Exotel Voice Websdk Bundle

Integration Guide

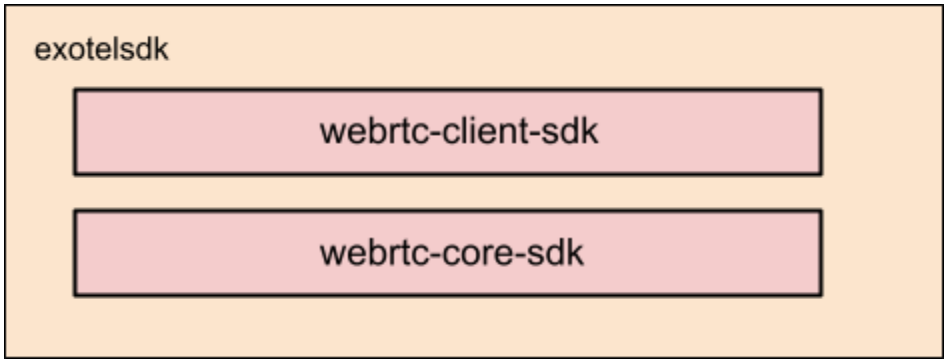
Version	Date	Changes
1.0	02-01-2023	Initial Upload

Content

1. Introduction

Exotel Webrtc SDK bundle enables you to add the voip calling feature into your web app. This document outlines the integration and usage. The Exotel Webrtc SDK package is layered. We have web-client-sdk that provides higher level APIs and callbacks to register/unregister, receive calls and operate on calls like mute/unmute, hold/unhold. And we have webrtc-core-sdk that provides the underlying state machine and SIP protocol stack.

The current document provides API and Callback details for the web-client-sdk.



2. Licensing

You need an exotel account to use the voip calling functionality with this websdk. Contact exotel support for demo and account creation.

3. Glossary

Terminology	Description
App	Web Application
VOIP	Voice over IP
Client	User / Subscriber / Client signing up to use the web app
Customer	Exotel's customer licensing the SDK

4. Getting Started

4.1. Software Package

The Exotel Webrtc SDK bundle software package includes

- Exotelsdk.js bundle with required media files
- Integration Guide
- Sample application(without npm) for reference

4.2. Add Web Client SDK Library to Project

Step 1: download the exotelsdk tar file from releases.

https://github.com/exotel/exotel-voip-websdk-sampleapp/releases

Step 2: untar the tar file and place the dist folder in your sample app.

4.3. Supported Browsers

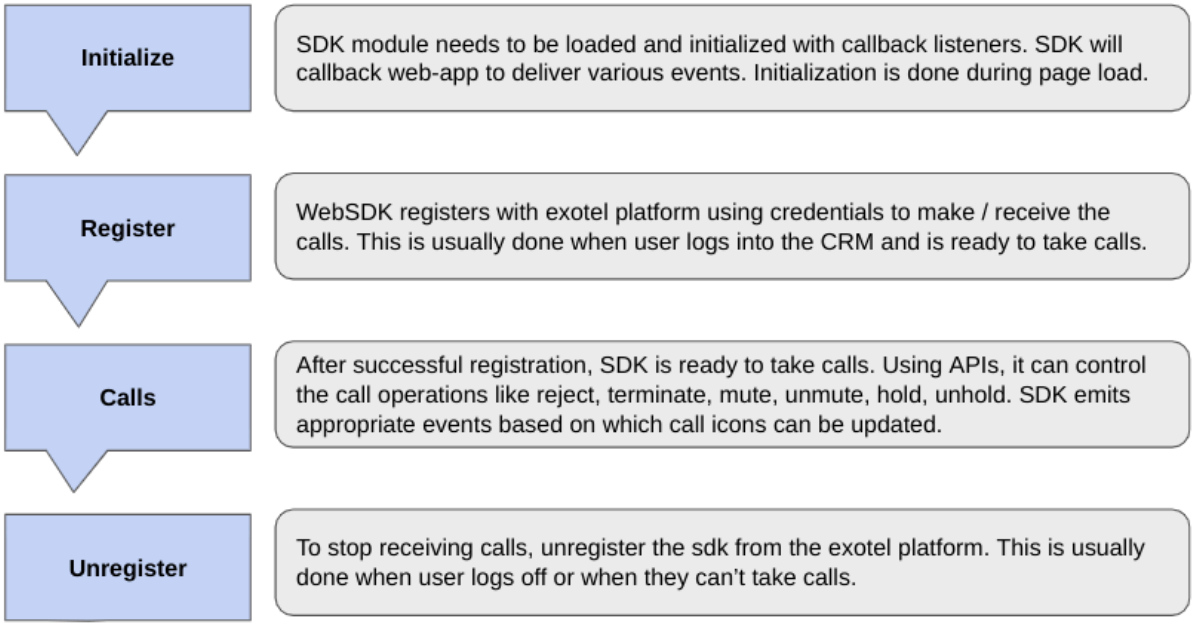
The SDK has been verified with the following browsers.

