Cisco Remote Expert Mobile Expert Assist Developer's Guide, Release 11.6 (1)

First Published: August 2017
Last Modified: January 2018

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

Contents

Preface

Change History

This table lists the major changes made to this guide. The most recent changes appear at the top.

Changes	Section	Date
Initial release of document for Release 11.6(1)		Aug 2017
Rearranged and revised chapters: ■ CSDK for Web (JavaScript) ■ CSDK for iOS (Objective C) ■ CSDK for Android (Java) chapters	Remote Expert Mobile—CSDK for Web (JavaScript) on page 23 Remote Expert Mobile—CSDK for iOS (Objective C) on page 45 Remote Expert Mobile—CSDK for Android (Java) on page 73	
Removed Callbacks and Annotations sections	Callbacks, Annotations	
Added sections: ■ Permissions ■ Consumer Session Creation sections	Permissions on page 19 Consumer Session Creation on page 15	
Updated iOS and Xcode Supported Versions section	iOS and Xcode Supported Versions on page 46	
Single stylesheet for all documents. Spelling and punctuation corrections.	Throughout	
Added Internationalization section	Internationalization on page 69	
Moved Using UUI sections	the Using UUI section	
Added Establishing an RE Mobile Session section including: ■ With Voice and Video ■ Co browse only (with correlation ID) ■ Co-browse only (with code)	Establishing a RE Mobile Session on page 7	
Added sections: ■ Expert Assists with no voice or video by using Correlation ID ■ Starting a Co-browse only Session sections	Co-browse only mode—Expert Assist with no voice or video by using Correlation ID on page 51 Starting a Co-browse only Session on page 79	
Updated Remote Expert Co-browse section	Remote Expert Co-browse on page 2	
Added C-browse only (with code) section	Co-browse only (with code) on page 10	

About this Guide

This document outlines the steps needed to develop mobile and web applications that leverage Cisco Remote Expert Mobile.

Developers using this guide should have experience in JavaScript, Objective C, or Java depending on the application type.

- Web—it is assumed that the developer is familiar with JavaScript, HTML and CSS
- iOS—it is assumed that the developer is familiar with iOS, Xcode, and Objective-C
- Android—it is assumed that the developer is familiar with Android, Java, and the Android SDK

This guide also assumes that you are familiar with basic contact center and unified communications terms and concepts.

Successful deployment of Remote Expert Mobile also requires familiarity with the information presented in the *Solution Design Guide for Unified Contact Center Enterprise, Release 11.6* (available at http://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucm/srnd/collab11/collab11.html). To review IP Telephony terms and concepts, see the documentation at the preceding link.

Audience

The primary audience for this guide is developers who need to use the co-browsingl features of Remote Expert Mobile in their applications.

Related Documents

Consult these documents for details of these subjects that are not covered in this guide.

Subject	Link
Compatibility Matrix for information on which versions of which products are supported for a contact center enterprise solution.	https://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/products-device-support-tables-list.html
Cisco Unified Contact Center Enterprise Features Guide for detailed information on the configuration and administration of integrated features in your solution.	http://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/products-feature-guides-list.html
Cisco Collaboration Systems Solution Reference Network Designs for detailed information on the Unified Communications infrastructure on which your solution is built.	http://www.cisco.com/c/en/us/support/unified-communications/unified-communications-manager-callmanager/products-implementation-design-guides-list.html

You can find the full documentation of each of the components in the Unified CCE solution at these sites:

Component	Link
Cisco Unified Contact Center Enterprise	http://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/tsd-products-support-series-home.html
Cisco Finesse	http://www.cisco.com/c/en/us/support/customer-collaboration/finesse/tsd-products-support-series-home.html
Cisco MediaSense	http://www.cisco.com/c/en/us/support/customer-collaboration/mediasense/tsd-products-support-series-home.html
Cisco SocialMiner	http://www.cisco.com/c/en/us/support/customer-collaboration/socialminer/tsd-products-support-series-home.html
Cisco Unified Customer Voice Portal	http://www.cisco.com/c/en/us/support/customer-collaboration/unified-customer-voice-portal/tsd-products-support-series-home.html

Component	Link
Cisco Unified Intelligence Center	http://www.cisco.com/c/en/us/support/customer-collaboration/unified-intelligence-center/tsd-products-support-series-home.html
Cisco Virtualized Voice Browser	http://www.cisco.com/c/en/us/support/customer-collaboration/virtualized-voice-browser/tsd-products-support-series-home.html

Organization of this Guide

The guide includes the following sections:

Introduction	Introduction and brief overview of Remote Expert Mobile.
Technologies	Description of the technologies used by Remote Expert Mobile.
Developer Overview	Describes the general components of the solution, provides an understanding and sequencing for Establishing a RE Mobile Sessions, Using Anonymous and Restricted Client Access as well as invoking co-browse only mode.
CSDK Security Highlights for Developers	A general overview of security concerns for developers.
Permissions	Overview of how to mask elements of a website from agents' view, using Permissions added to the HTML element.
Remote Expert Mobile—CSDK for Web	Quick start and specifics for embedding CSDK in web applications. Also includes details for masking and hiding sensitive information within co-browse sessions
Remote Expert Mobile—CSDK for iOS (Objective C)	Quick start and specifics for embedding CSDK in Apple iOS applications. Also includes details for masking and hiding sensitive information within co-browse sessions.
Remote Expert Mobile—CSDK for Android (Java)	Quick start and specifics for embedding CSDK in Android applications. Also includes details for masking and hiding sensitive information within co-browse sessions.
Acronym List	Lists some common industry and Cisco-specific acronyms relevant to Remote Expert Mobile.

Obtaining Documentation and Submitting a Service Request

To receive new and revised Cisco technical content directly to your desktop, you can subscribe to the What's New in Cisco Product Documentation RSS feed. RSS feeds are a free service.

Field Alerts and Field Notices

Cisco products may be modified or key processes may be determined to be important. These are announced through use of the Cisco Field Alerts and Cisco Field Notices. You can register to receive Field Alerts and Field Notices through the Product Alert Tool on Cisco.com. This tool enables you to create a profile to receive announcements by selecting all products of interest.

Log into www.cisco.com and then access the tool at http://www.cisco.com/cisco/support/notifications.html.

Documentation Feedback

To provide comments about this document, send an email message to the following address: contactcenterproducts_docfeedback@cisco.com.

We appreciate your comments.

Conventions

This document uses the following conventions:

Convention	Indication
boldface font	Boldface font is used to indicate commands, such as user entries, keys, buttons, and folder and submenu names. For example: ■ Choose Edit > Find. ■ Click Finish.
italic font	Italic font is used to indicate the following: ■ To introduce a new term. Example: A *skill group* is a collection of agents who share similar skills. ■ A syntax value that the user must replace. Example: IF (*condition, true-value, false-value*) ■ A book title. Example: See the *Cisco Unified Contact Center Enterprise Installation and Upgrade Guide*.
[ ]	Elements in square brackets are optional.
{ x | y | z }	Required alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A non-quoted sequence of characters. Do not use quotation marks around the string or the string will include the quotation marks.
`window font`	Window font, such as Courier, is used for the following: ■ Text as it appears in code or that the window displays. Example: `<html><title>Cisco Systems, Inc. </title></html>`
< >	Angle brackets are used to indicate the following: • For arguments where the context does not allow italic, such as ASCII output. • A character string that the user enters but that does not appear on the window such as a password.
[ ]	Default responses to system prompts are in square brackets.
!, #	An exclamation point (!) or a pound sign (#) at the beginning of a line of code indicates a comment line.

CHAPTER 1

Introduction

Cisco Remote Expert Mobile is a software solution that enables personal and actionable customer interactions within mobile and web applications. These interactions range from simple click-to call to a complete voice, video and Expert Assist customer engagement session interconnected to a full contact center environment. For example, Cisco Remote Expert can connect individual investors to the next available financial adviser within a mobile trading app (B2C—Business to Consumer) or a field employee's mobile app routing into an internal help-desk (B2E—Business to Employee).

Features

With Cisco Remote Expert Mobile you can deliver voice, video and Expert Assist co-browse and application sharing in mobile or web applications. Cisco Remote Expert Mobile is designed specifically for remote collaboration services provided through Cisco Unified Communications Manager, Cisco Unified Contact Center Enterprise (Unified CCE) and Cisco Unified Contact Center Express (Unified CCX). Remote Expert Mobile offers the following features and options that are pre-sized within core components. Core component features are:

- In-app voice and video communications (Over-the-Top WebRTC communications)
  - High definition video and audio
  - Bi-directional or one-way video
  - Mute audio, video or both
  - Client side call control
- WebRTC to SIP gateway (trunking into Cisco Unified Border Element and Unified Communications Manager)
- Expert Assist
  - Web co-browse
  - Mobile app sharing

  - Remote app control
  - Expert form editing and completion
  - Annotation by expert
  - Expert document push
  - Expert URL sharing
  - Protect sensitive data with field and data masking
- Media Handling:
  - STUN server (RFC 5389) for client external IP identification
  - UDP port multiplexing
  - Media encryption / decryption
  - Bidirectional audio
  - High definition video (H.264 or VP8 in CIF (352x288), nHD (640x480),720p (1280x720
  - High definition and narrowband audio codec support (Opus, G.711 ulaw or G.711 alaw)
  - Opus, G.711 ulaw, G.711 alaw and G.729a audio transcoding into the enterprise network
  - H.264 and VP8 video transcoding

Remote Expert Co-browse

With Cisco Remote Expert Co-browse (previously known as Meet Me), you can deliver Expert Assist co-browse and application sharing in mobile or web applications.The key component is Expert Assist (with all its elements as above).

