials on sign-in (e.g. via signInWithPopup).
  emulatorMode: true,
  // Observed only when emulatorMode is true.
  emulatorUser: {
    userId: 987654321,
    email: 'jane.smith@cresta.ai',
    displayName: 'Jane Smith',
    userRoles: ['AGENT', 'MANAGER'],
    username: 'jane.smith',
  }
};

const client = cresta.createClient(config);

// ...
```

An optional field **emulatorUser** is provided to allow the ability to set the profile of the mock signed in user. This may be useful when testing for specific users. When not specified, the client SDK will use a predefined profile.

The supported fields:

- **userId**: The numeric user ID.
- **email**: The user email address.
- **displayName**: The user display name.
- **userRoles**: The list of roles assigned to the user.
- **username**: The username.

You can toggle these configuration fields depending on whether their application is running in production or development mode. For example, when using webpack to bundle an application, environment variables can be used to toggle between these modes.

You can add a script to your `package.json` to run the emulator while running your sample app:

```
{
  "script": {
    "start": "NODE_ENV=prod parcel static/index.html",
    "start-dev": "NODE_ENV=test concurrently \"parcel static/index.html\"
\"cresta-emulator --port=5000\""
  },
  "devDependencies": {
    "@cresta/client-sdk-emulator": "latest",
    "concurrently": "7.2.2",
    ....
  }
}
```

Cresta Auth Server Sample (@cresta/client-auth-server-sample)

Note: if you already have a mechanism to generate OIDC ID tokens, you can skip this section.

In order to integrate with Cresta services, an agent has to be authenticated with Cresta. This enables API calls to the Cresta backend to be authenticated (via Bearer access tokens).

Cresta SDK supports authentication via popup. However, as agents have to also be separately authenticated to the platform, prompting the agent to sign in again to Cresta may not be convenient and may add unnecessary friction. As a result, Cresta SDK offers a seamless mechanism for identity federation based on the OAuth 2.0 Token Exchange rfc8693 standard, where the Cresta SDK will exchange a platform OIDC ID token for a Cresta access token and take care of authorizing API calls with that token.

The Cresta Auth Server sample provides a sample server to generate OIDC spec compliant ID tokens using RS256 signing algorithm, with publicly hosted public keys, that can be used for sign-in with the Cresta SDK.

The tokens are generated on sign-in with test username and password accounts defined in a JSON file.

The server is mostly stateless (apart from some logic to store revoked refresh tokens). It is provided as a sample code / guideline for generating spec-compliant OIDC ID tokens and is not meant to be used as is in production. However, this server can be substituted with other identity services (Okta, Auth0, Firebase Auth, etc) that provide a mechanism to generate short-lived ID tokens and long-lived refresh tokens.

When testing with Cresta, the Auth server needs to be hosted publicly as the corresponding public keys used to verify the minted tokens need to be accessible to the Cresta server.

The sample Auth server will only store tokens and keys in memory and will generate new keys on restart.

To avoid hosting the server during development, you can go around that and use a service like ngrok which makes it easy and straightforward to expose your server running locally on a specific port to the internet.

Cresta Setup

The following settings need to be registered with Cresta so it can verify the issued tokens are coming from the expected issuer and targeting the expected audience.

- The issuer URL for the minted OIDC ID tokens: Cresta will use this to verify that a token with this issuer belongs to the specified Cresta customer account.
- The ID token audience (The Cresta client ID): Cresta will verify the intended audience for the token. If this doesn't match the one configured in the Cresta customer account. The token will be rejected.

When using the client SDK, the Cresta client ID (the ID token audience) needs to be provided on Cresta client initialization:

```
const client = cresta.createClient({
  customerId: "CUSTOMER_ID",
  // This is used for the authentication URL.
  customerOrigin: "https://CUSTOMER_ID.cresta.com",
  // This is the Cresta service endpoint. Replace with proxy origin if using
  // proxy.
  serviceEndpoint: "https://api-CUSTOMER_ID.cresta.com",
  // The client ID. This is the ID token audience.
  clientId: "foobar",
});
```

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save-dev @cresta/client-auth-server-sample
```

Usage

To run the OIDC sample server, a bin script is provided:

```
npx cresta-auth-server-sample
# You can also run it by calling the script directly:
# ./node_modules/.bin/cresta-auth-server-sample
```

This will launch using the default port 5555 and the built-in clients/accounts data.

