aws

Developer Guide

Agent Workspace

Agent Workspace: Developer Guide

Table of Contents

What is the Connect Customer agent workspace?

Connect Customer agent workspace is a single, intuitive application that provides your agents with all of the tools and step-by-step guidance they need to resolve issues efficiently, improve customer experiences, and onboard faster. Contact center agents might be required to use more than seven applications to manage each customer interaction, digging through various tools to process simple requests, and frustrating customers on hold. Connect Customer agent workspace integrates all of your agent tools on one screen. You can customize the agent workspace to present agents with step-by-step guidance to resolve customer issues faster.

Topics

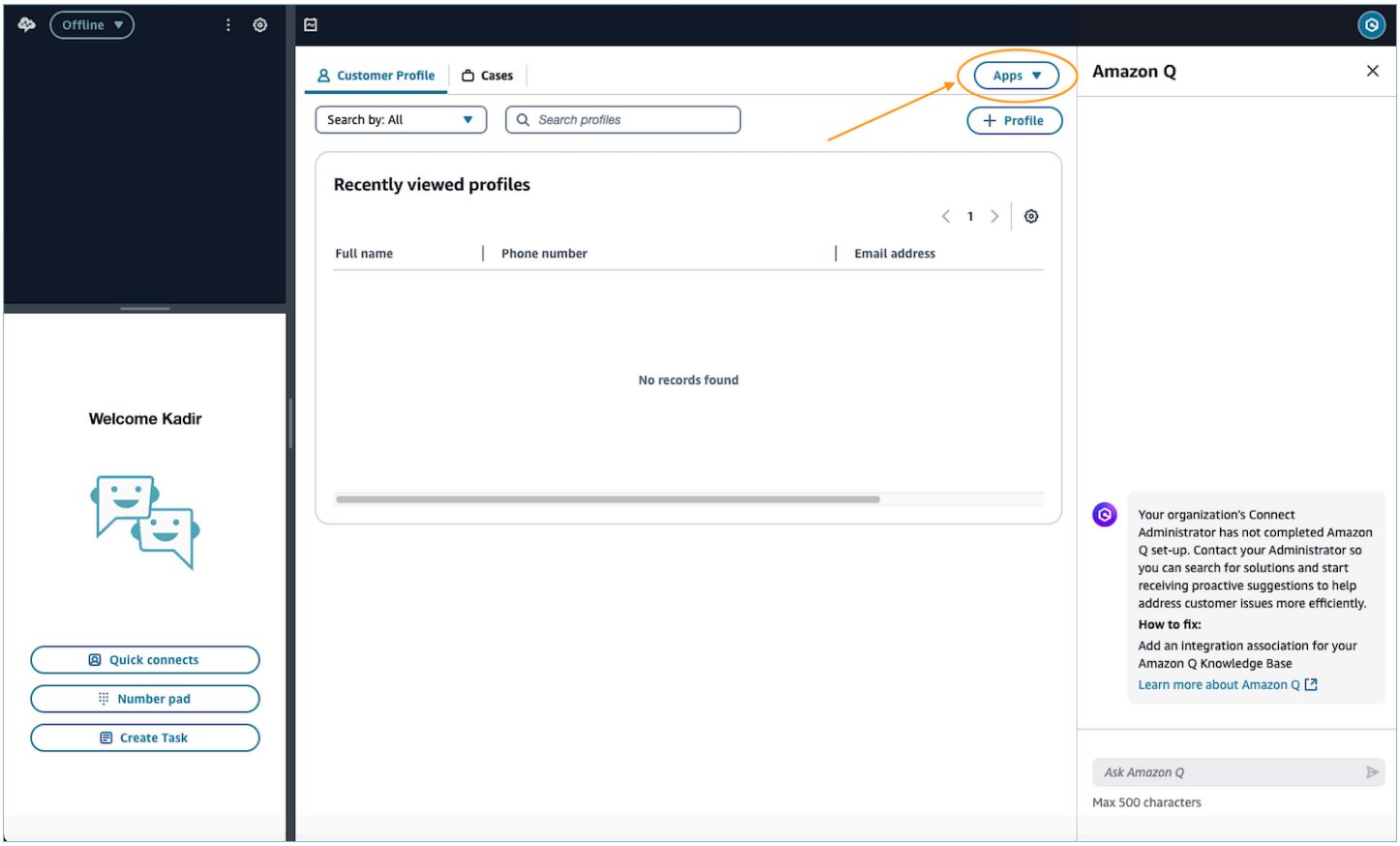
Are you a first-time Connect Customer agent workspace user?

If you are a first-time user of Connect Customer agent workspace, we recommend that you begin by reading the following sections:

- Customize the Connect Customer agent workspace
- Third-party applications (3P apps) in the Connect Customer agent workspace
- Working with third-party applications in the Connect Customer agent workspace

How applications are loaded in Connect Customer agent workspace

The agent workspace allows users to handle multiple contacts concurrently. They will have only one contact selected at a time though, and the agent workspace will update the experience based on the channel (call, chat, or task) of the contact and the applications opened for that contact. When a user switches to another contact, the set of application tabs are updated to what the user was doing last when they were on the previous contact.



An application can be opened by the user selecting the app launcher icon in the top right hand corner of the main agent workspace and select an application from the list. This will load your app in a new application tab for the contact the user has active at that time, or the idle state if the user doesn't have any active contacts. There will be new iframe created for each contact an application is opened with. That iframe will exist until the application tab is closed, for example, a user clicking on the x on the tab or the contact closing. At which point, the app will go through the destroy lifecycle process which gives apps a chance to clean up any resources before the iframe is unmounted from the DOM. The iframe will be hidden when a user selects another tab on the same contact or switches to another contact. This means that at any one time there can be multiple instances, for example, iframes, of the same application running for different contacts.

The agent workspace has a Content Security Policy (CSP) that only allows specific domains to be framed by setting frame-src. The domains configured in the *AccessUrl* and those added to *Approved Origins* will be included in the agent workspace's CSP. Ensure that all domains that your app uses for top level pages are included between *AccessUrl* and *Approved Origins*.

Events and data shared with an instance of an application will be for the contact the application is opened under and the other applications opened on the same contact. Events or data will not be shared between apps on different contacts.

Recommendations and best practices for Connect Customer agent workspace

Use the following recommendations and best practices to optimize applications in the Connect Customer agent workspace.

Topics

Ensuring that apps can only be embedded in the Connect Customer agent workspace

It is recommended that apps correctly set the Content Security Policy header with frame-ancestors to only allow Amazon Connect instances.

```
Content-Security-Policy: frame-ancestors https://*.awsapps.com https://
*.my.connect.aws;
```

Using multiple domains within an app

Apps that use multiple domains, such as those supporting login flows, must add additional domains to the approved origins list on the application configuration. Both the domain specified in the *AccessUrl* and any additional domains added to the *Approved Origins* will be incorporated into the Content Security Policy for the agent workspace, allowing iframe integration for these domains.

Initializing Streams

Initializing the CCP via Streams, even if hidden, is not supported in third-party applications. You must instead use contact and agent events when they are available.

Accessibility

The best practice is for your application to meet accessibility guidelines such as WCAG AA 2.1. The following are some examples of automated and manual tests that you can conduct to ensure that your app meets these guidelines.

Automated accessibility testing tools

1. **axe**: an open-source accessibility testing engine that can be integrated into your development workflow. It provides automated testing of web pages and applications for accessibility issues based on WCAG 2.1 standards.
2. **Pa11y**: a command-line interface that allows you to automate accessibility testing of web pages. It can be integrated into your continuous integration (CI) process to catch accessibility issues early in the development cycle.
3. **Lighthouse**: an open-source, automated tool for improving the quality of web pages. It includes an accessibility audit feature that can identify common accessibility issues and provide suggestions for improvement.
4. **WAVE**: a suite of evaluation tools that help authors make their web content more accessible to individuals with disabilities. It provides a browser extension and an online tool for automated accessibility testing.

Manual accessibility testing tools

1. **Screen Readers**: Use screen readers such as NVDA (NonVisual Desktop Access), JAWS (Job Access With Speech), and VoiceOver to manually test how users with visual impairments interact with your application.
2. **Keyboard Navigation**: Test the application using only a keyboard for navigation to ensure that all interactive elements, such as links and form controls, can be accessed and used without a mouse.
3. **Color Contrast Checkers**: Manual assessment of color contrast using tools like WebAIM's Contrast Checker to ensure that text and graphical elements have sufficient contrast for readability.

4. **User Testing**: Conduct manual accessibility testing with users who have disabilities to gain insights into how they interact with your application and to identify any barriers they may encounter. By using a combination of automated and manual tools, you can provide a comprehensive picture of your application's accessibility compliance. When documenting the testing process, be sure to include details about the tools used, the specific tests performed, and the results obtained to demonstrate your commitment to accessibility.

Theming and styling

The Amazon Connect SDK includes a standard Amazon Connect theme. We recommend that you use the theming package on top of Cloudscape, such that third-party applications match the overall look and feel of the Connect Customer agent workspace.

Sign up for an AWS account

To get started with AWS, you need an AWS account. For information about creating an AWS account, see Getting started with an AWS account in the *AWS Account Management Reference Guide*.

Working with third-party applications in the Connect Customer agent workspace

With Connect Customer agent workspace, you have the option to use first-party applications, such as Customer Profiles, Cases, Wisdom, and features such as step-by-step guides. With support for third-party applications (3P apps), you can unite your contact center software, built by yourself or by partners in one place. For example, you can integrate your proprietary reservation system or a vendor-provided metrics dashboard, into the Connect Customer agent workspace.

The following topics describe key concepts and procedures for developing applications for the Connect Customer agent workspace.

Topics

Prerequisites for developing third-party applications for Connect Customer agent workspace

To develop and test an application for use in Connect Customer agent workspace, you must have the following:

- An Connect Customer instance
- An IAM user that has the proper permissions for creating an application and associating it with the instance. For more information on the required user permissions, see the IAM role required for creating applications in Connect Customer agent workspace
- An Connect Customer user in that instance that has permissions to update security profiles

Topics

IAM role required for creating applications in Connect Customer agent workspace

On top of the `AmazonConnect_FullAccess` IAM policy, users need the following IAM permissions for creating an app and associating it with an Connect Customer instance.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "app-integrations:CreateApplication",
                "app-integrations:GetApplication",
                "iam:GetRolePolicy",
                "iam:PutRolePolicy",
                "iam:DeleteRolePolicy"
            ],
            "Resource": "arn:aws:app-integrations:us-
east-1:111122223333:application/*",
            "Effect": "Allow"
        }
    ]
}
```

Create your application for Connect Customer agent workspace

An application is a website that can be loaded from an HTTPS URL into an iframe in the Connect Customer agent workspace. It can be built using any frontend framework and hosted anywhere as long as it can be loaded by the user's browser and supports being embedded. In addition to being accessible by the user, the application must integrate the Amazon Connect SDK to establish secure communication between the application and the agent workspace allowing the application to receive events and data from the workspace.

Topics

Install the Amazon Connect SDK for developing applications for Connect Customer agent workspace

To develop applications for the Connect Customer agent workspace you must first install the Connect Customer SDK.

The *Connect Customer Amazon Connect SDK* can be installed from NPM. The Aamzon Connect SDK is made up of a set of modules that can be installed as separate packages, meaning that you should only pull in the packages that you need.

The *app* package provides core application features like logging, error handling, secure messaging, and lifecycle events, and must be installed by all applications at a minimum to integrate into the workspace.

Install from NPM

Install the app package from NPM by installing **@amazon-connect/app**.

```
% npm install --save @amazon-connect/app
```

> **Note**
>
> If you do not use NPM, refer to Using Amazon Connect SDK without pacakage manager

Using the Amazon Connect SDK without a package manager

This guide is intended for developers building Amazon Connect integrations who do not use npm, webpack, or other JavaScript package managers and bundlers in their web applications. This includes developers building custom StreamsJS-based contact center interfaces or third-party applications that run within the Connect Customer agent workspace.

Amazon Connect recommends using a package manager such as npm and a bundler such as webpack or Vite for SDK integration. These tools provide dependency management, automatic updates, tree-shaking, and a streamlined development workflow. If you choose not to use these tools, this guide describes an alternative approach.

The Amazon Connect SDK is distributed as npm packages. These packages use Node.js-style module resolution and cannot be loaded directly in a browser using a `<script>` tag. If your development environment does not include a package manager or bundler, you must create a prebuilt script file that bundles the SDK packages into a browser-compatible format.

Your responsibility

When working without a package manager, it becomes your responsibility to:

1. Set up a one-time build environment to create the bundle
2. Select the specific SDK packages your application requires
3. Build and maintain the bundled script file
4. Update the bundle when SDK versions change

Why a bundle is required

SDK packages use ES module imports like `import { ContactClient } from "@amazon-connect/contact"`. Browsers cannot resolve these package specifiers directly. A bundler resolves these imports, combines the code, and produces a single file the browser can execute.

Exposing the SDK as a global

When using `<script>` tags, there is no module system to share code between files. The bundle must attach the SDK to a global variable (such as `window.AmazonConnectSDK`) so your application scripts can access it. This is different from npm-based projects where you import directly from packages.

Available packages

The SDK consists of multiple packages. Select only the packages your application needs. Examples include:

Package	Purpose
`@amazon-connect/core`	Core SDK functionality and provider types
`@amazon-connect/contact`	ContactClient for contact operations
`@amazon-connect/email`	EmailClient for email channel operations
`@amazon-connect/app`	App initialization for third-party applications
`@amazon-connect/app-manager`	Plugin for hosting Connect first-party apps (Cases, Step-by-Step Guides)
`@amazon-connect/voice`	VoiceClient for voice channel operations

See the Amazon Connect SDK repository for the complete list of available packages.

Building the script file

This section provides step-by-step instructions for creating a browser-consumable bundle from the SDK npm packages.

Prerequisites

The following prerequisites are required:

- Node.js 18 or later installed
- npm (comes with Node.js)
- A text editor

Step 1: Create the build project directory

Create a new directory to hold your build configuration. This directory will contain npm tooling but the output bundle will be usable without npm.

```
mkdir connect-sdk-bundle
cd connect-sdk-bundle
```

Step 2: Initialize the npm project

```
npm init -y
```

Step 3: Install the SDK packages you need

For an email-based solution using EmailClient and ContactClient:

```
npm install @amazon-connect/core @amazon-connect/contact @amazon-connect/email
```

If you are building a third-party app (not embedded in StreamsJS), also install:

```
npm install @amazon-connect/app
```

If you are integrating with StreamsJS and want to host Connect first-party apps (such as Cases or Step-by-Step Guides), also install:

```
npm install @amazon-connect/app-manager
```

Step 4: Install the bundler

Install esbuild, a fast JavaScript bundler:

```
npm install --save-dev esbuild
```

Step 5: Create the entry file

Create a file that imports the SDK modules you need and exposes them as a global:

```
mkdir src
```

For StreamsJS integration (`src/entry-streams.js`):

```
// Entry file for StreamsJS integration
import { ContactClient } from "@amazon-connect/contact";
import { EmailClient } from "@amazon-connect/email";

// Expose the SDK on the window object
window.AmazonConnectSDK = {
  ContactClient,
  EmailClient,
};
```

For StreamsJS with first-party apps (`src/entry-streams-with-apps.js`):

If you want to host Connect first-party apps like Cases or Step-by-Step Guides alongside the CCP, include the app-manager plugin:

```
// Entry file for StreamsJS integration with 1P app support
import { ContactClient } from "@amazon-connect/contact";
import { EmailClient } from "@amazon-connect/email";
import { AppManagerPlugin } from "@amazon-connect/app-manager";

// Expose the SDK on the window object
window.AmazonConnectSDK = {
  AppManagerPlugin,
  ContactClient,
  EmailClient,
};
```

For third-party app (`src/entry-app.js`):

```
// Entry file for third-party app integration
import { AmazonConnectApp } from "@amazon-connect/app";
import { ContactClient } from "@amazon-connect/contact";
import { EmailClient } from "@amazon-connect/email";

// Expose the SDK on the window object
window.AmazonConnectSDK = {
  AmazonConnectApp,
  ContactClient,
  EmailClient,
};
```

Step 6: Add build scripts to package.json

Edit `package.json` to add build scripts:

```
{
  "name": "connect-sdk-bundle",
  "version": "1.0.0",
  "scripts": {
    "build:streams": "esbuild src/entry-streams.js --bundle --minify --sourcemap --
format=iife --target=es2020 --outfile=dist/connect-sdk-streams.bundle.js",
    "build:app": "esbuild src/entry-app.js --bundle --minify --sourcemap --format=iife
 --target=es2020 --outfile=dist/connect-sdk-app.bundle.js",
    "build": "npm run build:streams && npm run build:app"
  }
}
```

Step 7: Build the bundle

```
npm run build
```

This creates the following files in the `dist/` directory:

- `connect-sdk-streams.bundle.js` - Bundle for StreamsJS integration
- `connect-sdk-streams.bundle.js.map` - Source map for debugging

- `connect-sdk-app.bundle.js` - Bundle for third-party apps
- `connect-sdk-app.bundle.js.map` - Source map for debugging

Step 8: Copy the bundle to your project

Copy the appropriate `.js` file (and optionally the `.map` file for debugging) to your static website's assets folder:

```
cp dist/connect-sdk-streams.bundle.js /path/to/your/website/assets/vendor/
# or
cp dist/connect-sdk-app.bundle.js /path/to/your/website/assets/vendor/
```

Complete build project structure

After completing all steps, your build project should look like this:

```
connect-sdk-bundle/
### package.json
### package-lock.json
### node_modules/
### src/
#   ### entry-streams.js
#   ### entry-app.js
### dist/
    ### connect-sdk-streams.bundle.js
    ### connect-sdk-streams.bundle.js.map
    ### connect-sdk-app.bundle.js
    ### connect-sdk-app.bundle.js.map
```

Using the SDK with StreamsJS

This section explains how to use the prebuilt bundle in a solution that uses Amazon Connect Streams (StreamsJS).

Prerequisites

The following prerequisites are required:

- The Amazon Connect Streams library loaded on your page

- The `connect-sdk-streams.bundle.js` file from the building section

HTML setup

```
<!DOCTYPE html>
<html>
  <head>
    <title>Connect StreamsJS with SDK</title>
  </head>
  <body>
    <div id="ccp-container" style="width: 400px; height: 600px;"></div>

    <!-- Load Amazon Connect Streams first -->
    <script src="https://your-domain.com/amazon-connect-streams.min.js"></script>

    <!-- Load the SDK bundle -->
    <script src="/assets/vendor/connect-sdk-streams.bundle.js"></script>

    <!-- Your application code -->
    <script src="/app.js"></script>
  </body>
</html>
```

JavaScript implementation

In your `app.js` file:

```
// Initialize the CCP
var ccpContainer = document.getElementById("ccp-container");

connect.core.initCCP(ccpContainer, {
  ccpUrl: "https://your-instance.my.connect.aws/ccp-v2/",
  loginPopup: true,
  loginPopupAutoClose: true,
});

// Get the SDK provider from Streams after CCP initializes
connect.core.onInitialized(function () {
  // Retrieve the provider from the Streams SDK client config
  var sdkConfig = connect.core.getSDKClientConfig();
```

```
  var provider = sdkConfig.provider;

  // Create the SDK clients using the provider
  var contactClient = new AmazonConnectSDK.ContactClient(provider);
  var emailClient = new AmazonConnectSDK.EmailClient(provider);

  // Example: Subscribe to contact lifecycle events
  contactClient.onIncoming(function (event) {
    console.log("Incoming contact:", event.contactId);
  });

  contactClient.onConnected(function (event) {
    console.log("Contact connected:", event.contactId);
  });

  contactClient.onCleared(function (event) {
    console.log("Contact cleared:", event.contactId);
  });
});
```

Hosting Connect first-party apps (optional)

If you want to host Connect first-party apps like Cases or Step-by-Step Guides alongside your CCP, include the `@amazon-connect/app-manager` package in your bundle and apply the plugin during CCP initialization:

```
connect.core.initCCP(ccpContainer, {
  ccpUrl: "https://your-instance.my.connect.aws/ccp-v2/",
  loginPopup: true,
  loginPopupAutoClose: true,
  // Apply the plugin to enable 1P app hosting
  plugins: AmazonConnectSDK.AppManagerPlugin,
});
```

Key points for StreamsJS integration

1. Load the Streams library before the SDK bundle
2. Retrieve the provider using `connect.core.getSDKClientConfig().provider` after CCP initializes
3. Instantiate SDK clients with `new AmazonConnectSDK.ContactClient(provider)`

4. The `AppManagerPlugin` is only required if hosting Connect first-party apps

Using the SDK in a 3P app

This section explains how to use the prebuilt bundle in a third-party application that runs within the Connect Customer agent workspace.

Prerequisites

The following prerequisites are required:

- Your application is registered as a third-party app in Amazon Connect
- The `connect-sdk-app.bundle.js` file from the building section

HTML setup

```
<!DOCTYPE html>
<html>
  <head>
    <title>Connect Third-Party App</title>
  </head>
  <body>
    <div id="app-container"></div>

    <!-- Load the SDK bundle -->
    <script src="/assets/vendor/connect-sdk-app.bundle.js"></script>

    <!-- Your application code -->
    <script src="/app.js"></script>
  </body>
</html>
```

JavaScript implementation

In your `app.js` file:

```
// Initialize the third-party app - this must be called first
var initResult = AmazonConnectSDK.AmazonConnectApp.init({
```

```
  // Optional lifecycle callbacks
  onCreate: function (event) {
    console.log("App created");
  },
  onDestroy: function (event) {
    console.log("App destroyed");
  },
});

// Get the provider from the init result
var provider = initResult.provider;

// Create the SDK clients using the provider
var contactClient = new AmazonConnectSDK.ContactClient(provider);
var emailClient = new AmazonConnectSDK.EmailClient(provider);

// Example: Subscribe to contact lifecycle events
contactClient.onIncoming(function (event) {
  console.log("Incoming contact:", event.contactId);
});

contactClient.onConnected(function (event) {
  console.log("Contact connected:", event.contactId);
});

contactClient.onCleared(function (event) {
  console.log("Contact cleared:", event.contactId);
});
```

Key points for third-party apps

1. Call `AmazonConnectSDK.AmazonConnectApp.init()` before using any SDK functionality
2. The `init()` function returns an object containing the `provider`
3. Instantiate SDK clients with `new AmazonConnectSDK.ContactClient(provider)`
4. Lifecycle callbacks (`onCreate`, `onDestroy`) are optional but useful for managing app state

Updating the bundle

When a new version of the SDK is released:

1. Navigate to your build project directory

2. Update the SDK packages:

```
npm update @amazon-connect/core @amazon-connect/contact @amazon-connect/email
```

3. Rebuild the bundle:

```
npm run build
```

4. Copy the new bundle to your website
5. Test your application to verify compatibility

Troubleshooting

This section describes common issues and resolutions when using the SDK without a package manager.

Bundle is too large

If the bundle size is a concern, ensure you only import the packages you need. Each additional package increases bundle size.

"AmazonConnectSDK is not defined" error

Verify that the bundle script tag appears before your application script in the HTML, and that the path to the bundle file is correct.

Provider is undefined

For StreamsJS: Ensure you are accessing the provider after `connect.core.onInitialized()` fires.

For third-party apps: Ensure you call `AmazonConnectSDK.AmazonConnectApp.init()` and capture its return value.

SDK methods not working

Verify you passed the provider when creating the clients. The provider establishes the communication channel between your code and Amazon Connect.

Initialize the Amazon Connect SDK in your application for Connect Customer agent workspace

Initializing the Amazon Connect SDK in your app for the Connect Customer agent workspace requires calling `init` on the AmazonConnectApp module. This takes an `onCreate` and `onDestroy` callback, which will be invoked once the app has successfully initialized in the agent workspace and then when the agent workspace is going to destroy the iframe the app is running in. These are two of the lifecycle events that your app can integrate with. See Application lifecycle events in Connect Customer agent workspace for details on the other app lifecycle events that your app can hook into.

```
import { AmazonConnectApp } from "@amazon-connect/app";

const { provider } = AmazonConnectApp.init({
  onCreate: (event) => {
    const { appInstanceId } = event.context;
    console.log('App initialized: ', appInstanceId);
  },
  onDestroy: (event) => {
    console.log('App being destroyed');
  },
});
```

Note

Keep the reference to `{ provider }` which is required to create clients to interact with events and requests.

Doing a quick test locally by loading your app directly will produce an error message in the browser dev tools console that the app was unable to establish a connection to the workspace. This will happen when your app is correctly calling `init` when run outside of the workspace.

```
> App failed to connect to agent workspace in the allotted time
```

Events and requests in Connect Customer agent workspace

App developers can easily create applications that seamlessly integrate into the agent workspace experience in the Connect Customer agent workspace with the event and request functionality natively supported by Amazon Connect SDK. You can build an app by leveraging the Amazon Connect SDK to subscribe to agent/contact events (invoking a particular handler when the event occurs) and make requests to quickly retrieve agent/contact data.

This is the main module needed to integrate your app into the agent workspace and get exposure to its agent/contact data and make your app responsive throughout the contact-handling lifecycle.

- **Event**

  Refers to an asynchronous subscription-publication model, where the Amazon Connect SDK's client allows the 3P app to subscribe a callback to-be-invoked when a specific event occurs, such as an agent changing their state from *Available* to *Offline*. It then performs an application-defined action using the event context when said event fires. If and when an event fires is dependent on the event type. For more information, see the API Reference.
- **Request**

  Refers to a request-reply model, where the Amazon Connect SDK's client allows the 3P app to make requests on demand to retrieve data about the current contact or the logged-in agent.

Install from NPM

Install the contact package from NPM by installing **@amazon-connect/contact**.