OS	Chrome	Firefox	Safari (> V11)	Edge
Windows				
Linux				
MacOS				

Browser not compatible with OS

WebSDK supported

5. Web Client SDK APIs and Integration Workflow



5.1. Initialize Library

WebRTC client SDK library needs to be initialized prior to using. A sample code snippet to do so is as below.

Step1: Import the exotel client library as below.

in npm-module

```
const exotelSDK = require('./dist/exotelsdk.js');
const  exWebClient = new exotelSDK.ExotelWebClient();
```

in non-npm-module (refer demo-non-npm)

Load the **exotelsdk.js** in html

```
<script type="text/javascript" src="./dist/exotelsdk.js"></script>
```

inialize the webclient instance in js file.

```
const  exWebClient = new exotelSDK.ExotelWebClient();
```

Step2: Initialize the Exotel Web Client with SipAccountInfo and Callbacks using the API initWebrtc.

```
// Initialise sipAccountInfo  dictionary
   const sipAccountInfo = {
     'userName': 'Username',
     'authUser': 'Username',
     'sipdomain': 'Domain',
     'domain': 'HostServer' + ":" + Port
     'displayname': 'DisplayName',
     'secret': 'Password',
     'port': 'Port',
     'security': 'wss',
   }

// Initialize Callbacks:

exWebClient.initWebrtc(sipAccountInfo,
                RegisterEventCallBack,
                CallListenerCallback,
                SessionCallback)
```

This API takes one argument that contains the details of SIP Account during the initialization. Refer to the Messages section to know more about the parameters.

This API also takes as input three callbacks:

- RegisterEventCallBack handles registration states as they change.
- CallListenerCallback handles call events as they occur.
- SessionCallback handles notifications for multiple tab sessions.

The details of the callbacks are explained in the corresponding flows.

5.2. Register the SIP Phones

Before receiving / making any call, the SIP phone needs to be registered; and the API for this purpose is “DoRegister”.

API Name	ExotelWebClient.DoRegister
Args	None
Exported To	Application UI

```
//Ensure that initWebRTC is invoked before calling DoRegister

exWebClient.DoRegister();
```

Once the registration has been done, the response comes in the registrationCallback with the events, “registered ”/ ”terminated” as below.

API Name	RegisterEventCallBack
Args	state:"registered"/"unregistered"/'terminated'/ 'sent_request' phone: username
Exported To	Application UI

```
function RegisterEventCallBack (state, phone){
   if (state === 'registered') { // Successful registration
     setRegState(true)
   } else if (state === 'unregistered') { //Successful unregistration
     setRegState(false)
   } else if (state === 'terminated')  { //Registration/Unregistration failed
     setRegState(false)
   } else if (state === 'sent_request')  { //Registration/Unregistration Request Sent
   }
```

```
}
```

5.3. UnRegister the SIP Phones

To stop getting calls anymore, the SIP phones need to be unregistered. The API to do so is “UnRegister”.

API Name	ExotelWebClient.UnRegister
Args	None
Exported To	Application UI

```
//Ensure that initWebRTC is invoked before calling DoRegister.

exWebClient.UnRegister();
```

The response to unregistration comes in the registrationCallback as below.