To use a different port number:

```
npx cresta-auth-server-sample --port=5000
```

Built-in client/account data are already shipped with the server code but you can also specify your own data file:

```
npx cresta-auth-server-sample --data=/path/to/clients/accounts/data.json
```

By default the server will issue one hour long ID tokens. To customize the duration:

```
# Use 30min long tokens.
npx cresta-auth-server-sample --id-token-duration=1800
```

The server will validate requests and credentials against this file.

Data format:

```
{
  "clients": [
    {
      "clientId": "$CLIENT_ID",
      "audience": "$AUDIENCE",
      "accounts": [
        {
          "uid": "1234567890",
          "email": "john.smith@example.com",
          "emailVerified": true,
          "username": "john-smith",
          "roles": ["admin"],
          "displayName": "John Smith",
          "password": "super-secret-password123"
        },
        {
          "uid": "9876543210",
          "email": "jane.doe@example.com",
          "emailVerified": true,
          "username": "jane-doe",
          "roles": ["agent"],
          "displayName": "Jane Doe",
          "password": "super-secret-456-pass"
        }
      ]
    }
  ]
}
```

This will provide the auth server with the list of test accounts and the client ID and audience to use when minting tokens.

To see the command line manual:

```
npx cresta-auth-server-sample --help
```

Client SDK changes

You can add a script to your `package.json` to run the auth server while running your app:

```
{
  "script": {
    "start": "NODE_ENV=prod parcel static/index.html",
    "start-dev": "NODE_ENV=test concurrently \"parcel static/index.html\"
\"cresta-auth-server-sample --port=5000\""
  },
  "devDependencies": {
    "@cresta/client-auth-server-sample": "latest",
    "concurrently": "7.2.2",
    ....
  }
}
```

In order to use the Auth server to sign in with Cresta, an `ExternalCredentials` interface needs to be implemented and passed to the Cresta SDK.