```
% npm install --save @amazon-connect/contact
```

Authentication for applications in Connect Customer agent workspace

Apps in the Connect Customer agent workspace must provide their own authentication to their users. It is recommended that apps use the same identity provider that the Connect Customer instance has been configured to use when it was created. This will make it so users only need to log in once for both the agent workspace and their applications, since they both use the same single sign on provider.

Note

On Jul 22, 2024, Google announced that they no longer plan to deprecate third-party cookies **[1]**. With this announcement, there will be no impact to third-party applications embedded within Connect Customer agent workspace, unless third-party application users explicitly opt-in for deprecation. We advise third-party application developers to adopt the third-party cookie deprecation impact prevention solutions below as a forward-looking preventative measure.

If you have any questions or concerns, please contact AWS Support **[2]**.

[1] https://privacysandbox.com/news/privacy-sandbox-update/

[2] https://aws.amazon.com/support

For more information, see the 3P admin guide.

Third-party cookie deprecation

We are aware of the **Google Chrome** Third-Party Cookies Deprecation (3PCD) that may impact the third-party applications experience. If your application is embedded within the Connect Customer agent workspace in an iframe and uses cookie based Authentication/ Authorization, then your application is likely to be impacted by Third-Party Cookie Deprecation. You can test if your user experience will be impacted by 3PCD by using the following Test for Breakage guidance.

Here are the recommendations to ensure customers continue to have good experiences when accessing your application within the Connect Customer agent workspace with Google Chrome.

- **Temporary solution**: Allow 3P cookie access here .
- **Permanent solution**: Refer to the guidance from Chrome to choose the best option suitable for your application.

Integrate application with Connect Customer agent workspace agent data

To integrate your application with agent data from the Connect Customer agent workspace, instantiate the agent client as follows:

```
import { AgentClient } from "@amazon-connect/contact";
```

```
const agentClient = new AgentClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, see the API reference to customize your client's configuration.

Once the agent client is instantiated, you can use it to subscribe to events and make requests.

Example agent event

The code sample below subscribes a callback to the state change event topic. Whenever the agent's state is modified, the agent workspace will invoke your provided callback, passing in the event data payload for your function to operate on. In this example, it logs the event data to the console.

```
import { AgentStateChanged } from "@amazon-connect/contact";

// A simple callback that just console logs the state change event data
// returned by the workspace whenever the logged-in agent's state changes
const handler = async (data: AgentStateChanged) => {
    console.log(data);
};

// Subscribe to the state change topic using the above handler
agentClient.onStateChanged(handler);
```

Example agent request

The following code sample submits a `getARN` request and then logs the returned data to the console.

```
const arn = await agentClient.getARN();

console.log(`Got the arn value: ${arn}`);
```

The above agent event and request are non-exhaustive. For a full list of available agent events and requests, see the API Reference.

Integrate application with Connect Customer agent workspace contact data

To integrate your application with contact data from the Connect Customer agent workspace, instantiate the contact client as follows:

```
import { ContactClient } from "@amazon-connect/contact";

const contactClient = new ContactClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, see the API reference to customize your client's configuration.

Once the contact client is instantiated, you can use it to subscribe to events and make requests.

Contact scope

All ContactClient methods include an optional `contactId` parameter. If no value is provided, the client automatically defaults to the contact context from which the app was launched. Note that this requires the app to be opened within a contact's context.

- **Applications configured with Per Contact scope**

  For Per Contact scoped applications, the `contactId` of the current contact is provided in the `AppCreateEvent` which is supplied in the on`Create` callback.

```
const provider =  AmazonConnectApp.init({

    onCreate: async (event: AppCreateEvent) => {
        // Check if scope is defined and has contactId before accessing it
        if (event.context.scope && "contactId" in event.context.scope) {
            let contactId = event.context.scope.contactId;
            console.log("App launched for the contactId", contactId);
        }
    },

    onDestroy: async (event: AppDestroyEvent) => {
        console.log("App destroyed:", event);
    },

});
```

- **Applications configured with Cross Contact scope**

  Cross Contact scoped applications can retrieve the `contactId` by subscribing to any of the contact events like `onConnected` or `onIncomming`

```
const handler: ContactIncomingHandler = async (data: ContactIncoming) => {
    console.log("Contact incoming occurred! " + data);
    let contactId = data.contactId;
};

contactClient.onIncoming(handler);
```

Example contact event

The following code sample subscribes a callback to the connected event topic. Whenever a contact is connected to the agent, the agent workspace will invoke your provided callback, passing in the event data payload for your function to operate on. In this example, it logs the event data to the console.

```
import {
    ContactClient,
```

```
    ContactConnected,
    ContactConnectedHandler
} from "@amazon-connect/contact";

// A simple callback that just console logs the contact connected event data
// returned by the workspace whenever the current contact is connected
const handler: ContactConnectedHandler = async (data: ContactConnected) => {
    console.log(data);
};

// Subscribe to the contact connected topic using the above handler
contactClient.onConnected(handler, contactId);
```

Example contact request

The following code sample submits a `getQueue` request and then logs the returned data to the console.

```
import { ContactClient } from "@amazon-connect/contact";

const queue = await contact.getQueue(contactId);

console.log(`Got the queue: ${queue}`);
```

The above contact event and request are non-exhaustive. For a full list of available contact events and requests, see the API Reference.

Application lifecycle events in Connect Customer agent workspace

There are lifecycle states that an app can move between from when the app is initially opened to when it is closed in the Connect Customer agent workspace. This includes the initialization handshake that the app goes through with the agent workspace after it has loaded to establish the communication channel between the two. There is another handshake between the agent workspace and the application when the app will be shutdown. An application can hook into `onCreate` and `onDestroy` when calling `AmazonConnectApp.init()`.

The following section describe the create and destroy events in the Connect Customer agent workspace.

Topics

The create event in Connect Customer agent workspace

The create event in the Connect Customer agent workspace results in the `onCreate` handler passed into the `AmazonConnectApp.init()` to be invoked. `Init` should be called in an application once it has successfully loaded and is ready to start handling events from the workspace. The create event provides the *appInstanceId* and the *appConfig* .

- **appInstanceId**: The ID for this instance of the app provided by the workspace.
- **appConfig**: The application configuration being used by the instance for this app.
- **contactScope**: Provides the current `contactId` if the app is opened during an active contact.

The destroy event in Connect Customer agent workspace

The destroy event in the Connect Customer agent workspace will trigger the `onDestroy` callback configured during `AmazonConnectApp.init()`. The application should use this event to clean up resources and persist data. The agent workspace will wait for the application to respond that it has completed clean up for a period of time.

Apply a theme to your application in Connect Customer agent workspace

The theme package defines and applies the Connect Customer theme when developing with Cloudscape for the Connect Customer agent workspace.

Install from NPM

Install the theme package and Cloudscape global-styles from NPM by installing **@amazon-connect/theme** and **@cloudscape-design/global-styles**.

```
% npm install -P @amazon-connect/theme
% npm install -P @cloudscape-design/global-styles
```

Usage

The theme package must be imported once at the entry point of the application.

```
import { applyConnectTheme } from "@amazon-connect/theme";

await applyConnectTheme(provider);
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }` .

From then on Cloudscape components and design tokens can be used directly from Cloudscape.

```
// src/app.ts

import * as React from "react";
import Button from "@cloudscape-design/components/button";

export default () => {
  return <Button variant="primary">Button</Button>;
}
```

Application scoping in Connect Customer agent workspace

Every application has two scope settings that together determine how the application behaves with respect to contacts:

- A **registration-time scope**, selected by the application publisher when the application integration is registered with Amazon Connect.
- A **launch-time scope**, passed by the agent application in `AppLaunchOptions.scope` when it calls `launchApp`.

The launch-time scope always takes precedence. It is the value that AppManager uses to determine the lifecycle of the application instance, regardless of the registration-time scope. The registration-time scope is advisory: it indicates the launch-time scope that the publisher recommends for typical use of the application, but the agent application is free to launch the application with a different scope. For example, a registration-time "Cross contacts" application launched with `scope.type: "contact"` is destroyed when the associated contact ends.

Registration-time contact handling scope

The contact handling scope is selected by the application publisher when the application integration is registered with Amazon Connect. It is a property of the application and cannot be changed by the agent application at launch time. The following values are supported.

Scope	Description
Per contact	AppManager maintains a separate application instance for each contact. Each instance's data access is scoped to its associated contact, and the instance is destroyed when that contact ends. To launch the application with this behavior, set the launch-time scope to `contact`.
Cross contacts	AppManager maintains a single application instance that persists across contacts for the lifetime of the browser session. The instance is not bound to any specific contact and is not destroyed when contacts end. This scope is suitable for utilities, knowledge bases, dashboards, and other applications whose content is session-level rather than contact-level. To launch the application with this behavior, set the launch-time scope to `idle`.

Launch-time scope

The launch-time scope is the value that the agent application passes in `AppLaunchOptions.scope` when calling `launchApp`. It applies to both per-contact and cross-contact applications and determines the lifecycle of the resulting application instance. The following values are supported.

- **`contact`** – The application instance is bound to a single contact and can access data only for that contact. AppManager automatically destroys the instance when the associated contact ends,

and emits the `onDestroying` and `onDestroyed` lifecycle events so that the agent application can clean up the user interface. This behavior applies regardless of the application's registration-time scope. A cross-contact application launched with the `contact` scope is destroyed when the associated contact ends.

- **`idle`** – The *idle* state refers to the period when the agent is not handling any contact. An application launched with the `idle` scope is not bound to any contact, does not have access to contact data, and persists for the lifetime of the page or until the agent application calls `destroy()`. Use this scope for cross-contact applications that must remain open across contacts, and for per-contact applications that must remain available when the agent is not handling a contact.

The following table describes the launch-time scope to use for each registration-time scope, including the default behavior when the agent application does not pass a `scope` value at launch. An *active contact* exists whenever the agent is handling one or more contacts; the *idle* state is the period when the agent is not handling any contact.

Registration-time scope	Launch-time scope	Resulting behavior
Per contact	`contact`	AppManager creates one instance per contact and automatically destroys the instance when the contact ends.
	`idle`	The application is available when the agent is idle and is not destroyed when contacts end. The application cannot access contact data.
	Not specified	If an active contact exists at launch, behaves the same as `contact`. If the agent is idle at launch, behaves the same as `idle`.
Cross contacts	`idle`	The application persists across contacts and is destroyed only when the page session ends or the agent application calls `destroy()` .
	`contact`	The application can access contact data, but AppManager destroys the instance when the associate

Registration-time scope	Launch-time scope	Resulting behavior
		d contact ends. This prevents the application from persisting across contacts as the publisher intends.
	Not specified	If an active contact exists at launch, behaves the same as `contact`. If the agent is idle at launch, behaves the same as `idle`.

> **Note**
>
> The registration-time scope is **not** used as the implicit default when no launch-time scope is provided. To guarantee a specific behavior regardless of whether an active contact exists at launch time, set `scope.type` explicitly to `"contact"` or `"idle"`.

Launch an application with an explicit scope

To control the scope under which an application runs, provide a `scope` object in the `AppLaunchOptions` parameter of `launchApp`.

Launch with `contact` scope:

Provide the `contact` type and the ID of the contact to which the application is bound:

```
const launchOptions: AppLaunchOptions = {
    scope: {
        type: "contact",
        contactId: contactId  // The ID of the contact to scope this application to
    }
};

// Launch the application scoped to the specified contact
const appHost = await appManager.launchApp(app.arn, launchOptions);

// Create and configure the iframe for the application
const appIframe = document.createElement("iframe");
```

```
appHost.setIFrame(appIframe);
```

Launch with `idle` scope:

Provide the `idle` type to launch an application that is not bound to the lifecycle of any contact:

```
const launchOptions: AppLaunchOptions = {
    scope: {
        type: "idle"
    }
};

// Launch an application that does not have access to contact data and
// persists regardless of which contact the agent is handling
const appHost = await appManager.launchApp(app.arn, launchOptions);

// Create and configure the iframe for the application
const appIframe = document.createElement("iframe");
appHost.setIFrame(appIframe);
```

Test your application for Connect Customer agent workspace locally

Once you have a minimal version of the app that you want to use in the Connect Customer agent workspace with the Amazon Connect SDK that you want to test in the agent workspace, run your app locally and create an application in the AWS console with an *AccessUrl* using the localhost endpoint, like `http://localhost:3000` .

Creating an application and associating to your instance

> **Note**
>
> Detailed steps for creating and managing applications can be found in the admin guide under Third-party applications (3P apps) in the agent workspace (Preview).

1. Open the Connect Customer console (https://console.aws.amazon.com/connect/).
2. Navigate to **Third-party applications** in the left hand panel.

3. Choose **Add application**.
4. Fill out the necessary required information:

    a. **Name**: The name of the application is what will show up to agents in the app launcher in the agent workspace.

    b. **Namespace**: Namespace must be unique per application and, in the future, allow for applications to support custom events. Once an app is created, its namespace cannot be updated.

    c. **AccessUrl**: Set to the localhost url for your application.

    d. **Permissions**: A list of allowed functions that grants your application the ability to subscribe to agent/contact events that occur in the agent workspace or make requests for agent/contact data.
5. Select the Connect Customer instance you are testing with to associate the app with that instance.
6. Choose **Add application** to finish creating your app.
7. Log into your test instance as an admin user.
8. Navigate to **Security profiles** and select the `Admin` security profile.
9. Under **Agent applications** find your application and make sure the `View` permission is selected.

    - Open the agent application `/agent-app-v2`
10. Open your app by choosing the app launcher and selecting your application. Your app will be opened in a new application tab.

After following these steps you will have your app loaded from your local machine into the workspace. This will only work when loading the agent workspace on your local machine that has the app running on it. If you want to be able to load your app from any browser / computer, then you must deploy your app somewhere that is internet accessible.

Assuming the logging was included from the code snippet above, you should see the following in the console log of your browser's dev tools when you open your app in the workspace.

```
App initialized:  00420d405e
```

When your app is closed, for example, by closing the tab in the agent workspace, you should see the following series of logs entries.

```
> App destroyed: begin
> App being destroyed
> App destroyed
> App destroyed: end
```

If you see these, then your app correctly integrates with the *Connect Customer Amazon Connect SDK* and the The create event in Connect Customer agent workspace / The destroy event in Connect Customer agent workspacedestroy lifecycle events.

Test a deployed version of your application for Connect Customer agent workspace

When ready, deploy the app that you created for the Connect Customer agent workspace to a place that is internet accessible. Update your application configuration (or configure a new application) to point to the deployed version of your application. A simple way to deploy your app assuming it only has static assets is to host them on S3 and (optionally) use CloudFront.

Handle application errors in Connect Customer agent workspace

Applications can communicate errors back to the Connect Customer agent workspace by either calling `sendError` or `sendFatalError` on the `AmazonConnectApp` object. The agent workspace will shutdown an app if it sends a fatal error meaning that the app has reached an unrecoverable state and isn't functional. When an app sends a fatal error the agent workspace won't attempt to go through the destroy lifecycle handshake and will immediately remove the iframe from the DOM. Apps should do any clean up required prior to sending fatal errors.

Troubleshoot application setup in Connect Customer agent workspace

You can use the Amazon Connect SDK's `AppConfig` object to retrieve data about your applications's setup in the Connect Customer agent workspace, including its permissions. This

will allow you to inspect its state and determine which permissions were assigned to your app. Accessing its `permissions` property will return a list of strings, each representing a permissions that grants access to a set of events and requests. Performing an action, whether subscribing to an event or making a request, will fail if your app does not have the corresponding permission that grants the action. You may have to ask your account admin to assign the permissions required for your app to function. To review the full list of permissions assignable to apps, please see the admin guide.

Events

If your app uses the Amazon Connect SDK to subscribe to an event that it does not have permission for, the agent workspace will throw an error with a message formatted like below.

```
App attempted to subscribe to topic without permission - Topic {"key":
<event_name>,"namespace":"aws.connect.contact"}`
```

Requests

If your app uses the Amazon Connect SDK to make a request that it does not have permission for, the agent workspace will throw an error with a message formatted like below.

```
App does not have permission for this request
```

Building third-party services in the Connect Customer agent workspace

What is a third-party (3P) service?

third-party (3P) services are headless applications that customers can build and integrate into Connect Customer agent workspace. Services begin running when the agent workspace loads and remain active throughout the agent workspace session. They can perform various background tasks and enhance the agent experience, such as:

- Listening to contact events: Services can monitor events like contact connection, disconnection, or entering After Contact Work (ACW) state.
- Launching applications: Services can automatically open specific applications based on certain triggers or conditions.
- Implementing custom authentication flows: Services can ensure users complete necessary authentication processes for third-party applications as soon as the agent workspace starts.

Why use a third-party service?

third-party services allow you to implement background processes and automate tasks throughout the agent workspace session. They provide powerful capabilities for establishing contact event listeners, enabling custom authentication flows, and managing other agent workspace-wide functionality.

Common use cases for 3P services

These are common use cases for third-party services:

- Automatically launch specific applications when an agent enters After Contact Work (ACW).
- Ensure critical applications are always running by automatically launching them during agent workspace startup.
- Control application visibility by automatically bringing specific apps into focus based on contact events or other triggers.

- Implement custom contact handling logic to determine what happens when agents accept or leave contacts.
- Manage authentication flows that need to run when the agent workspace loads.

Understanding when to use each option

When to use a third-party application

Choose a third-party application when you need:

- Features that can be opened and closed during the agent's workflow
- Functionality that needs to be visible and directly interactive for the agent

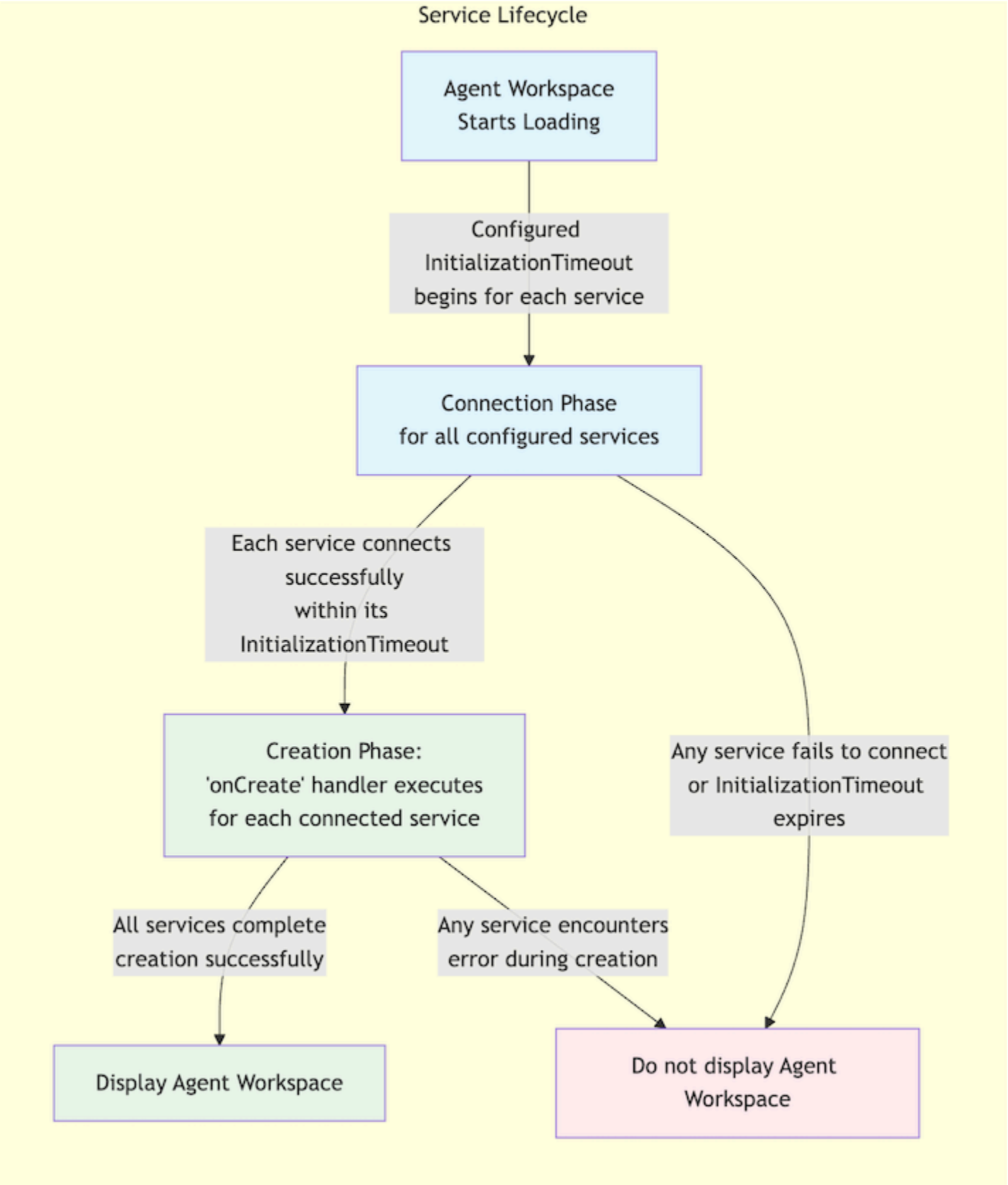
When to use a third-party service

Choose a third-party service when you need:

- To implement agent workspace-wide event listeners and handlers
- Automatic background processes that should run throughout the agent workspace session
- To automate tasks that don't require direct agent interaction
- To control the behavior and state of other applications in the agent workspace

Agent workspace startup process

Third-party services follow this process in the Connect Customer agent workspace:


Service Lifecycle
Agent Workspace Starts Loading
Configured InitializationTimeout begins for each service
Connection Phase for all configured services
Each service connects successfully within its InitializationTimeout
Creation Phase: 'onCreate' handler executes for each connected service
Any service fails to connect or InitializationTimeout expires
All services complete creation successfully
Any service encounters error during creation
Display Agent Workspace
Do not display Agent Workspace

1. **Agent workspace startup**: When an agent logs in and the agent workspace starts loading, all configured services will begin their startup process.
   a. The configured InitializationTimeout will be in effect until the third-party service has officially connected to the agent workspace.
2. **Agent workspace loading**: The agent workspace will not fully load and become accessible to the agent until all services have successfully connected.
3. **Service startup**: Once connected, the logic within the onCreate handler will begin to run.
4. **Service runtime**: Once created, services continue running for the remainder of the agent workspace session.

> **Important**
>
> Services directly impact the agent workspace startup process. If any service fails to start within its configured timeout, an error will be displayed which will prevent agents from accessing the agent workspace.

Creating a third-party service

Third-party service setup

Following the instructions here to properly integrate your service with the agent workspace. First, install the app package:

```
% npm install --save @amazon-connect/app
```

> **Note**
>
> If you do not use NPM, refer to Using Amazon Connect SDK without pacakage manager

Then, add the following initialization code to your app:

```
import { AmazonConnectService } from "@amazon-connect/app";
```

```
const { provider } = AmazonConnectService.init({
  onCreate: (event) => {
    const { instanceId } = event.context;
    console.log('Service creation complete: ', instanceId);
  }
});
```

When building a third-party service:

1. Use AmazonConnectService from @amazon-connect/app to initialize your service
2. Complete all initialization within the timeout period (30 seconds)
   a. This is a separate timeout from the initializationTimeout configured for the service
   b. initializationTimeout is the time the agent workspace will wait for a service to successfully connect, before onCreate is triggered
3. Handle initialization failures to prevent agent workspace loading issues

Here is an example of proper initialization with timeout handling:

```
import { AmazonConnectService } from "@amazon-connect/app";

export function initService() {
  AmazonConnectService.init({
    onCreate: async (event) => {
      // This is where you set up your service's functionality
      // Add all of your service setup code here - examples include:
      // - Set up event listeners
      // - Establish connections
      // - Load configurations
      // - etc.
      let timeoutId: NodeJS.Timeout;

      const INIT_TIMEOUT = 25000; // 25 seconds
      const { context } = event;
      const { instanceId } = event.context;
      const provider = context.getProvider();

      const cleanup = () => {
        if (timeoutId) {
          clearTimeout(timeoutId);
```

```
        }
        // Add any other cleanup logic here
        // For example: close connections, remove event listeners, etc.
      };

      try {
        // Create a promise that will reject after the timeout
        const timeoutPromise = new Promise((_, reject) => {
          timeoutId = setTimeout(() => {
            reject(new Error('Service creation timed out'));
          }, INIT_TIMEOUT);
        });

        // Race between your code and the timeout
        await Promise.race([
          executeServiceCode(),
          timeoutPromise
        ]);

        console.log('Service creation complete: ', instanceId);

      } catch (error) {
        console.error('Service creation failed:', error);
        // Send fatal error to terminate the service
        provider.sendFatalError('Service creation failed');
      } finally {
        cleanup();
      }
    }
  });
}

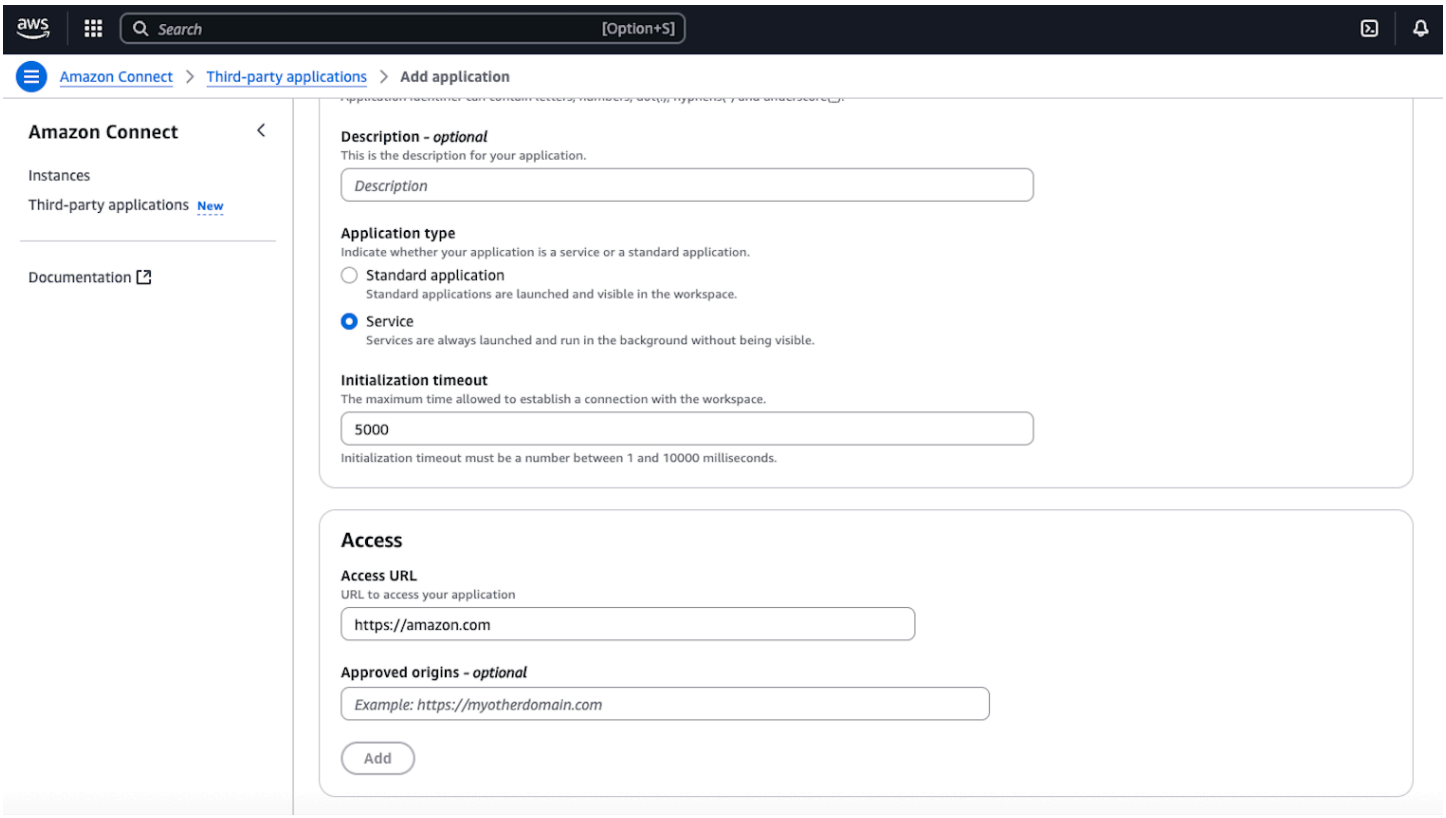
async function executeServiceCode() {
  // Your service implementation goes here
  console.log('Executing service code');
}

initService();
```

AWS console setup

Create a new third-party service by creating a third-party application with the following settings either via APIs or in the AWS Console:

- `isService` set to true
  - An app that is configured as a service will be started when the agent workspace is loaded and will run hidden for the lifetime of the agent workspace. An application that runs as a service in the agent workspace must be integrated with the app Amazon Connect SDK package. If the service fails to start before the InitializationTimeout, then an error will be sent to agent workspace causing the agent workspace to fail
- Set a `InitializationTimeout` in milliseconds up to 10000 (10 seconds)
  - The InitializationTimeout parameter controls the maximum time allowed for the initial handshake/connection between the service and the agent workspace. This is required to be set for applications configured with isService to true.



