aws

Developer Guide

Agent Workspace

Agent Workspace: Developer Guide

Table of Contents

What is the Amazon Connect Agent Workspace?

Amazon Connect 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. Amazon Connect Agent Workspace integrates all of your agent tools on one screen. You can customize the workspace to present agents with step-by-step guidance to resolve customer issues faster.

Are you a first-time Amazon Connect Agent Workspace user?

If you are a first-time user of Amazon Connect Agent Workspace, we recommend that you begin by reading the following sections:

- Customize the Amazon Connect Agent Workspace.
- Third-party applications (3p apps) in the agent workspace (Preview).
- Getting started with third-party development in the Amazon Connect Agent Workspace.

Getting started with third-party development in the Amazon Connect 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 Amazon Connect agent workspace.

Topics

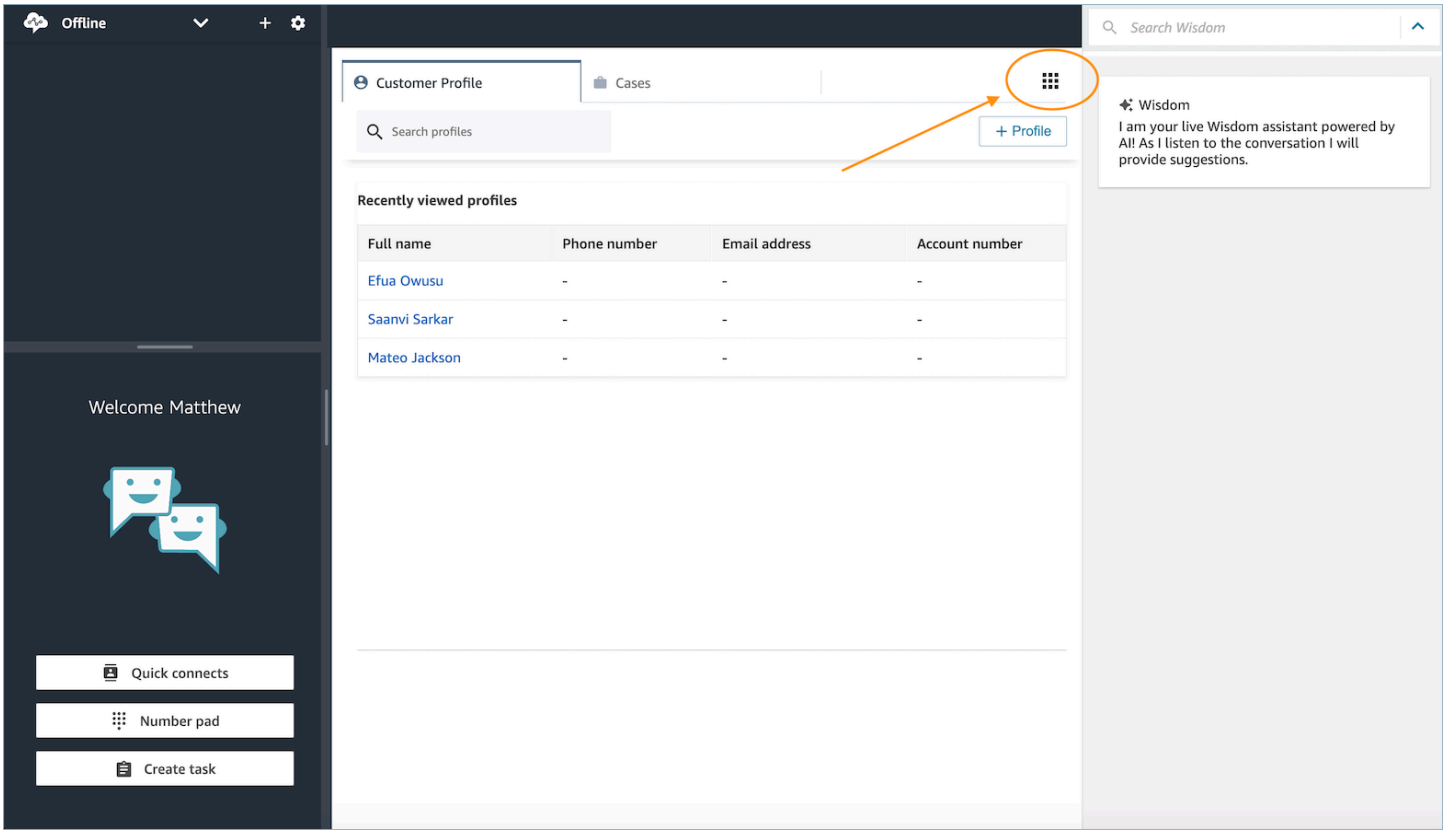
Prerequisites for 3P apps

Developing and testing an application for Connect requires:

- A Connect 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 Appendix.
- A Connect user in that instance that has permissions to update security profiles.

How applications are loaded in the 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 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 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.

Creating your application

An application is a website that can be loaded from an HTTPS URL into an iframe in the 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 application SDK to establish secure communication between the application and the workspace allowing the application to receive events and data from the workspace.

Installing the Amazon Connect SDK

The *Amazon Connect SDK* can be installed from NPM. The 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.

> **Note**
>
> Only ECMAScript modules are supported at this time.

Install from NPM

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

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

Events and Requests

The contact package provides the functionality required to allow the app to subscribe callbacks to agent andcontact events and to request agent and 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.

Install from NPM

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

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

Theme

The theme package defines and applies the Connect theme when developing with Cloudscape.

Install from NPM

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

```
% npm install -P @amazon-connect/theme
```

Initialize the SDK in your application

Initializing the SDK in your app 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 workspace and then when the 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 Lifecycle events 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');
  },
});
```

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 workspace in the allotted time
```

Events and Requests

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

- **Event**

  Refers to an asynchronous subscription-publication model, where the 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 SDK's client allows the 3P app to make requests on demand to retrieve data about the current contact or the logged-in agent.

Authentication

Apps must provide their own authentication to their users. It is recommended that apps use the same identity provider that the Connect 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**
>
> **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 Amazon Connect's 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 Amazon Connect 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 with agent data

Instantiating the agent client

You can instantiate the agent client as follows:

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

const agentClient = new AgentClient();
```

Note

For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. 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 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 { AgentStateChangedEventData } 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: AgentStateChangeEventData) => {
    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 with contact data

Instantiating the contact client

You can instantiate the contact client as follows:

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

```
const contactClient = new ContactClient();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. 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

For all ContactClient event methods which have the optional parameter `contactId` but do not receive an argument for this parameter, the client will default to using the scope of the contact in which the app was opened, for example, the *current contact* from `AppContactScope`. You can also use `AppContactScope` *current contact* value as an argument to the contact request methods to retrieve data about the contact loaded into the workspace. This requires the app being opened in the context of a contact.

Example contact event

The code sample below subscribes a callback to the accepted event topic. Whenever a contact is accepted by the agent, the 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,
    ContactAcceptedEventData,
    ContactAcceptedHandler
} from "@amazon-connect/contact";
import { AppContactScope } from "@amazon-connect/app";

// A simple callback that just console logs the contact accepted event data
// returned by the workspace whenever the current contact is accepted
const handler: ContactAcceptedHandler = async (data: ContactAcceptedEventData) => {
```

```
    console.log(data);
};

// Subscribe to the contact accepted topic using the above handler
contactClient.onAccepted(handler, AppContactScope.CurrentContactId);
```

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

const queue = await contact.getQueue(AppContactScope.CurrentContactId);

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.

Integrate with user data

Instantiating settings client

You can instantiate the voice client as follows:

```
import { SettingsClient } from "@amazon-connect/user";
const settingsClient = new SettingsClient();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, see the API reference to customize your client's configuration. Once the user client is instantiated, you can use it to make requests.

Example user request

The following user event and request are non-exhaustive. For a full list of available voice events and requests, see the API reference.

```
import { SettingsClient } from "@amazon-connect/user";

const settingsClient = new SettingsClient();
const language = await settingsClient.getLanguage();

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

Integrate with voice data

Instantiating the voice client

You can instantiate the voice client as follows:

```
import { VoiceClient } from "@amazon-connect/voice";
const voiceClient = new VoiceClient();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, see the API reference to customize your client's configuration. Once the voice client is instantiated, you can use it to make requests.