Note that the code below is applicable to other sources of OIDC ID tokens. Only the logic for refreshing and obtaining new ID tokens would need to be updated.

```
import { AuthToken, ExternalCredentials } from "@cresta/client";

/** Plain object representation of an OidcCredentials. */
interface JsonOidcCredentials {
  oidcServerOrigin?: string;
  clientId?: string;
  idToken?: string;
  refreshToken?: string;
  expirationTime?: number;
}

/** OIDC auth and token refresh endpoint JSON response format. */
interface TokenRefreshResponse {
  id_token: string;
  refresh_token: string;
  expires_in: number;
}

/** One second in milliseconds. */
const ONE_SEC_MS = 1000;
/** HTTP status success code. */
const HTTP_STATUS_SUCCESS = 200;
/** Used to generate a random number for the nonce. */
const NONCE_RANDOM_NUM_RADIX = 36;
// This will trim the first 2 characters '0.' from the randomly generated
// string.
const NONCE_RANDOM_NUM_STARTING_INDEX = 2;

/**
```

```
 * OidcCredentials implements the ExternalCredentials expected by the Cresta
 * client SDK using the Cresta Auth server sample.
 */
export class OidcCredentials
  extends EventTarget
  implements ExternalCredentials
{
  readonly #sessionId: string;
  #pendingTokenPromise: Promise<void> | null = null;

  /**
   * Initializes the OidcCredentials.
   * @param oidcServerOrigin - The origin of the OIDC provider.
   * @param clientId - The client ID to be used by the OIDC provider.
   * @param idToken - The initial OIDC ID token.
   * @param refreshToken - The initial OIDC refresh token.
   * @param expirationTime - The initial OIDC ID token expiration time in number
   *     of milliseconds since Epoch time.
   */
  constructor(
    private readonly oidcServerOrigin: string,
    private readonly clientId: string,
    public idToken: string,
    public refreshToken: string,
    public expirationTime: number
  ) {
    super();

    const components = idToken.split(".");
    const payload = atob(components[1]!.replace(/_/g, "/").replace(/-/g, "+"));

    const claims = JSON.parse(payload) as {
      sub: string;
    };

    this.#sessionId = claims.sub;
  }

  /**
   * The session ID is used by the Cresta client SDK to determine whether to
   * reuse cached credentials (if the session ID didn't change from last page
   * reload).
   */
  get sessionId() {
    return this.#sessionId;
  }
```

```
/**
 * Refresh logic is dependent on the underlying mechanism for getting the
 * ID token.
 */
async #refresh(): Promise<void> {
  const data = new URLSearchParams();

  data.append("refresh_token", this.refreshToken);
  data.append("grant_type", "refresh_token");
  data.append(
    "nonce",
    Math.random()
      .toString(NONCE_RANDOM_NUM_RADIX)
      .slice(NONCE_RANDOM_NUM_STARTING_INDEX)
  );
  const response = await fetch(
    `${this.oidcServerOrigin}/v1/clients/${this.clientId}:token`,
    {
      method: "POST",
      mode: "cors",
      cache: "no-cache",
      headers: {
        "Content-Type": "application/x-www-form-urlencoded",
      },
      body: data.toString(),
    }
  );

  const jsonResponse = await response.json();

  if (response.status === HTTP_STATUS_SUCCESS) {
    const {
      // eslint-disable-next-line @typescript-eslint/naming-convention
      id_token,
      // eslint-disable-next-line @typescript-eslint/naming-convention
      refresh_token,
      // eslint-disable-next-line @typescript-eslint/naming-convention
      expires_in,
    } = jsonResponse as TokenRefreshResponse;

    this.idToken = id_token;
    this.refreshToken = refresh_token;
    this.expirationTime = new Date().getTime() + expires_in * ONE_SEC_MS;

    this.dispatchEvent(new CustomEvent("refresh"));
  } else {
    const message = jsonResponse.error_description;
    throw new Error(message);
  }
}

/**
 * Whether the current ID token is expired or not.
 */
#isExpired(): boolean {
  return this.expirationTime <= new Date().getTime();
}

/**
 * Returns the requested OIDC ID token.
 * @param forceRefresh Whether to force refresh of the tokens regardless
 *     if they are expired or not.
 * @returns A promise that resolves with the requested ID token in AuthToken
 *     format.
 */
async getToken(forceRefresh: boolean): Promise<AuthToken> {
  if (this.#isExpired() || forceRefresh) {
    // Store the pending token request promise. This guards against multiple
    // refresh requests being processed in parallel using the same refresh
    // token. This may cause failure as the refresh token may be one-time use.
    if (!this.#pendingTokenPromise) {
      this.#pendingTokenPromise = this.#refresh()
        .then(() => {
          this.#pendingTokenPromise = null;
        })
        .catch((error) => {
          this.#pendingTokenPromise = null;
          throw error;
        });
    }

    // Return the pending token request promise.
    await this.#pendingTokenPromise;
  }

  return {
    token: this.idToken,
    tokenType: "urn:ietf:params:oauth:token-type:id_token",
    expirationTimeMs: this.expirationTime,
  };
}

/** Revokes the current credentials. */
```

```
async revoke() {
  const data = new URLSearchParams();

  data.append("token", this.refreshToken);
  data.append("token_type_hint", "refresh_token");
  const response = await fetch(
    `${this.oidcServerOrigin}/v1/clients/${this.clientId}:revoke`,
    {
      method: "POST",
      mode: "cors",
      cache: "no-cache",
      headers: {
        "Content-Type": "application/x-www-form-urlencoded",
      },
      body: data.toString(),
    }
  );

  if (response.status !== HTTP_STATUS_SUCCESS) {
    const message = (await response.json()).error_description;

    throw new Error(message);
  }

  this.dispatchEvent(new CustomEvent("revoke"));
}

/**
 * Used to serialize the credentials.
 * @returns The plain object representation of the credentials.
 */
toJSON() {
  return {
    oidcServerOrigin: this.oidcServerOrigin,
    clientId: this.clientId,
    idToken: this.idToken,
    refreshToken: this.refreshToken,
    expirationTime: this.expirationTime,
  };
}

/**
 * Used to deserialize the credentials.
 * @param json - The potential plain object representation of the credentials.
 * @returns The corresponding OidcCredentials if the provided object is a
 *     serialized OidcCredentials, null otherwise.
 */
```

```
  static fromJSON(json: JsonOidcCredentials): OidcCredentials | null {
    if (
      json.oidcServerOrigin &&
      json.clientId &&
      json.idToken &&
      json.refreshToken &&
      json.expirationTime
    ) {
      return new OidcCredentials(
        json.oidcServerOrigin,
        json.clientId,
        json.idToken,
        json.refreshToken,
        json.expirationTime
      );
    }

    return null;
  }
}
```