There should be an existing communication channel between the two parties (consumer and agent), which can be a voice and video call, a chat session, and so on.

SDKs

Cisco Remote Expert Mobile includes Software Development Kits (SDKs) to add voice over IP, video over IP, and Expert Assist (app share and web co-browse, annotation, and document push) features to existing mobile and web applications.

Whether making or receiving calls in client web applications, RE Mobile’s Client SDK for Web supports major browsers such as Google Chrome, Mozilla Firefox, Opera, Internet Explorer (Windows desktop only, not tablet), and Apple Safari. With WebRTC at its core, Remote Expert Mobile enables communications without the need for browser plugins in those browsers which support WebRTC. In those browsers (Internet Explorer and Safari) which do not support WebRTC natively, Remote Expert Mobile provides WebRTC plugins for voice and video.

Cisco Remote Expert Mobile also delivers integrated communications in iOS and Android applications through native libraries.

Technologies

WebRTC

WebRTC (Web Real Time Communications) is a standard set of APIs which enable browsers and mobile applications to send and receive real time communications streams, particularly voice and video. Using these APIs, web developers can implement video and audio applications in a way that is both interoperable with other clients and does not require a plug-in. WebRTC can use a variety of codecs, such as G.711, Opus,H.264, and VP8.

Expert Assist

With Expert Assist, remote users can share the screen of their tablet, smartphone, or browser tabs with an agent. Sensitive information on the regions or fields of a user's web page or application can be masked to the agents.

Agents can also control the app or website of the user through a simple point and click. The remote control feature allows the agent to select from menus, go to specific information, fill in a form, or guide users

through an important process. Agents can also move the live video window to ensure that it does not interfere with the other elements on the screen.

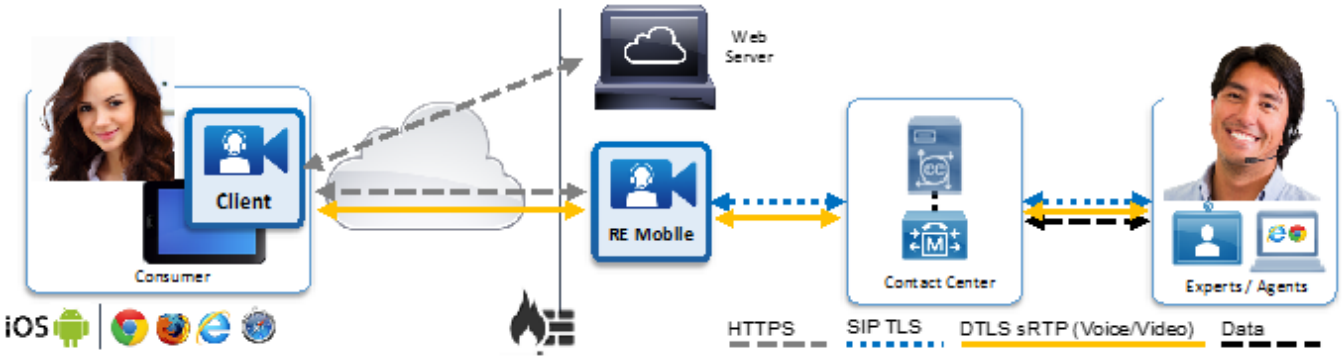
Unlike most co-browsing technologies, Expert Assist does not share the Document Object Model (DOM) between the user and the agent. Expert Assist technologies ensure that any differences between the browsers do not affect the user experience during the session. Expert Assist supports native iOS and Android apps.

CHAPTER 2

Developer Overview

With Remote Expert Mobile, consumers are connected to an expert or agent. Developers embed the Remote Expert Mobile Client SDKs (CSDK) in a native iOS, native Android or web based application code.

Figure 1:



Note For detail architecture diagrams, protocol flows please refer to the *Cisco Remote Expert Mobile Design Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-installation-guides-list.html).

- **Consumer**—A consumer is the user of that connects to an agent. The Client Application is installed on the consumer's mobile iOS or Android phone or tablet or accessed via their web browser.
- **Client or Client Application**—With only a few lines of code, a developer can integrate WebRTC voice/video and Expert Assist into a new or existing application. These Client Applications (Clients) can then connect consumers to agents with voice over IP, video over IP and Expert Assist (app share and web co-browse, annotation and document push) features within pre-existing mobile and web applications. Each connection from a client to an agent is referred to as a session or call.

  Whether placing or receiving calls, Cisco Remote Expert Mobile supports every major browser. With WebRTC at its core, in-app communications are enabled without the need for plugins. Where WebRTC is yet to be supported in Internet Explorer and Safari, WebRTC plugins are provided. Cisco Remote Expert Mobile also delivers integrated communications in iOS and Android apps through native libraries.

  - CSDK for Web - JavaScript for web browser applications (Chrome (Desktop and Android), Firefox (Desktop and Android), Opera (Desktop and Android), Microsoft Internet Explorer (Windows desktop only, not tablet) and Apple Safari (Desktop)
  - CSDK for iOS- Objective C for Apple iOS native applications (iPhone, iPad, iPod touch)
  - CSDK for Android- Java for Android (Phone, Phablet and Tablets)
- **Remote Expert Mobile (RE Mobile)**—RE Mobile also includes server software components that run on the 'company standard' Virtual Machine (VM) hardware platform for ease of management and deployment within an existing data center. These server component provide the call signaling to establish calls, the ability to handle voice, video and Expert Assist media as well as WebRTC and associated firewall-traversal.

  - Remote Expert Mobile Application Server (REAS)
  - Remote Expert Mobile Media Broker (REMB)
- **Contact Center**—Typically calls from client applications, passing through the RE Mobile server software are then placed in queue as part of a contact center solution, such as Cisco Unified Contact Center Enterprise (Unified CCE) or Cisco Unified Contact Center Express (Unified CCX) and a UC solution such as Cisco Unified Communications Manager (Unified CM). The contact center solution then routes the call to the best and available agent to handle the consumer's issues.
- **Expert or Agent**—Experts or agents are connected to consumers. Experts and contact center agents typically have an agent desktop application such as Cisco Finesse and a VoIP or video capable endpoint (ex. Cisco DX70). With the agent desktop and endpoint, the agent resolves consumer issues by communicating with the consumer and viewing the consumer's mobile app or browser. Experts and contact center agents may take advantage of Remote Expert Mobile within their Finesse agent user interface or use the Expert Assist Web Agent Console in UC specific deployments.

Quick JavaScript Example Overview

CSDK for Web allows a consumer on a website to be connected over their Wi-Fi or Internet connection to an agent and engage in live communications. Developers can easily embed voice, video, and Expert Assist sessions in their web application with JavaScript. There are several options that can be configured through various parameters, but for the most part developers require two simple lines of JavaScript code to include Expert Assist functionality:

#1—including the `assist.js` JavaScript library to enable the website:

```
<script src=" https://<REAS
IP>:8443/assistserver/sdk/web/consumer/assist.js"></script>
```

#2—establish an Expert Assist session in connection with a link or image:

```
<a title="Expert Assist" onclick="AssistSDK.startSupport({destination :
'agent1'})"><img src="img/support.png"></a>
```

After the user clicks the image, the consumer call is routed to an agent who has logged in as 'agent1' into the Expert Assist Web agent console or Finesse gadget. More details for the JavaScript CSDK is provided in **Remote Expert Mobile—CSDK for Web (JavaScript) on page 23**.

Note The above invocation of `startSupport` starts a call with voice and video. For co-browse only sessions, see **Co-browse only (with code) on page 10** or **Co-browse only (with correlation ID) on page 9**

Establishing a RE Mobile Session

The application can establish a session for voice and video to include co-browsing, or it can establish a session for co-browsing only. The difference between establishing a co-browse only session and establishing a session with voice and video depends upon how the session is started. In both cases, the application calls `startSupport`, but provides different parameters in the configuration object supplied to the method call. The creation of the configuration object is platform specific.

Note After being established, voice and video sessions and co-browse only sessions are identical as far as the Expert Assist developer is concerned. All the Expert Assist functions such as co-browsing, document push, annotations, screen sharing, remote control, and form completion, are available in the same way using the same APIs. For control of the voice and video call in a voice and video session, see the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html).