API Name	RegisterEventCallBack
Args	state:"registered"/"unregistered"/'terminated'/ 'sent_request' phone: username
Exported To	Application UI

```
function RegisterEventCallBack (state, phone){
  if (state === 'registered') { // Successful registration
    setRegState(true)
  } else if (state === 'unregistered') { //Successful unregistration
    setRegState(false)
  } else if (state === 'terminated')  { //Registration/Unregistration failed
    setRegState(false)
  } else if (state === 'sent_request')  { //Registration/Unregistration Request Sent
```

```
    if (unregisterWait === "true") {
      unregisterWait = "false";
      setRegState(false)
    }
  }
}
```

Note 1: If you have more than one phone, the second argument “phone” would give the indication as to which phone got unregistered.

Note 2: In some cases there won't be any successful response for the “unregister”. In such situations, it is advised to handle “unregistration” flow with the “sent_request” event as shown in the above example.

5.4. Receive Calls

Once the registration has happened, and when there is an incoming call, the callback registered for “Call Events” would get an event along with the details of the incoming call.

API Name	CallListenerCallback
Args	callObj, eventType, phone
Exported To	Application UI

Description	callObj = {
	callId,
	callState,
	callDirection,
	callStartedTime,
	remoteDisplayName
	}
	
	eventType =incoming/connected/callEnded/activeSession
	incoming: show incoming call message
	connected : open dialer
	callEnded: close dialer
	activeSession: session continuity.
	
	Phone = username identifying the phone to which the call is coming.

In the following snippet,the UI states are appropriately modified based on the call events.

```
function CallListenerCallback(callObj, eventType, phone) {
  if (eventType === 'incoming') { //Incoming Call
    setCallComing(true)
  }  else if (eventType === 'connected') { //Call in connected state
    setCallComing(false)
    setCallState(true)
  }  else if (eventType === 'callEnded') {//Call ended
    setCallComing(false)
    setCallState(false)
  } else if (eventType === 'terminated') {//Call terminated
    setCallComing(false)
    setCallState(false)
  }
}
```

5.5. Accept Calls

Once the incoming call event is received, based on the user action the call can be accepted. The API to do so is “Call.Answer” as below. The call object could be obtained by invoking “getCall()” method on exWebClient object.

API Name	Call.Answer
Args	None
Exported To	Application UI

```
function acceptCallHandler() {
  call =  exWebClient.getCall()
  call.Answer();
}
```

The response event to accept calls comes in “CallListenerCallback”. Successful acceptance results in a “connected” event. Other possible events are:

“callEnded” : Call ended locally.

“terminated” : Call terminated remotely.

```
function CallListenerCallback(callObj, eventType, phone) {
  if (eventType === 'incoming') { //Incoming Call
    setCallComing(true)
  }  else if (eventType === 'connected') { //Call in connected state
    setCallComing(false)
    setCallState(true)
  }  else if (eventType === 'callEnded') {//Call ended
    setCallComing(false)
    setCallState(false)
  } else if (eventType === 'terminated') {//Call terminated
    setCallComing(false)
    setCallState(false)
  }
}
```

5.6. Hangup / Reject Calls

Once the incoming call event is received, based on the user action the call can be rejected. The API to do so is “Call.Hangup” as below.

API Name	Call.Hangup
Args	None
Exported To	Application UI

```
function rejectCallHandler() {
  call =  exWebClient.getCall()
  call.Hangup();
}
```

For call hangup by local user, the response comes as a “callEnded” event. For call hangup by remote user,, the event comes as “terminated”.

```
function CallListenerCallback(callObj, eventType, phone) {
  if (eventType === 'callEnded') {//Call ended
    setCallComing(false)
    setCallState(false)
  } else if (eventType === 'terminated') {//Call terminated
    setCallComing(false)
    setCallState(false)
  }
}
```

5.7. Mute / Unmute Calls

Once the call is in progress, based on the user action the call can be muted/unmuted. The APIs to do so are “Call.Mute” and “Call.UnMute” as below.