Service implementation patterns

third-party services can implement various patterns to extend the agent workspace functionality. The following examples demonstrate some of our key Amazon Connect SDK capabilities:

Launching an application on startup

```
import {
  AmazonConnectService,
  ServiceContext,
  ServiceCreatedEvent,
} from "@amazon-connect/app";
import { ContactClient } from "@amazon-connect/contact";

const provider = AmazonConnectService.init({
  onCreate: onCreateHandler,
});

const onCreateHandler = async (event: ServiceCreatedEvent) => {
  const context: ServiceContext = event.context;
  console.log(`${SDK_LOG_PREFIX} Service created: `, context.instanceId);
  await registerEventHandlers();

  const appName = 'TargetAppName';
  console.log(`Launching app in ACW`, { event: contactEvent, appName: appName });
  const appControllerClient = new AppControllerClient(context.getProvider());

  // Get list of all applications available to the agent
  const apps: AppConfig[] = await appControllerClient.getAppCatalog();

  // Find your application by name
  const appArn = apps.find(
    (app) => app.name === appName
  )?.arn;

  if (!appArn) {
    throw new Error(`${appName} not found!`);
  }

  await appControllerClient.launchApp(appArn);
};
```

Contact event listening with application launching functionality

```
import {
```

```
  AmazonConnectService,
  ServiceContext,
  ServiceCreatedEvent,
} from "@amazon-connect/app";
import { ContactClient } from "@amazon-connect/contact";

const provider = AmazonConnectService.init({
  onCreate: onCreateHandler,
});

const onCreateHandler = async (event: ServiceCreatedEvent) => {
  const context: ServiceContext = event.context;
  console.log(`${SDK_LOG_PREFIX} Service created: `, context.instanceId);
  await registerEventHandlers();
  return Promise.resolve();
};

async function registerEventHandlers() {
  const contactClient = new ContactClient(provider);

  // Listen for connected contacts
  contactClient.onConnected(async (contactEvent) => {
    console.log(`Contact connected!`, { event: contactEvent });
  });

  // Listen for contacts that have entered After Contact Work (ACW)
  contactClient.onStartingAcw(async (contactEvent) => {
    const appName = 'TargetAppName';
    console.log(`Launching app in ACW`, { event: contactEvent, appName: appName });
    const appControllerClient = new AppControllerClient(provider);

    // Get list of all applications available to the agent
    const apps: AppConfig[] = await appControllerClient.getAppCatalog();

    // Find your application by name
    const appArn = apps.find(
        (app) => app.name === appName
    )?.arn;

    if (!appArn) {
        throw new Error(`${appName} not found!`);
    }

    await appControllerClient.launchApp(appArn);
```

```
  });

  // Listen for contacts that are cleared
  contactClient.onCleared(async (contactEvent) => {
    try {
      console.log(`Contact Cleared:`, { event: contactEvent, type:
 contactEvent.type });
    } catch (error) {
      console.error(`Error handling incoming contact:`, { event: contactEvent, error:
 error });
    }
  });

  // This is emitted when the service unsubscribes from the cleared event
  contactClient.offCleared(async (contactEvent) => {
    console.log(`Contact no longer in the incoming state`, { event: contactEvent });
  });
}
```

Authentication popup functionality

```
import { AmazonConnectService } from "@amazon-connect/app";
import { AppControllerClient } from "@amazon-connect/app-controller";
import { AppConfig } from "@amazon-connect/workspace-types";

interface IdpMessage {
  type: 'IDP_MESSAGE';
  payload: {
    message: string;
    timestamp: string;
  };
}

const SDK_LOG_PREFIX = '[TestService]';
const APP_NAME = 'TestApp'

const IDP_POPUP_CONFIG = {
  width: 600,
  height: 400,
  title: 'IDP Simulation'
};
```

```
let provider;

// Start the authentication service
export function initIDPService() {
  provider = AmazonConnectService.init({
    onCreate: onCreateHandler
  });
}

const onCreateHandler = async () => {
  console.log(`${SDK_LOG_PREFIX} Service running...`);

  try {
    // Start the authentication flow
    runIdpFlow();
  } catch(e) {
    // This reports an error in the auth flow but allows the service
    // to continue running.
    console.error(`${SDK_LOG_PREFIX} IDP flow failed:`, error);
  }

  console.log(`${SDK_LOG_PREFIX} Service creation complete`);
};

/**
  * Creates the IDP authentication popup
  * Returns a promise that resolves when authentication is complete
  * or rejects if authentication fails/times out
  */
export const initIdpSimulator = (): Promise<void> => {
  return new Promise((resolve, reject) => {

    // Clean up event listeners and timers
    const cleanup = () => {
      window.removeEventListener('message', handleIdpMessage);
    };

    // Handle messages received from the IDP popup
    const handleIdpMessage = async (event: MessageEvent<IdpMessage>) => {
      if (event.data.type === 'IDP_MESSAGE') {
        // Implementation for a successful authentication
        console.log(`${SDK_LOG_PREFIX} Message received from IDP:`,
 event.data.payload);
```

```
        try {
          // Launch the application after successful authentication
          await launchApp(APP_NAME);
          console.log(`${SDK_LOG_PREFIX} Successfully launched app`);
          resolve();
        } catch (error) {
          console.error(`${SDK_LOG_PREFIX} Failed to launch app:`, error);
          reject(error);
        } finally {
          cleanup();
        }
      }
    };

    // Open the authentication popup window
    const openIdpPopup = () => {
      // https://developer.mozilla.org/en-US/docs/Web/API/Window/open
      const popup = window.open('/popup.html', // URL to your authentication page
        IDP_POPUP_CONFIG.title,
        `width=${IDP_POPUP_CONFIG.width},height=${IDP_POPUP_CONFIG.height}`
      );

      if (!popup) {
        cleanup();
        reject(new Error('Failed to open IDP popup'));
        return;
      }
    };

    // Listen for messages from the popup
    window.addEventListener('message', handleIdpMessage);

    // Open the popup
    openIdpPopup();
  });
};

/**
  * Manages the complete authentication flow
  * Handles errors and logging
  */
export const runIdpFlow = async () => {
  try {
```

```
    console.log(`${SDK_LOG_PREFIX} Starting IDP flow...`);
    await initIdpSimulator();
    console.log(`${SDK_LOG_PREFIX} IDP flow completed successfully`);
  } catch (error) {
    console.error(`${SDK_LOG_PREFIX} IDP flow failed!`, error);
    throw error;
  }
};

/**
  * Launches the specified application using the AppController
  * Verifies the app exists in the catalog before attempting to launch
  */
export async function launchApp(targetAppName: string) {
  const appControllerClient = new AppControllerClient(provider);

  // Get list of all applications available to the agent
  const apps: AppConfig[] = await appControllerClient.getAppCatalog();
  console.log(`${SDK_LOG_PREFIX} App Catalog: `, apps);

  // Find your application by name
  const testAppArn = apps.find(
    (app) => app.name === targetAppName
  )?.arn;

  if (!testAppArn) {
    throw new Error(targetAppName + " not found!");
  }

  await appControllerClient.launchApp(testAppArn);
}
```

This is the HTML template for the authentication popup:

```
<!DOCTYPE html>
<html>
  <head>
      <title>IDP Simulation</title>
      <style>
          body {
              font-family: Arial, sans-serif;
              padding: 20px;
```

```
            background: #f5f5f5;
        }
        .container {
            background: white;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 2px 10px rgba(0,0,0,0.1);
        }
        button {
            padding: 10px 20px;
            margin: 5px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            background: #007bff;
            color: white;
        }
        button:hover {
            background: #0056b3;
        }
    </style>
</head>
<body>
    <div class="container">
        <h2>IDP Simulation</h2>
        <p>Click the button below to simulate IDP authentication</p>
        <button id="authButton">Authenticate</button>
    </div>
    <script>
        // When authentication button is clicked, send success message
        // back to parent window
        document.getElementById('authButton').onclick = function() {
            window.opener.postMessage({
                type: 'IDP_MESSAGE',
                payload: {
                    message: 'User authenticated via IDP',
                    timestamp: new Date().toISOString()
                }
            }, '*');
            window.close();
        };
    </script>
</body>
```

```
</html>
```

Best practices and recommendations

Service creation management

- Keep onCreate operations lightweight to ensure a reasonable loading time for Agents
  - Use Promise timeouts for any external API calls during initialization to ensure fail-fast behavior
- Handle service errors gracefully
  - Any uncaught error encountered during service initialization will be considered as a service failure, which will prevent agents from accessing the workspace

Authentication

- Prompt for Authentication during agent workspace startup with a third-party service
  - Begin authentication process without blocking service execution
  - Centralize authentication prompting in your service to avoid redundant implementations in your third-party applications
- Implement visual authentication interfaces (e.g., pop-ups) for agent interaction
- Set appropriate authentication timeouts to prevent infinite retry loops
- Ensure applications share the same origin as the third-party service

Service coordination

- Consolidate interdependent behaviors within a single service
  - For example, any applications launched on the startup of the agent workspace should be done by one service to ensure a consistent launch order for agents

Integrating AWS-managed applications with Amazon Connect Streams

This guide demonstrates how to integrate AWS-managed applications with your existing applications built using Amazon Connect Streams. This integration extends your custom agent application with AWS-managed applications from the Connect Customer agent workspace. By embedding AWS-managed applications into your custom agent application, you can leverage their features without additional development effort, while maintaining control over application access through Security Profiles.

Amazon Connect Streams

Amazon Connect Streams is a JavaScript library that integrates the Contact Control Panel (CCP) and other agent functionality into existing web applications. This library enables you to embed the CCP user interface as well as handle agent and contact state events so that you can build a custom agent application. See the Amazon Connect Streams documentation.

AWS-managed applications

Amazon Connect provides AWS-managed applications, such as Worklist that are accessible in the Connect Customer agent workspace.

Amazon Connect SDK

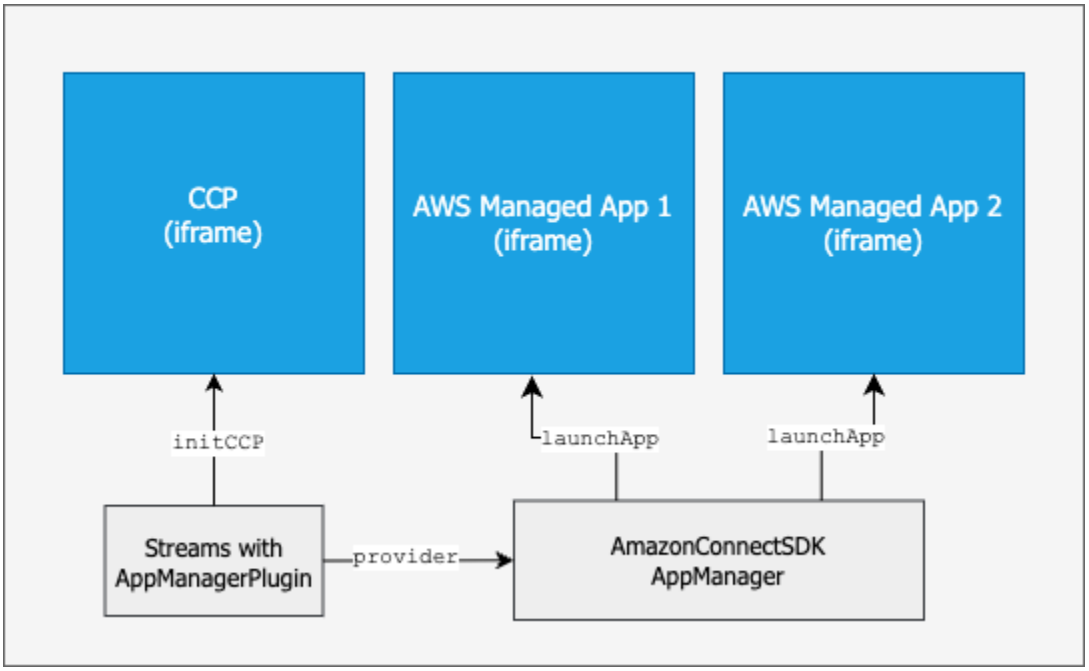
The Amazon Connect SDK is a collection of packages that helps you build applications applications that interact with and extend Amazon Connect's native functionality. See the Amazon Connect SDK repository on GitHub. .

AppManager

AppManager provides APIs to discover, launch, and manage AWS-managed applications. It's available within the Amazon Connect SDK `@amazon-connect/app-manager` package.

Integration architecture

The following diagram illustrates the components and integration flow for AWS-managed applications using Streams and AppManager.



Application launch follows this sequence:

1. Your web application initializes the CCP with the AppManager plugin.
2. The `launchApp` method in AppManager is invoked with the application name or Amazon Resource Name (ARN).
3. AppManager creates an `AppHost` object to manage the application instance.
4. An iframe element is provided to the `AppHost`.
5. AppManager configures the iframe with appropriate URL and security attributes.
6. The application loads and establishes secure communication.

Implementation guide

This section describes the steps to integrate AWS-managed applications using Streams and AppManager. The Worklist AWS-managed application is used for demonstration purposes.

Prerequisites

The following prerequisites are required:

- A working web application integrated with Amazon Connect Streams version 2.20 or above
- Security Profile permissions configured for the Worklist AWS-managed application

Step 1: Install required packages

Choose the appropriate package based on your integration needs:

- **For basic application management** — Install the `@amazon-connect/app-manager` package:

```
npm install @amazon-connect/app-manager
```

- **For contact-scoped applications** — If you need to manage applications that are tied to specific customer contacts, install the `@amazon-connect/app-manager-agent` package instead. This package includes all the functionality of `@amazon-connect/app-manager` plus contact-scoped application support. For more information, see the section called "Contact-scoped applications".

```
npm install @amazon-connect/app-manager-agent
```

> **Note**
>
> If you do not use NPM, refer to Using Amazon Connect SDK without pacakage manager

Step 2: Add the AppManager plugin in CCP initialization

Update the existing CCP initialization code to include the AppManager plugin.

Before:

```
import "@amzn/amazon-connect-streams";
```

```
const containerElement = document.getElementById("ccp-container");
connect.core.initCCP(containerElement, {
    ccpUrl: "https://<connect-instance-alias>.my.connect.aws/ccp-v2/",
    // Other initialization parameters
});
```

After:

Use the code example that corresponds to the package you installed in the section called “Step 1: Install packages”. The two examples differ only in the side-effect import on the second line.

Option A — For basic application management (using `@amazon-connect/app-manager`):

```
import "@amzn/amazon-connect-streams";
import "@amazon-connect/app-manager";
import { AppManagerPlugin } from "@amazon-connect/app-manager";

const containerElement = document.getElementById("ccp-container");
connect.core.initCCP(containerElement, {
    ccpUrl: "https://<connect-instance-alias>.my.connect.aws/ccp-v2/",
    // Other initialization parameters

    plugins: [AppManagerPlugin], // Enables AppManager
});

// Retrieve the provider for accessing AppManager
const provider = connect.core.getSDKClientConfig().provider;
```

Option B — For contact-scoped applications (using `@amazon-connect/app-manager-agent`):

```
import "@amzn/amazon-connect-streams";
import "@amazon-connect/app-manager-agent";
import { AppManagerPlugin } from "@amazon-connect/app-manager";

const containerElement = document.getElementById("ccp-container");
connect.core.initCCP(containerElement, {
    ccpUrl: "https://<connect-instance-alias>.my.connect.aws/ccp-v2/",
    // Other initialization parameters

    plugins: [AppManagerPlugin], // Enables AppManager
```

```
});

// Retrieve the provider for accessing AppManager
const provider = connect.core.getSDKClientConfig().provider;
```

Note

- The AppManager plugin is backward compatible and does not affect existing CCP functionality.
- Replace `<connect-instance-alias>` with your Amazon Connect instance alias.

Step 3: Embed a page for AWS-managed application

Add an iframe element to the desired location for displaying the Worklist AWS-managed application:

```
<iframe id="aws-app-iframe" height="1080px" width="1920px"/>
```

Important

AppManager configures the iframe source, do not manually set the `src` attribute in iframe.

Step 4: Launch AWS-managed application

Launch the Worklist AWS-managed application using the AppManager `launchApp` API:

```
// Launch the Worklist AWS-managed application
const appHost = await provider.appManager.launchApp("Worklist");

// Retrieve the iframe element for displaying the Worklist AWS-managed application
const awsAppIframe = document.getElementById("aws-app-iframe");

// AppManager uses the iframe to render the AWS-managed application
appHost.setIFrame(awsAppIframe);
```

Step 5: Build and deploy

After you build and deploy your web application, verify that the Worklist application is loaded in the iframe and available for usage.

Retrieve available applications

The `getAppCatalog` method retrieves applications available to the authenticated user. AppManager filters the list based on the user's Security Profile permissions. Use this method to verify if user has permission for the AWS-managed application prior to launching.

```
import { AppConfig } from "@amazon-connect/workspace-types";

// Retrieve applications filtered by Security Profile permissions
const applications: AppConfig[] = await appManager.getAppCatalog();
```

Application configuration properties:

Each `AppConfig` object contains the following properties:

- `arn` - Amazon Resource Name (ARN) that uniquely identifies the application
- `name` - Display name for the application

> **Note**
>
> This is not required if you know the name of the AWS-managed application that you want to launch and the user is guaranteed to have access.

Application lifecycle management

Application lifecycle management is essential when adding and removing applications throughout a page's life. If you launch an application once at startup and keep it open, lifecycle management is optional. However, if you launch and close apps as part of the user's workflow, then these lifecycle states help you create a better user experience.

Managing application lifecycle

Application creation begins after an iframe is set in the AppHost. You can subscribe to AppHost's on`Created` event to control when the iframe appears to the user, though showing it during application creation has no functional impact.

AppManager handles application state transitions throughout the lifecycle. To close an application, invoke the `destroy` method on the `AppHost` object. The `AppHost` then emits lifecycle events (on`Destroying` , on`Destroyed`). Your web application handles these events to update the user interface accordingly.

The following table describes the application lifecycle states.

State	Description
`Created`	The application iframe is configured and secure communication is established. The application is ready for use. This state occurs immediately following successful `launchApp()` invocation.
`Destroying`	Application cleanup is in progress. The application is no longer usable. This state occurs after you invoke `destroy()` method.
`Destroyed`	The application is fully destroyed. This state occurs up to 5 seconds following the `destroy()` method invocation. The iframe can be safely removed from the DOM.

Important

When destroying an application, do not remove it from the Document Object Model (DOM) until the destruction process completes. You can hide it from users, but premature removal from the DOM might interrupt critical cleanup processes such as flushing log buffers or saving state. The on`Destroyed` event indicates when it's safe to completely remove the application.

Application visibility states

When managing multiple applications, based on how you arrange them, some applications may not be visible at any given time. When an application is temporarily hidden, it's recommended to notify the application by invoking `stop()` on `AppHost` and then invoking `start()` when making it visible again.

State	Description
`Started`	The application is visible and actively synchronizing data.
`Stopped`	The application is not visible. Background operations are paused.

Handle lifecycle events

Implement application lifecycle event handlers to manage iframe visibility and cleanup operations:

```
// Handle destroying event
appHost.onDestroying((event) => {
    console.log(`Application ${appHost.config.name} is being destroyed`);

    // Hide the iframe as the application is no longer usable
    appIframe.style.display = "none";

    return Promise.resolve();
});

// Handle destroyed event
appHost.onDestroyed((event) => {
    console.log(`Application ${appHost.config.name} has been destroyed`);

    // Remove the iframe from the DOM
    appIframe.remove();

    return Promise.resolve();
});
```

Advanced configuration

Prevent duplicate application instances

When you launch an application multiple times, AppManager creates multiple application instances by default. You can use launch keys when starting applications to prevent multiple instances of the same application.

```
const launchOptions: AppLaunchOptions = {
    launchKey: 'Worklist-singleton'
};

// Returns existing instance if launch key matches a running application
const appHost = await appManager.launchApp("Worklist", launchOptions);
```

The launch key is a caller-generated value that prevents duplicate instances. When an application with a matching launch key is already running, AppManager returns the existing instance rather than creating a new instance and triggers `onAppHostFocused` event. This event could be leveraged to bring the application to focus when it's being launched again.

Dynamic application launch and management

AppManager provides `onAppHostAdded` and `onAppHostRemoved` notifications to indicate when a new application is launched or destroyed. These events could be leveraged to dynamically create and destroy iframes. For an example implementation, see the section called “Dynamic application management” section below.

Attach metadata to appHost

If you are managing iframes dynamically and want the storage of any application specific UI state or routing information in the `AppHost`, you can attach a custom JSON metadata at the time of application launch. This metadata is accessible in all event callbacks for application state management.

```
const launchOptions: AppLaunchOptions = {
    appManagerData: {
        // Custom JSON data
  }
```

```
};

const appHost = await appManager.launchApp("Worklist", launchOptions);

// Access the metadata
console.log("AppManagerData attached to AppHost", appHost.appManagerData);
```

Support Global Resiliency

Amazon Connect Global Resiliency features provide automatic failover between AWS Regions. When your Amazon Connect instance is configured for Global Resiliency, you should implement the following handlers to ensure agent workspace responsiveness to failover events:

```
// Handle pending failover events
connect.globalResiliency?.onFailoverPending(() => {
   // Destroy all applications before failover
    appManager.clearAll();
});

// Handle completed failover events
connect.globalResiliency?.onFailoverCompleted(() => {
    // Re-launch necessary applications following failover
    // Implementation depends on your state management approach
});
```

See Set up Amazon Connect Global Resiliency in the *Amazon Connect Administrator Guide*.

Example implementation of dynamic application management with React

The following example demonstrates how to dynamically manage AWS-managed applications in a React application. This implementation uses onAppHostAdded and onAppHostRemoved events to automatically update the user interface when applications are launched or destroyed. This example demonstrates how to position applications one after the other vertically on a web page.

Iframe container component

```
const IFrameAppContainer: React.FC<{ appHost: AppHost }> = ({ appHost }) => {
  const iframeRef = useRef<HTMLIFrameElement>(null);

  useEffect(() => {
    const iframe = iframeRef.current;
    if (iframe) {
      (appHost as IFrameAppHost).setIFrame(iframe);
    }
  }, [appHost]);

  const handleClose = (): void => {
    void appHost.destroy();
  };

  return (
    <div className="app-container">
      <div className="app-header">
        <h2>{appHost.config.name}</h2>
        <button onClick={handleClose}> Close </button>
      </div>
      <iframe ref={iframeRef} className="app-iframe" title={appHost.config.name} />
    </div>
  );
};
```

Application container component

```
const ApplicationContainer: React.FC<{provider: AmazonConnectProvider}>
  = ({provider}) => {

  const [activeApps, setActiveApps] = useState<Set<AppHost>>(new Set());

  useEffect(() => {
    const appManager = provider.appManager;

    // Handle new applications being added
    const handleAppHostAdded = ({appHost}: AppHostAdded): Promise<void> => {
      setActiveApps((prevApps) => new Set(prevApps).add(appHost));
      return Promise.resolve();
    };

    // Handle applications being removed
```

```
    const handleAppHostRemoved = ({appHost}: AppHostRemoved): Promise<void> => {
      setActiveApps((prevApps) => {
        const newApps = new Set(prevApps);
        newApps.delete(appHost);
        return newApps;
      });

      return Promise.resolve();
    };

    // Register event handlers
    appManager.onAppHostAdded(handleAppHostAdded);
    appManager.onAppHostRemoved(handleAppHostRemoved);

    return () => {
        // Un-register event handlers
        appManager.offAppHostAdded(handleAppHostAdded);
        appManager.offAppHostRemoved(handleAppHostRemoved);
    }

  }, [provider.appManager]);

  return (
    <div className="application-container">
      {Array.from(activeApps).map((appHost) => (
        <IFrameAppContainer key={appHost.instanceId} appHost={appHost} />
      ))}
    </div>
  );
};
```

Contact-scoped applications

This section describes how to manage AWS-managed applications that are tied to specific customer contacts. Contact scoping itself is not specific to AWS-managed applications — for an explanation of the registration-time and launch-time scope settings that determine how any application behaves with respect to contacts, including `contact` and `idle` launch examples, see the section called “Application scoping”.

