3.11 Desktop Integration API .NET Version Tutorial

Bright Pattern Documentation

Generated: 10/31/2024 11:51 pm

Table of Contents

Purpose

This tutorial will teach you how to use the ServicePattern Desktop Integration API to control agent states and handle calls from .NET-based applications. The complete specification for the .NET version of this API can be found here: http://www.brightpattern.com/doc/client-API/.

For general information about the Agent Desktop functions discussed this tutorial, see corresponding sections of the *ServicePattern Agent Guide*.

Next >

Audience

This guide is intended for the IT personnel responsible for the data infrastructure of the ServicePattern-based contact centers. Readers of this guide are expected to have expertise in web application development as well as solid understanding of contact center operations.

< Previous | Next >

General Information

The Bright Pattern Contact Center Desktop Integration API .NET Version is encapsulated in the class *ADAPI.AgentPlace* provided from the assembly *ADAPI.dll*.

View the complete specification for the .NET version of this API.

Project and Code Setup

1. In the Visual Studio project where you wish to use the ServicePattern Desktop Integraiton API (.NET Version), add references to the assembly *ADAPI.dll* and *ADUtils.dll*.
2. Instantiate an *ADAPI.AgentPlace* object.
3. Attach event handlers to events generated in ADAPI. This should be done before agent logs in. For a full list of callback events raised by ADAPI, please consult the API specification.

Threading Model

ADAPI will create its own threads automatically once the *AgentPlace* object is created. Unless otherwise stated with respect to a particular element, most of the public methods in *AgentPlace* and other ADAPI objects like *Call* are asynchronous, and can be invoked safely in any thread. The events in *AgentPlace* would be raised in the same thread that creates the *AgentPlace* object.

< Previous | Next >

Agent Login

The *AgentPlace* object provides the following methods to log in to Agent Desktop for different phone device types:

- for softphone (*isSoftphone=true*) or default phone (*isSoftphone=false*) option

  public void loginOnDefaultPhone(string adHost, string loginId, string domain, string password, bool isSoftphone, bool force = false);

- for internal phone (*isInternal=false*) or external phone (*isInternal=false*) option

  public void loginOnHardPhone(string adHost, string loginId, string domain, string password, string phoneNum, bool isInternal, bool force = false);

- for nailed connection (option "dial-in and keep line open")

  public void loginOnVirtualPhone(string adHost, string loginId, string domain, string password, bool force = false);

- for no phone option

  public void loginNoPhone(string adHost, string loginId, string domain, string password, bool force = false);

After authentication, the following event will be raised to indicate whether the login is successful:

public event LoggedInCallback loggedInCallback;

The following method is used to log out:

public override ResultCode logout()

And the following event will be raised once logout is completed:

public event LoggedOutCallback loggedOutCallback;

< Previous | Next >

Agent State

Whenever the agent state changes, the following event will be emitted:

public event StateChangedCallback stateChangedCallback;

The *UserState* object from the callback provides information about the current agent state and the next agent state.

The following methods are used to set the agent state:

- for the Ready state

  public ResultCode setReady();

- for the Not Ready state with optional indication of the reason (the list of configured Not Ready reasons can be obtained in *AgentPlace.notReadyReasons* after a successful login)

  public ResultCode setNotReady(string reason);

< Previous | Next >

Services

After a successful login, the list of services available to the agent can be obtained from the following property:

public SortedDictionary<string, Service> services;

< Previous | Next >

Making a call

The following method is used to make a call:

public ResultCode makeCall(string dnis, Service service, out ADAPI.Item outItem);

The *dnis* parameter should not contain any non-numeric characters such as ":", "+", or "-".

The output parameter *outItem* corresponds to the new ACL item generated for the call.

The following event will be raised when the call is dialed:

public event CallDialingCallback callDialingCallback;

When the other party answers the call and the call is connected, the following event will be raised:

public event CallConnectedCallback callConnectedCallback;

< Previous | Next >

Transfers

The following method in *Call* is used to transfer a primary call to a consultation party (assuming that there is an active consult call):

public ResultCode consultTransfer(Call consultCall)

The following method in *Call* is used to make a single-step transfer:

public ResultCode sstepTransfer(string dest)

< Previous | Next >

Conference

The following code snippet shows how to merge calls into a conference.