In order to generate the credentials above, the user has to sign in with a username and password to the Auth server.

```
/**
 * Signs in using the username and password and returns the corresponding
 * OidcCredentials object.
 * @param oidcServerOrigin - The origin of the OIDC provider.
 * @param clientId - The client ID to be used by the OIDC provider.
 * @param username - The username to sign in with.
 * @param password - The password to sign in with.
 * @returns A promise that resolves with the OidcCredentials corresponding to
 *     the username/password credentials provided.
 */
async function signInWithUsernameAndPassword(
  oidcServerOrigin: string,
  clientId: string,
  username: string,
  password: string
): Promise<OidcCredentials> {
  const response = await fetch(
    `${oidcServerOrigin}/v1/clients/${clientId}:auth`,
    {
      method: "POST",
      mode: "cors",
      cache: "no-cache",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        username,
        password,
        nonce: Math.random()
          .toString(NONCE_RANDOM_NUM_RADIX)
          .slice(NONCE_RANDOM_NUM_STARTING_INDEX),
      }),
    }
  );

  const jsonResponse = await response.json();

  if (response.status === HTTP_STATUS_SUCCESS) {
    const {
      // eslint-disable-next-line @typescript-eslint/naming-convention
      id_token,
      // eslint-disable-next-line @typescript-eslint/naming-convention
      refresh_token,
      // eslint-disable-next-line @typescript-eslint/naming-convention
      expires_in,
    } = jsonResponse as TokenRefreshResponse;
    const idToken = id_token;
    const refreshToken = refresh_token;
    const expirationTime = new Date().getTime() + expires_in * ONE_SEC_MS;

    return new OidcCredentials(
      oidcServerOrigin,
      clientId,
      idToken,
      refreshToken,
      expirationTime
    );
  }

  const message = jsonResponse.error_description;

  throw new Error(message);
}
```

The external credentials can then be passed to the Cresta client SDK. This should be always done on page load if tokens are available. Cresta SDK will cache the result on page navigation and will only make an API call to refresh when needed (cached credentials are expired or the credentials session ID changes).

```
async function signInWithCresta(client: CrestaClient, creds: OidcCredentials) {
  await client.auth.signInWithExternalCredentials(creds);
}
```