With Voice and Video

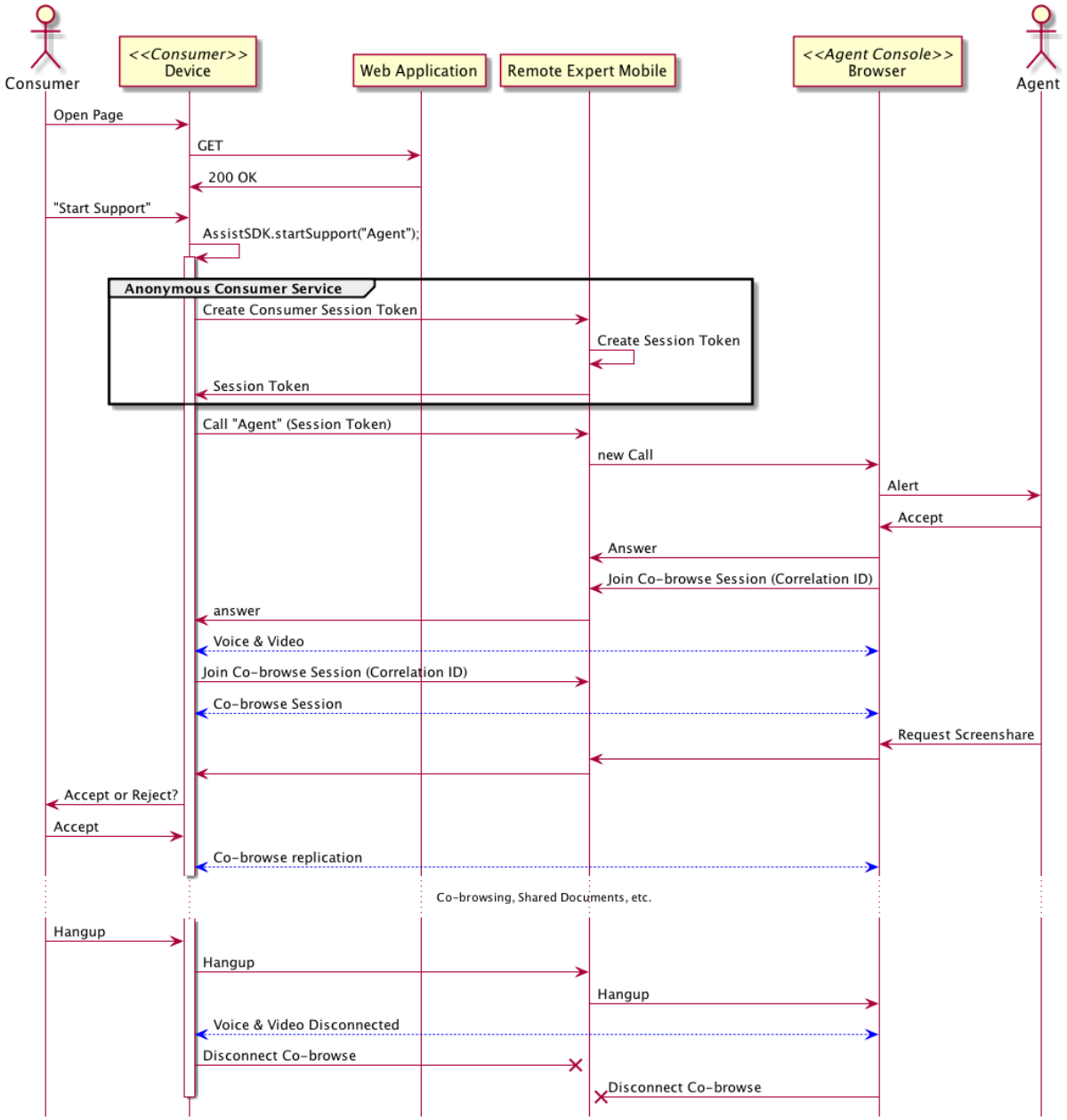
This section describes the sequence of events that establishes a voice and video support session between a consumer and an agent when each uses anonymous access.

Note For increased security, Anonymous Agent Access is disabled by default. See the *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-installation-guides-list.html) for details on how to enable Anonymous Agent Access.

When voice and video is enabled, the call itself is used to transport an identifier (the Correlation ID) that correlates the consumer and the agent so that they can join the same co-browse session automatically. The Correlation ID is allocated by Remote Expert Mobile and transported as the `username` part of the consumer's `From` address in the SIP messages which set up the call. A client (such as a Finesse application) which receives those SIP messages, can extract the Correlation ID from the `From` address.

When an agent is available, the consumer can request support as shown in **Figure 2:**

Figure 2:



Trusted Anonymous Consumer Access Mode

If the Anonymous Consumer Access mode is set to Trusted, you can pass UUI data to the call using the `uui` configuration property (see the *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* for details of how to set Anonymous Consumer Access to Trusted). Note the following:

- Anonymous Consumer Access mode should only be set to Trusted if you can be sure that:
  - the client (Web, iOS, or Android consumer application) has not been tampered with, or
  - modifying the Correlation ID or UUI will not affect the system or other users

Because JavaScript is in plain text in the browser, it is open to abuse, and sensitive values such as the Correlation ID and UUI could be modified by a malicious user. The issue is less likely on iOS and Android platforms, because applications are compiled binaries, but the risk should not be discounted completely. The Anonymous Consumer Access mode is Enabled by default.

- UUI may be used to transfer customer specific information such as account numbers and username. The correctness and validity of this value should only be relied upon if the server-side application is managing the Session Tokens. Otherwise, ensure that either the network is secure enough (for instance, a private network used only by trusted employees), or that the correctness of the UUI is not critical.

Co-browse only (with correlation ID)

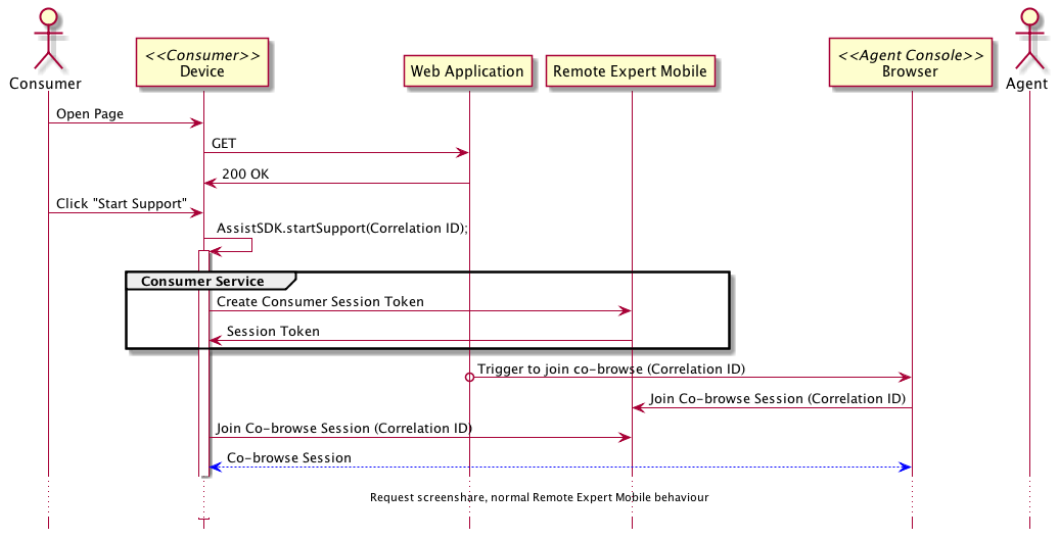
You can use the Remote Expert SDK to start a co-browsing session without starting a voice and video call at the same time. The application allocates a correlation ID for the session and signals it to the appropriate agent, so that both parties can join the same session. How to allocate the correlation ID and transmit it to the agent is a matter for the application developer.

The application includes the correlation ID when it calls `startSupport`; the details of how to call `startSupport` and include the correlation ID differ between SDKs, so see **Remote Expert Mobile—CSDK for Android (Java) on page 73**, **Remote Expert Mobile—CSDK for iOS (Objective C) on page 45**, and **Remote Expert Mobile—CSDK for Web (JavaScript) on page 23**). The Remote Expert SDK includes it as the `CID` URL parameter when it makes an HTTP request to the *Anonymous Consumer Service* to create and return a session token.

Initiation

The generalized sequence of events to establish a co-browsing session is shown in **Figure 3:**

Figure 3:



In the sequence diagram above, the Web Application triggers the agent to join a co-browse session; however, this signal may come from somewhere else within the infrastructure and is only shown here for illustrative purposes.

When selecting a suitable Correlation ID, consider the following:

- **Uniqueness**—Different active sessions (in most cases from different customers) must not use the same correlation ID.
- **Randomness**—The value should be random enough to be difficult to guess. As the CSDK does not authenticate a user, if the correlation ID was easily guessable it could be possible for an attacker to guess another consumer's correlation ID and eavesdrop on their co-browse session.

Destruction

Both the agent and consumer can join and leave the co-browsing session independently of each other; the co-browsing session remains open for as long as there is at least one active connection.

Co-browse only (with code)

The Expert Assist server exposes short code REST services to help developers communicate the correlation ID. The short code services can create a correlation ID and session token, and associate them with a short code, which can then be communicated to the other party.