API Name	Call.Mute
Args	None
Exported To	Application UI

API Name	Call.UnMute
Args	None
Exported To	Application UI

You can also use a single API to toggle the mic using Call.MuteToggle.

API Name	Call.MuteToggle
Args	None
Exported To	Application UI

```
function muteHandler() {
  call =  exWebClient.getCall()

  //call.MuteToggle(); // or

  if (!callOnMute) {
    call.Mute()
    callOnMute = true
  } else {
    call.UnMute()
    callOnMute = false
  }

}
```

There is no callback event for mute. Call remains “connected” but with the mic muted. This is a toggle operation.

5.8. Hold / Resume Calls

Once the call is in progress, based on the user action the remote user can be set on hold/unhold. The API to do so is “Call.Hold” and “Call.UnHold” as below.

API Name	Call.Hold
Args	None
Exported To	Application UI

API Name	Call.UnHold
Args	None
Exported To	Application UI

You can also use a single API to hold the remote caller using Call.HoldToggle.

```
function holdHandler() {
  call =  exWebClient.getCall()

  //call.HoldToggle(); // or

  if (!callOnHold) {
    call.Hold()
    callOnHold = true
  } else {
    call.UnHold()
    callOnHold = false
  }

}
```

There is no callback event for call hold. Call remains in “connected” but in sendonly mode. This is a toggle operation.

5.9. Multitab Scenarios

When the webrtc sdk is loaded in multiple tabs then all the instances will register with the backend and receive incoming call alerts. There are two ways to avoid this,

- Maintain a single login session for the user in your webapp so that only one instance of the webrtc sdk is loaded. This is preferred.
- Maintain a parent-child relationship across the tabs in your webapp so that call is handled only in the parent tab.

Exotel Webrtc-SDK supports multiple tab sessions using Broadcast Channel. In this feature, session listener and session callbacks can be used to get the indication on child tabs. A SessionListener creates a broadcast channel and sends a broadcast event to each child tab for the following events, incoming / connected / callEnded / re-register. When a child tab (as per the logic maintained by the “Application Backend”.) receives a session event, it may notify but not handle the callback.

When the parent tab is destroyed, a re-register event comes to child tabs, based on the logic of the client, a child can opt to be the new parent.

Note 1: The logic of maintaining the parent and child tabs has to be in “Application Backend” by the “Customer”.

Note 2: If multitab scenario is not used, there is no need to handle the events in the session callback.

API Name	SessionListener
Args	None
Exported To	Application UI

```
SessionListener();  // To be called during initialization.
```

API Name	SessionCallback
Args	callState, phone
Exported To	Application UI
Description	callState =incoming/connected/callEnded/re-register incoming : child tab to show a notification message connected : child tab to close the notification message callEnded : child tab to close the notification message re-register : child tab to register when parent tab is closed Phone = username identifying the phone to which the call is coming.

A sample code snippet for session callback is as below.

```
function SessionCallback(callState, phone) {
    /**
     * SessionCallback is triggered whenever an incoming call arrives
     * which needs to be handled across tabs
     */
    switch(callState){
      case 'incoming':
        console.log('incoming call' + phone)
```

```
        /**
         * Display a different notification popup in case of child tabs
         */
        if(window.sessionStorage.getItem('activeSessionTab') !== 'parent0'){
          const message = 'Incoming call from ' + phone + ' ,Switch tab to find dialpad'
          setMessage(message);
        }
      break;
      case 'callEnded':
        /**
         * When call is either accepted or rejected then this is gets shutdown
         */
        console.log('call ended' + phone)
        setOpen(false);
      break;
      case 'connected':
        /**
         * When call is connected close the notification popup on child tabs
         */
        console.log('call connected' + phone)
        setOpen(false);
      break;
      case 're-register':
        /**
         * In case if the main/parent tab is closed then make the subsequent tab in the
tab list as the parent tab
         * and send register for the same and also make that tab as the master
         */
          window.sessionStorage.removeItem('activeSessionTab');
          window.sessionStorage.setItem('activeSessionTab', 'parent0');
          sendAutoRegistration();
        }
        break;
    }
  };
```

5.10. Device and Network Diagnostics

5.10.1. Initialize diagnostics.

Diagnostics can be initialized by passing two callbacks, one trouble shooting logs and one to get the responses of diagnostics back.