Note

Scoped application launches require the `@amazon-connect/app-manager-agent` package. For installation instructions, see the section called “Step 1: Install packages”.

Manage the active contact

AppManager binds applications to the active contact when no `scope` value is provided at launch. Therefore, when an agent handles multiple contacts simultaneously, you must keep the active contact accurate so that the correct contact context is used when launching applications without an explicit `contactId` in the `scope` option. Use `setActiveContact` to update the active contact in AppManager:

```
// Update AppManager with the currently active contact
await appManager.setActiveContact(contactId);
```

To keep AppManager automatically in sync with the contact the agent is currently viewing, subscribe to the `connect.core.onViewContact` event and call `setActiveContact` when the active contact changes:

```
// Listen for contact selection events from the CCP
connect.core.onViewContact((event) => {
    const contactId = event.contactId;
    if (contactId) {
        // Keep AppManager in sync with the currently viewed contact
        void appManager.setActiveContact(contactId).catch((error) => {
            console.error("Failed to set active contact:", error);
        });
    }
});
```

Note

The subscription to `connect.core.onViewContact` that invokes `appManager.setActiveContact` should be done after CCP initialization and before

launching any contact-scoped applications to ensure the correct contact context is always available.

Troubleshooting

This section describes common issues and resolutions when integrating AWS-managed applications.

Application launch failures

Symptoms

Applications fail to launch, display error messages, or the iframe remains blank.

Possible causes and solutions

- CCP not initialized:
  - Ensure the CCP is fully initialized before launching applications.
- Missing AppManager plugin:
  - Verify that the AppManager plugin is properly configured during CCP initialization.
  - ```
    connect.core.initCCP(container, {
        ccpUrl: instanceUrl,
        plugins: [AppManagerPlugin], // Required
    });
    ```
- Security Profile permissions:
  - Confirm the user has necessary Security Profile permissions to access the applications.
- Cross-origin issues:
  - Verify that your domain is properly allowlisted in Amazon Connect.
  - Check the browser console for Cross-Origin Resource Sharing (CORS) errors.
  - Ensure third-party cookies are not blocked in the browser settings.
- Iframe configuration:
  - Verify that iframes have the necessary permissions in `allow` and `sandbox` attributes.
  - Allow AppManager to configure the iframe. Do not manually set the `src` attribute.
- Content Security Policy (CSP):
    ```

- Ensure CSP allows communication with Amazon Connect domains either via response headers or meta tags in the HTML head.

Application not appearing in catalog

Symptoms

The `getAppCatalog()` method returns an empty array or does not include expected applications.

Possible causes and solutions

- Applications not enabled: Verify that the required applications are enabled in the Amazon Connect instance.
- Security Profile permissions: Confirm that the user has necessary Security Profile permissions to access the applications.

Connect Customer agent workspace API reference

This Connect Customer agent workspace API reference enumerates the agent events, agent requests, contact events, and contact requests that are supported by the Amazon Connect SDK.

Contents

Connect Customer agent workspace AI Agents API

The Amazon Connect SDK provides an `AIAgentsClient` which serves as an interface that your app can use to interact with AI Agent functionality. This client provides methods for real-time agent assistance use cases like checking AI Agent support, sending messages to AI Agents, retrieving message history, fetching knowledge content, and subscribing to real-time AI Agent message events.

The `AIAgentsClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
```

```
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the client as follows:

```
import { AIAgentsClient } from "@amazon-connect/ai-agents";

const aiAgentsClient = new AIAgentsClient();
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, you can provide a constructor argument:

```
import { AIAgentsClient } from "@amazon-connect/ai-agents";

const aiAgentsClient = new AIAgentsClient({
    context: sampleContext,
    provider: sampleProvider
});
```

The following sections describe the API calls for working with the AI Agents API.

Contents

- Subscribe to AI Agent message status changes in Connect Customer agent workspace
- Unsubscribe from AI Agent message status changes

Check if AI Agent is supported

Checks whether the Connect Customer instance has an AI Agent configured.

Signature

```
isAIAgentSupported(): Promise<AIAgentSupportedResult>
```

Usage

```
const result = await aiAgentsClient.isAIAgentSupported();
console.log("AI Agent supported:", result.isSupported);

// AIAgentSupportedResult Structure
{
  isSupported: boolean;
}
```

Permissions required:

```
*
```

Subscribe to AI Agent contact ready event in Connect Customer agent workspace

Subscribes a handler that fires when the session and agentic state have been resolved for a contact. Use this event to know when AI Agents features are available for a specific contact.

Signature

```
onAIAgentContactReady(handler: AIAgentContactReadyHandler, contactId: string): void
```

Usage

```
const handler: AIAgentContactReadyHandler = (data: AIAgentContactReadyEvent) => {
    console.log("AI Agent ready for contact:", data.contactId);
```

```
    console.log("Agentic enabled:", data.isAgenticEnabled);
};

aiAgentsClient.onAIAgentContactReady(handler, contactId);

// AIAgentContactReadyEvent Structure
{
  contactId: string;
  isAgenticEnabled: boolean;
}
```

Permissions required:

```
*
```

Unsubscribe from AI Agent contact ready event

Removes a previously registered on`AIAgentContactReady` handler subscription.

Signature

```
offAIAgentContactReady(handler: AIAgentContactReadyHandler, contactId: string): void
```

Usage

```
aiAgentsClient.offAIAgentContactReady(handler, contactId);
```

Permissions required:

```
*
```

Send a message to the AI Agent

Sends a message to the AI Agent. Returns a handle containing the assigned `inputId` for correlating responses.

Signature

```
sendAIAgentMessage(params: SendAIAgentMessageParams): Promise<SendAIAgentMessageResult>
```

Usage

```
const handle = await aiAgentsClient.sendAIAgentMessage({
    contactId: "contact-123",
    text: "How do I reset my password?",
    metadata: {
        RECOMMENDATION_TYPE: "DETECTED_INTENT"
    }
});

console.log("Request ID:", handle.inputId);

// SendAIAgentMessageParams Structure
{
  contactId?: string;       // Contact identifier
  text: string;             // The message text (max 512 characters)
  metadata?: {              // Optional attribution metadata (max 50 entries)
    RECOMMENDATION_TYPE?: "DETECTED_INTENT" | "SUGGESTED_MESSAGE";
    INTENT_ID?: string;
  };
}

// SendAIAgentMessageResult Structure
{
  inputId: string;          // Unique ID for correlation
}
```

Note

Responses to a sent message are delivered via the onAIAgentMessageReceived event. Use the `inputId` returned in the result to correlate incoming messages with the original request.

Note

The `text` field has a maximum length of 512 characters. The `metadata` object supports a maximum of 50 entries, with keys up to 4,096 characters and values up to 4,096 characters. Identifier fields such as `contactId` have a maximum length of 256 characters.

Permissions required:

```
*
```

List AI Agent messages for a contact

Retrieves the AI Agent message history for a contact, ordered by sequence number.

Signature

```
listAIAgentMessages(params: ListAIAgentMessagesParams):
 Promise<ListAIAgentMessagesResult>
```

Usage

```
const result = await aiAgentsClient.listAIAgentMessages({
    contactId: "contact-123"
});

console.log("Messages:", result.messages);

// ListAIAgentMessagesParams Structure
{
  contactId?: string;
}

// ListAIAgentMessagesResult Structure
{
  contactId: string;
  messages: AIAgentMessageEvent[];
}
```

Permissions required:

```
*
```

Retrieve knowledge article content

Retrieves the full content of a knowledge article, including a pre-signed URL for rendering. Use this to display the source content referenced by citations in AI Agent responses.

Signature

```
getKnowledgeContent(params: GetContentParams): Promise<ContentResult>
```

Usage

```
const content = await aiAgentsClient.getKnowledgeContent({
    knowledgeBaseId: "kb-12345",
    contentId: "content-67890"
});

console.log("Title:", content.title);
console.log("URL:", content.url);

// GetContentParams Structure
{
  knowledgeBaseId: string;  // Knowledge base ID or ARN (max 256 characters)
  contentId: string;        // Content ID or ARN (max 256 characters)
}

// ContentResult Structure
{
  contentId: string;
  contentArn: string;
  title?: string;
  contentType?: KnowledgeContentType;
  url?: string;             // Pre-signed URL, re-fetch if expired
  body?: string;            // Populated for text/plain and text/html
}
```

Permissions required:

```
*
```

Subscribe to AI Agent message events in Connect Customer agent workspace

Subscribes to the unified AI Agents message stream. The handler fires for all message types including `USER_INPUT`, `RESPONSE_CHUNK`, `INTENT`, and `PREDEFINED`. Use the `contactId` parameter to scope the subscription to a specific contact.

Signature

```
onAIAgentMessageReceived(handler: AIAgentMessageHandler, contactId?: string): void
```

Usage

```
const handler: AIAgentMessageHandler = (data: AIAgentMessageEvent) => {
    console.log("Message type:", data.message.type);
    console.log("Content:", data.message.response.value);
    if (data.message.response.citations) {
        console.log("Citations:", data.message.response.citations);
    }
};

aiAgentsClient.onAIAgentMessageReceived(handler, contactId);

// AIAgentMessageEvent Structure
{
  inputId: string;
  message: {
    type: "RESPONSE_CHUNK" | "USER_INPUT" | "INTENT" | "PREDEFINED";
    status: "IN_FLIGHT" | "RECEIVED" | "SUCCESS" | "BLOCKED" | "FAILED";
    response: {
      value: string;
      citations?: Citation[];
      isGuardrailBlocked?: boolean;
    };
    isFinalResponse: boolean;
  };
  contactId?: string;
}

// Citation Structure
{
  citationSpan: {
    beginOffsetInclusive: number;
    endOffsetExclusive: number;
  };
  referenceType: CitationReferenceType;
  contentId?: string;
  knowledgeBaseId?: string;
  sourceURL?: string;
  title?: string;
}
```

Permissions required:

```
*
```

Unsubscribe from AI Agent message events

Removes a previously registered on`AIAgentMessageReceived` handler subscription.

Signature

```
offAIAgentMessageReceived(handler: AIAgentMessageHandler, contactId?: string): void
```

Usage

```
aiAgentsClient.offAIAgentMessageReceived(handler, contactId);
```

Permissions required:

```
*
```

Subscribe to AI Agent message status changes in Connect Customer agent workspace

Subscribes to status lifecycle transitions for AI Agent messages. Use this to track whether messages have been received, succeeded, were blocked by guardrails, or failed.

Signature

```
onAIAgentMessageStatusChanged(handler: AIAgentMessageStatusHandler, contactId?:
 string): void
```

Usage

```
const handler: AIAgentMessageStatusHandler = (data: AIAgentMessageStatusEvent) => {
    console.log("Input ID:", data.inputId);
    console.log("Status:", data.status);
};

aiAgentsClient.onAIAgentMessageStatusChanged(handler, contactId);
```

```
// AIAgentMessageStatusEvent Structure
{
  inputId: string;
  status: "IN_FLIGHT" | "RECEIVED" | "SUCCESS" | "BLOCKED" | "FAILED";
  contactId?: string;
}
```

Permissions required:

```
*
```

Unsubscribe from AI Agent message status changes

Removes a previously registered on`AIAgentMessageStatusChanged` handler subscription.

Signature

```
offAIAgentMessageStatusChanged(handler: AIAgentMessageStatusHandler, contactId?:
 string): void
```

Usage

```
aiAgentsClient.offAIAgentMessageStatusChanged(handler, contactId);
```

Permissions required:

```
*
```

Connect Customer agent workspace Activity API

The Amazon Connect SDK provides a `SessionExpirationWarningClient` which serves as an interface that your app in the Connect Customer agent workspace can use to subscribe to events related to session expiration due to inactivity and to signal the Connect Customer that the agent is active.

The `SessionExpirationWarningClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the client as follows:

```
import { SessionExpirationWarningClient } from "@amazon-connect/activity";

const sessionExpirationWarningClient = new SessionExpirationWarningClient();
```

Note

You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, you can provide a constructor argument:

```
import { SessionExpirationWarningClient } from "@amazon-connect/activity";

const sessionExpirationWarningClient = new SessionExpirationWarningClient({
    context: sampleContext,
    provider: sampleProvider
});
```

The following sections describe the API calls for working with the SessionExpirationWarning API.

Contents

- Subscribe to session extension errors in Connect Customer agent workspace
- Inform Connect Customer that the agent is active

Unsubscribe a callback function from the expiration warning event

Unsubscribes a callback function from the expiration warning event that is triggered when the agent is nearing expiration due to inactivity.

Signature

```
offExpirationWarning(handler: ExpirationWarningHandler);
```

Usage

```
sessionExpirationWarningClient.offExpirationWarning(handler);
```

Unsubscribe a callback function from the expiration warning cleared event

Unsubscribes a callback function from the expiration warning cleared event that is triggered when the expiration warning is dismissed due to the agent choosing to stay logged in.

Signature

```
offExpirationWarningCleared(handler: ExpirationWarningClearedHandler);
```

Usage

```
sessionExpirationWarningClient.offExpirationWarningCleared(handler);
```

Unsubscribe a callback function from the session extension error event

Unsubscribes a callback function from the session extension error event that is triggered when the agent's session fails to update.

Signature

```
offSessionExtensionError(handler: SessionExtensionErrorHandler);
```

Usage

```
sessionExpirationWarningClient.offSessionExtensionError(handler);
```

Subscribe to session expiration warning event in Connect Customer agent workspace

Subscribes a callback function to be invoked whenever the agent's session is about to expire due to inactivity.

Signature

```
onExpirationWarning(handler: ExpirationWarningHandler);
```

Usage

```
const handler: ExpirationWarningHandler = (data: SessionExpirationInformation) => {
    console.log("Agent's session expiring at:", data);
}

sessionExpirationWarningClient.onExpirationWarning(handler);

// SessionExpirationInformation Structure
{
   expiration: number;
}
```

Subscribe to expiration warning cleared event in Connect Customer agent workspace

Subscribes a callback function to be invoked when the agent has acknowledged the expiration warning and chooses to update their session.

Signature

```
onExpirationWarningCleared(handler: ExpirationWarningClearedHandler);
```

Usage

```
const handler: ExpirationWarningClearedHandler = () => {
```

```
    console.log("My session was extended after I was warned!");
}

sessionExpirationWarningClient.onExpirationWarningCleared(handler);
```

Subscribe to session extension errors in Connect Customer agent workspace

Subscribes a callback function to be invoked when an attempt to extend the agent's session fails.

Signature

```
onSessionExtensionError(handler: SessionExtensionErrorHandler);
```

Usage

```
const handler: SessionExtensionErrorHandler = (details: SessionExtensionErrorData) => {
    console.log("Failed to extend my session!", details);
}

sessionExpirationWarningClient.onSessionExtensionError(handler);

// SessionExtensionErrorData Structure
{
    isWarningActive: boolean;
    errorDetails: Record<string, unknown>;
}
```

Inform Connect Customer that the agent is active

Sends a signal to the Connect Customer indicating that the agent is active and should not be logged out. It takes a provider as a parameter.

Signature

```
sendActivity(provider): void
```

Usage

```
import { sendActivity } from '@amazon-connect/activity';
```

```
const handleActivity = () => {
   sendActivity(sampleProvider);
};

window.addEventListener("click", handleActivity);
```

Connect Customer agent workspace Agent API

The Amazon Connect SDK provides an `AgentClient` which serves as an interface that your app in the Connect Customer agent workspace can use to subscribe to agent events and make agent data requests.

The `AgentClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { AgentClient } from "@amazon-connect/contact";

const agentClient = new AgentClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { AgentClient } from "@amazon-connect/contact";

const agentClient = new AgentClient({
    context: sampleContext,
    provider: sampleProvider
});
```

The following sections describe API calls for working with the Agent API.

Contents

- Unsubscribe a callback function when an Connect Customer agent workspace agent state changes - Deprecated
- Get the default outbound queue for the agent in Connect Customer agent workspace
- Get the queues on the agent's routing profile in Connect Customer agent workspace
- Get the current availability state of the agent in Connect Customer agent workspace
- List the contact-handling security profile permissions for the agent in Connect Customer agent workspace
- Subscribe a callback function when an Connect Customer agent workspace agent's availability state changes
- Unsubscribe a callback function when an Connect Customer agent workspace agent's availability state changes
- Subscribe a callback function when an Connect Customer agent workspace agent's next queued availability state changes
- Get the current network connection status of the agent in Connect Customer agent workspace
- Subscribe a callback function when the Connect Customer agent workspace agent's network connection status changes
- Unsubscribe a callback function when the Connect Customer agent workspace agent's network connection status changes
- Unsubscribe a callback function when an Connect Customer agent workspace agent's next queued availability state changes

Get the ARN of the agent in Connect Customer agent workspace

Returns the Amazon Resource Name(ARN) of the user that's currently logged in to the Connect Customer agent workspace.

```
async getARN(): Promise<string>
```

Permissions required:

```
User.Details.View
```

Get the limit of contacts for the agent in Connect Customer agent workspace

Returns a map of `ChannelType`-to-number indicating how many concurrent contacts can an Connect Customer agent workspace agent have on a given channel. 0 represents a disabled channel.

```
async getChannelConcurrency(): Promise<AgentChannelConcurrencyMap>
```

Permissions required:

```
User.Configuration.View
```

Get the extension of the agent in Connect Customer agent workspace

Returns phone number of the agent currently logged in to the Connect Customer agent workspace. This is the phone number that is dialed by the Connect Customer to connect calls to the agent for incoming and outgoing calls if soft phone is not enabled.

```
async getExtension(): Promise<string | null>
```

Permissions required:

```
User.Configuration.View
```

Get the name of the agent in Connect Customer agent workspace

Returns the name of the user that's currently logged in to the Connect Customer agent workspace.

```
async getName(): Promise<string>
```

Permissions required:

```
User.Details.View
```

Get the routing profile of the agent in Connect Customer agent workspace

Returns the routing profile of the agent currently logged in to the Connect Customer agent workspace. The routing profile contains the following fields:

- `channelConcurrencyMap`: See agent.Get the limit of contacts for the agent in Connect Customer agent workspace for more info.
- `defaultOutboundQueue`: The default queue which should be associated with outbound contacts. See queues for details on properties.
- `name`: The name of the routing profile.
- `queues`: The queues contained in the routing profile. Each queue object has the following properties:
  - `name`: The name of the queue.
  - `queueARN`: The ARN of the queue.
  - `queueId`: Alias for queueARN.
- `routingProfileARN`: The routing profile ARN.
- `routingProfileId`: Alias for `routingProfileARN`.

```
async getRoutingProfile(): Promise<AgentRoutingProfile>
```

Permissions required:

```
User.Configuration.View
```

Get the current state of the agent in Connect Customer agent workspace - Deprecated

> **Note**
>
> This API is deprecated, use getAvailabilityState() instead.

Returns the Connect Customer agent workspace agent's current `AgentState` object indicating their availability state type. This object contains the following fields:

- `agentStateARN`: The agent's current state ARN.
- `name`: The name of the agent's current availability state.
- `startTimestamp`: A `Date` object that indicates when the state was set.
- `type`: The agent's current availability state type, as per the `AgentStateType` enumeration. The different values are as follows:
  - `routable`
  - `not_routable`
  - `after_call_work`
  - `system`
  - `error`
  - `offline`

```
async getState(): Promise<AgentState>
```

Permissions required:

```
User.Status.View
```

Get all the availability states configured for the current agent in Connect Customer agent workspace

Get all the availability states configured for the current agent.

Signature

```
listAvailabilityStates(): Promise<AgentState[]>
```

Usage

```
const availabilityStates: AgentState[] = await agentClient.listAvailabilityStates();
```

Output - AgentState

Parameter	Type	Description
agentStateARN	string	Amazon Reference Number of agent state
type	string	It could be "routable" | "not_routable" | "after_call_work" | "system" | "error" | "offline"
name	string	Name of the agent state like `Available` or `Offline`
startTimestamp	Date	A `Date` object that indicates when the state was set.

Permissions required:

```
User.Configuration.View
```

Get the list of Quick Connect endpoints associated with a given queue in Connect Customer agent workspace

Get the list of Quick Connect endpoints associated with the given queue(s). Optionally you can pass in a parameter to override the default max-results value of 500.

Signature

```
listQuickConnects(
    queueARNs: QueueARN | QueueARN[],
    options?: ListQuickConnectsOptions,
  ): Promise<ListQuickConnectsResult>
```

Usage

```
const routingProfile: AgentRoutingProfile = await agentClient.getRoutingProfile();
const quickConnects: ListQuickConnectsResult = await
 agentClient.listQuickConnects(routingProfile.queues[0].queueARN);
```

Input

Parameter	Type	Description
queueARNs *Required*	string | string[]	One or more Queue ARNs for which the Queue Connects need to be retrieved
options.maxResults	number	The maximum number of results to return per page. The default value is 500
options.nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request to retrieve the next set of results.

Output - ListQuickConnectsResult

Parameter	Type	Description
quickConnects	QuickConnect[]	Its either AgentQuickConnect or QueueQuickConnect or PhoneNumberQuickConnect which contains endpointARN and name. Additionally PhoneNumberQuickConnect contains phoneNumber
nextToken	string	If there are additional results, this is the token for the next set of results.

Permissions required:

```
User.Configuration.View
```

Unsubscribe from agent enabled channel list changes in Connect Customer agent workspace

Unsubscribes from EnabledChannelListChanged event.

Signature

```
offEnabledChannelListChanged(handler: EnabledChannelListChangedHandler): void
```

Unsubscribe from agent routing profile changes in Connect Customer agent workspace

Unsubscribes from RoutingProfileChanged event.

Signature

```
offRoutingProfileChanged(handler: RoutingProfileChangedHandler): void
```

Subscribe to agent enabled channel list changes in Connect Customer agent workspace

Creates a subscription for EnabledChannelListChanged event. This gets triggered when an Agent's enabled channels get updated.

Signature

```
const handler: EnabledChannelListChangedHandler = async (data:
 EnabledChannelListChanged) => {
    console.log("Agent channel list change occurred! " + data);
};

agentClient.onEnabledChannelListChanged(handler);

// EnabledChannelListChanged Structure
{
    enabledChannels: AgentRoutingProfileChannelTypes[];
    previous?: {
        enabledChannels: AgentRoutingProfileChannelTypes[];
    };
}

// AgentRoutingProfileChannelTypes
type AgentRoutingProfileChannelTypes = "VOICE" | "CHAT" | "TASK" | "EMAIL";
```

Permissions required:

```
*
```

Subscribe to agent routing profile changes in Connect Customer agent workspace

Creates a subscription for RoutingProfileChanged event. This gets triggered when an Agent's routing profile gets updated.

Signature

```
const handler: RoutingProfileChangedHandler = async (data: AgentRoutingProfileChanged)
 => {
    console.log("Agent routing profile change occurred! " + data);
};

agentClient.onRoutingProfileChanged(handler);

// AgentRoutingProfileChanged Structure
{
    routingProfile: AgentRoutingProfile;
    previous?: {
        routingProfile: AgentRoutingProfile;
    };
}
```

Permissions required:

```
*
```

Set the agent state with the given agent state ARN in Connect Customer agent workspace

Set the agent state with the given agent state ARN. By default, the promise resolves after the agent state is set in the backend. The response status is either `updated` or `queued` based on the current agent state.

Signature

```
 setAvailabilityState(
    agentStateARN: string,
  ): Promise<SetAvailabilityStateResult>
```

Usage

```
const availabilityStates: AgentState[] = await agentClient.listAvailabilityStates();
const availabilityStateResult:SetAvailabilityStateResult = await
 agentClient.setAvailabilityState(availabilityStates[0].agentStateARN);
```

Input

Parameter	Type	Description
agentStateARN *Required*	string	The ARN of the agent state

Output - SetAvailabilityStateResult

Parameter	Type	Description
status	string	The status will be `updated` or `queued` depending on if the agent is currently handling an active contact.
current	AgentState	Reperesents the current state of the agent.
next	AgentState	It'll be the target state if the agent is handling active contact. Applicable when the status is `queued`.

Permissions required:

```
User.Configuration.Edit
```

Set the agent state with the given agent state name in Connect Customer agent workspace

Sets the agent state with the given agent state name. The promise resolves after the agent state is set in the backend. The response status is either `updated` or `queued` based on the current agent state.

Signature

```
setAvailabilityStateByName(
    agentStateName: string,
  ): Promise<SetAvailabilityStateResult>
```

Usage

```
const availabilityStateResult: SetAvailabilityStateResult = await
 agentClient.setAvailabilityStateByName('Available');
```

Input

Parameter	Type	Description
agentStateName *Required*	string	The name of the agent state

Output - SetAvailabilityStateResult

Parameter	Type	Description
status	string	The status will be "updated" or "queued" depends on if the agent is currently handling an active contact.
current	AgentState	Reperesents the current state of the agent.
next	AgentState	It'll be the target state if the agent is handling active contact. Applicable when the status is queued

Permissions required:

```
User.Configuration.Edit
```

Sets the agent state to Offline in Connect Customer agent workspace

Sets the agent state to Offline. The promise resolves after the agent state is set in the backend.

Signature

```
  setOffline(): Promise<SetAvailabilityStateResult>
```

Usage

```
const availabilityStateResult: SetAvailabilityStateResult = await
 agentClient.setOffline();
```

Output - SetAvailabilityStateResult

Parameter	Type	Description
status	string	The status will be updated or queued depending on if the agent is currently handling an active contact.
current	AgentState	Represents the current state of the agent.
next	AgentState	It'll be the target state if the agent is handling active contact. Applicable when the status is queued.

Permissions required:

```
User.Configuration.Edit
```

Subscribe a callback function when an Connect Customer agent workspace agent state changes - Deprecated

> **Note**
>
> This API is deprecated, use onAvailabilityStateChanged() instead.

Subscribes a callback function to-be-invoked whenever an agent state changed event occurs in the Connect Customer agent workspace.

Signature

```
onStateChanged(handler: AgentStateChangedHandler)
```

Usage

```
const handler: AgentStateChangedHandler = async (data: AgentStateChangedEventData) => {
    console.log("Agent state change occurred! " + data);
};

agentClient.onStateChanged(handler);

// AgentStateChangedEventData Structure
{
  state: string;
  previous: {
    state: string;
  };
}
```

Permissions required:

```
User.Status.View
```

Unsubscribe a callback function when an Connect Customer agent workspace agent state changes - Deprecated

> **Note**
>
> This API is deprecated, use offAvailabilityStateChanged() instead.

Unsubscribes the callback function from the agent stated change event in the Connect Customer agent workspace.

Signature

```
offStateChanged(handler: AgentStateChangedHandler)
```

Usage

```
agentClient.offStateChanged(handler);
```

Get the default outbound queue for the agent in Connect Customer agent workspace

Returns the default outbound queue for the agent currently logged in to the Connect Customer agent workspace. This is the queue used for outbound contacts the agent originates when no other queue is specified. The returned `Queue` contains `name`, `queueARN`, and `queueId`.