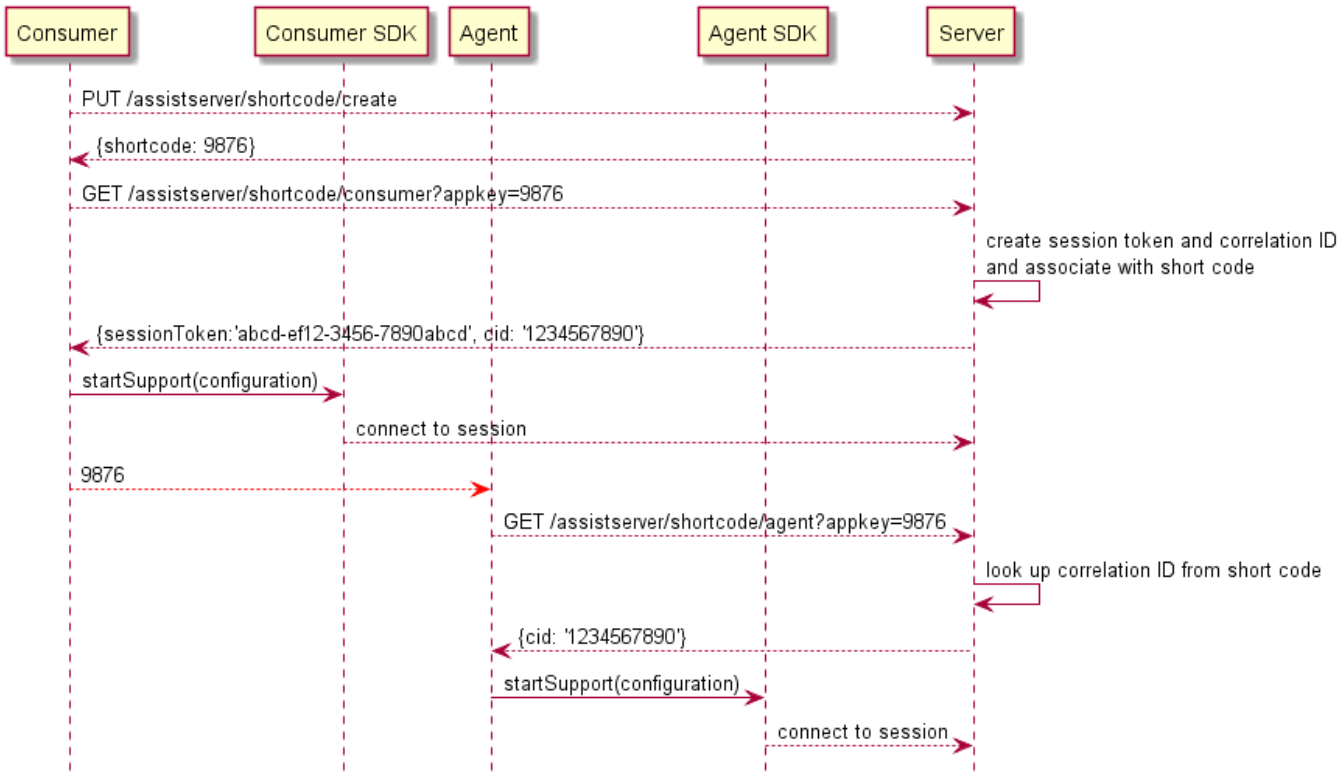
The advantage of communicating a short code, rather than the correlation ID, is that the short code generated by the server is guaranteed to be both unique while the correlation ID is being communicated, and short enough to be easily communicated by voice (or whatever other out-of-band communication channel is in use) without error.

Note After a short code has been used by both agent and consumer to communicate a correlation ID, it is discarded, and can be used by a different agent and consumer to communicate a different correlation ID; it also expires 5 minutes after creation. Therefore you should use the short code as soon as possible after creation.

Co-browse only

In this scenario, a communication channel already exists between the two parties who need to set up a co-browse session. The consumer uses the REST service to create the short code, and then uses another REST service to create the session token and the correlation ID, which are associated with the short code; the consumer uses the session token and correlation ID in the configuration in their call to `startSupport`. The consumer then communicates the short code to the agent using the out-of-band mechanism, who uses it to connect to the same session by retrieving the correlation ID associated with the short code.

Figure 4:



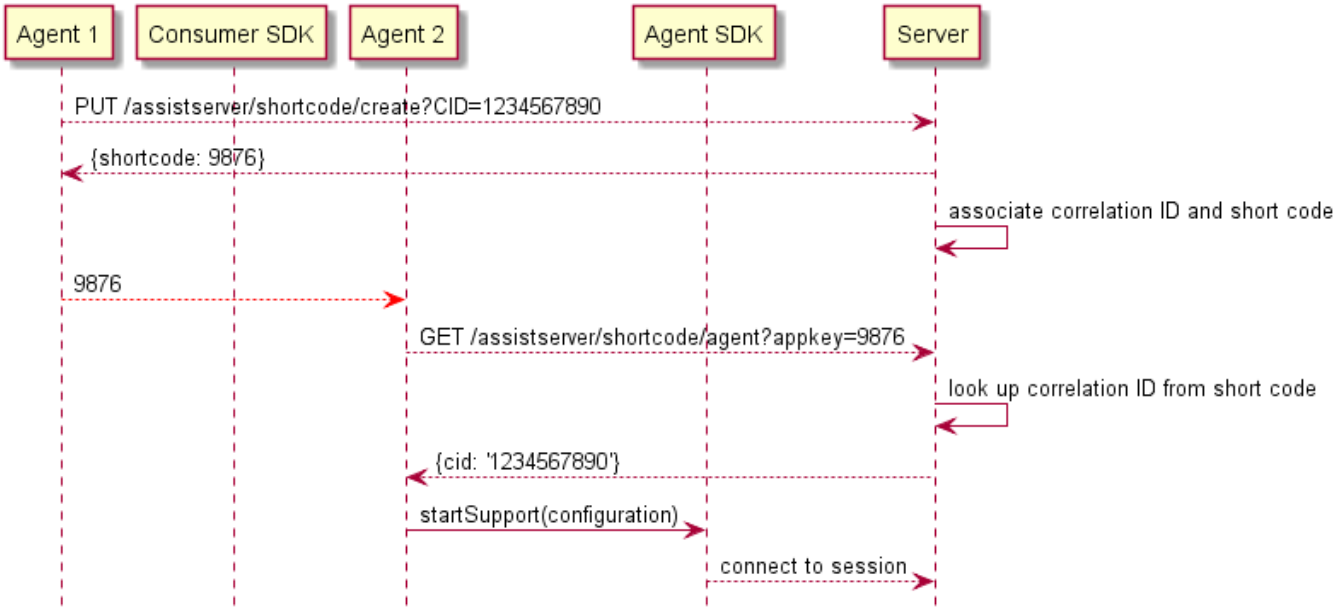
Escalating a call to co-browse

The sequences here are the same as the diagram above, except that the session token is already known to the consumer.

Including a second agent

In this scenario, a consumer is in a co-browse session with an agent, and the agent includes another agent into the same co-browse (similar to a consultation call). The first agent already knows the correlation ID in use for the co-browse session, and can use the `CID` URL parameter in the initial call to the REST service to associate that correlation ID with a newly created short code:

Figure 5:

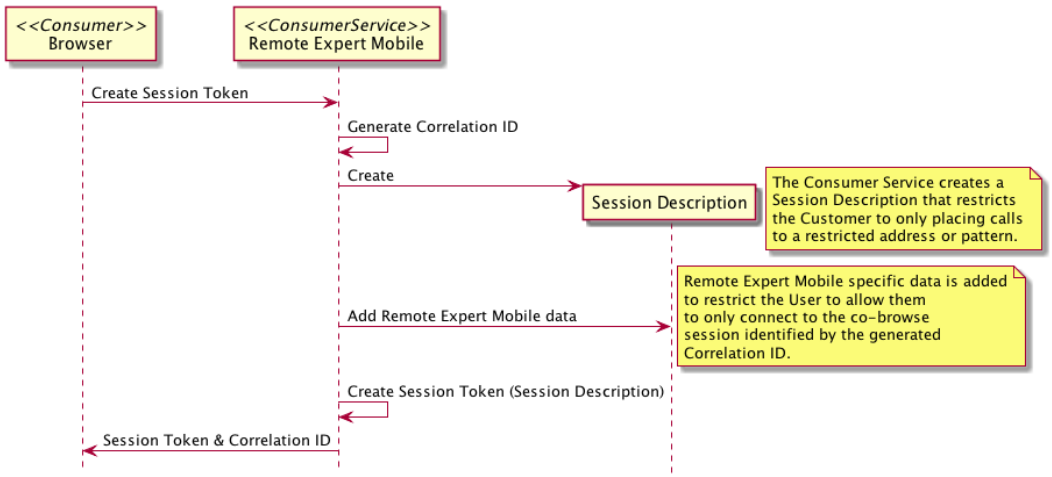


Anonymous Client Access (default)

By default CSDK provides client applications anonymous access to Remote Expert Mobile allowing any user to use voice, video or Expert Assist from a developer's application. The CSDK generates a correlation ID that is suitably unique and random, in the form of "assist-" followed by a 25 character alphanumeric string (for example `assist-m2v7r3jpb0jsk5j28ok4b5o4s`).