However, the above will not persist Cresta credentials across tabs/windows or on page reload. To do so, you will need to host a SharedWorker and listen to `cresta#getToken` events and respond to them with fresh OIDC tokens.

```
let pendingTokenRequestPromise: Promise<AuthToken> | null = null;

(self as unknown as SharedWorkerGlobalScope).addEventListener(
  "connect",
  (e) => {
    const port = e.ports[0];

    port?.addEventListener("message", async (event) => {
      // This event is required to be handled in the SharedWorker. This
      // message will be sent by the Cresta SDK to get a valid unexpired
      // OIDC ID token.
      if (event.data.type === "cresta#getToken") {
        const { forceRefresh, eventId, audience } = event.data;

        try {
          // Cache pending request for tokens to avoid multiple token requests
          // running at the same time in parallel.
          if (!pendingTokenRequestPromise) {
            pendingTokenRequestPromise = getTokens(forceRefresh);
          }

          const token = await pendingTokenRequestPromise;

          pendingTokenRequestPromise = null;
          // Return the updated tokens to the caller. Pass the audience and
          // eventId back too.
          port.postMessage({
            audience,
            type: "cresta#getToken",
            status: "success",
            token,
            eventId,
          });
        } catch (error) {
          pendingTokenRequestPromise = null;
          // Notify caller of the token error. Pass the audience and eventId
          // back too.
          port.postMessage({
            audience,
            eventId,
            type: "cresta#getToken",
            status: "error",
            error: error.message,
          });
        }
      }
    });

    // Start sending message queued on the port.
    port?.start();
  }
);
```

You would then need to pass the `SharedWorker` info to the Cresta SDK afterthe user signs in with the OIDC provider. This only needs to be done once and the Cresta SDK will cache this information and persist it across tabs and page reloads:

When signing out

```
async function signOut(client: CrestaClient, creds: OidcCredentials) {
  // Make sure to revoke the OIDC credentials.
  await creds.revoke();
  // Sign out from Cresta. This will also ensure Cresta stops trying to get
  // new tokens from the provided ExternalCredentials.
  await client.auth.signOut();
}
```

REST APIs

The Cresta Auth server sample supports the following APIs and endpoints:

API to authenticate with the Auth server using a username and password. An optional `nonce` parameter can also be provided. This will be added to the returned ID token.

```
curl -X POST http://localhost:5555/v1/clients/example:auth \
    -H 'Content-Type: application/json' \
    -d '{"username":"john-smith","password":"super-secret-password123"}'
```

Where **example** is a test client ID.

The response format:

```
{
  "id_token": "eyJhbGciOiJ...",
  "refresh_token": "ef04853ed...",
  "expires_in": 3600
}
```

API to refresh the ID token before expiration.
An optional `nonce` parameter can also be provided. This will be added to the returned ID token.

```
curl -X POST http://localhost:5555/v1/clients/example:token \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -d 'grant_type=refresh_token&refresh_token=97e65e3...&nonce=123456'
```

Where **example** is a test client ID.
The response format:

```
{
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZC...",
  "refresh_token": "ef04853edfd16feac5...",
  "expires_in": 3600
}
```

API to revoke a refresh token, typically called on sign out.

```
curl -X POST http://localhost:5555/v1/clients/example:revoke \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    -d 'token=97e65e3da6a32f486527...&token_type_hint=refresh_token'
```

Where **example** is a test client ID.
API to get the OpenID configuration file.

```
curl http://localhost:5555/example/.well-known/openid-configuration
```

Where **example** is a test client ID.
The response format:

```
{
  "issuer": "http://localhost:5555/example",
  "jwks_uri": "http://localhost:5555/example/.well-known/jwks.json",
  "response_types_supported": ["id_token"],
  "subject_types_supported": ["public"],
  "id_token_signing_alg_values_supported": ["RS256"]
}
```

API to get the ID token public keys in JWKS format.

```
curl http://localhost:5555/example/.well-known/jwks.json
```

Where **example** is a test client ID.
The response format:

```
{
  "keys": [
    {
      "kty": "RSA",
      "n": "61MmB3MA-S4iY9zndfP7cLX9nCw5VOe1BW...",
      "e": "AQAB",
      "alg": "RS256",
      "use": "sig",
      "kid": "Wyp9aCQqLmSiMLTaiWkSRvXAQjP9qD"
    }
  ]
}
```