Example voice request

The following voice event and request are non-exhaustive. For a full list of available voice events and requests, see the API reference.

```
import { VoiceClient } from "@amazon-connect/voice";
import { AppContactScope } from "@amazon-connect/app";

const voiceClient = new VoiceClient();
const phoneNumber = await voiceClient.getPhoneNumber(AppContactScope.CurrentContactId);

console.log(`Got the phone number: ${phoneNumber}`);
```

Lifecycle events

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

Create

The create event 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.

Destroy

The destroy event will trigger the `onDestroy` callback configured during `AmazonConnectApp.init()`. The application should use this event to clean up resources and

persist data. The workspace will wait for the application to respond that it has completed clean up for a period of time.

Theme

Usage

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

```
// src/index.ts

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

applyConnectTheme();
```

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>;
}
```

Error handling

Apps can communicate errors back to the workspace by either calling `sendError` or `sendFatalError` on the `AmazonConnectApp` object. The 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 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.

Troubleshooting

You can use the SDK's AppConfig object to retrieve data about your applications's setup, 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 SDK to subscribe to an event that it does not have permission for, the 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 SDK to make a request that it does not have permission for, the workspace will throw an error with a message formatted like below.

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

Testing your application locally

Once you have a minimal version of the app with the 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 Amazon Connect 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 workspace data.
5. Select the Amazon Connect 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 *Amazon Connect SDK* and the Create / Destroydestroy lifecycle events.

Test with a deployed version of your application

When ready, deploy your app to a place that is internet accessible and 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.

API reference for 3P apps events and requests

This API reference enumerates the agent events, agent requests, contact events, and contact requests that are supported by the AmazonConnectSDK.

Contents

Agent

Client

The SDK provides an `AgentClient` which serves as an interface that your app 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();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, providing a constructor argument:

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

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

Events

Contents

StateChanged (Subscribing)

Subscribes a callback function to-be-invoked whenever an agent state changed event occurs.

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
```

StateChanged (Unsubscribing)

Unsubscribes the callback function from the agent stated change event.

Signature

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

Usage

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

Requests

Contents

- getARN()
- getName()
- getState()
- getRoutingProfile()
- getChannelConcurrency()
- getExtension()
- getDialableCountries()

getARN()

Returns the Amazon Resource Name(ARN) of the current logged-in user.

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

Permissions required:

```
User.Details.View
```

getName()

Returns the name of the current logged-in user.

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

Permissions required:

```
User.Details.View
```

getState()

Returns the 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.

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

Permissions required:

```
User.Status.View
```

getRoutingProfile()

Returns the agent's routing profile. The routing profile contains the following fields:

- `channelConcurrencyMap`: See agent.getChannelConcurrency() 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
```

getChannelConcurrency()

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

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

Permissions required:

```
User.Configuration.View
```

getExtension()

Returns the agent's phone number. This is the phone number that is dialed by Amazon Connect 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
```

getDialableCountries()

Returns a list of eligible countries to be dialed / desk phone redirected.

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

Permissions required:

```
User.Configuration.View
```

Contact

Client

The SDK provides an `ContactClient` which serves as an interface that your app 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();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, providing a constructor argument:

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

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

Events

Contents

StartingAcw (Subscribing)

Subscribes a callback function to-be-invoked whenever a contact StartingAcw event occurs. 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: ContactStartingAcwEventData) =>
 {
    console.log("Contact StartingAcw occurred! " + data);
};

contactClient.onStartingAcw(handler);

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

Permissions required:

```
Contact.Details.View
```

StartingAcw (Unsubscribing)

Unsubscribes the callback function from the contact StartingAcw event.

Signature

```
offStartingAcw(handler: ContactStartingAcwHandler, contactId?: string)
```

Usage

```
contactClient.offStartingAcw(handler);
```

Connected (Subscribing)

Subscribes a callback function to-be-invoked whenever a contact Connected event occurs. 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: ContactConnectedEventData) => {
    console.log("Contact Connected occurred! " + data);
};

contactClient.onConnected(handler);

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

Permissions required:

```
Contact.Details.View
```

Connected (Unsubscribing)

Unsubscribes the callback function from Connected event.