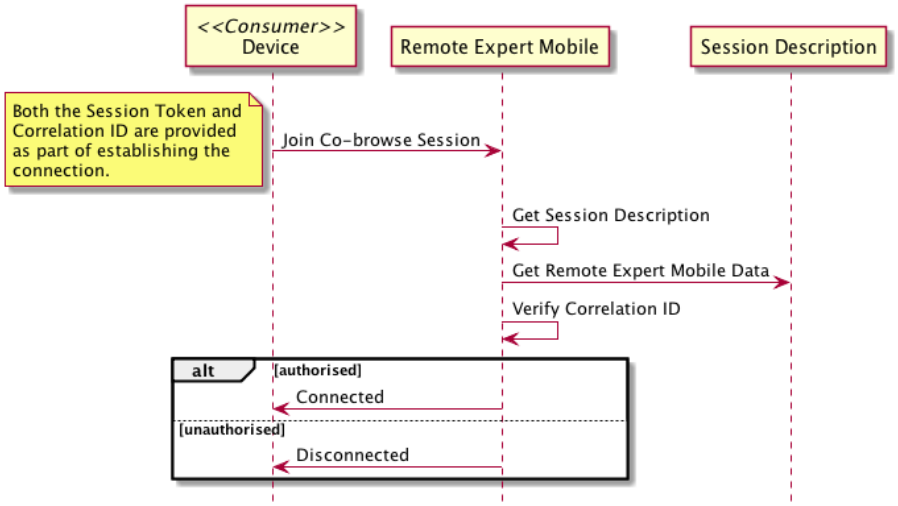
The sequence of events is shown in **Figure 6:**

Figure 6:



When the session is established from the consumer's device to CSDK, CSDK verifies that the session associated with the Session Token is also associated with the Correlation ID, rejecting the connection if it is not.

Figure 7:



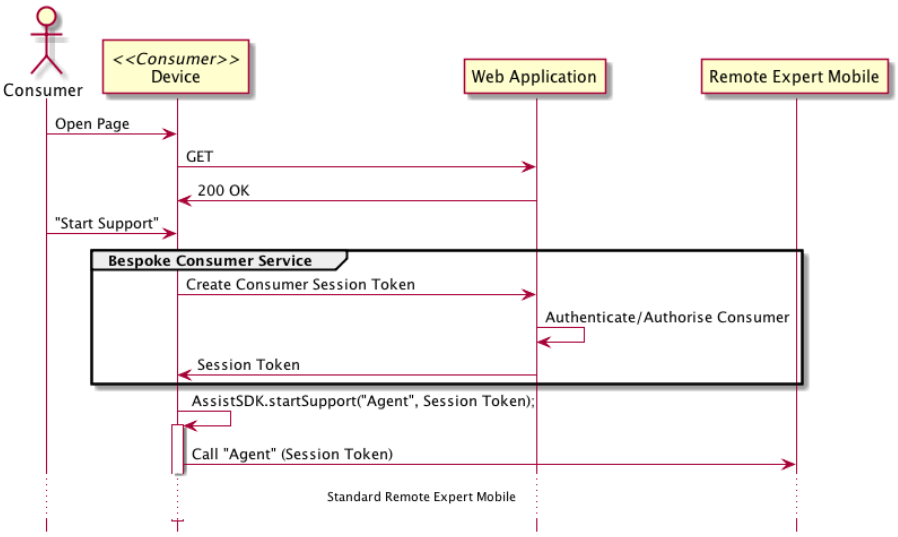
Restricted Client Access

It may not be appropriate in all use cases for clients to have anonymous access to agents. Consequently, it is possible to disable anonymous and create a custom implementation. It becomes the responsibility of this implementation to create and manage Session Tokens, which are subsequently provided by the application to the CSDK for it to use when establishing sessions.

For example, Expert Assist may only be available to users that are logged in to a CSDK for Web application. The server-side web application can validate a consumer to ensure they are logged in, create a Session Token on the REAS with the appropriate details and provide that token back to the client-side code on the consumer's device. That token is then passed to the CSDK, which uses it to establish the Expert Assist session.

When the implementation is restricted for consumers, the sequence of events differs slightly from the Anonymous Access:

Figure 8:



The key difference is that it is the responsibility of the developer's application to create the Session Token using the REAS; the generated token is provided to CSDK for it to use. Due to the presence of the Session Token, CSDK does not attempt to create a token itself, and uses the one provided.

Integrating the Remote Expert Advanced SDK

Remote Expert Mobile SDKs use facilities from Remote Expert Advanced SDKs, and rely on an instance of the REM Advanced SDK being available, so all the facilities of REM Advanced SDK are available for you to use if you want.

In the case of browser applications written in JavaScript, REM Advanced SDK objects are available from the global `UC` object:. How to access the `UC` object depends upon how you initialized the Remote Expert Mobile SDK.

- If you called `startSupport` with a configuration object which does *not* include a session token, it automatically requests a session token and initializes REM Advanced SDK with it. In this case, the `UC` object is automatically available as a global object for you to use.
- If you obtained a session token from a session token REST API (see **Restricted Client Access on the previous page**) in order to initialize `UC`, you can use this session token and a correlation ID in the configuration object which you pass to `startSupport` to escalate the session for Remote Expert Mobile after a call has been established (see **Escalating a call to co-browse on page 11**). In this case, you have created the global `UC` object, and can use it for call control.
- If you started a co-browse only session, there is no call under the control of the REM Advanced SDK, so the REM Advanced SDK objects are not available.

Having obtained a `UC` object, you can use the facilities available from its `phone` object to control the call:

```
var call = UC.phone.getCall(CALL_ID);
 ⋮
call.end();
```

In the case of Android and iOS applications, there is no simple way to get the REM Advanced SDK objects from the Remote Expert SDK. If you want to control the call using the REM Advanced SDK, you need to create a voice and video call first, and then escalate it to co-browse. See **Integrating an Android Application with the Advanced SDK on page 96** and **Integrating an iOS Application with the Advanced SDK on page 70**.

See the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)*(available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) for details on what facilities are available and how to use them.

CHAPTER 3

CSDK Security Highlights for Developers

Security within CSDK is achieved through the following mechanisms:

- Socket Security—HTTPS, Secure Web Sockets (WSS).
- Session Tokens—Session Tokens created with restricted policies.
- Configuration—Behavior restricted by System Configuration administrated by an administrator.

CSDK by default provides anonymous access to consumers and agents and does not implement user authentication or authorization; however, it does take precautions to allow only the appropriate participants to join a session. If client access is restricted, the following items must be considered:

- JavaScript is in Plain Text when running in a user's browser. Consequently, values such as the Correlation ID are readily accessible. As discussed in **Initiation on page 9**, the application should use a suitably unique and random value, if one is specified manually.
- REAS can create session tokens with varying degrees of restriction. You should apply as much restriction as possible.
- The Session Token API exposed by the REAS must be accessed by a Server Side Web Application and that API must not be exposed publicly through a Reverse Proxy.

Note For more details on security and certificate setup, please refer to the *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-installation-guides-list.html).

Consumer Session Creation

An REM Advanced SDK Web Gateway *session token* (see the *Client SDK documentation*) and a *correlation ID* is required to establish a co-browsing session. When the application calls `startSupport`, Remote Expert Mobile

uses a built-in mechanism to create a session token for the voice and video call, and associates it with a correlation ID for the co-browse. The built-in mechanism provides a standalone, secure mechanism for creating a session token and a correlation ID, but the process is not integrated with any pre-existing authentication and authorization system, and assumes that if a client can invoke `startSupport`, it is permitted to do so.

If you wish to integrate your Remote Expert Mobile application with an existing authentication and authorization system, you can disable the built-in mechanism (by setting the **Anonymous Consumer Access** setting to `disabled` using the Web Administration service), and replace it with a bespoke implementation which uses the existing system to authorize and authenticate the client.

Once you have authenticated and authorized the application using the pre-existing system, the application needs to create a session token and associate it with a correlation ID.

Session Token Creation

For a bespoke implementation, the following general steps are required:

1. Create a Web Application that can invoke the Session Token API REST Service, exposed by the Web Gateway.
2. Provide the appropriate Remote Expert SDK (if in use) configuration in a JSON object (the *session description*).
3. Add Remote Expert Mobile-specific data to the session description:
   - `AED2.metadata.role`

     This should be set to `consumer`
   - `AED2.allowedTopic`

     A regular expression which limits the correlation IDs which the session token can be used to connect to. A value of `.*` allows the session token to be used to connect to any support session with any correlation ID. For security reasons, we recommend that this should be set to the value of the correlation ID which will actually be used:

```
{
   ...
   "voice": {
      ...
   },
   "aed": {
      "accessibleSessionIdRegex": "customer-ABCDE",
      ...
   },
   ...
   "additionalAttributes": {
   {
      "AED2": {
         "metadata": {
            "role": "consumer"
         },
         "allowedTopic": "customer-ABCDE"
      }
   },
   ...
}
```

4. Request a session token by sending an HTTP `POST` request to the Session Token API, providing the session description in the body of the `POST`.

For steps 1, 2, and 4, see the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)*.

Note The *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)* documents both `voice` and `aed` sections - at least one of these must be present for the session token to be created. However, if a `voice` section is included (voice and video functionality is required), then only the `AED2` entries are necessary for Remote Expert Mobile functionality. If voice and video functionality is not needed, and a `voice` section is omitted, then there must be an `aed` section as well as the `AED2` section entries.

CHAPTER 4

Permissions

You can use permissions to prevent an agent from interacting with, or even seeing, a UI control. Whether an agent can see a particular control or not depends upon both the agent's and the control element's *permissions*.