```
async getDefaultOutboundQueue(): Promise<Queue>
```

Permissions required:

```
*
```

Get the queues on the agent's routing profile in Connect Customer agent workspace

Returns the list of queues on the routing profile of the agent currently logged in to the Connect Customer agent workspace. Each `Queue` contains `name`, `queueARN`, and `queueId`.

```
async getRoutingProfileQueues(): Promise<Queue[]>
```

Permissions required:

```
*
```

Get the current availability state of the agent in Connect Customer agent workspace

Returns the current availability state of the agent currently logged in to the Connect Customer agent workspace, along with the next pending state if one is queued because the agent is handling an active contact.

This API supersedes getState(), which is now deprecated.

```
async getAvailabilityState(): Promise<GetAvailabilityStateResult>
```

Output - GetAvailabilityStateResult

Parameter	Type	Description
agentStateARN	string (optional)	The ARN of the agent's current availability state.

Parameter	Type	Description
name	string	The name of the agent's current availability state.
type	AgentStateType	The agent's current availability state type. One of `routable`, `not_routable` , `after_call_work` , `system`, `error`, or `offline`.
startTimestamp	Date (optional)	The time at which the agent entered this state.
nextState	AgentState (optional)	The next state the agent will transition to once all active contacts are cleared, when one is queued.

Permissions required:

```
*
```

List the contact-handling security profile permissions for the agent in Connect Customer agent workspace

Returns the list of contact-handling security profile permissions granted to the agent currently logged in to the Connect Customer agent workspace. Each permission is returned as a string identifier (for example, `outboundCall`, `outboundEmail`). Apps can use this to gate contact-handling functionality based on what the agent is authorized to do.

```
async listSecurityProfilePermissions(): Promise<string[]>
```

Permissions required:

```
*
```

Subscribe a callback function when an Connect Customer agent workspace agent's availability state changes

Subscribes a callback function to be invoked whenever the agent's availability state changes in the Connect Customer agent workspace.

This API supersedes onStateChanged(), which is now deprecated.

Signature

```
onAvailabilityStateChanged(handler: AvailabilityStateChangedHandler)
```

Usage

```
const handler: AvailabilityStateChangedHandler = async (data:
 AgentAvailabilityStateChanged) => {
    console.log("Agent availability state changed! " + data.state.name);
};

agentClient.onAvailabilityStateChanged(handler);

// AgentAvailabilityStateChanged Structure
{
  state: AgentState;
  previous?: {
    state: AgentState;
  };
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when an Connect Customer agent workspace agent's availability state changes

Unsubscribes the callback function from the agent availability state change event in the Connect Customer agent workspace.

Signature

```
offAvailabilityStateChanged(handler: AvailabilityStateChangedHandler)
```

Usage

```
agentClient.offAvailabilityStateChanged(handler);
```

Subscribe a callback function when an Connect Customer agent workspace agent's next queued availability state changes

Subscribes a callback function to be invoked whenever the agent's next queued availability state changes in the Connect Customer agent workspace. The next state is the availability state that is queued to apply once all of the agent's active contacts are cleared. This event fires when a new next state is queued, when a queued next state is replaced, or when a queued next state is cleared.

Signature

```
onNextAvailabilityStateChanged(handler: NextAvailabilityStateChangedHandler)
```

Usage

```
const handler: NextAvailabilityStateChangedHandler = async (data:
 NextAgentAvailabilityStateChanged) => {
```

```
    console.log("Next availability state changed! " + data.nextState?.name);
};

agentClient.onNextAvailabilityStateChanged(handler);

// NextAgentAvailabilityStateChanged Structure
{
  nextState: AgentState | null;
  previous?: {
    nextState: AgentState;
  };
}
```

Permissions required:

```
*
```

Get the current network connection status of the agent in Connect Customer agent workspace

Returns the current network connection health status of the agent's connection to Connect Customer backend services.

```
async getNetworkConnectionStatus(): Promise<NetworkConnectionStatusChanged>
```

Output - NetworkConnectionStatusChanged

Parameter	Type	Description
status	NetworkConnectionStatus	The connection health status. One of `"connected"`, `"connecting"`, `"disconnected"`, or `"failed"`.

Parameter	Type	Description
timestamp	number	Epoch milliseconds when the status was reported.

Permissions required:

```
*
```

Subscribe a callback function when the Connect Customer agent workspace agent's network connection status changes

Subscribes a callback function to be invoked whenever the agent's network connection status changes in the Connect Customer agent workspace.

Signature

```
onNetworkConnectionStatusChanged(handler: NetworkConnectionStatusChangedHandler)
```

Usage

```
const handler: NetworkConnectionStatusChangedHandler = async (data:
 NetworkConnectionStatusChanged) => {
    console.log("Network connection status changed! " + data.status);
};

agentClient.onNetworkConnectionStatusChanged(handler);

// NetworkConnectionStatusChanged Structure
{
  status: NetworkConnectionStatus;
  timestamp: number;
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when the Connect Customer agent workspace agent's network connection status changes

Unsubscribes the callback function from the network connection status change event in the Connect Customer agent workspace.

Signature

```
offNetworkConnectionStatusChanged(handler: NetworkConnectionStatusChangedHandler)
```

Usage

```
agentClient.offNetworkConnectionStatusChanged(handler);
```

Unsubscribe a callback function when an Connect Customer agent workspace agent's next queued availability state changes

Unsubscribes the callback function from the agent next availability state change event in the Connect Customer agent workspace.

Signature

```
offNextAvailabilityStateChanged(handler: NextAvailabilityStateChangedHandler)
```

Usage

```
agentClient.offNextAvailabilityStateChanged(handler);
```

Connect Customer agent workspace AppController API

The Amazon Connect SDK provides an `AppControllerClient` to control applications in the Connect Customer agent workspace.

The `AppControllerClient` accepts an optional argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the AppControllerClient as follows:

```
import { AppControllerClient } from "@amazon-connect/app-controller";

const appControllerClient = new AppControllerClient({ provider });
```

The following sections describe API calls for working with the App Controller API.

Contents

- Launch an application in Connect Customer agent workspace

Close an application in Connect Customer agent workspace

Closes the application for the given application instance ID in the Connect Customer agent workspace.

Signature

```
closeApp(instanceId: string): Promise<void>
```

Usage

```
await appControllerClient.closeApp(appInstanceId);
```

Input

Parameter	Type	Description
appInstanceId *Required*	string	The instance ID of the application

Permissions required:

```
*
```

Focus an application in Connect Customer agent workspace

Brings the application into focus in the Connect Customer agent workspace for the given application instance ID.

Signature

```
focusApp(instanceId: string): Promise<AppFocusResult>
```

Usage

```
const applicationFocusResult: AppFocusResult = await
 appControllerClient.focusApp(appInstanceId);
```

Input

Parameter	Type	Description
appInstanceId *Required*	string	The instance ID of the application

Output - AppFocusResult

Parameter	Type	Description
instanceId	string	The AmazonResourceName (ARN) of the application
result	"queued" | "completed" | "failed"	Indicates if the request is queued, completed or failed

Permissions required:

```
*
```

Get application information in Connect Customer agent workspace

Returns the application information for the given application instance ID in the Connect Customer agent workspace.

Signature

```
getApp(instanceId: string): Promise<AppInfo>
```

Usage

```
const applicationInfo: AppInfo = await appControllerClient.getApp(appInstanceId);
```

Input

Parameter	Type	Description
appInstanceId *Required*	string	The instance ID of the application

Output - AppInfo

Parameter	Type	Description
instanceId	string	Unique ID of the application instance
config	Config	The configuration of the application
startTime	Date	Time when the application is launched
state	AppState	Current state of the applicati on
appCreateOrder	number	Sequentially incremented counter upon new application launches
accessUrl	string	Access URL of the application
parameters	AppParameters | undefined	Key value pair of parameters passed to the application
launchKey	string	A unique id to avoid duplicate application being launched

Parameter	Type	Description
		with multiple invocation of launchApp API
scope	ContactScope | IdleScope	Indicates if the application is launched with idle i.e when there are no contacts or launched during an active contact

Permissions required:

```
*
```

Get the application catalog in Connect Customer agent workspace

Returns all the applications that are available in the Connect Customer agent workspace for the current logged-in user.

Signature

```
getAppCatalog(): Promise<AppConfig[]>
```

Usage

```
const applications: AppConfig[] = await appControllerClient.getAppCatalog();
```

Output - AppConfig

Parameter	Type	Description
arn	string	The AmazonResourceName (ARN) of the application

Parameter	Type	Description
namespace	string	The immutable application namespace used at the time of registration
id	string	The unique identifier of the application
name	string	Name of the application
description	string	Description of the application
accessUrl	string	URL to access the application
initializationTimeout	number	The maximum time allowed in milliseconds to establish a connection with the workspace
contactHandling.scope	PER_CONTACT | CROSS_CONTACTS	Indicates whether the application refreshes for each contact

Permissions required:

```
*
```

Get the application configuration in Connect Customer agent workspace

Returns the application configuration for the given application ARN in the Connect Customer agent workspace.

Signature

```
getAppConfig(appArn: string): Promise<AppConfig>
```

Usage

```
const applicationConfig: AppConfig = await appControllerClient.getAppConfig(arn);
```

Input

Parameter	Type	Description
arn *Required*	string	The ARN of the application

Output - AppConfig

Parameter	Type	Description
arn	string	The AmazonResourceName (ARN) of the application
namespace	string	The immutable application namespace used at the time of registration
id	string	The unique identifier of the application
name	string	Name of the application
description	string	Description of the application
accessUrl	string	URL to access the application
initializationTimeout	number	The maximum time allowed to establish a connection with the workspace

Parameter	Type	Description
contactHandling.scope	PER_CONTACT | CROSS_CONTACTS	Indicates whether the application refreshes for each contact

Permissions required:

```
*
```

Get all active application information in Connect Customer agent workspace

Returns the application information for all active application instances in the Connect Customer agent workspace.

Signature

```
getApps(): Promise<AppInfo[]>
```

Usage

```
const applicationInfo: AppInfo[] = await appControllerClient.getApps();
```

Output - AppInfo

Parameter	Type	Description
instanceId	string	Unique ID of the application instance
config	Config	The configuration of the application

Parameter	Type	Description
startTime	Date	Time when the application is launched
state	AppState	Current state of the applicati on
appCreateOrder	number	Sequentially incremented counter upon new application launches
accessUrl	string	Access URL of the application
parameters	AppParameters | undefined	Key value pair of parameters passed to the application
launchKey	string	A unique id to avoid duplicate application being launched with multiple invocation of launchApp API
scope	ContactScope | IdleScope	Indicates if the applicati on is launched with idle i.e when there are no contacts or launched during an active contact

Permissions required:

```
*
```

Launch an application in Connect Customer agent workspace

Launch the application in the agent workspace for the given application ARN or name. It supports optional launch options to fine tune the launch behavior.

For details on how the `options.scope` value controls an application's relationship to contacts, see the section called “Application scoping”.

Signature

```
launchApp(arnOrName: string, options?: AppLaunchOptions): Promise<AppInfo>
```

Usage

```
const applicationsConfig: AppConfig[] = await appControllerClient.getAppCatalog();
const appInfo: AppInfo = await
 appControllerClient.launchApp(applicationsConfig[0].arn);
```

Input

Parameter	Type	Description
arnOrName *Required*	string	The ARN or name of the application
options.parameters	AppParameters	Key value pair of parameters passed to the application
options.launchKey	string	A unique id to avoid duplicate application being launched with multiple invocation of launchApp API
options.openInBackground	boolean	If set to true, the application won't be set to focus after its launched
options.scope	ContactScope | IdleScope	Indicates if the applicati on is launched with idle i.e when there are no contacts or launched during an active contact

Output - AppInfo

Parameter	Type	Description
instanceId	string	Unique ID of the application instance
config	Config	The configuration of the application
startTime	Date	Time when the application is launched
state	AppState	Current state of the applicati on
appCreateOrder	number	Sequentially incremented counter upon new application launches
accessUrl	string	Access URL of the application
parameters	AppParameters | undefined	Key value pair of parameters passed to the application
launchKey	string	A unique id to avoid duplicate application being launched with multiple invocation of launchApp API
scope	ContactScope | IdleScope	Indicates if the applicati on is launched with idle i.e when there are no contacts or launched during an active contact

Permissions required:

```
*
```

Connect Customer agent workspace Contact API

The Amazon Connect SDK provides an `ContactClient` which serves as an interface that your app in the Connect Customer agent workspace can use to subscribe to contact events and make contact data requests.

The `ContactClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
 };
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { ContactClient } from "@amazon-connect/contact";
const contactClient = new ContactClient({ provider });
```

Note

You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }` . This is the recommended option.

Alternatively, providing a constructor argument:

```
import { ContactClient } from "@amazon-connect/contact";
```

```
        const contactClient = new ContactClient({
            context: sampleContext,
            provider: sampleProvider
    });
```

The following sections describe API calls for working with the Contact API.

Contents

- Check if contact is in preview mode in Connect Customer agent workspace
- List all contacts for the current agent in Connect Customer agent workspace
- List all participants for a contact in Connect Customer agent workspace
- Subscribe a callback function when an Connect Customer agent workspace contact is missed
- Unsubscribe a callback function when an Connect Customer agent workspace contact is missed
- Unsubscribe from incoming contact events in Connect Customer agent workspace
- Subscribe to incoming contact events in Connect Customer agent workspace
- Subscribe to participant added events in Connect Customer agent workspace
- Unsubscribe from participant added events in Connect Customer agent workspace
- Subscribe to participant disconnected events in Connect Customer agent workspace
- Unsubscribe from participant disconnected events in Connect Customer agent workspace
- Subscribe to participant state change events in Connect Customer agent workspace
- Subscribe a callback function when an Connect Customer agent workspace contact starts ACW
- Unsubscribe a callback function when an Connect Customer agent workspace contact starts ACW
- Transfer a contact to another agent in Connect Customer agent workspace
- Reject the incoming contact for the given contactId in Connect Customer agent workspace
- Disconnect the agent from the given contact in Connect Customer agent workspace
- Check whether auto-accept is enabled for the given contact in Connect Customer agent workspace
- Subscribe a callback function when an Connect Customer agent workspace contact turns to Connecting state
- Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Connecting state
- Subscribe a callback function when an Connect Customer agent workspace contact turns to Error state
- Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Error state
- Subscribe a callback function when an Connect Customer agent workspace contact turns to Pending state
- Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Pending state

Accept the incoming contact for the given contactId in Connect Customer agent workspace

Accept the incoming contact for the given contactId.

Signature

```
accept(contactId: string): Promise<void>
```

Usage

```
await contactClient.accept(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact that needs to be accepted.

Permissions required:

```
Contact.Details.Edit
```

Add another participant to a contact in Connect Customer agent workspace

Add another participant to the contact. Multi-party only works for Voice at this time. For Voice, the existing participants will be put on hold when a new participant is added.

Signature

```
addParticipant(
    contactId: string,
    quickConnect: QuickConnect,
```

```
  ): Promise<AddParticipantResult>
```

Usage

```
const routingProfile: AgentRoutingProfile = await agentClient.getRoutingProfile();
const quickConnectResult: ListQuickConnectsResult = await
 agentClient.listQuickConnects(routingProfile.queues[0].queueARN);
const quickConnect: QuickConnect = quickConnectResult.quickConnects[1];
const addParticipantResult: AddParticipantResult = await
 contactClient.addParticipant(contactId, quickConnect);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact to which a participant needs to be added.
quickConnect *Required*	QuickConnect	Its either AgentQuickConnect or QueueQuickConnect or PhoneNumberQuickConnect which contains endpointA RN and name. Additionally PhoneNumberQuickConnect contains phoneNumber

Output - AddParticipantResult

Parameter	Type	Description
participantId	string	The id of the newly added participant

Permissions required:

```
Contact.Details.Edit
```

Clears the contact for the given contactId in Connect Customer agent workspace

Clears the contact for the given contactId.

Signature

```
clear(contactId: string): Promise<void>
```

Usage

```
await contactClient.clear(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact that needs to be cleared.

Permissions required:

```
Contact.Details.Edit
```

Creates a subscription whenever a contact cleared event occurs in Connect Customer agent workspace

It creates a subscription whenever a contact cleared event occurs in Connect Customer agent workspace. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

onCleared(handler: ContactClearedHandler, contactId?: string)

Usage

```
const handler: ContactClearedHandler = async (data: ContactCleared) => {
    console.log("Contact cleared occurred! " + data);
};

contactClient.onCleared(handler);

// ContactCleared Structure
{
    contactId: string;
}
```

Permissions required:

```
Contact.Details.View
```

Unsubscribes the callback function from the contact cleared event in Connect Customer agent workspace

Unsubscribes the callback function from the contact cleared event in Connect Customer agent workspace.

Signature

```
       offCleared(handler: ContactClearedHandler, contactId?: string)
```

Usage

```
 contactClient.offCleared(handler);
```

Subscribe a callback function when an Connect Customer agent workspace contact is connected

Subscribes a callback function to-be-invoked whenever a contact Connected event occurs in the Connect Customer agent workspace. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onConnected(handler: ContactConnectedHandler, contactId?: string)
```

Usage

```
const handler: ContactConnectedHandler = async (data: ContactConnected) => {
    console.log("Contact Connected occurred! " + data);
};

contactClient.onConnected(handler);

// ContactConnected Structure
{
    contactId: string;
}
```

Permissions required:

```
Contact.Details.View
```

Unsubscribe a callback function when an Connect Customer agent workspace contact is connected

Unsubscribes the callback function from Connected event in the Connect Customer agent workspace.

Signature

```
offConnected(handler: ContactConnectedHandler)
```

Usage

```
contactClient.offConnected(handler);
```

Disconnect a participant from a contact in Connect Customer agent workspace

Disconnects a specific participant from the contact.

Signature

```
disconnectParticipant(participantId: string): Promise<void>
```

Usage

```
await contactClient.disconnectParticipant("participant-456");
console.log("Participant disconnected");
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant to disconnect

Permissions required:

```
*
```

Engage the preview contact for the given contactId in Connect Customer agent workspace

When an agent is previewing a preview contact, this API will actually initiate the outbound dial to the end customer, ending the preview experience.

Signature

```
engagePreviewContact(contactId: string): Promise<AddParticipantResult>
```

Usage

```
const addParticipantResult: AddParticipantResult = await
 contactClient.engagePreviewContact(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact which is being previewed by the agent to which a participant needs to be added.

Output - AddParticipantResult

Parameter	Type	Description
participantId	string	The id of the newly added participant

Permissions required:

```
*
```

Get specific attributes for a contact in Connect Customer agent workspace

Returns the requested attribute associated with the contact in the Connect Customer agent workspace.

```
async getAttribute(
  contactId: string,
  attribute: string,
): Promise<string | undefined>
```

Permissions required:

```
Contact.Attributes.View
```

Get the attributes of a contact in Connect Customer agent workspace

Returns a map of the attributes associated with the contact in the Connect Customer agent workspace. Each value in the map has the following shape: `{ name: string, value: string }`.

```
// example { "foo": { "name": "foo", "value": "bar" } }
```

```
getAttributes(
  contactId: string,
  attributes: ContactAttributeFilter,
): Promise<Record<string, string>>
```

```
ContactAttributeFilter is either string[] of attributes or '*'
```

Permissions required:

```
Contact.Attributes.View
```

Get the type of contact in Connect Customer agent workspace

Get the type of the contact in Connect Customer agent workspace. This indicates what type of media is carried over the connections of the contact.

Signature

```
 getChannelType(contactId: string): Promise<ContactChannelType>
```

Usage

```
const contactType: ContactChannelType = await contactClient.getChannelType(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact.

Output - ContactChannelType

Parameter	Type	Description
type	string	The possible values are `voice, queue_callback, chat, task, email`
subtype	string	For the types `voice` & `queue_callback` , it will be `connect:Telephony | connect:WebRTC` . For the type `chat`, it will be `connect:Chat | connect:SMS |connect:Apple | connect:Guide` . For the type `task`, it will be `connect:Task` . For the type `email`, it will be `connect:Email` .

Permissions required:

```
Contact.Details.View
```

Get detailed contact information in Connect Customer agent workspace

Retrieves detailed information for a specific contact by its ID.

Signature

```
getContact(contactId: string): Promise<ContactData>
```

Usage

```
const contactData = await contactClient.getContact("contact-123");
console.log(`Contact type: ${contactData.type}`);
console.log(`Queue: ${contactData.queue.name}`);
```

Input

Parameter	Type	Description
contactId *Required*	string	The unique identifier for the contact

Output - ContactData

The ContactData interface includes:

- `contactId`: string - Unique identifier for the contact
- `type`: ContactType - Type of contact (voice, chat, task)
- `subtype`: string - Subtype providing additional classification
- `initialContactId?`: string - Initial contact ID for transferred contacts
- `queue`: Queue - Queue information

Permissions required:

```
*
```

Get the initial ID of the contact in Connect Customer agent workspace

Returns the original (initial) contact id from which this contact was transferred in the Connect Customer agent workspace, or none if this is not an internal Connect transfer. This is typically a contact owned by another agent, thus this agent will not be able to manipulate it. It is for reference and association purposes only, and can be used to share data between transferred contacts externally if it is linked by originalContactId.

```
async getInitialContactId(contactId: string): Promise<string | undefined>
```

Permissions required:

```
Contact.Details.View
```

Get specific participant information in Connect Customer agent workspace

Retrieves information for a specific participant.

Signature

```
getParticipant(participantId: string): Promise<ParticipantData>
```

Usage

```
const participant = await contactClient.getParticipant("participant-456");
console.log(`Participant type: ${participant.type.value}`);
console.log(`Is initial: ${participant.isInitial}`);
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant

Output - ParticipantData

Returns a ParticipantData object with participant details.

Permissions required:

```
*
```

Get participant state in Connect Customer agent workspace

Retrieves the current state of a specific participant.

Signature

```
getParticipantState(participantId: string): Promise<ParticipantState>
```

Usage

```
const state = await contactClient.getParticipantState("participant-456");
if (state.value === "connected") {
    console.log("Participant is connected");
} else if (state.value === "hold") {
    console.log("Participant is on hold");
}
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant

Output - ParticipantState

The ParticipantState type can be:

- { value: ParticipantStateType } where ParticipantStateType includes: connecting, connected, hold, disconnected, rejected, silent_monitor, barge
- { value: "other"; actual: string } for unknown states

Permissions required:

```
*
```

Get preview configuration for the given contactId in Connect Customer agent workspace

This gets configuration information related to the preview experience.

Signature

```
getPreviewConfiguration(contactId: string): Promise<GetPreviewConfigurationResponse>
```

Usage

```
const isPreview  = await contactClient.isPreviewMode(contactId);
if (isPreview) {
    const {autoDialTimeout, canDiscardPreview} = await
 contactClient.getPreviewConfiguration(contactId);
```

```
}
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact which is in preview.

Output - GetPreviewConfigurationResponse

Parameter	Type	Description
autoDialTimeout	number	The number of seconds the agent has to preview the contact before the auto-dial triggers.
canDiscardPreview	boolean	Whether the agent has permission to discard the contact during preview. Use this to control whether the agent should be presented the option to discard the contact without dialing the end customer.

Permissions required:

```
*
```

Get the queue of the contact in Connect Customer agent workspace

Returns the queue associated with the contact in the Connect Customer agent workspace. The `Queue` object has the following fields:

- `name`: The name of the queue.
- `queueARN`: The ARN of the queue.
- `queueId`: Alias for `queueARN`.

```
async getQueue(contactId: string): Promise<Queue>
```

Permissions required:

```
Contact.Details.View
```

Get the timestamp of the contact in Connect Customer agent workspace

Returns a `Date` object with the timestamp associated with when the contact was placed in the queue in the Connect Customer agent workspace.

```
async getQueueTimestamp(contactId: string): Promise<Date | undefined>
```

Permissions required:

```
Contact.Details.View
```

Get the duration of the contact state in Connect Customer agent workspace

Returns the duration of the contact state in milliseconds relative to local time, in the Connect Customer agent workspace. This takes into account time skew between the JS client and the Connect Customer backend servers.

```
async getStateDuration(contactId: string): Promise<number>
```

Permissions required:

```
Contact.Details.View
```

Check if contact is in preview mode in Connect Customer agent workspace

Returns whether the contact is being previewed. During this time, calling engagePreviewContact will trigger the outbound dial to the end customer and end preview mode.

Signature

```
isPreviewMode(contactId: string): Promise<boolean>
```

Usage

```
const isPreviewMode = await contactClient.isPreviewMode(currentContactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact.

Permissions required:

```
*
```

List all contacts for the current agent in Connect Customer agent workspace

Lists all contacts for the current agent.

Signature

```
listContacts(): Promise<ListContactsResult>
```

Usage

```
const contacts = await contactClient.listContacts();
console.log(`Active contacts: ${contacts.length}`);
contacts.forEach((contact) => {
    console.log(`Contact ${contact.contactId}: ${contact.type}`);
});
```

Output - ListContactsResult

Returns an array of contact data objects (currently typed as CoreContactData[]).

Permissions required:

```
*
```

List all participants for a contact in Connect Customer agent workspace

Retrieves all participants associated with a specific contact.

Signature

```
listParticipants(contactId: string): Promise<ParticipantData[]>
```

Usage

```
const participants = await contactClient.listParticipants("contact-123");
participants.forEach((p) => {
    console.log(`Participant ${p.participantId}: ${p.type.value}`);
    if (p.isSelf) {
        console.log("This is the current user");
    }
});
```

Input

Parameter	Type	Description
contactId *Required*	string	The unique identifier for the contact

Output - ParticipantData[]

The ParticipantData interface includes:

- `participantId`: string - Unique identifier for the participant
- `contactId`: string - Contact this participant belongs to

- `type`: ParticipantType - Type of participant (agent, outbound, inbound, monitoring, other)
- `isInitial`: boolean - Whether this is the initial participant
- `isSelf`: boolean - Whether this participant is associated with the current user

Permissions required:

```
*
```

Subscribe a callback function when an Connect Customer agent workspace contact is missed

Subscribes a callback function to-be-invoked whenever a contact missed event occurs in the Connect Customer agent workspace. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onMissed(handler: ContactMissedHandler, contactId?: string)
```

Usage

```
const handler: ContactMissedHandler = async (data: ContactMissed) => {
    console.log("Contact missed occurred! " + data);
};

contactClient.onMissed(handler);

// ContactMissed Structure
{
  contactId: string;
}
```

Permissions required:

```
Contact.Details.View
```

Unsubscribe a callback function when an Connect Customer agent workspace contact is missed

Unsubscribes the callback function from the contact missed event.