Using with ngrok

To avoid hosting the server during development, you can go around that and use a service like ngrok which makes it easy and straightforward to expose your server running locally on a specific port to the internet.

```
ngrok http 5555
```

This will output an HTTPS ngrok URL. This will be used to access the Auth server.

```
https://4ae6-2601-646-8e81-dd0-f438-1bcd-ebd4-4567.ngrok.io ->
http://localhost:5555
```

You can then use this origin when calling the server:

```
curl -X POST
https://4ae6-2601-646-8e81-dd0-f438-1bcd-ebd4-4567.ngrok.io/v1/clients/example:auth
\
    -H 'Content-Type: application/json' \
    -d '{"username":"john-smith","password":"super-secret-password123"}'
```

Cresta External Utilties (@cresta/client-sdk-utils)

This package provides customer-related utilities that can be used with the Cresta SDK. This includes support for HMAC client authentication that may be used when calling a proxy server to ensure data integrity (messages not tampered with) and to verify the authenticity of the request (coming from the expected sender).

Installation

Follow the step in the Before you begin.
Install the package:

```
npm install --save @cresta/client-sdk-utils
```

API Details

```
/**
 * Client Authentication Using HMAC.
 *
 * Example of the authorization header:
 * MAC ID="gfFb4K8esqZgMpzwF9SXzKLCCbPYV8bR", ts="1463772177193",
 * nonce="61129a8d-ca24-464b-8891-9251501d86f0",
 * bodyhash="YJpz6NdGP0aV6aYaa+6qKCjQt46of+Cj4liBz90G6X8=",
 * mac="uzybzLPj3fD8eBZaBzb4E7pZs+l+IWS0w/w2wwsExdo="
 *
 * Where:
 *
 * - MAC ID: is the HmacSecret key ID.
 * - ts: ts: The timestamp the hash is generated. The format is Unix Epoch time
 *   (ms).
 * - nonce: Unique identifier string. The value of nonce must be unique for each
 *   request.
 * - bodyhash: Hash of the message body using the HMAC Sha256 algorithm.
 * - mac: The mac is generated using the HMAC Sha256 algorithm. The HMAC
 *   secret is used in the generation of the hash/signature. The base string
 *   used to create the mac consists of multiple lines of data. Each line is
 *   delimited by a new line \n character.
 *   Timestamp + \n + nonce + \n + httpmethod + \n +
 *   path + \n + host + \n + port + \n + bodyhash + \n
 *
 * @param request - The HTTP request.
 * @param origin - The request origin (this includes the port number too if
 *     explicitly specified), e.g. http://www.example.com:8080
 *     This should not include a trailing slash for the path.
 * @param hmacSecret - The HMAC secret and its key ID.
 * @returns The generated signature.
 */
```

```
export declare function generateHmacHeader1(request: HttpRequest, origin: string,
hmacSecret: HmacSecret): string;

/** HMAC secret and its key ID. */
export declare interface HmacSecret {
    readonly secret: string;
    readonly keyId: string;
}

/**
 * Interface representing the intercepted serialized API request.
 */
export declare interface HttpRequest {
    /** The optional HTTP request body. */
    readonly body?: string;
    /** The HTTP request headers. */
    readonly headers: Record<string, string>;
    /** The HTTP request method. The default is GET. */
    readonly method?: 'GET' | 'POST' | 'DELETE' | 'PATCH' | 'HEAD' | 'PUT';
    /** The HTTP request URL path. */
    readonly urlPath: string;
}

export { }
```

Client Authentication Using HMAC

The package provides an implementation of client authentication using HMAC.