- **Control element permissions**

  Each UI element has at most one permission marker value. Elements which do not have a permission marker inherit their parent element's permission marker; an element which does not have a permission marker either assigned explicitly or inherited from its parent, is assigned the `default` permission marker. The `default` permission is explained further in the **Default Permission on page 22**.

- **Agent permissions**

  Agents have two sets of permissions, *viewable permissions* and *interactive permissions*. Each set may contain an arbitrary number of values. Agents which are not assigned any permissions have the `default` permission for both interactive and viewable permission sets.

  Remote Expert Mobile grants permissions to the agent when the agent presents a *Session Token Description* to the Remote Expert Mobile server (see the *Remote Expert Mobile Agent Console Developer-Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) for more information about setting agent permissions, and under what circumstances the agent can be implicitly assigned the `default` permission).

Note If the agent specifies *permissions* in the *Session Token Description*, but leaves both the viewable set and interactive set empty, the agent will end up with no permissions, not even the `default` permission.

The combination of the element's and the agent's permissions determines the visibility of a UI element to an agent. A UI element is visible to a specific agent if, and only if, the agent's set of viewable permissions contains the permission marker assigned to or inherited by that element. Similarly, an agent may interact with a UI element if and only if the agent's set of interactive permissions contains the element's permission marker.

Permissions and permission markers are free-form text, which (apart from the reserved `default` permission) are in the control of the application developer. Remote Expert Mobile will show to the agent those, and only those, elements which the agent has permission to view; but it is up to the application developer to ensure that each agent has the permissions they need, and that the UI elements have corresponding permission markers assigned.

Remote Expert Mobile assumption: When an agent wishes to establish a co-browse, the permissions the agent should have, as defined by the organization's infrastructure, are known, and can be translated into an equivalent set of permissions in the Session Description.

The methods used to set the permission on a control element and read the permissions of an agent differ between the different SDKs (Web, Android, and iOS); see the Permissions section of the relevant SDK section in this guide.

Agent and Element Permissions

Permissions are compound such that:

Permission marker on element	Agent viewable permission set	Agent interactive permission set	Result
X	["X"]	["X"]	Agent can view and interact with an element marked with `X`.
X	["X"]	[]	Agent can view the element marked with `X` but cannot interact with it.
X	[]	["X"]	Agent can neither view nor interact with the element, because it does not have `X` in its viewable set. (In order to interact with an element, and agent must first be able to view it.)
X	[]	[]	Element marked with `X` is masked or redacted, as Agent does not have the `X` permission in its viewable or interactive set.
X	["default"]	["default"]	Element marked with `X` is masked or redacted, because Agent does not have the `X` permission in its viewable or interactive set.
X	["default"]	["X"]	Agent can neither view nor interact with the element, because it does not have `X` in its viewable set.
X	["X"]	["default"]	Agent can view the element, because it has the `X` permission in its viewable set; it cannot interact with it, because it does not have the `X` permission in its interactive set.
B	["X"]	["X"]	Element marked with `B` is masked or redacted, because Agent has `X` permission and not `B` in their permission set.

Permission marker on element	Agent viewable permission set	Agent interactive permission set	Result
	["X","default"]	["X","default"]	Agent can view and interact with the element because they have the `default` permission in their viewable and interactive sets, and the element implicitly has the `default` permission.
	["X"]	["X"]	Element is masked or redacted, because Agent's sets do not contain the `default` permission
	["default"]	["default"]	Agent can view and interact with the element, because they have the `default` permission set for their viewable and interactive set.
	[]	[]	Element is masked or redacted, because Agent's sets do not contains `default` permission
	["default"]	["X"]	Agent can see the element because they have the `default` permission in their viewable set. They cannot interact with it because they do not have the `default` permission in their interactive set.
B	["X"]	["B"]	Element is masked or redacted because the agent's viewable set does not contain `B`. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.
B	["B"]	["X"]	Element is viewable, because the agent's viewable set contains `B`; the element is not interactive, as the agent's interactive set does not contain `B`.

An agent is granted a permission if a permission (such as `A`, `B`, or `X`) configured in their Session Description matches the permission-marker of the UI element in the application.

Note In some circumstances an agent can be granted the `default` permission *implicitly*, but that ***that is not the same thing as having an empty set of permissions***. In the preceding table, an empty set of agent permissions means exactly that; a set of permissions containing *only* the default permission may have been granted either implicitly or explicitly.

Parent and Child Permissions

An element can also inherit permissions through the UI hierarchy: UI elements that are a child of a parent UI element inherit the permission marker of the parent, unless the child specifies a permission marker of its own.

A child element can override its parent permission marker, but it will only be effective if the agent's viewable permission set contains the parent's permission marker as well as the child's (the agent must be able to see the container in order to interact with an element inside it). This allows the developer to make a child element interactive and the parent element not. An example use of this could be a child button within a parent container, where only the button needs to be interactive.

Permission marker set on parent element	Permission marker set on child element	Agent viewable permission set	Agent interactive permission set	Result
A		["A"]	["A"]	Agent can view and interact with both parent and child element. Child inherits permission marker `A`.

Permission marker set on parent element	Permission marker set on child element	Agent viewable permission set	Agent interactive permission set	Result
A	A	["A"]	["A"]	Agent can view and interact with both parent and child element.
A	B	["A"]	["A"]	Agent cannot view or interact with child element marked with `B`.
A	B	["A","B"]	["A"]	Agent can view child element but cannot interact with it
A	B	["A","B"]	["B"]	Agent can view and interact with the child element but cannot interact with the parent.
A	B	["B"]	["B"]	Agent cannot view or interact with child or parent element as they do not have the parent's permission marker in their viewable permission set. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.
		["default"]	["default"]	Agent can view and interact with both parent and child elements as they have the `default` permission in their viewable and interactive permission sets, and both parent and child elements implicitly have the `default` permission.
	B	["B"]	["B"]	Agent cannot view or interact with child element, because the parent has an implicit `default` permission marker, and they do not have the `default` permission in their viewable permission set. The agent may not interact with an element which they cannot see, even though they have the appropriate permission in their interactive permission set.

Default Permission

You do not have to assign a permission marker to every UI element which you want agents to view or interact with; every element which does not have or inherit a permission automatically has the `default` permission marker.

Elements which have the `default` permission marker will be viewable and interactive for any agent which has the `default` permission. Any agent which has the `default` permission will include the reserved word `default` among its set of permissions, as found in the viewable and interactive permissions of the agent.

Not every agent has the `default` permission, and an agent might have the `default` permission in its viewable permissions, but not in its interactive permissions.

CHAPTER 5

Remote Expert Mobile—CSDK for Web (JavaScript)

This is an overview of the tasks and development required to integrate RE Mobile with a pre-existing web application using the CSDK for Web. Developers can easily embed voice, video and or Expert Assist sessions in their website or web application with JavaScript.

Integration with an Existing Application

The steps needed to integrate Remote Expert Mobile with a Web application is described in the following sections.

Note On Windows, web-based applications are supported on desktop only, not tablet.

Packaging JavaScript

The Remote Expert Mobile JavaScript SDK is available as part of the Remote Expert Mobile server component, so the necessary JavaScript library to load the SDK can be included on a web page directly from the server. This is the recommended mechanism.

You should not include the contents of the `expert_assist_web_consumer_SDK-11.6.1.10000-7-ES3.zip` package in your web application as the source of the SDK, because important updates to the SDK, available on server upgrade, will not be available to the client application.

Making Pages Supportable

Every page that is to allow support to start or continue must include the `assist.js` file from the Remote Expert Mobile SDK, and have the `<DOCTYPE html>` declaration. Add the following lines should to the HTML page, where `<reas address>` is the host name or IP address of the Remote Expert Mobile server:

```
<DOCTYPE html>
 ⋮
<script src='<reas address>/assistserver/sdk/web/consumer/assist.js'/>
 ⋮
```

We suggest that you add these lines to the template for the site, if there is one.

Note When developing with Remote Expert Mobile, remember that the SDK also requires cookies and JavaScript to be enabled on the browser.

Supporting Iframes

By default, Remote Expert Mobile ignores iframes within the supported page, because it is not possible to include iframes as part of the support session without an additional implementation step.

If you want to include iframe support, add the `assist-iframe.js` script to the body of the iframe's source (that is, the webpage targeted by the iframe must include the `assist iframe.js` script), and initialize `AssistIFrameSDK` with an object containing an `allowedOrigins` element:

```
<script src='<reas address>/assistserver/sdk/web/consumer/js/assist-
iframe.js'/>
 ⋮
AssistIFrameSDK.init({allowedOrigins: '*'});
 ⋮
```

The `allowedOrigins` element should be an array of origin domains, including scheme and port, in the form `scheme:host:port` (for example `http://127.0.0.1:8080`), which is typically set to match the origin of the page that includes the iframe. This facilitates safe communication between the iframe and its parent. The special value `"*"` (as above) specifies that the iframe will communicate with a parent from any origin address.