Signature

```
offMissed(handler: ContactMissedHandler, contactId?: string)
```

Usage

```
contactClient.offMissed(handler);
```

Unsubscribe from incoming contact events in Connect Customer agent workspace

Unsubscribes the callback function from the contact incoming event in Connect Customer agent workspace.

Signature

```
offIncoming(handler: ContactIncomingHandler, contactId?: string): void
```

Usage

```
contactClient.offIncoming(handler);
```

Subscribe to incoming contact events in Connect Customer agent workspace

Creates a subscription whenever a new incoming event occurs in the Connect Customer agent workspace.

Signature

```
onIncoming(handler: ContactIncomingHandler, contactId?: string): void
```

Usage

```
const handler: ContactIncomingHandler = async (data: ContactIncoming) => {
    console.log("Contact incoming occurred! " + data);
};

contactClient.onIncoming(handler);

// ContactIncoming Structure
{
    contactId: string;
    initialContactId: string | undefined;
    type: ContactChannelType["type"];
    subtype: ContactChannelType["subtype"];
}
```

Permissions required:

```
*
```

Subscribe to participant added events in Connect Customer agent workspace

Subscribes to participant added events. This event fires when a new participant joins a contact.

Signature

```
onParticipantAdded(handler: ParticipantAddedHandler, contactId?: string): void
```

Usage

```
const handleParticipantAdded = (event) => {
    console.log(`New participant added: ${event.participant.participantId}`);
    console.log(`Type: ${event.participant.type.value}`);
};

// Subscribe to all contacts
contactClient.onParticipantAdded(handleParticipantAdded);

// Or subscribe to a specific contact
contactClient.onParticipantAdded(handleParticipantAdded, "contact-123");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantAddedHandler	Event handler function to call when participants are added
contactId	string	Optional contact ID to filter events for a specific contact

Event Structure - ParticipantAdded

The handler receives a ParticipantAdded event with:

- `participant`: ParticipantData - The participant that was added

Permissions required:

```
*
```

Unsubscribe from participant added events in Connect Customer agent workspace

Unsubscribes from participant added events.

Signature

```
offParticipantAdded(handler: ParticipantAddedHandler, contactId?: string): void
```

Usage

```
contactClient.offParticipantAdded(handleParticipantAdded);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantAddedHandler	Event handler function to remove
contactId	string	Optional contact ID to unsubscribe from specific contact events

Subscribe to participant disconnected events in Connect Customer agent workspace

Subscribes to participant disconnected events. This event fires when a participant leaves or is removed from a contact.

Signature

```
onParticipantDisconnected(handler: ParticipantDisconnectedHandler, contactId?: string):
 void
```

Usage

```
const handleParticipantDisconnected = (event) => {
    console.log(`Participant disconnected: ${event.participant.participantId}`);
};

contactClient.onParticipantDisconnected(
    handleParticipantDisconnected,
    "contact-123"
);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantDisconnectedHandler	Event handler function to call when participants disconnect
contactId	string	Optional contact ID to filter events for a specific contact

Event Structure - ParticipantDisconnected

The handler receives a ParticipantDisconnected event with:

- `participant`: ParticipantData - The participant that was disconnected

Permissions required:

```
Contact.Details.View
```

Unsubscribe from participant disconnected events in Connect Customer agent workspace

Unsubscribes from participant disconnected events.

Signature

```
offParticipantDisconnected(handler: ParticipantDisconnectedHandler, contactId?:
 string): void
```

Usage

```
contactClient.offParticipantDisconnected(handleParticipantDisconnected);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantDisconnectedHandler	Event handler function to remove
contactId	string	Optional contact ID to unsubscribe from specific contact events

Subscribe to participant state change events in Connect Customer agent workspace

Subscribes to participant state change events. This event fires when a participant's state changes (e.g., from connecting to connected, or to hold).

Signature

```
onParticipantStateChanged(handler: ParticipantStateChangedHandler, participantId?:
 string): void
```

Usage

```
    const handleStateChanged = (event) => {
    console.log(
    `Participant ${event.participantId} state changed to: ${event.state.value}`
    );

    if (event.state.value === "connected") {
    console.log("Participant is now connected");
    } else if (event.state.value === "hold") {
    console.log("Participant is now on hold");
    }
    };

    // Subscribe to all participants
    contactClient.onParticipantStateChanged(handleStateChanged);

    // Or subscribe to a specific participant
    contactClient.onParticipantStateChanged(handleStateChanged, "participant-456");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantStateChangedHandler	Event handler function to call when participant states change
participantId	string	Optional participant ID to filter events for a specific participant

Event Structure - ParticipantStateChanged

The handler receives a ParticipantStateChanged event with:

- `participantId`: string - The ID of the participant whose state changed
- `state`: ParticipantState - The new state of the participant

Permissions required:

```
Contact.Details.View
```

Subscribe a callback function when an Connect Customer agent workspace contact starts ACW

Subscribes a callback function to-be-invoked whenever a contact StartingAcw event occurs in the Connect Customer agent workspace. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onStartingAcw(handler: ContactStartingAcwHandler, contactId?: string)
```

Usage

```
const handler: ContactStartingAcwHandler = async (data: ContactStartingAcw) => {
    console.log("Contact StartingAcw occurred! " + data);
};

contactClient.onStartingAcw(handler);

// ContactStartingAcw Structure
{
  contactId: string;
}
```

Permissions required:

```
Contact.Details.View
```

Unsubscribe a callback function when an Connect Customer agent workspace contact starts ACW

Unsubscribes the callback function from the contact StartingAcw event in the Connect Customer agent workspace.

Signature

```
offStartingAcw(handler: ContactStartingAcwHandler, contactId?: string)
```

Usage

```
contactClient.offStartingAcw(handler);
```

Transfer a contact to another agent in Connect Customer agent workspace

Performs a cold transfer by transferring the given contact to another agent using a quick connect and disconnecting from the contact. The quick connect type has to be either `agent` or `queue`. Supports voice, chat, task, and email channels.

Signature

```
  transfer(
    contactId: string,
    quickConnect: AgentQuickConnect | QueueQuickConnect,
  ): Promise<void>
```

Usage

```
const routingProfile: AgentRoutingProfile = await agentClient.getRoutingProfile();
```

```
const quickConnectResult: ListQuickConnectsResult = await
 agentClient.listQuickConnects(routingProfile.queues[0].queueARN);
const quickConnect: QuickConnect = quickConnectResult.quickConnects[1];
await contactClient.transfer(contactId, quickConnect);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact to which a participant needs to be transferred.
quickConnect *Required*	QuickConnect	Its either AgentQuickConnect or QueueQuickConnect

Permissions required:

```
Contact.Details.Edit
```

Reject the incoming contact for the given contactId in Connect Customer agent workspace

Reject the incoming contact for the given contactId. The contact will not be connected to the agent and will be routed back to the queue.

Signature

```
reject(contactId: string): Promise<void>
```

Usage

```
await contactClient.reject(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the incoming contact to reject.

Permissions required:

```
*
```

Disconnect the agent from the given contact in Connect Customer agent workspace

Disconnects the currently logged-in agent from the given contact. Other participants on the contact (such as the customer) remain connected.

Signature

```
disconnectSelf(contactId: string): Promise<void>
```

Usage

```
await contactClient.disconnectSelf(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact to disconnect the agent from.

Permissions required:

```
*
```

Check whether auto-accept is enabled for the given contact in Connect Customer agent workspace

Returns whether auto-accept is enabled for the given contact. When auto-accept is enabled, an incoming contact is automatically accepted on the agent's behalf without requiring an explicit accept() call.

Signature

```
isAutoAcceptEnabled(contactId: string): Promise<boolean>
```

Usage

```
const enabled: boolean = await contactClient.isAutoAcceptEnabled(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact to check.

Permissions required:

```
*
```

Subscribe a callback function when an Connect Customer agent workspace contact turns to Connecting state

Subscribes a callback function to-be-invoked whenever a contact turns to Connecting state in the Connect Customer agent workspace. The Connecting state means the contact is being routed to

the agent and has not yet been fully assigned. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onConnecting(handler: ContactConnectingHandler, contactId?: string)
```

Usage

```
const handler: ContactConnectingHandler = async (data: ContactConnecting) => {
    console.log("Contact Connecting occurred! " + data.contactId);
};

contactClient.onConnecting(handler);

// ContactConnecting Structure
{
    contactId: string;
    initialContactId: string | undefined;
    type: string;
    subtype: string;
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Connecting state

Unsubscribes the callback function from Connecting event in the Connect Customer agent workspace.

Signature

```
offConnecting(handler: ContactConnectingHandler)
```

Usage

```
contactClient.offConnecting(handler);
```

Subscribe a callback function when an Connect Customer agent workspace contact turns to Error state

Subscribes a callback function to-be-invoked whenever a contact turns to Error state in the Connect Customer agent workspace. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onError(handler: ContactErrorHandler, contactId?: string)
```

Usage

```
const handler: ContactErrorHandler = async (data: ContactError) => {
    console.log("Contact Error occurred! " + data.contactId);
};

contactClient.onError(handler);

// ContactError Structure
{
    contactId: string;
    initialContactId: string | undefined;
    type: string;
    subtype: string;
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Error state

Unsubscribes the callback function from Error event in the Connect Customer agent workspace.

Signature

```
offError(handler: ContactErrorHandler)
```

Usage

```
contactClient.offError(handler);
```

Subscribe a callback function when an Connect Customer agent workspace contact turns to Pending state

Subscribes a callback function to-be-invoked whenever a contact turns to Pending state in the Connect Customer agent workspace. For Queued Callback contacts, the Pending state is transient and requires no action from the agent. For preview contacts, the agent must call engagePreviewContact() to proceed to the Connecting state, or disconnectSelf() to decline it. If no contact ID is provided, then it uses the context of the current contact that the 3P app was opened on.

Signature

```
onPending(handler: ContactPendingHandler, contactId?: string)
```

Usage

```
const handler: ContactPendingHandler = async (data: ContactPending) => {
    console.log("Contact Pending occurred! " + data.contactId);
};

contactClient.onPending(handler);

// ContactPending Structure
{
    contactId: string;
    initialContactId: string | undefined;
    type: string;
    subtype: string;
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when an Connect Customer agent workspace contact turns to Pending state

Unsubscribes the callback function from Pending event in the Connect Customer agent workspace.

Signature

```
offPending(handler: ContactPendingHandler)
```

Usage

```
contactClient.offPending(handler);
```

Connect Customer agent workspace Contact UI API

With the Amazon Connect SDK, your app in the Connect Customer agent workspace can intercept contact-related UI actions by using the `ContactInterceptorService` in the `@amazon-connect/contact-ui` package. Use contact interceptors to override or augment built-in contact behaviors. For example, you can present a custom transfer UI instead of the default Quick Connect menu. You can also require agents to submit a disposition code before clearing a contact.

You can instantiate the contact interceptor service as follows:

```
import { ContactInterceptorService } from '@amazon-connect/contact-ui';

const service = new ContactInterceptorService(provider);
```

> **Note**
>
> The `provider` is your `AmazonConnectProvider` instance — the same provider you use to initialize other `@amazon-connect/*` packages.

The `ContactInterceptorService` provides action-specific methods that use the `ExtensibilityService` interceptor pipeline. Each method targets one contact UI action and provides the callback context as the section called “ContactInterceptorContext”. To learn how an interceptor allows or blocks a default action, see the section called “Interceptor callback”.

To scope an interceptor to a single contact, pass its `contactId`. If you omit the `contactId`, the interceptor applies to all contacts.

Contents

Intercept the Quick Connect action in Connect Customer agent workspace

Registers an interceptor for the Quick Connect action, which runs when the user chooses the Quick Connect or transfer button. To block the built-in Quick Connect experience and present your own transfer UI, return `{ continue: false }` (or `false`).

Signature

```
addOpenQuickConnectInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  options?: RegisterInterceptorOptions
): Promise<void>

addOpenQuickConnectInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  contactId: string,
  options?: RegisterInterceptorOptions
): Promise<void>
```

Usage

```
// Block the Quick Connect menu from opening
await service.addOpenQuickConnectInterceptor(async (context) => {
  return { continue: false };
});

// Scoped to a specific contact
await service.addOpenQuickConnectInterceptor(myInterceptor, contactId);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	An asynchronous callback that receives a `{ contactId?: string }` context

Parameter	Type	Description
		and returns an `InterceptorResult` . See the section called “Interceptor callback”.
contactId *Optional*	string	The `contactId` of the contact to scope the intercept or to. If you omit this value, the interceptor applies to all contacts.
options *Optional*	`RegisterInterceptorOptions`	The registration options. If you omit this value, the service uses the default timeout of 5000 milliseconds and the default maximum consecutive block limit of 5. See the section called “Options”.

Output

`Promise<void>`

Unregister a Quick Connect interceptor in Connect Customer agent workspace

Unregisters an interceptor that you registered with `addOpenQuickConnectInterceptor`. Pass the same interceptor reference and the same `contactId` that you used at registration.

Signature

```
removeOpenQuickConnectInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  contactId?: string
): Promise<void>
```

Usage

```
await service.removeOpenQuickConnectInterceptor(myInterceptor);

// If scoped to a contact, pass the same contactId
await service.removeOpenQuickConnectInterceptor(myInterceptor, contactId);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	The same function reference that you passed to `addOpenQuickConnectInterceptor` .
contactId *Optional*	string	The contact ID that the interceptor was scoped to. Pass the same value that you used at registration. If you omit this value, the framework removes the interceptor that applies to all contacts.

Output

`Promise<void>`

Intercept the outbound dialer action in Connect Customer agent workspace

Registers an interceptor for the outbound dialer action, which runs when the user chooses the number pad or outbound dialer. To block the built-in outbound dialer and present your own dialer UI, return `{ continue: false }` (or `false`).

Signature

```
addOpenOutboundDialerInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  options?: RegisterInterceptorOptions
): Promise<void>

addOpenOutboundDialerInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  contactId: string,
  options?: RegisterInterceptorOptions
): Promise<void>
```

Usage

```
// Block the outbound dialer from opening
await service.addOpenOutboundDialerInterceptor(async (context) => {
  return { continue: false };
});

// Scoped to a specific contact
await service.addOpenOutboundDialerInterceptor(myInterceptor, contactId);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	An asynchronous callback that receives a `{ contactId?: string }` context and returns an `InterceptorResult`. See the section called “Interceptor callback”.
contactId *Optional*	string	The `contactId` of the contact to scope the intercept

Parameter	Type	Description
		or to. If you omit this value, the interceptor applies to all contacts.
options *Optional*	`RegisterInterceptorOptions`	The registration options. If you omit this value, the service uses the default timeout of 5000 milliseconds and the default maximum consecutive block limit of 5. See the section called "Options".

Output

`Promise<void>`

Unregister an outbound dialer interceptor in Connect Customer agent workspace

Unregisters an interceptor that you registered with `addOpenOutboundDialerInterceptor`. Pass the same interceptor reference and the same `contactId` that you used at registration.

Signature

```
removeOpenOutboundDialerInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  contactId?: string
): Promise<void>
```

Usage

```
await service.removeOpenOutboundDialerInterceptor(myInterceptor);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	The same function reference that you passed to `addOpenOutboundDialerInterceptor` .
contactId *Optional*	string	The contact ID that the interceptor was scoped to. Pass the same value that you used at registration. If you omit this value, the framework removes the interceptor that applies to all contacts.

Output

`Promise<void>`

Intercept the clear contact action in Connect Customer agent workspace

Registers an interceptor for the clear contact action, which runs when the user chooses to clear or end a contact. To block clearing the contact until a disposition code is submitted, return `{ continue: false }` (or `false`).

Signature

```
addClearContactInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  options?: RegisterInterceptorOptions
): Promise<void>

addClearContactInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
```

```
  contactId: string,
  options?: RegisterInterceptorOptions
): Promise<void>
```

Usage

```
// Require a disposition code before clearing the contact
await service.addClearContactInterceptor(async (context) => {
  const done = await collectDisposition(context.contactId);
  return { continue: done };
});

// Scoped to a specific contact
await service.addClearContactInterceptor(myInterceptor, contactId);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	An asynchronous callback that receives a `{ contactId?: string }` context and returns an `InterceptorResult`. See the section called “Interceptor callback”.
contactId *Optional*	string	The `contactId` of the contact to scope the intercept or to. If you omit this value, the interceptor applies to all contacts.
options *Optional*	`RegisterInterceptorOptions`	The registration options. If you omit this value, the service uses the default timeout of 5000 milliseconds and the default maximum

Parameter	Type	Description
		consecutive block limit of 5. See the section called "Options".

Output

`Promise<void>`

Unregister a clear contact interceptor in Connect Customer agent workspace

Unregisters an interceptor that you registered with `addClearContactInterceptor`. Pass the same interceptor reference and the same `contactId` that you used at registration.

Signature

```
removeClearContactInterceptor(
  interceptor: Interceptor<ContactInterceptorContext>,
  contactId?: string
): Promise<void>
```

Usage

```
await service.removeClearContactInterceptor(myInterceptor);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptor *Required*	`Interceptor<ContactInterceptorContext>`	The same function reference that you passed to `addClearContactInterceptor` .

Parameter	Type	Description
contactId *Optional*	string	The contact ID that the interceptor was scoped to. Pass the same value that you used at registration. If you omit this value, the framework removes the interceptor that applies to all contacts.

Output

`Promise<void>`

ContactInterceptorContext in Connect Customer agent workspace

Defines the invocation context that the contact interceptor methods receive.

Signature

```
type ContactInterceptorContext = {
  contactId?: string;
}
```

Properties

The following table describes the properties.

Property	Type	Description
contactId *Optional*	string	The identifier of the contact that triggered the action. The service omits this value when no specific contact initiated the action.

Connect Customer agent workspace Extensibility API

With the Amazon Connect SDK, your app in the Connect Customer agent workspace can register and manage interceptors for UI actions by using `ExtensibilityService`. `ExtensibilityService` is the generic, low-level interceptor API, and it accepts any interceptor key. Domain-specific interceptor services define typed methods and callback contexts for a specific set of actions. When a domain-specific service covers your use case, prefer it over `ExtensibilityService`.

> **Note**
>
> To intercept contact-related UI actions, use the contact-specific methods in the section called “Contact UI”. Use `ExtensibilityService` directly only when no domain-specific service covers your interceptor key.

You can instantiate the extensibility service as follows:

```
import { ExtensibilityService } from '@amazon-connect/extensibility';

const service = new ExtensibilityService(provider);
```

> **Note**
>
> The `provider` is your `AmazonConnectProvider` instance — the same provider you use to initialize other `@amazon-connect/*` packages.

Interceptors are asynchronous callbacks. They run before a default UI action runs. To allow the default action, return `{ continue: true }` (or `true`). For example, you can require a disposition code in your app before a contact is cleared. To block the default action, return `{ continue: false }` (or `false`). When you block the default action, you can provide a custom experience. For example, you can present your own Quick Connect experience instead of the built-in one.

The framework accepts any string as an interceptor key, but only keys that the UI actively invokes have an effect. The following table lists the currently supported keys.

Key	Triggers when	Context
`openQuickConnect`	The user chooses the Quick Connect or transfer button	`{ contactId?: string }`
`openOutboundDialer`	The user chooses the number pad or outbound dialer	`{ contactId?: string }`
`clearContact`	The user chooses to clear or end a contact	`{ contactId?: string }`

Contents

Register an interceptor in Connect Customer agent workspace

Registers an asynchronous interceptor callback for the specified action key. When the UI triggers the action, your interceptor runs before the default behavior.

Signature

```
addInterceptor<T>(
  interceptorKey: InterceptorKey,
  interceptor: Interceptor<T>,
  options?: RegisterInterceptorOptions
): Promise<void>

addInterceptor<T>(
  interceptorKey: InterceptorKey,
  interceptor: Interceptor<T>,
  parameter: string,
  options?: RegisterInterceptorOptions
```

```
): Promise<void>
```

Usage

```
// Without parameter
await service.addInterceptor('clearContact', myInterceptor, { timeout: 5000 });

// With parameter (scoped to a specific contact)
await service.addInterceptor('openQuickConnect', myInterceptor, contactId, { timeout:
 5000 });
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptorKey *Required*	`InterceptorKey` (string)	The action key to intercept. Valid values: • `openQuickConnect` – The Quick Connect or transfer action. • `openOutboundDialer` – The number pad or outbound dialer action. • `clearContact` – The clear or end contact action.
interceptor *Required*	`Interceptor<T>`	An asynchronous callback that receives a context object and returns an `InterceptorResult`. See the section called “Interceptor callback”.
parameter *Optional*	string	A scoping parameter. For the currently supported intercept or keys, pass the `contactId`

Parameter	Type	Description
		of the contact that the interceptor applies to. If you omit this value, the intercept or applies to all contacts.
options *Optional*	`RegisterInterceptorOptions`	The configuration options for this interceptor. If you omit this value, the service uses the default timeout of 5000 milliseconds and the default maximum consecutive block limit of 5. See the section called “Options”.

Output

`Promise<void>`

Unregister an interceptor in Connect Customer agent workspace

Removes a previously registered interceptor. You must pass the same function reference and the same optional parameter that you used at registration.

Signature

```
removeInterceptor<T>(
  interceptorKey: InterceptorKey,
  interceptor: Interceptor<T>,
  parameter?: string
): Promise<void>
```

Usage

```
await service.removeInterceptor('clearContact', myInterceptor);
```

```
// If registered with a parameter, pass the same parameter
await service.removeInterceptor('openQuickConnect', myInterceptor, contactId);
```

Input

The following table describes the input parameters.

Parameter	Type	Description
interceptorKey *Required*	`InterceptorKey` (string)	The action key that you registered the interceptor for.
interceptor *Required*	`Interceptor<T>`	The same function reference that you passed to `addInterceptor` .
parameter *Optional*	string	The same scoping parameter that you passed to `addInterceptor` . For the currently supported interceptor keys, pass the same `contactId` you used when you registered the interceptor. If you omit this value, the interceptor applies to all contacts.

Output

`Promise<void>`

Interceptor callback type in Connect Customer agent workspace

Defines the callback signature and return value for an interceptor. An interceptor receives a context object and returns a result that indicates whether the action proceeds.

Signature

```
type Interceptor<T> = (context: T) => Promise<InterceptorResult>

type InterceptorResult = boolean | { continue: boolean }
```

Context

For the currently supported interceptor keys, the context object is:

```
{ contactId?: string }
```

Return value

The following table describes the return values.

Return value	Shorthand	Effect
`{ continue: true }`	`true`	Allow the default action to proceed.
`{ continue: false }`	`false`	Block the default action.

RegisterInterceptorOptions in Connect Customer agent workspace

Contains the configuration options for interceptor registration, including timeout and consecutive-block behavior.

Signature

```
interface RegisterInterceptorOptions {
  timeout?: number;
  maxConsecutiveBlocks?: number;
}
```

Properties

The following table describes the properties.

Parameter	Type	Description
timeout *Optional*	number	The maximum time in milliseconds to wait for the interceptor to resolve. Default: 5000 (5 seconds). Valid range: 1–10000 milliseconds (hard ceiling: 10 seconds). The service clamps values outside this range to the default. If the intercept or exceeds the timeout, the default action proceeds as if the interceptor returned `{ continue: true }`.
maxConsecutiveBlocks *Optional*	number	The maximum number of consecutive times the interceptor can block the action before the system overrides and allows the action. Default: 5. Valid range: 0–100. The service clamps values outside this range to the default. The counter resets when the interceptor allows an action. Set to `0` to disable the consecutive block limit.

Interceptor behavior in Connect Customer agent workspace

The following rules describe how the framework evaluates interceptors at runtime:

- **Thrown error** – If your interceptor throws an error, the framework catches it and allows the default action to proceed.

- **Multiple interceptors** – When you register multiple interceptors for the same key, the framework evaluates all of them. If any interceptor returns `{ continue: false }`, the framework blocks the action.

Connect Customer agent workspace Email API

The Amazon Connect SDK provides an `EmailClient` which serves as an interface that your app can use to subscribe to email contact events and make email contact requests.