API Name	ExotelWebClient.initDiagnostics
Args	diagnosticsReportCallback, keyValueSetCallback
Exported To	Application UI

The signature of the callbacks are as below.

diagnosticsReportCallback is for logs

```
function diagnosticsReportCallback(logStatus, logData) {
 //logStatus : Additional information on the logs, can be ignored as of now.
  //logData : Troubleshooting log to save in a file..
}
```

diagnosticsKeyValueCallback is for test responses, described in the following sections.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key : Indicates the type of response
  //status: the value/status specific to key
  //description : description specific to key
}
```

Immediately after invoking the initDiagnostics, three parameters are returned through the diagnosticsKeyValueCallback callback.

browserVersiom	browserName/browserVersion. Eg. Chrome/101.0.0.0
micInfo	Mic name returned by the browser. Eg. “Built-in Audio Analog Stereo”
speakerInfo	Speaker Name returned by the browser. Eg. “Built-in Audio Analog Stereo”

5.10.2. Speaker Test

Speaker test can be started by invoking the API “startSpeakerDiagnosticsTest”.

API Name	ExotelWebClient.startSpeakerDiagnosticsTest
Args	None
Exported To	Application UI

This starts a test by playing a ring tone. And as the ring tone gets played, the volume levels are passed through the callback function “diagnosticsKeyValueCallback” which can then be used to render the UI to show volume meter.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:“speaker”
  //status: a floating point value
  //description : "speaker ok"/”speaker error”
}
```

Once the user response is captured, it can be passed back to the API, “stopSpeakerDiagnosticsTest " as below. The responses are passed as arguments. THe

API Name	ExotelWebClient.stopSpeakerDiagnosticsTest
Args	Optional, if present, “yes”/”no”.
Exported To	Application UI

The response “Yes” indicates that the user has heard the speaker's sound. “No” indicates that the user did not hear the speaker's sound. This response is further used to update the troubleshooting logs.

If no arguments are passed, only the test is terminated. No updates would be made to

troubleshooting logs.

5.10.3. Mic Test

Mic test can be started by invoking the API “startMicDiagnosticsTest”.

API Name	ExotelWebClient.startMicDiagnosticsTest
Args	None
Exported To	Application UI

This starts a test by capturing the audio spoken on the mic. And as the audio is analysed, the volume levels are passed through the callback function “diagnosticsKeyValueCallback” with key as “mic” which can then be used to render the UI to show mic volume meter.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:“mic”
  //status: a floating point value
  //description : "mic ok"/”mic error”
}
```

Once the user response is captured, it can be passed back to the API, “stopMicDiagnosticsTest " as below. The responses are passed as arguments. THe

API Name	ExotelWebClient.stopMicDiagnosticsTest
Args	Optional, if present, “yes”/”no”.
Exported To	Application UI

The response “Yes” indicates that the user’s has been captured by the mic. “No” indicates that the user's voice could not be captured. This response is further used to update the troubleshooting logs.

If no arguments are passed, only the test is terminated. No updates would be made to troubleshooting logs.

5.10.4. Network Diagnostics

Network diagnosis can be started by invoking the API “startNetworkDiagnostics”.

API Name	ExotelWebClient.startNetworkDiagnostics
Args	None
Exported To	Application UI

This API starts network operations testing. The callback “diagnosticsKeyValueCallback” is called with appropriate keys after each test completion.

a. Web Socket Connection Callback

Returns a WSS url with status “connected” on successful connectivity.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:“wss”
  //status = connected/disconnected
  //description = WSS URL
}
```

b. User Registration Status Callback

Returns a status “connected” on successful registration of the configured “username”. This is the callback from the background registration requests. No explicit registration requests are sent specifically for diagnostics purposes.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:“userReg”
  //status - "registered"/"unregistered"
  //description - userName
}
```

c. TCP connectivity callback

Returns a key value “tcp”with ice candidate information as description.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:"tcp"
  //status: connected/disconnected
  //description : ice candidate line for tcp connectivity/empty string
}
```