Remote Expert Mobile supports both local-origin and cross-origin iframes, allowing agents to see the content of iframes; however remote agent interaction with iframes is currently not supported.

allowedIframeOrigins

Including the `allowedOrigins` member in the configuration object passed in to `AssistIFrameSDK.init()` enables the programmer to protect the iframe from rogue pages which may attempt to embed the iframe (see **Supporting Iframes above**). The similar `allowedIframeOrigins` member is a list of pages which embed the iframe (acting as the iframe's parents), passed in to the configuration object when the application calls `startSupport` (see **Session Configuration on the facing page**). Set it either to `false` (to disable iframe support in Remote Expert Mobile), or to an array containing either all the URLs which embed the iframe (`['http://192.168.0.1:8080', 'http://www.server.net']`), or the wildcard (`['*']`). The default value (if `allowedIframeOrigins` is not specified) is the wildcard, which allows the iframe to be embedded in any page.

Note

- The use of the wildcard as the default is a temporary measure to preserve backward compatibility. In a future release it will be removed, so that in order for iframes to be co-browse enabled, the correct origins will need to be supplied both inside the iframe and on the parent page containing the iframe, using the two SDKs (`AssistSDK` and `AssistIframeSDK`).
- When explicitly setting `allowedIframeOrigins` to the wildcard, remember to include it as the only element of an array.

Starting a Support Session

The application starts a support session, normally in response to the user clicking on a **Help** or **Request Support** button, using the `AssistSDK.startSupport()` function, passing in a configuration object. To start a simple support session with default values, the application only needs to specify the `destination`:

```
<a title='Remote Expert Mobile' onclick='AssistSDK.startSupport({destination : "agent1"})'>Support</a>
```

The above code provides a link which a user can click on for support; when a consumer clicks the link, Remote Expert Mobile starts a call and co-browse session with the support agent named `agent1`.

Typically, customer support services provide a queue, which is serviced by a number of support agents. The `destination` parameter can also specify a queue instead of an individual agent:

```
var config;
config.destination = 'customer-support';
config.videoMode = 'agentOnly';
 ⋮
AssistSDK.startSupport(config);
```

The configuration object is a JavaScript object with a number of properties which control aspects of the session (see **Session Configuration below**).

Session Configuration

The configuration object passed in to `startSupport` may contain the following properties:

Property	Default Value/Behavior	Description
`destination`		Username of agent or agent group if that agent or agent group is local to the web gateway; otherwise full SIP URI of agent or queue
`videoMode`	`full`	Sets whether and from which parties video should be shown. Allowed values are: ■ `full` ■ `agentOnly` ■ `none`
`correlationId`	Generated	ID of the co-browsing session
`url`	Calculated from `src` attribute of `script` tag	Base URL of Remote Expert Mobile server and REM Advanced SDK Gateway, including only scheme, hostname or IP address, and port number. You should include this if the `assist.js`*JavaScript* file included with the <script> tag is not on the Remote Expert Mobile server. URIs of shared documents (see **Sharing Documents on page 36**) will also be resolved

Property	Default Value/Behavior	Description
		against this URL.
`sdkPath`	Calculated from `src` attribute of `script` tag	URL of the base directory of the consumer SDK. As with the url property, you need to include this if the Remote Expert Mobile SDK is not on the same server as the `assist.js` file.
`popupCssUrl`		URL of CSS stylesheet containing styles for the Remote Expert Mobile popup window. This allows you to customize the Remote Expert Mobile user interface (see **Customizing the Remote Expert Mobile popup Window on page 39**).
`popupInitialPosition`		Object containing values to position the popup window on the screen (see **Customizing the Remote Expert Mobile popup Window on page 39**).
`sessionToken`		Web gateway session token
`uui`		The value specified is placed in the SIP User-to-User Interface header in hex-encoded form. **Note:** The UUI can only be used when **Anonymous Consumer Access** is set to `trusted` mode (see the *Remote Expert Mobile Design Guide, Release 11.6 (1)* for more information). The UUI is ignored if the *session token* is provided.
`allowedIframeOrigins`	`*`	List of pages which will host iframes. See **Supporting Iframes on page 24** for details.
`retryIntervals`	`[1.0,2.0, 4.0,8.0, 16.0,32.0]`	Indicates the number of automatic reconnection attempts, and the time in seconds between each attempt. If an empty array is specified, then no reconnection attempt is made.
`connectionStatusCallbacks`		A set of callback functions which allow the application to control or monitor the status of the current connection. See **Connection Callbacks on page 41**.

Note

If the `sessionToken` property is not provided, the Remote Expert Mobile SDK will automatically create a session with the Remote Expert SDK server, and that session will be used for co-browsing and the Remote Expert voice/video call (if any); we expect this to be the normal case.

If the `sessionToken` property *is* provided (for instance, if a session token is provided separately using a bespoke security mechanism (see **Consumer Session Creation on page 15**), or the Remote Expert SDK has previously been used to initiate a call which is now being escalated to provide co-browsing (see **Escalating a Call to Co-browsing on page 28**)), then the configuration object passed to `startSupport` is used as is, and the session identified by the session token will be used for co-browsing. You will need to specify any non-default values for the other properties.

Note If `startSupport` is called programmatically, it will trigger the popup blocker that is built into most browsers; however, if it is called as a direct consequence of a user interaction (such as pressing a button in the UI), it is not.

With Voice and one-way video from agent

To invoke support without video from the consumer side, but with video from the agent side and with audio from both sides, include the configuration parameter `videoMode` and set it to `agentOnly`. For example:

```
AssistSDK.startSupport({destination : "agent1", videoMode : "agentOnly"});
```

With voice only

To invoke support with voice but without video, include the configuration parameter "`videoMode`" and set it to "`none`". For example:

```
AssistSDK.startSupport({destination : "agent1", videoMode : "none"});
```

Using UUI

The value specified in the `uui` element is placed in the SIP `User-to-User` Interface header exactly as it is passed. The application must ensure that the encoding is correct.

```
// Triggered by clicking a button, link, etc.
// 5465737420555549 = Hex encoded String "Test UUI"
AssistSDK.startSupport({"destination":"agent1", "uui":"5465737420555549"});
```

With the URL for the REAS

Ordinarily, the URL for the REAS is derived from the URL for the client web page, using the same scheme, host, and port as the client. However, sometimes developers may need to specify a different URL for the application server. In that case, you can do so by including a "`url`" property in the configuration object passed to `startSupport()`. For example:

```
AssistSDK.startSupport({destination : "agent1", url :
"https://myserver.test.com:8443"});
```

The URL should only include a scheme, hostname, and port number. Similarly, when the URL configuration parameter is present, documents that are shared with the consumer by the agent are resolved against this URL.

Specifying path to CSDK

Normally, the CSDK automatically detects its path to the SDK based on the URL used to include the `assist.js` JavaScript. However, in some circumstances it may be necessary to specify the path to the JavaScript CSDK manually. You can do this by providing the "`sdkPath`" parameter; for example:

```
AssistSDK.startSupport({destination : "agent1", sdkPath :
"http://myserver.com/sdk/"});
```

Co-browse only mode—Expert Assist with no voice or video by using Correlation ID

Remote Expert Mobile does not need a call to be made using Remote Expert SDK, and can support existing infrastructure such as a traditional phone system (PSTN), or Instant Messaging and Chat. If the client application does not need Remote Expert Mobile to place a call using REM Advanced SDK, the application can provide an ID that Remote Expert Mobile uses to correlate the consumer and agent side. This *correlation ID* provides a way to join agent and consumer sessions without prior knowledge of the domain-specific way that sessions are identified. This allows an application to leverage the features of

Remote Expert Mobile (co-browsing, document push or share, annotations, and remote control) without voice and video.

For example, to add a link to click for support:

```
<a title="Remote Expert Mobile"
   onclick="AssistSDK.startSupport({correlationId : 'your_correlation_ID'})">
   Remote Expert Mobile</a>
```

where the parameter specified is the unique ID used to correlate agent and consumer sessions. The newly created session for co-browsing will be associated with the correlation ID which you have supplied.

Note The correlation ID needs to be known to both parties in the call, and needs to be unique enough that the same correlation ID is not used by two support calls at the same time. The application developer must decide the mechanism by which this happens, but possible ways are for both parties to calculate a value from data about the call known to both of them, or that one side calculates it and communicates it to the other. There is also a REST service provided by Remote Expert Mobile which will create a correlation ID and associate it with a short code; see **Co-browse only (with correlation ID) on page 9** and **Co-browse only (with code) on page 10**.

Escalating a Call to Co-browsing

Note Escalating a call to include co-browsing is not relevant to co-browse only session, though it in fact uses the same mechanisms; see **Co-browse only (with correlation ID) on page 9** or **Co-browse only (with code) on page 10**.

In most cases, the application calls `startSupport` with an agent name, and allows Remote Expert Mobile to set up a call to the agent and implicitly add Remote Expert Mobile support to that call. However, there may be cases where a call to an agent already exists, and the application needs to add Remote Expert Mobile support capabilities. To do this, you need to supply the *session token* and a *correlation ID* in the configuration object which you supply to `startSupport`; and the agent needs to connect to the same session. The Remote Expert Mobile server provides some support for doing this.