Signature

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

Usage

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

Destroyed (Subscribing)

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

Signature

```
onDestroyed(handler: ContactDestroyedHandler, contactId?: string)
```

Usage

```
const handler: ContactDestroyedHandler = async (data: ContactDestroyedEventData) => {
    console.log("Contact destroyed occurred! " + data);
};

contactClient.onDestroyed(handler);

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

Permissions required:

```
Contact.Details.View
```

Destroyed (Unsubscribing)

Unsubscribes the callback function from the contact destroyed event.

Signature

```
offDestroyed(handler: ContactDestroyedHandler, contactId?: string)
```

Usage

```
contactClient.offDestroyed(handler);
```

Missed (Subscribing)

Subscribes a callback function to-be-invoked whenever a contact missed event occurs. 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: ContactMissedEventData) => {
    console.log("Contact missed occurred! " + data);
};

contactClient.onMissed(handler);
```

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

Permissions required:

```
Contact.Details.View
```

Missed (Unsubscribing)

Unsubscribes the callback function from the contact missed event.

Signature

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

Usage

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

Requests

Contents

- getQueueTimestamp()

getAttributes()

Returns a map of the attributes associated with the contact. 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
```

getAttribute()

Returns the requested attribute associated with the contact.

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

Permissions required:

```
Contact.Attributes.View
```

getInitialContactId()

Returns the original (initial) contact id from which this contact was transferred, 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 | null>
```

Permissions required:

```
Contact.Details.View
```

getType()

Returns the type of the contact. This indicates what type of media is carried over the connections of the contact.

```
async getType(contactId: string): Promise<ContactType>
```

```
enum ContactType {
  VOICE = "voice",
  QUEUE_CALLBACK = "queue_callback",
  CHAT = "chat",
  TASK = "task",
}
```

Permissions required:

```
Contact.Details.View
```

getStateDuration()

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

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

Permissions required:

```
Contact.Details.View
```

getQueue()

Returns the queue associated with the contact. 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
```

getQueueTimestamp()

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

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

Permissions required:

```
Contact.Details.View
```

User

Client

The SDK provides an `SettingsClient` which serves as an interface that your app 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();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { SettingsClient } from "@amazon-connect/user";

const settingsClient = new SettingsClient({
    context: sampleContext,
    provider: sampleProvider
});
```

Events

Contents

LanguageChanged (Subscribing)

Subscribes a callback function to-be-invoked whenever a user LanguageChanged event occurs.

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
```

LanguageChanged (Unsubscribing)

Unsubscribes the callback function from LanguageChanged event.

Signature

```
offLanguageChanged(handler: UserLanguageChangedHandler)
```

Usage

```
settingsClient.offLanguageChanged(handler);
```

Requests

Contents

- getLanguage()

getLanguage()

Returns the language setting for the current user.

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

Permissions required:

```
User.Configuration.View
```

Voice

Client

The SDK provides an `VoiceClient` which serves as an interface that your app 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();
```

> **Note**
>
> For the zero-arg constructor demonstrated above to work correctly, you must first instantiate the app which will set up the default AmazonConnectProvider. This is the recommended option.

Alternatively, providing a constructor argument:

```
import { VoiceClient } from "@amazon-connect/voice";

const voiceClient = new VoiceClient({
    context: sampleContext,
    provider: sampleProvider
});
```

Requests

Contents

getPhoneNumber()

Returns the voice contact's phone number they are calling from.

```
async getPhoneNumber(contactId: string): Promise<string>
```

Permissions required:

```
Contact.CustomerDetails.View
```

Recommendations and best practices

Authentication

For more information on authentication, see Authentication under the Creating your application section of the *Agent Workspace developer guide*.

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

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

Our App SDK includes a standard 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 Amazon Connect agent workspace.

Appendix

Role required for creating applications

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "app-integrations:CreateApplication",
                "app-integrations:GetApplication",
                "iam:GetRolePolicy",
                "iam:PutRolePolicy",
                "iam:DeleteRolePolicy"
            ],
            "Resource": "arn:aws:app-integrations:<aws-region>:<aws-account-
Id>:application/*",
            "Effect": "Allow"
        }
    ]
}
```

Document history for the Agent Workspace Developer Guide

The following table describes the documentation releases for Agent Workspace.

Change	Description	Date
Initial release	Initial release of the Agent Workspace Developer Guide	October 27, 2023