The `EmailClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { EmailClient } from "@amazon-connect/email";

const emailClient = new EmailClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { EmailClient } from "@amazon-connect/email";

const emailClient = new EmailClient({
    context: sampleContext,
    provider: sampleProvider
});
```

Note

Third-party applications must be configured with Cross Contact scope in order to utilize the EmailClient APIs, and * permission is required.

The following sections describe API calls for working with the Email API.

Contents

Subscribe to accepted email notifications in Connect Customer agent workspace

Subscribes a callback function to-be-invoked whenever an inbound email contact has been accepted.

Signature

```
onAcceptedEmail(handler: SubscriptionHandler<EmailContactId> contactId?: string): void
```

Usage

```
const handler: SubscriptionHandler<EmailContactId> = async (emailContact:
 EmailContactId) => {
   const { contactId } = emailContact;
   console.log(`Accepted Email Contact with Id: ${contactId}`);
}

emailClient.onAcceptedEmail(handler);

// EmailContactId Structure
{
   contactId: string;
}
```

Unsubscribe from accepted email notifications in Connect Customer agent workspace

Unsubscribes a callback function from the event that is fired when an inbound email contact is accepted.

Signature

```
offAcceptedEmail(handler: SubscriptionHandler<EmailContactId>, contactId?: string):
 void
```

Usage

```
emailClient.offAcceptedEmail(handler);
```

Create a draft email contact in Connect Customer agent workspace

Creates a draft outbound email contact; can either be an agent initiated outbound draft email, an agent reply draft email, or a forwarded email. Upon successful draft creation, the email contact will be in connected state. Returns an object that includes:

- `contactId: string`: The contact id of the newly created draft email contact

Signature

```
createDraftEmail(contactCreation: CreateDraftEmailContact): Promise<EmailContactId>
```

CreateDraftEmailContact Properties

Parameter	Type	Description
initiationMethod	"AGENT_REPLY" | "OUTBOUND"	"OUTBOUND" indicates that this draft email is the start of a new email conversation; "AGENT_REPLY" indicates that this draft email is being sent in response to an incoming email contact
relatedContactId	string	The id of the contact that is the reason for creating the new draft email; this is required when initiatio nMethod="AGENT_REPLY" and should be the contact id of the email that this email is being sent in response to. Also required when messageType="FORWARD" and should be the contact id of the email being forwarded.
messageType	"FORWARD"	Optional. The type of message being created. Use with initiationMethod=" OUTBOUND" to create a forward email contact.
expiryDurationInMinutes	number	Length of time before an unsent contact expires; Minimum is 1 minute,

Parameter	Type	Description
		Maximum is 1 week; Default is 12 hours.
attributes	Record<string, string>	A custom key-value pair using an attribute map. The attributes are standard Amazon Connect attributes, and can be accessed in flows just like any other contact attributes.
references	Record<string, { type: string; value: string; }>	Well-formed data on a contact, used by agents to complete a contact request.

Usage for Agent Initiated Outbound

```
const contact: EmailContactId = await emailClient.createDraftEmail({
   initiationMethod: "OUTBOUND",
});

const { contactId } = contact;
```

Usage for Agent Reply

```
const acceptedInboundEmailContactId = "exampleContactId";

const contact: EmailContactId = await emailClient.createDraftEmail({
   initiationMethod: "AGENT_REPLY",
   relatedContactId: acceptedInboundEmailContactId,
});

const { contactId } = contact;
```

Usage for Forward Email

```
const emailContactIdToBeForwarded = "exampleContactId";

const contact: EmailContactId = await emailClient.createDraftEmail({
   initiationMethod: "OUTBOUND",
   relatedContactId: emailContactIdToBeForwarded,
   messageType: "FORWARD",
});

const { contactId } = contact;
```

Subscribe to draft email creation notifications in Connect Customer agent workspace

Subscribes a callback function to-be-invoked whenever a draft email contact has been created.

Signature

```
onDraftEmailCreated(handler: SubscriptionHandler<EmailContactId>, contactId?: string):
 void
```

Usage

```
const handler: SubscriptionHandler<EmailContactId> = async (emailContact:
 EmailContactId) => {
   const { contactId } = emailContact;
   console.log(`Draft Email Contact Created with Id: ${contactId}`);
}

emailClient.onDraftEmailCreated(handler);

// EmailContactId Structure
{
   contactId: string;
}
```

Unsubscribe from draft email creation notifications in Connect Customer agent workspace

Unsubscribes a callback function from the event that is fired when a draft email contact is created.

Signature

```
offDraftEmailCreated(handler: SubscriptionHandler<EmailContactId>, contactId?: string):
 void
```

Usage

```
emailClient.offDraftEmailCreated(handler);
```

Get the metadata for an email contact in Connect Customer agent workspace

Returns the metadata for an email contact id while handling an active contact. The activeContactId is the id of the email contact the agent is actively viewing while contactId is the id of the email contact whose metadata should be retrieved.

Signature

```
getEmailData({ contactId, activeContactId }: { contactId: string; activeContactId:
 string; }): Promise<EmailContact>
```

Output - *EmailContact Properties*

Parameter	Type	Description
contactId	string	The id of the email contact
contactArn	string	The ARN of the email contact
contactAssociationId	string	The root contactId which is used as a unique identifier for all subsequent contacts in a contact tree. Use this value

Parameter	Type	Description
		with the EmailClient.getEmailThread api.
relatedContactId	string	This contact is in response or related to the specified related contact.
initialContactId	string	If this contact is related to other contacts, this is the id of the initial contact.
subject	string	The subject of the email
from	EmailAddress	An object that includes the from email address; this value could be undefined when the email contact has not been sent.
to	EmailAddress[]	An array of objects, each including an email address the email contact was sent to
cc	EmailAddress[]	An array of objects, each including an email address that was carbon copied on the email contact
deliveredTo	EmailAddress	An object that includes the email address associated with Amazon Connect that received this message; this is only applicable when direction=INBOUND.

Parameter	Type	Description
direction	"INBOUND" | "OUTBOUND"	INBOUND means the email contact was delivered to Amazon Connect; OUTBOUND means the email contact is from Amazon Connect
bodyLocation	EmailArtifactLocation	An object that includes the id and associated resource ARN of the file that the email contact's body is stored in; this value could be undefined when the email contact has not been sent.
attachmentLocations	EmailArtifactLocation[]	An array of objects, each including the id and associate d resource ARN, of files that have been attached to the email contact

EmailAddress Properties

Parameter	Type	Description
emailAddress	string	The email address
displayName	string	The name that is displayed inside the recipient's mailbox

EmailArtifactLocation Properties

Parameter	Type	Description
fileId	string	The id of the attached file
associatedResourceArn	string	The Amazon Connect resource to which the attached file is related to

Usage

```
const activeContactId: string = "exampleActiveContactId"; // The contact the agent is
 actively handling
const contactId: string = "contactIdToDescribe"; // The email contact id whose metadata
 should be retrieved

const emailMetadata: EmailContact = await emailClient.getEmailData({ contactId,
 activeContactId });

// Get the body of the email through the File Client
const bodyLocation = emailMetadata.bodyLocation;
if (bodyLocation) {
   const body: DownloadableAttachment = await fileClient.getAttachedFileUrl({
        attachment: bodyLocation,
        activeContactId,
   });

   const { downloadUrl } = body;
   const response: Response = await fetch(downloadUrl, { method: "GET" });
   const bodyContent: string = (await response.json()).messageContent;
}
```

Note

The EmailContact object will contain bodyLocation and attachmentLocations, both of which will require use of the FileClient's getAttachedFileUrl to get the relevant data for those objects.

Get a list of email contacts in an email contact's tree in Connect Customer agent workspace

Returns an array of EmailThreadContact objects (for the provided contactAssociationId) that represent that contact's email thread. The contactAssociationId is the root contact id which is used as a unique identifier for all subsequent contacts in a contact tree. Returns an object that includes:

- `contacts: EmailThreadContact[]`: an array of EmailThreadContact objects, each an email contact in the thread
- `nextToken?: string`: The token for the next set of results; use the value returned in the previous response in the next request to retrieve the next set of results

Signature

```
getEmailThread(getEmailThreadParams: GetEmailThreadParams): Promise<{ contacts:
 EmailThreadContact[]; nextToken?: string; }>
```

EmailThreadContact Properties

Parameter	Type	Description
contactId	string	The id of the email contact
contactArn	string	The ARN of the email contact
previousContactId	string	If this contact is not the first contact, this is the ID of the previous contact.
initialContactId	string	If this contact is related to other contacts, this is the ID of the initial contact.
relatedContactId	string	The contactId that is related to this contact.
initiationMethod	string	Indicates how the contact was initiated; Supported values:

Parameter	Type	Description
		"INBOUND" ,"OUTBOUND", "AGENT_REPLY", or "TRANSFER"
initiationTimestamp	Date	The date and time this contact was initiated, in UTC time.
disconnectTimestamp	Date | undefined	The date and time that the customer endpoint disconnected from the current contact, in UTC time. In transfer scenarios, the DisconnectTimestamp of the previous contact indicates the date and time when that contact ended.

GetEmailThreadParams Properties

Parameter	Type	Description
contactAssociationId	string	The contact association id to get the thread for.
maxResults	number	The max number of email threads to return; Default is 100. Minimum value of 1. Maximum value of 100.
nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request

Parameter	Type	Description
		to retrieve the next set of results.

Usage

```
const inboundEmailData = await emailClient.getEmailData({
   contactId: sampleContactId, // The inbound email contact you've accepted (or is
 still connecting)
   activeContactId: sampleContactId, // The email contact you're actively working; in
 this example, its the same as the accepted inbound email
});

const emailThreadContacts = await emailClient.getEmailThread({
  contactAssociationId: inboundEmailData.contactAssociationId,
});

// OPTIONAL: Filter out contacts that have been transferred to avoid displaying
 duplicated email content
const previousContactIdsSet = new Set(
    emailThreadContacts
        .map(emailThreadContact => emailThreadContact.previousContactId)
        .filter(Boolean)
);

const filteredEmailContactsInEmailThread = emailThreadContacts.filter(emailContact =>
    emailContact.contactId === sampleContactId ||
    !previousContactIdsSet.includes(emailContact.contactId)
);
```

Note

Each time an email contact is transferred, a new contact ID is created with initiationMethod === 'TRANSFER' and its previousContactId is the contact id before the transfer. You may optionally filter out these transferred contacts to avoid duplicate content when rendering the email thread.

Send a draft email contact in Connect Customer agent workspace

Sends both agent initiated and agent reply draft email contacts. Upon successfully sending the email, the contact will transition to ENDED state.

Signature

```
sendEmail(emailContact: DraftEmailContact): Promise<void>
```

DraftEmailContact Properties

Parameter	Type	Description
to	EmailAddress[]	An array of destination email addresses; max length supported is 1
emailContent	EmailContent	The content of the email
from	EmailAddress	The email contact will be sent from this email address; if no from address is provided in the request, the queue MUST have a default email address specified in the Outbound email configuration
cc	EmailAddress[]	Additional recipients to receive a carbon copy of the email; Max length supported is 10
contactId	string	The id of the draft email contact

EmailAddress Properties

Parameter	Type	Description
emailAddress	string	The email address
displayName	string	The name that is displayed inside the recipient's mailbox

EmailContent Properties

Parameter	Type	Description
subject	string	The email contact's subject
body	string	The body/content of the email, either in plain text or HTML
bodyType	"text/plain" | "text/html"	The body type of the email; can either be "text/plain" or "text/html"

Error Handling

When sending draft emails, agents may encounter issues. The @amazon-connect/email library provides methods to handle common errors:

- `isOutboundEmailAddressNotConfiguredError()`: Handle errors when the routing profile's default outbound queue does not have a default outbound email address and the sendEmail() request does not include a from address.
- `isEmailBodySizeExceededError()`: Handle errors when the size of the email body exceeds the limit.
- `isTotalEmailSizeExceededError()`: Handle errors when the total size of the email (email body and all attachments) exceeds the limit.

Usage

```
import {
    isOutboundEmailAddressNotConfiguredError,
    isEmailBodySizeExceededError,
    isTotalEmailSizeExceededError,
} from "@amazon-connect/email";

/* ... */

const toEmailAddress = {
  emailAddress: sampleRecipientAddress,
};

const emailContent = {
  subject: "Hello!",
  body: "Thank you!",
  bodyType: "text/plain",
}

const draftContact = {
  to: [toEmailAddress]
  emailContent,
  contactId: draftContactId, // This is the contact ID of the draft contact created via
 createDraftEmail()
};

try {
  await emailClient.sendEmail(draftContact);
} catch (e) {
  if (isOutboundEmailAddressNotConfiguredError(e)) {
    // Handle error when the routing profile's default outbound queue does not have a
 default
    // outbound email address and the request to `sendEmail` does not include a `from`
 address.
  } else if (isEmailBodySizeExceededError(e)) {
    // Handle error when the size of the email body exceeds the limit
  } else if (isTotalEmailSizeExceededError(e)) {
    // Handle error when total size of the email (email body and all attachments)
 exceeds the limit
  }
}
```

Connect Customer agent workspace File API

The Amazon Connect SDK provides a `FileClient` which serves as an interface that you can use to make file requests to upload, retrieve, and delete attached files.

The `FileClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { FileClient } from "@amazon-connect/file";

const fileClient = new FileClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { FileClient } from "@amazon-connect/file";

const fileClient = new FileClient({
    context: sampleContext,
    provider: sampleProvider
```

```
});
```

> **Note**
>
> Third-party applications must be configured with * permission in order to utilize the FileClient APIs.

The following sections describe API calls for working with the File API.

Contents

Get metadata about multiple attached files in Connect Customer agent workspace

Get metadata about multiple attached files on an associated resource while handling an active contact. The activeContactId is the id of the contact the agent is actively viewing. Each attached file provided in the input list must be associated with the associatedResourceArn in the RelatedAttachments request object.

Signature

```
batchGetAttachedFileMetadata({ relatedAttachments, activeContactId }:
 { relatedAttachments: RelatedAttachments; activeContactId: string; }):
 Promise<BatchGetAttachedFileMetadataResponse>
```

BatchGetAttachedFileMetadataResponse Properties

Parameter	Type	Description
files	AttachmentMetadata[]	Array of file metadata objects that were successfully retrieved
errors	AttachmentError[]	Array of errors of attached files that could not be retrieved

AttachmentMetadata Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
fileId	string	The unique identifier of the attached file resource
fileArn	string	The unique identifier of the attached file resource (ARN).
fileName	string	A case-sensitive name of the attached file being uploaded.
fileStatus	FileStatus	The current status of the attached file. Supported values: "APPROVED", "REJECTED", "PROCESSING", "FAILED"
fileSizeInBytes	number	The size of the attached file in bytes.

Parameter	Type	Description
creationTime	string	The time of Creation of the file resource as an ISO timestamp. It's specified in ISO 8601 format: yyyy-MM-ddThh:mm:ss.SSSZ. For example, 2024-05-03T02:41:28.172Z.

AttachmentError Properties

Parameter	Type	Description
errorCode	string	Status code describing the failure
errorMessage	string	Why the attached file couldn't be retrieved
fileId	string	The unique identifier of the attached file resource

RelatedAttachments Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
fileIds	string[]	The unique identifiers of the attached file resources

Usage

```
const relatedAttachments: RelatedAttachments = {
  fileIds: [sampleFileId1, sampleFileId2],
  associatedResourceArn: sampleAssociatedResourceArn,
};

const response: BatchGetAttachedFileMetadataResponse = await
 fileClient.batchGetAttachedFileMetadata({
  relatedAttachments,
  activeContactId: sampleActiveContactId, // The contact the agent is actively handling
});

const { files, errors } = response;

// Add logic to handle response
```

Confirm that an attached file has been uploaded in Connect Customer agent workspace

Allows you to confirm that the attachment has been uploaded using the pre-signed URL provided in the startAttachedFileUpload API. The request accepts an Attachment object, which has the following properties:

- `associatedResourceArn: string`: Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
- `fileId: string`: ID in Connect's File record

Signature

```
completeAttachedFileUpload(attachment: Attachment): Promise<void>
```

Usage

```
/* Logic with startAttachedFileUplaod and uploading attachment to pre-signed URL */

/* ... */
```

```
await fileClient.completeAttachedFileUpload({
  associatedResourceArn: sampleAssociatedResourceArn, // Get this from the response
 from `startAttachedFileUpload`
  fileId: sampleFileId // Get this from the response from `startAttachedFileUpload`
});
```

Delete an attached file in Connect Customer agent workspace

Deletes an attached file along with the underlying S3 Object. The attached file is permanently deleted if S3 bucket versioning is not enabled. The request accepts an Attachment object, which has the following properties:

- `associatedResourceArn: string`: Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
- `fileId: string`: ID in Connect's File record

Signature

```
deleteAttachedFile(data: Attachment): Promise<void>
```

Usage

```
await fileClient.deleteAttachedFile({
  associatedResourceArn: sampleAssociatedResourceArn, // Get this from the response
 from `startAttachedFileUpload`
  fileId: sampleFileId // Get this from the response from `startAttachedFileUpload`
});
```

Get a pre-signed S3 URL to download an approved attached file in Connect Customer agent workspace

Returns a pre-signed URL to download an approved attached file while handling an active contact. The activeContactId is the id of the contact the agent is actively viewing. This API also returns metadata about the attached file and it will only return a downloadUrl if the status of the attached file is APPROVED.

Signature

```
getAttachedFileUrl({ attachment, activeContactId }: { attachment: Attachment;
 activeContactId: string; }): Promise<DownloadableAttachment>
```

DownloadableAttachment Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
fileId	string	The unique identifier of the attached file resource.
downloadUrl	string	A pre-signed URL that should be used to download the attached file.
fileArn	string	The unique identifier of the attached file resource (ARN).
fileName	string	A case-sensitive name of the attached file being uploaded.
fileStatus	FileStatus	The current status of the attached file. Supported values: "APPROVED", "REJECTED", "PROCESSING", "FAILED"
fileSizeInBytes	number	The size of the attached file in bytes.
creationTime	string	The time of Creation of the file resource as an ISO timestamp. It's specified in ISO 8601 format: yyyy-

Parameter	Type	Description
		MM-ddThh:mm:ss.SSSZ. For example, 2024-05-0 3T02:41:28.172Z.

Attachment Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
fileId	string	The unique identifier of the attached file resource.

Usage

```
const downloadableAttachment = await fileClient.getAttachedFileUrl({
  attachment: {
    associatedResourceArn: sampleAssociatedResourceArn,
    fileId: sampleFileId,
  },
  activeContactId: sampleActiveContactId, // The contact the agent is actively handling
});

const { downloadUrl } = downloadableAttachment;
const response: Response = await fetch(downloadUrl, { method: "GET" });
```

Start uploading a file to Connect Customer agent workspace

Provides a pre-signed Amazon S3 URL in response to upload a new attached file.

Signature

```
startAttachedFileUpload(data: NewAttachment): Promise<UploadableAttachment>
```

UploadableAttachment Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. Could be a Connect Email Contact ARN or a Connect Case ARN
fileId	string	ID in Connect's File record
uploadUrl	string	A pre-signed S3 URL that should be used for uploading the attached file.
uploadHeaders	Record<string, string>	A map of headers that should be provided in the request when uploading the attached file.
uploadMethod	"PUT"	The upload request must be a PUT.
fileStatus	FileStatus	The current status of the attached file. Supported values: "APPROVED", "REJECTED", "PROCESSING", "FAILED"

NewAttachment Properties

Parameter	Type	Description
associatedResourceArn	string	Amazon Connect ARN of the resource that the file is attached to. This could be a Connect Email Contact ARN or a Connect Case ARN
fileName	string	A case-sensitive name of the attached file being uploaded. Minimum length of 1; Maximum length of 256. Supported pattern: ^\P{C}*$
fileSizeInBytes	number	The size of the attached file in bytes. Minimum value of 1.
fileUseCaseType	"ATTACHMENT"	The use case for the file. Must be "ATTACHMENT"

Error Handling

When beginning the process to upload attached files, agents may encounter issues. The @amazon-connect/file library provides methods to handle common errors:

- `isInvalidFileNameError()`: Handle errors when the name of the file is not valid
- `isInvalidFileTypeError()`: Handle errors when the file type is not supported
- `isInvalidFileSizeError()`: Handle errors when the size of the file is invalid
- `isTotalFileSizeExceededError()`: Handle errors when the total size of all files (being uploaded) exceeds the limit.
- `isTotalFileCountExceededError()`: Handle errors when the total number of files (being uploaded) exceeds the limit.

Usage

```
import {
    isInvalidFileNameError,
    isInvalidFileTypeError,
    isInvalidFileSizeError,
    isTotalFileSizeExceededError,
    isTotalFileCountExceededError
} from "@amazon-connect/file";

/* ... */

const newAttachment: NewAttachment = {
  associatedResourceArn: sampleAssociatedResourceArn, // This could be an email contact
 ARN or case ARN that you are uploading the attached file to
  fileName: sampleFileName,
  fileSizeInBytes: sampleFileSizeInBytes,
  fileUseCaseType: "ATTACHMENT"
};

let uploadableAttachment: UploadableAttachment;
try {
  uploadableAttachment = await fileClient.startAttachedFileUpload(newAttachment);
} catch (e) {
  if (isInvalidFileNameError(e)) {
    // Handle InvalidFileName error
  } else if (isInvalidFileTypeError(e)) {
    // Handle InvalidFileType error
  } else if (isInvalidFileSizeError(e)) {
    // Handle InvalidFileSize error
  } else if (isTotalFileSizeExceededError(e)) {
    // Handle TotalFileSizeExceeded error
  } else if (isTotalFileCountExceededError(e)) {
    // Handle TotalFileCountExceeded error
  }
}

// Assuming startAttachedFileUpload succeeded, we upload the attached file to the pre-
signed S3 URL
const { uploadUrl, uploadHeaders, uploadMethod } = uploadableAttachment;

await fetch(uploadUrl, {
  method: uploadMethod,
  headers: uploadHeaders,
  body: file, // This is the file you're uploading
```

```
});
```

Connect Customer agent workspace Message Template API

The Amazon Connect SDK provides a `MessageTemplateClient` which serves as an interface that you can use to make requests to search and get content from your Amazon Connect Message Template Knowledge Base.

The `MessageTemplateClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { MessageTemplateClient } from "@amazon-connect/message-template";

const messageTemplateClient = new MessageTemplateClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { MessageTemplateClient } from "@amazon-connect/message-template";
```

```
const messageTemplateClient = new MessageTemplateClient({
    context: sampleContext,
    provider: sampleProvider
});
```

Note

Third-party applications must be configured with * permission in order to utilize the MessageTemplateClient APIs.

The following sections describe API calls for working with the MessageTemplate API.

Contents

Get content of a message template in Connect Customer agent workspace

Gets the content of a message template. This includes plaintext or html content of the body of the message template as a string, the subject of the message template, and attachments if they are configured on the message template. Attributes in the message template will be filled if they are system attributes, agent attributes, or custom attributes set up in the contact flow. More information on the attributes can be found here: https://docs.aws.amazon.com/connect/latest/adminguide/personalize-templates.html

Signature

```
getContent(messageTemplateId: string, contactId: string):
 Promise<MessageTemplateContent>
```

messageTemplateId

The messageTemplateId can be either the ID or the ARN of a message template

- Passing in the ARN returned by searchMessageTemplates is recommended here, since this will get the content of the active version of the message template.
- Passing in the ID will return the content of the latest version of the message template. A qualifier can be appended to the messageTemplateId to get the content of a different version of the message template.

More information on qualifiers can be found here:

https://docs.aws.amazon.com/connect/latest/APIReference/API_amazon-q-connect_GetMessageTemplate.html

More information on versioning can be found here:

https://docs.aws.amazon.com/connect/latest/adminguide/about-version-message-templates.html

contactId

The current contact to add the message template to. This is used to populate attributes in the message template

MessageTemplateContent Properties

Parameter	Type	Description
subject	string	Message subject populated in the template
body	MessageTemplateBody	Message body content populated in the template. This can include plainText or html or both
attachments	MessageTemplateAttachment[]	Attachments populated in the template
attributesNotInterpolated	string[]	List of attributes that were not automatically populated in the message template. If all attributes were automatic

Parameter	Type	Description
		ally populated, this list will be empty

MessageTemplateBody Properties

Parameter	Type	Description
plainText	string	Plain text content of the message template as a string. It is possible for both the plain text and html to be populated, or for only the plain text or html content to be populated
html	string	HTML content of the message template as a string

MessageTemplateAttachment Properties

Parameter	Type	Description
fileName	string	Name of the attachment
fileId	string	ID of the attachment
downloadUrl	string	URL to download the attachment from

Determine if the Message Template feature is enabled in Connect Customer agent workspace

Returns the MessageTemplateEnabledState object, which indicates if the message template feature is enabled for the Connect instance. The Message Template feature is considered enabled if there

is a knowledge base for message templates configured for the instance. The object contains the following fields:

- `isEnabled`: A boolean indicating if the feature is enabled
- `knowledgeBaseId`: The id of the Message Template Knowledge Base (if the feature is enabled)

Signature

```
isEnabled(): Promise<MessageTemplateEnabledState>
```

Retrieve message templates that match a search query in Connect Customer agent workspace

Returns the SearchMessageTemplatesResponse object, which contains the matching message templates and a token to retrieve the next page of results, if available. The SearchMessageTemplatesParams object is used to configure the search, allowing you to filter by various criteria, as well as specify the channels and pagination options. If no filter text is provided, all active message templates for the agent's routing profile and the channel(s) specified are returned.

Signature

```
searchMessageTemplates(request: SearchMessageTemplatesParams):
 Promise<SearchMessageTemplatesResponse>
```

SearchMessageTemplatesResponse Properties

Parameter	Type	Description
messageTemplate	MessageTemplate[]	List of message templates matching the search criteria specified in the request
nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request

Parameter	Type	Description
		to retrieve the next set of results.

MessageTemplate Properties

Parameter	Type	Description
messageTemplateArn	string	The ARN of the message template. This contains the active version qualifier at the end of the ARN
messageTemplateId	string	The ID of the message template. This does NOT contain a qualifier with the version of the message template.
name	string	Name of the message template
description	string	Description of the message template

SearchMessageTemplatesParams Properties

Parameter	Type	Description
channels	MessageTemplateChannel[]	The channel(s) to return message templates for. If the list is empty, no message templates will be returned. Supported values: "EMAIL"

Parameter	Type	Description
queries	MessageTemplateQueryField[]	Queries are used to filter the returned message templates by name or description. Leaving the queries empty will return all message templates associated with the agent's routing profile
maxResults	number	Maximum number of message templates to return
nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request to retrieve the next set of results.

MessageTemplateQueryField Properties

Parameter	Type	Description
name	"name" | "description"	The message templates will be filtered by the values matching the text in the name field provided
values	string[]	The values of the attribute to query the message templates by
priority	"HIGH" | "MEDIUM" | "LOW"	The importance of the attribute field when calculating query result relevancy scores. The value set for

Parameter	Type	Description
		this parameter affects the ordering of search results.
allowFuzziness	boolean	Whether the query expects only exact matches on the attribute field values. The results of the query will only include exact matches if this parameter is set to false.
operator	"CONTAINS" | "CONTAINS_AND_PREFIX"	Include all templates that contain the values or only templates that contain the values as the prefix.

Connect Customer agent workspace Quick Responses API

The Amazon Connect SDK provides a `QuickResponsesClient` which serves as an interface that you can use to make requests to search your Amazon Connect Quick Responses Knowledge Base.

The `QuickResponsesClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { QuickResponsesClient } from "@amazon-connect/quick-responses";
```

```
const quickResponsesClient = new QuickResponsesClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { QuickResponsesClient } from "@amazon-connect/quick-responses";

const quickResponsesClient = new QuickResponsesClient({
    context: sampleContext,
    provider: sampleProvider
});
```

> **Note**
>
> Third-party applications must be configured with * permission in order to utilize the QuickResponsesClient APIs.