Example of the authorization header:

```
MAC ID="gfFb4K8esqZgMpzwF9SXzKLCCbPYV8bR", ts="1463772177193",
nonce="61129a8d-ca24-464b-8891-9251501d86f0",
bodyhash="YJpz6NdGP0aV6aYaa+6qKCjQt46of+Cj4liBz90G6X8=",
mac="uzybzLPj3fD8eBZaBzb4E7pZs+l+IWS0w/w2wwsExdo="
```

Where:

- MAC ID: is the HmacSecret key ID.
- ts: The timestamp the hash is generated. The format is Unix Epoch time (ms).
- nonce: Unique identifier string. The value of nonce must be unique for each request.
- bodyhash: Hash of the message body using the HMAC Sha256 algorithm.
- mac: The mac is generated using the HMAC Sha256 algorithm. The HMAC secret is used in the generation of the hash/signature. The base string used to create the mac consists of multiple lines of data. Each line is delimited by a new line `\n` character.

```
Timestamp + \n + nonce + \n + httpmethod + \n + path + \n + host + \n
+ port + \n + bodyhash + \n
```

Usage

Add the `@cresta/client-sdk-utils` package in the `package.json` file.

To compute the HMAC client authentication signature:

```
import { generateHmacHeader1 } from "@cresta/client-sdk-utils";

// The HTTP request whose HMAC authentication header is to be generated.
const request = {
  body: JSON.stringify({ foo: "bar" }),
  headers: { Authorization: "Bearer ACCESS_TOKEN" },
  method: "POST",
  urlPath: "/v1/chats/api",
};
const origin = "https://api-CUSTOMER_ID.cresta.com";
// The HMAC secret and the corresponding key ID.
const hmacSecret = {
  secret: process.env.SECRET,
  keyId: process.env.KEY_ID,
};
// Generate the signature.
const signature = generateHmacHeader1(request, origin, hmacSecret);
```

The Cresta SDK offers HTTP request interceptors that will expose hooks to modify HTTP requests to the Cresta server. The above utilities can be used to generate the HMAC auth header and then inject it in the HTTP request header.

```
import { generateHmacHeader1 } from "@cresta/client-sdk-utils";
import { cresta } from "@cresta/client";

// The HMAC secret and the corresponding key ID.
const hmacSecret = {
  secret: process.env["SECRET"] as string,
  keyId: process.env["KEY_ID"] as string,
};
// Additional headers to include (The API key value and its header name).
const apiKey = process.env["API_KEY"] as string;
const apiKeyHeaderName = process.env["API_KEY_NAME"] as string;

// Initialize the Cresta client with the CrestaConfig.
cresta.createClient({
  customerId: "CUSTOMER_ID",
  customerOrigin: "https://CUSTOMER_ID.cresta.com",
  // The service endpoint. This would use the proxy origin.
  serviceEndpoint: "https://api.dev.proxy.com",
});

// Configure the HTTP request interceptor.
cresta.configure({
  network: {
    // The callback is triggered right before the HTTP request is processed.
    httpRequestInterceptors: [
      async (request) => {
        // Find the authz header key.
        // eslint-disable-next-line no-restricted-syntax
        for (const key in request.headers) {
          if (key.toLowerCase() === "authorization") {
            // Set Cresta AuthZ header in a new field.
            request.headers["Cresta-Authorization"] = request.headers[
              key
            ] as string;
            // Delete the authz header since it will be overwritten with the
            // HMAC header.
            delete request.headers[key];
          }
        }

        // Replace Authorization header with the HMAC auth header:
        request.headers["Authorization"] = generateHmacHeader1(
          request,
          // Proxy origin.
          cresta.getClient()!.config.serviceEndpoint,
          hmacSecret
        );

        // Include additional headers.
        request.headers[apiKeyHeaderName] = apiKey;

        // Return the modified HTTP request.
        return {
          ...request,
          // Modify the URL path to match the proxy's expected one.
          // The request origin should use the one defined in the CrestaConfig,
          // e.g. serviceEndpoint https://api.dev.proxy.com. Modifying that
          // should be done in the CrestaConfig.
          urlPath: request.urlPath.replace(
            "/v1/",
            "/servicing/v1/cresta_outbound/"
          ),
        };
      },
    ],
  },
});
```