- For an conference via consultation, it assumes existence of a held primary call and an active consult call. Execution of the code snippet would be triggered by an explicit user action (e.g., pressing of a *Merge* button).
- For a single-step conference, it assumes existence of an active primary call and another call attempt initiated by the logged-in user. Execution of the code snippet would be triggered by connection of the second call.

```
public void MergeCalls(List<Call> calls) {
    Call conferencingCall = FindConferencingCall(calls, null);
    if (conferencingCall == null)
        return;

    List<Call> callsToMerge = new List<Call>();
    foreach (Call call in calls) {
        if (call.id != conferencingCall.id && call.state != CallState.Disconnected) {
            callsToMerge.Add(call);
        }
    }
    conferencingCall.mergeConference(callsToMerge);
}

private Call FindConferencingCall(List<Call> calls) {
    Call bestCall = null;
```

```
        DateTime bestTime = DateTime.Now;

        foreach (Call call in calls) {
            if (call.state != CallState.Disconnected && call.state != CallState.Unknown) {
                foreach (CallParty cp in call.parties.Values) {
                    switch (cp.phoneType) {
                        case PhoneType.Trunk:
                            return call;
                        case PhoneType.DialOut:
                            return call;
                        case PhoneType.AccessNumber:
                            return call;
                        case PhoneType.Phone:
                        case PhoneType.Softphone:

                            if (bestCall == null) {
                                bestCall = call;
                                bestTime = call.startTime;
                            }
                            else if (call.startTime < bestTime) {
                                bestCall = call;
                                bestTime = call.startTime;
                            }
                            break;
                    }
                }
            }
        }
        return bestCall;
    }
```

To remove a participant from a conference, the *removeParty* method from Call should be used as follows:

```
void RemoveFromConference(string callId, string partyId)

{

    Call call = AgentPlace.call(callId);

        if (call != null)
        {

            CallParty party = call.parties[partyId];
            if (party != null)
            {

                call.removeParty(party);

            }

        }

}
```

To leave a conference (remove oneself from a conference), use the *drop* method from *Call*:

```
public void drop();
```

To end a conference, use the *endConference* method from *Call*:

```
public void endConference();
```

< Previous | Next >

Receiving a call

When a call is made to an agent directly or offered to an agent via an ACD queue, the following events would be raised (in the specified order):

```
public event ItemArrivedCallback itemArrivedCallback;
public event CallOfferedCallback callOfferedCallback;
```

Your application can use *itemArrivedCallback* to create a new item in its active communications list (ACL), while *callOfferedCallback*, which offers more information about the call (e.g. remote name and number), should be used to update the application about the call. The *itemId* property in the *Call* object can be used to find the ACL item by the *id* property in the *Item* object.

When the user answers and the call is connected, the following event will be raised:

```
public event CallConnectedCallback callConnectedCallback;
```

Call States

The *Call* object has a set of methods to control call states.

Answer a ringing call	*public void answer();*
Place the call on hold	*public void hold();*
Retrieve the call from hold	*public void resume();*
Mute the microphone	*public void mute();*
Unmute the microphone	*public void unmute();*
Start or resume call recording	*public ResultCode startRecording();*
Stop or pause call recording	*public ResultCode stopRecording();*
Send DTMF digits to other parties in the call	*public void sendDtmf(string dtmf);*
Disconnect the call	*public void drop();*

In order to reflect changes in call states to the user, the application should subscribe to the following events:

public event CallConnectedCallback callConnectedCallback;

public event CallDataChangedCallback callDataChangedCallback;

public event CallMutedCallback callMutedCallback;

public event CallUnmutedCallback callUnmutedCallback;

public event CallLocalHeldCallback callLocalHeldCallback;

public event CallLocalResumedCallback callLocalResumedCallback;

public event CallRemoteHeldCallback callRemoteHeldCallback;

public event CallRemoteResumedCallback callRemoteResumedCallback;

public event CallDisconnectedCallback callDisconnectedCallback;

The following properties of the *Call* object can be used to check the current status of call recording:

public bool isRecording - specifies it the call is being recorded
public bool isRecordingMuted - specifies if the call is being recorded with actual voice replaced by silence (muted recording)

The following *Call* property specifies the URL postfix which can be used to download the recording.

public string recordingPlaybackUrl

The full URL should be composed by joining the web server address with this postfix.

Assuming that the agent handles one interaction at a time, event *callDisconnectedCallback* will normally be followed by the *stateChangedCallback* event. Depending on the contact center configuration this event may indicate that the agent is in Ready, Not Ready, or After-call Work state.

< Previous | Next >

Wrapping Up After-call Work

Depending on the service configuration, agent state may be changed to After-call Work after the call is disconnected.