The following sections describe API calls for working with the QuickResponses API.

Contents

Determine if the Quick Responses feature is enabled in Connect Customer agent workspace

Returns the QuickResponsesEnabledState object, which indicates if the quick responses feature is enabled for the Connect instance. Quick responses is considered enabled if there is a knowledge base for quick responses configured for the instance. The object contains the following fields:

- `isEnabled`: A boolean indicating if the feature is enabled
- `knowledgeBaseId`: The id of the Quick Responses Knowledge Base (if the feature is enabled)

Signature

```
isEnabled(): Promise<QuickResponsesEnabledState>
```

Retrieve quick responses that match a search query in Connect Customer agent workspace

Returns the SearchQuickResponsesResult object, which contains the matching quick response results and a token to retrieve the next page of results, if available. The SearchQuickResponsesRequest object is used to configure the search, allowing you to filter by various criteria, as well as specify the channels, user-defined contact attributes, and pagination options. If no queries are provided, the method will return all quick responses associated with the agent's routing profile.

Signature

```
searchQuickResponses(queryRequest: SearchQuickResponsesRequest):
 Promise<SearchQuickResponsesResult>
```

SearchQuickResponsesResult Properties

Parameter	Type	Description
results	QuickResponsesSearchResultData[]	The results of the quick responses search
nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request to retrieve the next set of results.

QuickResponsesSearchResultData Properties

Parameter	Type	Description
contents	QuickResponseContents	The contents of the quick response.
knowledgeBaseId	string	The identifier of the knowledge base.
name	string	The name of the quick response.
quickResponseArn	string	The Amazon Resource Name (ARN) of the quick response.
quickResponseId	string	The identifier of the quick response.
description	string	The description of the quick response.
shortcutKey	string	The shortcut key of the quick response. The value should be unique across the knowledge base.
attributesNotInterpolated	string[]	The user defined contact attributes that are not resolved when the search result is returned.

QuickResponseContents Properties

Parameter	Type	Description
markdown	string	The content of the quick response stored in markdown

Parameter	Type	Description
plainText	string	The content of the quick response stored in plain text

SearchQuickResponsesRequest Properties

Parameter	Type	Description
queries	QuickResponsesQuery[]	Query used to filter quick responses; if no queries are provided, the client will return all quick responses associate d with the agent's routing profile.
channels	QuickResponseChannel[]	The channels to filter the request by. Supported values: "Chat" or "Email"
attributes	Record<string, string>	The user-defined Amazon Connect contact attributes to be resolved when search results are returned.
debounceMS	number	The default value is set to 250ms; set it to 0 to disable debounced input change
maxResults	number	The number of results to be returned. Minimum value of 1. Maximum value of 100.
nextToken	string	The token for the next set of results. Use the value returned in the previous response in the next request

Parameter	Type	Description
		to retrieve the next set of results.

QuickResponsesQuery Properties

Parameter	Type	Description
name	QuickResponsesQueryFieldName	The name of the attribute to query the quick responses by. Supported values: "content", "name", "description", or "shortcutKey"
values	string[]	The values of the attribute to query the quick responses by.
operator	QuickResponsesQueryOperator	The operator to use for matching attribute field values in the query. Supported values: "CONTAINS" or "CONTAINS_AND_PREFIX"
priority	QuickResponsesQueryPriority	The importance of the attribute field when calculating query result relevancy scores. The value set for this parameter affects the ordering of search results. Supported values: "HIGH", "MEDIUM", or "LOW"
allowFuzziness	boolean	Whether the query expects only exact matches on the attribute field values. The results of the query will only

Parameter	Type	Description
		include exact matches if this parameter is set to false.

Connect Customer agent workspace User API

The Amazon Connect SDK provides an `SettingsClient` which serves as an interface that your app in Connect Customer agent workspace can use to make data requests on user settings.

The `SettingsClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
 };
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { SettingsClient } from "@amazon-connect/user";
const settingsClient = new SettingsClient({ provider });
```

Note

You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }` . This is the recommended option.

Alternatively, providing a constructor argument:

```
    import { SettingsClient } from "@amazon-connect/user";

    const settingsClient = new SettingsClient({
        context: sampleContext,
        provider: sampleProvider
});
```

The following sections describe API calls for working with the User API.

Contents

Get the language of a user in Connect Customer agent workspace

Returns the language setting for the current user in the Connect Customer agent workspace.

```
async getLanguage(): Promise<Locale | null>
```

Permissions required:

```
User.Configuration.View
```

Subscribe a callback function when an Connect Customer agent workspace user changes languages

Subscribes a callback function to-be-invoked whenever a user LanguageChanged event occurs in the Connect Customer agent workspace.

Signature

```
onLanguageChanged(handler: UserLanguageChangedHandler)
```

Usage

```
const handler: UserLanguageChangedHandler = async (data: UserLanguageChanged) => {
    console.log("User LanguageChange occurred! " + data);
};

settingsClient.onLanguageChanged(handler);

// UserLanguageChanged Structure
{
  language: string;
  previous: {
    language: string;
  };
}
```

Permissions required:

```
User.Configuration.View
```

Unsubscribe a callback function when an Connect Customer agent workspace user changes languages

Unsubscribes the callback function from LanguageChanged event in the Connect Customer agent workspace.

Signature

```
offLanguageChanged(handler: UserLanguageChangedHandler)
```

Usage

```
settingsClient.offLanguageChanged(handler);
```

Get the ARN of the current user in Connect Customer agent workspace

Returns the Amazon Resource Name (ARN) of the user that's currently logged in to the Connect Customer agent workspace.

```
async getUserArn(): Promise<string>
```

Permissions required:

```
*
```

Get the instance ID of the current Connect Customer instance in Connect Customer agent workspace

Returns the Connect Customer instance ID associated with the user that's currently logged in to the Connect Customer agent workspace.

```
async getInstanceId(): Promise<string>
```

Permissions required:

```
*
```

Get the network type of the current Connect Customer instance in Connect Customer agent workspace

Returns the network type of the Connect Customer instance associated with the user that's currently logged in to the Connect Customer agent workspace. The returned `NetworkType` is either `"DUAL_STACK"` or `"IPV4"`.

```
async getNetworkType(): Promise<NetworkType>
```

Permissions required:

```
*
```

Set the language of a user in Connect Customer agent workspace

Sets the language preference for the user that's currently logged in to the Connect Customer agent workspace. The promise resolves once the language change has been persisted, with the resulting language echoed back in the result.

Signature

```
setLanguage(language: Locale): Promise<SetLanguageResult>
```

Usage

```
const result: SetLanguageResult = await settingsClient.setLanguage("en_US");
```

Input

Parameter	Type	Description
language *Required*	Locale	The locale to set for the current user. One of `en_US`, `de_DE`, `es_ES`, `fr_FR`, `ja_JP`, `it_IT`, `ko_KR`, `pt_BR`, `zh_CN`, or `zh_TW`.

Output - SetLanguageResult

Parameter	Type	Description
language	Locale (optional)	The locale that was set, echoed from the request.

Permissions required:

```
*
```

Set the visual mode of a user in Connect Customer agent workspace

Sets the visual mode (light, dark, or auto) for the user that's currently logged in to the Connect Customer agent workspace. The promise resolves once the visual mode change has been persisted.

Signature

```
setVisualMode(visualMode: VisualMode): Promise<void>
```

Usage

```
await settingsClient.setVisualMode("dark");
```

Input

Parameter	Type	Description
visualMode *Required*	VisualMode	The visual mode to set. One of `"light"`, `"dark"`, or `"auto"`.

Permissions required:

```
*
```

Subscribe a callback function when an Connect Customer agent workspace user's visual mode changes

Subscribes a callback function to-be-invoked whenever the user's visual mode changes in the Connect Customer agent workspace.

Signature

```
onVisualModeChange(handler: VisualModeChangedHandler)
```

Usage

```
const handler: VisualModeChangedHandler = async (data: VisualModeChanged) => {
    console.log("Visual mode changed! " + data.visualMode);
};

settingsClient.onVisualModeChange(handler);

// VisualModeChanged Structure
{
  visualMode: VisualMode;
  previous: {
    visualMode: VisualMode | null;
  };
}
```

Permissions required:

```
*
```

Unsubscribe a callback function when an Connect Customer agent workspace user's visual mode changes

Unsubscribes the callback function from the visual mode change event in the Connect Customer agent workspace.

Signature

```
offVisualModeChange(handler: VisualModeChangedHandler)
```

Usage

```
settingsClient.offVisualModeChange(handler);
```

Connect Customer agent workspace Voice API

The Amazon Connect SDK provides an `VoiceClient` which serves as an interface that your app in the Connect Customer agent workspace can use to make data requests on voice contact.

The `VoiceClient` accepts an optional constructor argument, `ConnectClientConfig` which itself is defined as:

```
export type ConnectClientConfig = {
    context?: ModuleContext;
    provider?: AmazonConnectProvider;
};
```

If you do not provide a value for this config, then the client will default to using the **AmazonConnectProvider** set in the global provider scope. You can also manually configure this using **setGlobalProvider**.

You can instantiate the agent client as follows:

```
import { VoiceClient } from "@amazon-connect/voice";

const voiceClient = new VoiceClient({ provider });
```

> **Note**
>
> You must first instantiate the AmazonConnectApp which initializes the default AmazonConnectProvider and returns `{ provider }`. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { VoiceClient } from "@amazon-connect/voice";

const voiceClient = new VoiceClient({
```

```
    context: sampleContext,
    provider: sampleProvider
});
```

Note

Third-party applications must be configured with * permission in order to utilize the VoiceClient APIs.

The following sections describe API calls for working with the Voice API.

Contents

- Unsubscribe from voice enhancement mode change events in Connect Customer agent workspace
- Subscribe to participant resume capability change events in Connect Customer agent workspace
- Subscribe to self resume capability change events in Connect Customer agent workspace
- Subscribe to participant hold events in Connect Customer agent workspace
- Subscribe to participant resume events in Connect Customer agent workspace
- Subscribe to self hold events in Connect Customer agent workspace
- Subscribe to self resume events in Connect Customer agent workspace
- Subscribe to voice enhancement mode change events in Connect Customer agent workspace
- Resume a participant from hold in Connect Customer agent workspace
- Set the voice enhancement mode in Connect Customer agent workspace

Check if a participant can be resumed from hold in Connect Customer agent workspace

Checks whether a specific participant can be resumed from hold.

Signature

```
canResumeParticipant(participantId: string): Promise<boolean>
```

Usage

```
const canResume = await voiceClient.canResumeParticipant("participant-456");
if (canResume) {
  await voiceClient.resumeParticipant("participant-456");
}
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant

Output

Returns a Promise that resolves to a boolean: true if the participant can be resumed, false otherwise

Check if the current user can be resumed from hold in Connect Customer agent workspace

Checks whether the current user's participant can be resumed from hold for a specific contact.

Signature

```
canResumeSelf(contactId: string): Promise<boolean>
```

Usage

```
const canResume = await voiceClient.canResumeSelf("contact-123");
if (canResume) {
  // Resume logic here
}
```

Input

Parameter	Type	Description
contactId *Required*	string	The unique identifier for the contact

Output

Returns a Promise that resolves to a boolean: true if the current user can be resumed, false otherwise

Conference all participants on a contact in Connect Customer agent workspace

Conferences all participants on a contact together, removing any hold states and enabling all participants to communicate with each other.

Signature

```
conferenceParticipants(contactId: string): Promise<void>
```

Usage

```
await voiceClient.conferenceParticipants("contact-123");
console.log("All participants are now conferenced");
```

Input

Parameter	Type	Description
contactId *Required*	string	The unique identifier for the contact

Create an outbound call to phone number in Connect Customer agent workspace

Creates an outbound call to the given phone number and returns the contactId. It takes an optional parameter queueARN which specifies the outbound queue associated with the call, if omitted the default outbound queue defined in the agent's routing profile will be used.

Signature

```
  createOutboundCall(
    phoneNumber: string,
    options?: CreateOutboundCallOptions,
  ): Promise<CreateOutboundCallResult>
```

Usage

```
const outboundCallResult:CreateOutboundCallResult = await
 voiceClient.createOutboundCall("+18005550100");
```

Input

Parameter	Type	Description
phoneNumber *Required*	string	The phone number specified in E.164 format
options.queueARN	string	It specifies the outbound queue associated with the call, if omitted the default outbound queue defined in the agent's routing profile will be used.
options.relatedContactId	string	Optional parameter to supply related contactId

Output - *CreateOutboundCallResult*

Parameter	Type	Description
contactId	string	The contactId of the created outbound call.

Permissions required:

```
Contact.Details.Edit
```

Gets the phone number of the initial customer connection in Connect Customer agent workspace

Gets the phone number of the initial customer connection. Applicable only for voice contacts.

Signature

```
getInitialCustomerPhoneNumber(contactId: string): Promise<string>
```

Usage

```
const initialCustomerPhoneNumber: string = await
 voiceClient.getInitialCustomerPhoneNumber(contactId);
```

Input

Parameter	Type	Description
contactId *Required*	string	The id of the contact for which the data is requested.

Permissions required:

```
Contact.CustomerDetails.View
```

Gets the outbound call permission configured for the agent in Connect Customer agent workspace

Gets true if the agent has the security profile permission for making outbound calls, false otherwise.

Signature

```
getOutboundCallPermission(): Promise<boolean>
```

Usage

```
const outboundCallPermission: boolean = await voiceClient.getOutboundCallPermission();
```

Permissions required:

```
User.Configuration.View
```

Place a participant on hold in Connect Customer agent workspace

Places a specific participant on hold.

Signature

```
holdParticipant(participantId: string): Promise<void>
```

Usage

```
await voiceClient.holdParticipant("participant-456");
console.log("Participant is now on hold");
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant to place on hold

Get the voice enhancement mode in Connect Customer agent workspace

Gets the voice enhancement mode of the user that's currently logged in to Connect Customer agent workspace. The voice enhancement mode can have the following values:

- `VOICE_ISOLATION`: it suppresses background noise and isolates the agent's voice. This mode should only be enabled if the agent uses a wired headsets.
- `NOISE_SUPPRESSION`: it suppresses the background noise. We recommend using this mode with any type of headset.
- `NONE`: no voice enhancement applies.

Signature

```
async getVoiceEnhancementMode(): Promise<VoiceEnhancementMode>
```

Usage

```
const voiceEnhancementMode: VoiceEnhancementMode = await
 voiceClient.getVoiceEnhancementMode();
```

Permissions required:

```
*
```

Get voice enhancement model paths in Connect Customer agent workspace

Returns the voice enhancements models static assets URL paths.

Signature

```
async getVoiceEnhancementPaths(): Promise<VoiceEnhancementPaths>
```

Usage

```
voiceClient.getVoiceEnhancementPaths();

// VoiceEnhancementPaths structure
interface VoiceEnhancementPaths {
  processors: string;
  workers: string;
  wasm: string;
  models: string;
}
```

Permissions required:

```
*
```

Check if a participant is on hold in Connect Customer agent workspace

Checks whether a specific participant is currently on hold.

Signature

```
isParticipantOnHold(participantId: string): Promise<boolean>
```

Usage

```
const isOnHold = await voiceClient.isParticipantOnHold("participant-456");
if (isOnHold) {
  console.log("Participant is on hold");
} else {
  console.log("Participant is active");
}
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant

Output

Returns a Promise that resolves to a boolean: true if the participant is on hold, false otherwise

Get a list of dialable countries in Connect Customer agent workspace

Get a list of `DialableCountry` that contains the country code and calling code that the Connect Customer instance is allowed to make calls to.

Signature

```
listDialableCountries(): Promise<DialableCountry[]>
```

Usage

```
const dialableCountries:DialableCountry[] = await voiceClient.listDialableCountries();
```

Output - *DialableCountry*

Parameter	Type	Description
countryCode	string	The ISO country code
callingCode	string	The calling code for the country
label	string	The name of the country

Permissions required:

```
User.Configuration.View
```

Unsubscribe from participant resume capability change events in Connect Customer agent workspace

Unsubscribes from participant capability change events.

Signature

```
offCanResumeParticipantChanged(
  handler: CanResumeParticipantChangedHandler,
  participantId?: string
): void
```

Usage

```
voiceClient.offCanResumeParticipantChanged(handleCanResumeChanged);
```

Input

Parameter	Type	Description
handler *Required*	CanResumeParticipantChangedHandler	Event handler function to remove

Parameter	Type	Description
participantId	string	Optional participant ID to unsubscribe from specific participant events

Unsubscribe from self resume capability change events in Connect Customer agent workspace

Unsubscribes from capability change events for the current user.

Signature

```
offCanResumeSelfChanged(
  handler: CanResumeParticipantChangedHandler,
  contactId?: string
): void
```

Usage

```
voiceClient.offCanResumeSelfChanged(handleCanResumeSelfChanged);
```

Input

Parameter	Type	Description
handler *Required*	CanResumeParticipantChangedHandler	Event handler function to remove
contactId	string	Optional contact ID to unsubscribe from specific contact events

Unsubscribe from participant hold events in Connect Customer agent workspace

Unsubscribes from participant hold events.

Signature

```
offParticipantHold(
  handler: ParticipantHoldHandler,
  participantId?: string
): void
```

Usage

```
voiceClient.offParticipantHold(handleParticipantHold);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantHoldHandler	Event handler function to remove
participantId	string	Optional participant ID to unsubscribe from specific participant events

Unsubscribe from participant resume events in Connect Customer agent workspace

Unsubscribes from participant resume events.

Signature

```
offParticipantResume(
  handler: ParticipantResumeHandler,
  participantId?: string
): void
```

Usage

```
voiceClient.offParticipantResume(handleParticipantResume);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantResumeHandler	Event handler function to remove
participantId	string	Optional participant ID to unsubscribe from specific participant events

Unsubscribe from self hold events in Connect Customer agent workspace

Unsubscribes from self hold events.

Signature

```
offSelfHold(
  handler: ParticipantHoldHandler,
  contactId?: string
): void
```

Usage

```
voiceClient.offSelfHold(handleSelfHold);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantHoldHandler	Event handler function to remove
contactId	string	Optional contact ID to unsubscribe from specific contact events

Unsubscribe from self resume events in Connect Customer agent workspace

Unsubscribes from self resume events.

Signature

```
offSelfResume(
  handler: ParticipantResumeHandler,
  contactId?: string
): void
```

Usage

```
voiceClient.offSelfResume(handleSelfResume);
```

Input

Parameter	Type	Description
handler *Required*	ParticipantResumeHandler	Event handler function to remove
contactId	string	Optional contact ID to unsubscribe from specific contact events

Unsubscribe from voice enhancement mode change events in Connect Customer agent workspace

Unsubscribes a callback function registered for voice enhancements mode changed event.

Signature

```
offVoiceEnhancementModeChanged(handler: VoiceEnhancementModeChangedHandler)
```

Usage

```
const handler: VoiceEnhancementModeChangedHandler = async (data:
 VoiceEnhancementModeChanged) => {
  console.log("User VoiceEnhancementMode changed! " + data);
}

// subscribe a callback for mode change
voiceClient.onVoiceEnhancementModeChanged(handler);

// unsubsribes a callback for mode change
voiceClient.offVoiceEnhancementModeChanged(handler);
```

Permissions required:

```
*
```

Subscribe to participant resume capability change events in Connect Customer agent workspace

Subscribes to events when a participant's capability to be resumed from hold changes.

Signature

```
onCanResumeParticipantChanged(
  handler: CanResumeParticipantChangedHandler,
  participantId?: string
): void
```

Usage

```
const handleCanResumeChanged = (event) => {
  console.log(`Participant ${event.participantId} resume capability:
 ${event.canResumeConnection}`);
  if (event.canResumeConnection) {
    // Enable resume button for this participant
  } else {
    // Disable resume button for this participant
  }
};
voiceClient.onCanResumeParticipantChanged(handleCanResumeChanged, "participant-456");
```

Input

Parameter	Type	Description
handler *Required*	CanResumeParticipantChangedHandler	Event handler function to call when the capability changes
participantId	string	Optional participant ID to filter events for a specific participant

Subscribe to self resume capability change events in Connect Customer agent workspace

Subscribes to events when the current user's capability to be resumed from hold changes.

Signature

```
onCanResumeSelfChanged(
  handler: CanResumeParticipantChangedHandler,
  contactId?: string
): void
```

Usage

```
const handleCanResumeSelfChanged = (event) => {
  if (event.canResumeConnection) {
    console.log("You can now be resumed from hold");
    // Enable resume button in UI
  } else {
    console.log("You cannot be resumed at this time");
    // Disable resume button in UI
  }
};
voiceClient.onCanResumeSelfChanged(handleCanResumeSelfChanged, "contact-123");
```

Input

Parameter	Type	Description
handler *Required*	CanResumeParticipantChangedHandler	Event handler function to call when the capability changes
contactId	string	Optional contact ID to filter events for a specific contact

Subscribe to participant hold events in Connect Customer agent workspace

Subscribes to events when any participant is put on hold.

Signature

```
onParticipantHold(
  handler: ParticipantHoldHandler,
  participantId?: string
): void
```

Usage

```
const handleParticipantHold = (event) => {
  console.log(`Participant ${event.participantId} is now on hold`);
  console.log(`Contact: ${event.contactId}`);
};
// Subscribe to all participants
voiceClient.onParticipantHold(handleParticipantHold);
// Or subscribe to a specific participant
voiceClient.onParticipantHold(handleParticipantHold, "participant-456");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantHoldHandler	Event handler function to call when participants are put on hold

Parameter	Type	Description
participantId	string	Optional participant ID to filter events for a specific participant

Subscribe to participant resume events in Connect Customer agent workspace

Subscribes to events when any participant is taken off hold.

Signature

```
onParticipantResume(
  handler: ParticipantResumeHandler,
  participantId?: string
): void
```

Usage

```
const handleParticipantResume = (event) => {
  console.log(`Participant ${event.participantId} has been resumed`);
};
voiceClient.onParticipantResume(handleParticipantResume, "participant-456");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantResumeHandler	Event handler function to call when participants are taken off hold
participantId	string	Optional participant ID to filter events for a specific participant

Subscribe to self hold events in Connect Customer agent workspace

Subscribes to events when the current user's participant is put on hold.

Signature

```
onSelfHold(
  handler: ParticipantHoldHandler,
  contactId?: string
): void
```

Usage

```
const handleSelfHold = (event) => {
  console.log("You have been put on hold");
  console.log(`Contact: ${event.contactId}`);
};
// Subscribe to all contacts
voiceClient.onSelfHold(handleSelfHold);
// Or subscribe to a specific contact
voiceClient.onSelfHold(handleSelfHold, "contact-123");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantHoldHandler	Event handler function to call when the current user's participant is put on hold
contactId	string	Optional contact ID to filter events for a specific contact

Subscribe to self resume events in Connect Customer agent workspace

Subscribes to events when the current user's participant is taken off hold.

Signature

```
onSelfResume(
```

```
  handler: ParticipantResumeHandler,
  contactId?: string
): void
```

Usage

```
const handleSelfResume = (event) => {
  console.log("You have been resumed from hold");
};
voiceClient.onSelfResume(handleSelfResume, "contact-123");
```

Input

Parameter	Type	Description
handler *Required*	ParticipantResumeHandler	Event handler function to call when the current user's participant is taken off hold
contactId	string	Optional contact ID to filter events for a specific contact

Subscribe to voice enhancement mode change events in Connect Customer agent workspace

Subscribes a callback function whenever voice enhancements mode is changed in user's profile.

Signature

```
onVoiceEnhancementModeChanged(handler: VoiceEnhancementModeChangedHandler)
```

Usage

```
const handler: VoiceEnhancementModeChangedHandler = async (data:
 VoiceEnhancementModeChanged) => {
  console.log("User VoiceEnhancementMode changed! " + data);
}

voiceClient.onVoiceEnhancementModeChanged(handler);
```

```
// VoiceEnhancementModeChanged structure
{
  voiceEnhancementMode: string
  previous: {
     voiceEnhancementMode: string
  }
}
```

Permissions required:

```
*
```

Resume a participant from hold in Connect Customer agent workspace

Resumes a specific participant from hold.

Signature

```
resumeParticipant(participantId: string): Promise<void>
```

Usage

```
await voiceClient.resumeParticipant("participant-456");
console.log("Participant has been resumed");
```

Input

Parameter	Type	Description
participantId *Required*	string	The unique identifier for the participant to resume

Set the voice enhancement mode in Connect Customer agent workspace

Sets the voice enhancement mode of the user that's currently logged in to Connect Customer agent workspace. The voice enhancement mode can have the following values:

- VOICE_ISOLATION: it suppresses background noise and isolates the agent's voice. This mode should only be enabled if the agent uses a wired headsets.
- NOISE_SUPPRESSION: it suppresses the background noise. We recommend using this mode with any type of headset.
- NONE: no voice enhancement applies.

Signature

```
async setVoiceEnhancementMode(voiceEnhancementMode: VoiceEnhancementMode):
 Promise<void>
```

Usage

```
await voiceClient.setVoiceEnhancementMode(VoiceEnhancementMode.NOISE_SUPPRESSION);
```

Input

Parameter	Type	Description
voiceEnhancementMode *Required*	VoiceEnhancementMode	The mode to set on the user. Values accepted: VOICE_ISOLATION , NOISE_SUPPRESSION , NONE

Permissions required:

```
*
```

Document history for the agent workspace developer guide

The following table describes the documentation releases for agent workspace.

Change	Description	Date
Integration of AWS-managed applications and additional APIs	Connect Customer agent workspace introduces support for integrating AWS-manag ed applications with existing applications built using Streams. For details, see Building third-party services.	December 22, 2025
Enhanced third-party capabilities and new AppController API	Connect Customer agent workspace introduces support for third-party services and a new App Controlle r API. Third-party services are headless applications that run in the backgroun d of the agent workspace, enabling automated tasks and enhanced workflows . The App Controller API allows developers to manage third-party applications programmatically. For details, see Building third-party services and App Controller API.	July 25, 2025
API version 1.0.5	API version 1.0.5 released. For more information, see the Connect Customer agent	April 24, 2025

	workspace API reference for third-party applications.	
Initial release	Initial release of the agent workspace developer guide	October 27, 2023