d. UDP connectivity callback

Returns a key value "udp" with ice candidate information as description.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:"udp"
  //key: connected/disconnected
  //description : ice candidate line for udp connectivity/empty string
}
```

e. Host connectivity callback

Returns a key value "host" with ice candidate information as description for internal network.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:"host"
  //key: connected/disconnected
  //description : ice candidate for the host connectivity (local facing)/empty string
}
```

f. Reflexive connectivity callback

Returns a key value "srflx" with ice candidate information as description for external network.

```
function diagnosticsKeyValueCallback(key, status, description) {
  //key:"srflx"
  //key: connected/disconnected
  //description : ice candidate for the reflex connectivity (remote facing)/empty string
}
```

5.11. Auto reconnect in case of websocket connection failure

Sometimes due to network issue, websocket connection get disconnected. In that case application has to retry the connection. To implement it we can store the state for shouldAutoRetry, and during doRegistration it could be set as true, and during explicit unregistration it could be set as false, and based on unregistered event we can invoke doRegister API.

```
shouldAutoRetry = false;

/* Event Handlers */

const registerHandler = () => {

  registrationRef.current = "Sent register request:" + phone.Username;

  setRegistrationData(registrationRef.current)



  if (!configUpdated) {

    updateConfig();

  }



  unregisterWait = "false";



  initialise_callbacks();

  console.log("App.js: Calling DoRegister");

  shouldAutoRetry = true;

  exWebClient.DoRegister();



};



const unregisterHandler = () => {
```

```
  registrationRef.current = "Sent unregister request:" + phone.Username;
  setRegistrationData(registrationRef.current)


  if (!configUpdated) {
    updateConfig();
  }


  initialise_callbacks();


  unregisterWait = "true";
  shouldAutoRetry = false;
  exWebClient.UnRegister();
};
```

```
function RegisterEventCallBack(state, sipInfo) {
  document.getElementById("status").innerHTML = state;
  if (state === 'registered') {
      document.getElementById("registerButton").innerHTML = "UNREGISTER";
  } else {
      document.getElementById("registerButton").innerHTML = "REGISTER";
      if (shouldAutoRetry) {
          exWebClient.DoRegister();
      }
  }
```

```
}
```

5.12. Check SDK Readiness

- To check the SDK readiness, wether SDK is ready to receive a call or not. We can invoke checkClientStatus API with callback method.
- First it checks if microphone is available or not, then it checks websocket is connected or not, then it check if user is registered or not.

Event	Event Decription
media_permission_denined	either media device not available, or permission not given
not_intialized	sdk is not intialied
websocket_connection_failed	websocket connection is failing, due to network connectivity
unregistered,terminated	either your credential is invalid or registration keep alive failed.
initial	sdk registration is progress
registered	ready
unknown	something went wrong
disconnected	websocket is not connected
connecting	Trying to connect the websocket

```
exWebClient.checkClientStatus(function (status) {

      console.log("SDK Status " + status);

  });
```

6. Messages

sipAccountInfo	
authUser	SIP username to register.
userName	Used as a unique map index for phones. Same as authUser.
displayName	Local Displayname on Dialer.
secret	SIP Password
sipdomain	SIP public domain
security	“wss”/”ws” typically “wss”
port	443 for Websockets.
sipUri	Complete SipURI (Internally constructed, Customer Can leave this blank)
contactHost	IP Address to contact back (Internally found through STUN discovery, customer can leave this blank)

7. Integration with Exotel APIs

Refer to https://developer.exotel.com/api/make-a-call-api for exotel platform integration APIs. For example, to make a outbound call from a webclient to a pstn phone the request will be

```
curl -s -X POST
https://<your_api_key>:<your_api_token><subdomain>/v1/Accounts/<your_account_sid>/Calls/connect -d "From=<sip_user_id>" -d "CallerId=<caller_id>" -d "To=<phone number>"
```

8. Support Contact

Please write to hello@exotel.in for any support required with integration.