For manual completion of interaction while in After-call Work, use this method for the *Item* object:

```
public void complete();
```

The following event will be raised when the interaction is completed:

```
public event ItemCompletedCallback itemCompletedCallback;
```

Note that the same event will be raised if the interaction is finished automatically (by timeout).

< Previous | Next >

Entering Dispositions and Notes

The following code snippet shows how to obtain a list of dispositions for an interaction filtered according to the disposition type:

```
List<Disposition> dispositions = new List<Disposition>();

List<Disposition> allDispositions = item.service.dispositions.Values.ToList();

switch (item.mediaType)

        {

            case ADAPI.MediaType.Preview:

                dispositions.AddRange(allDispositions.Where(x => x.preview && x.showToAgent));
                break;

            case ADAPI.MediaType.Voice:

                if (String.IsNullOrEmpty(item.workitemId))

                    {

                        dispositions.AddRange(allDispositions.Where(x => x.inbound && x.showToAgent));
```

```
                }
            else
                {
                    dispositions.AddRange(allDispositions.Where(x => x.outbound && x.showToAgent));
                }
            break;
        default:
            dispositions.AddRange(allDispositions.Where(x => x.showToAgent));
            break;
    }
```

The method below is used for:

- entering a disposition and notes for a processed call
- rescheduling a call attempt
- indicating whether the voice recording of this call contains a voice signature

```
public ResultCode disposition((string dispositionId, string notes, DateTime reschFrom, DateTime reschTo, string reschPhone, string reschTZ, bool bVoiceSignature);
```

This method can be used both while the call is in progress and after it is disconnected while the user is in the After-call Work.

< Previous | Next >

Directory

There are two types of directory items in the *AgentPlace* object that are available immediately after agent logs in, users grouped in **teams** and static entries grouped in **categories** (folders).

The list of Team objects can be accessed via:

```
public SortedDictionary<string, Team> teams;
```

This list corresponds to the teams in the Contact Center Administrator application. The *User* list associated with a *Team* is loaded on demand by the following method in *Team*:

```
public void requestMembers(bool includeLoggedOut);
```

When the *User* list is available, the following event from the *AgentPlace* would be raised:

public event TeamMembersReceivedCallback teamMembersReceivedCallback;

If your application needs to receive notifications about changes in attributes of a team member, the following method in *Team* should be invoked:

public void subscribe();

When any change in attributes of a team member occurs in that team, the following method would be raised:

public event UserInfoUpdatedCallback userInfoUpdatedCallback;

Note that only one team subscription is allowed at a time (a new subscription will cancel the previous team subscription).

The list of DirectoryCategory objects can be accessed via:

public SortedDictionary<string, DirectoryCategory> directoryCategories;

This list corresponds to list of folders of static entries in the Contact Center Administrator application. The *DirectoryItem* list associated with *DirectoryCategory* is loaded on demand by the following method in *DirectoryCategory*:

public void requestItems();

When the DirectoryItem objects are available, the following event from*AgentPlace* would be raised:

public event DirectoryItemsReceivedCallback directoryItemsReceivedCallback;

< Previous | Next >

Recent Interactions

The list of recent interactions for the logged user is available in the following property in*AgentPlace* after agent login:

public SortedDictionary<string, Recent> recents;

The following code snippet shows how to sort the recent interactions according to the timestamp of the interactions:

```
List<Recent> recentList = _AgentPlace.recents.Values.ToList();

recentList.Sort((x, y) => x.timestamp.CompareTo(y.timestamp));
```

< Previous | Next >

Favorites

The following method from *AgentPlace* is used to add a contact to the favorites list of the logged user:

public Favorite addToFavorites(string name, string number);

The list of favorites is available in the following property in *AgentPlace* after agent login:

public SortedDictionary<string, Favorite> favorites;

The following method from AgentPlace is used to remove a contact from the favorites list:

public ResultCode removeFromFavorites(string number);

< Previous | Next >

Notifications

The following events in *AgentPlace* are be raised when the corresponding conditions are detected in the logged user's Agent Desktop environment. Your application may use these events to alert the users.

When user's soft phone status changes:

public event SoftphoneStatusCallback softphoneStatusCallback;

When input or output audio device is changes:

public event AudioDeviceChangedCallback audioDeviceChangedCallback;

When user's phone capabilities change:

public event PhoneCapabilitiesChangedCallback phoneCapabilitiesChangedCallback;

When the audio quality problem is detected for the current call:

public event CallAudioQualityAlertCallback callAudioQualityAlertCallback;

When an error occurs:

public event ErrorCallback errorCallback;

< Previous >]]