1. The application connects to a specific URL on the Remote Expert Mobile server, to request a *short code* (error handling omitted):

```
var request = new XMLHttpPRequest();
request.onreadystatechange = function() {
   if (request.readyState == 4) {
      if (request.status == 200) {
         var shortcode = JSON.parse(request.responseText).shortCode;
         start(shortcode);
      }
   }
}
request.open('PUT', '<reas address>/assistserver/shortcode/create',
true);
request.send();
```

2. The application uses the short code in another call to a URL on the Remote Expert Mobile server, and receives a JSON object containing a session token and a correlation ID:

```
var start = function(shortcode) {
   var request = new XMLHttpRequest();
   request.onreadystatechange = function() {
      if (request.readyState == 4) {
```

```
            if (request.status == 200) {
               var response = JSON.parse(request.responseText);
                ⋮
               }
            }
         }
      }
      request.open('GET', '<reas
      address>/assistserver/shortcode/consumer?appkey=' + shortcode, true);
      request.send();
```

3. The application includes those values in the configuration object and passes it to `startSupport`:

```
var configuration;
configuration.sessionToken = response['session-token'];
configuration.correlationId = response.cid;
 ⋮
AssistSDK.startSupport(configuration);
```

   More configuration can be set in the configuration object.

4. The agent uses the same short code to get a JSON object containing the session token and correlation ID, which it can then use to connect to the same Remote Expert Mobile support session (see the *Remote Expert Mobile Agent Console DeveloperGuide, Release 11.6 (1)*). Informing the agent of the short code is a matter for the application. It could be something as simple as having it displayed on the consumer's screen and having the consumer read it to the agent on the existing call (this is how the sample application does it).

The sample application supplied with the SDK includes a JavaScript file called `short-code-assist.js`, which contains a function called `ShortCodeAssist.startSupport`, which contains the necessary code and takes a callback function and a configuration object:

```
ShortCodeAssist.startSupport(function() {
    ⋮
},
configuration);
```

The SDK calls the callback function when the support session starts successfully. You can take this code and adjust it as you need for your own purposes.

Note

- When escalating an existing call, the `destination` property should *not* be set on the configuration object; in this case, the destination is known implicitly from the existing call.
- The short code expires after 5 minutes, or when it has been used by both agent and consumer to connect to the same session.

The sample presents the following options when a customer clicks the Help button:

- Share my screen with support agent
- Call support and then share
- Already on the call, want to share
  - Selecting this option displays a short code on the screen (for example, 962013) that the customer can read out to the agent. The agent then enters this code and connects to co-browse the customer's screen.

The typical scenario for this functionality is as follows:

1. Agent prompts consumer to click the button
2. Consumer reads the generated short numeric code to the agent

3. Agent enters the code into their console
4. Consumer and agent are connected to the same co-browse—agent can now request co-browsing

Ending a Support Session

When voice and video is enabled, the user can end the session using the default UI that Remote Expert Mobile adds; the application can also end the session programmatically using the `AssistSDK.endSupport` function. In co-browse-only mode, Remote Expert Mobile does not add a default UI, so the application must call `AssistSDK.endSupport` to end the support session.

During a Co-browsing Session

While a co-browsing session is active (after the application has called `startSupport` successfully, and before either it calls `endSupport` or receives the `onEndSupport` notification to indicate that the agent has ended the support session), the application may:

- Accept an agent into, or expel the agent from, the co-browsing session
- Pause and resume the co-browsing session
- Receive a document from the agent
- Push a document to the agent
- Receive an *annotation* (a piece of text or drawing to show on the device's screen, overlaid on the application's view) from the agent
- Have a form on its screen wholly or partly filled in by the agent

Actions which are initiated by the application (such as pushing a document to the agent) require it to call one of the methods on the `AssistSDK` object.

Actions initiated by the agent (such as annotating the consumer's screen) can in general be allowed to proceed without interference from the application, as the Remote Expert Mobile SDK manages them, overlaying the user's screen with its own user interface where necessary.

However, the application can receive notifications of these events by defining one or more of the callback functions on the `AssistSDK` object:

```
window.AssistSDK = {
   onEndSupport: function() {
       ⋮
   };
}
```

Callbacks

When it calls `startSupport`, the consumer application can provide callback functions to the consumer web SDK. The SDK calls these functions to notify the application of events; the application can respond to the events, and in some cases can tell the SDK what to do next.

onConnectionEstablished

A consumer application can implement the `onConnectionEstablished` callback to receive notification when an agent first joins a Remote Expert Mobile session. Once this has happened, the agent may request permission to co-browse.

```
AssistSDK.onConnectionEstablished = function() {
   console.log("Connection Established");
};
```

Note By default, Remote Expert Mobile presents the request for permission to the user; however, the application can override this behavior; see **onScreenshareRequest below**.

onWebcamUseAccepted

When Remote Expert Mobile establishes a voice and video call, it prompts the consumer to allow the application to use their webcam; the application receives this callback after the user has given permission. You might use it to update the user interface to remove a warning message, or to update a progress indicator:

```
AssistSDK.onWebcamUseAccepted = function() {
   // Hide the warning
   hideWebcamWarning();
};
```

onScreenshareRequest

The `onScreenshareRequest` callback notifies the application when an agent asks to co-browse the consumer's screen. It gives the application an opportunity (by returning `true`) to allow the screenshare without asking the consumer (for example, there could be a flag in the application's configuration which gives permanent permission for screen sharing):

```
AssistSDK.onScreenshareRequest = function() {
   if (screenshareAllowed) {
      return true;
   }
    ⋮
   return false;
};
```

By default, Remote Expert Mobile displays a dialog box when an agent requests co-browsing, allowing the consumer to accept or reject the co-browse.

onInSupport

The application can receive an `onInSupport` callback when it accepts a screenshare, and the agent has joined the co-browse. It gives the application an opportunity to change its own UI to reflect the fact that a co-browsing session is active, or to log events.

```
AssistSDK.onInSupport = function() {
   // Show user extra UI as they're in a Remote Expert Mobile session
   showCobrowseUI();
};
```

onPushRequest

When the agent pushes a document to the consumer, by default Remote Expert Mobile displays a dialog box, allowing the user to accept or reject the document; if they accept it, it shows the document to the consumer. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

The application developer can override this behavior using the `onPushRequest` callback. The SDK calls this function when the agent pushes a document to the consumer, before it displays it. The callback function receives two functions: an `allow` function and a `deny` function. The callback function should call the `allow` function to show the pushed document to the consumer, or the `deny` function to reject the pushed document:

```
AssistSDK.onPushRequest = function(allow, deny) {
```

```
    var result = confirm("The agent wants to send you a document or image. Would
    you like to view it?");
    if (result)
      allow();
    else
      deny();
}
```

The above function's behavior is very similar to the default behavior. It shows a confirmation prompt on the screen, and lets the user click **OK** or **Cancel**, depending on whether they want to view the document. To always show documents without prompting:

```
AssistSDK.onPushRequest = function(allow, deny) {
  allow();
}
```

Document Callbacks

By default, after it receives a document, Remote Expert Mobile opens a window to display the shared document; if there is an error loading or parsing the shared document file, it displays an error window. It does this without any interaction from the application.

If it successfully load, parses, and displays the shared document, the SDK calls the `onDocumentReceivedSuccess` callback function; the function receives a `sharedDocument` object (described below). If an error occurs while trying to load or parse the shared document, it calls the `onDocumentReceivedError` callback function, which also receives a `sharedDocument` object. The two callback functions are optional - the SDK does nothing if you do not supply them.

The `sharedDocument` object that the SDK passes to the `onDocumentReceivedSuccess` and `onDocumentReceivedError` functions contains an `id` property, which is a unique identifier for the document; and may contain a `metadata` property, which contains additional information about the document supplied by the agent. It also has a `close` method, which the application can call to close the shared document window or error window. Additionally, the application may add an `onClosed` handler to the `sharedDocument` object, to receive notification when the window closes due to a user (consumer or agent) closing it from the UI.

The following code creates `onDocumentReceivedSuccess` and `onDocumentReceivedError` callbacks, adds an `onClosed` handler to the `sharedDocument` object, and sets a timer to call `sharedDocument.close()`:

```
AssistSDK.onDocumentReceivedSuccess = function(sharedDocument) {
  console.log("*** shared item opened successfully: " + sharedDocument.id);
  sharedDocument.onClosed = function(actor) {
    alert("Shared document window has closed by " + actor + ".");
  };
  console.log("Setting shared item " + sharedDocument.id + " to close in 15
  secs.");
  setTimeout(function() {
    console.log("*** Closing shared item " + sharedDocument.id);
    sharedDocument.close();
  }, 15 * 1000);
};

AssistSDK.onDocumentReceivedError = function(sharedDocument) {
  console.log("*** shared item opened with error: " + sharedDocument.id);
  sharedDocument.onClosed = function(actor) {
    alert("Shared document error window has been closed by " + actor + ".");
  };
  setTimeout(function() { sharedDocument.close();}, 5 * 1000);
};
```

Annotation Callbacks

There are two callbacks whichnotify the application when an agent draws on a shared screen:

- `onAnnotationAdded(annotation, sourceName)`

  Called when an annotation is received from an agent. The `annotation` object contains the following properties:

Property	Description
`stroke`	The color of the annotation
`strokeOpacity`	A number between 0.0 and 1.0 indicating the opacity of the annotation
`strokeWidth`	A number giving the width of the line of the annotation
`points`	An array of points representing the path of the annotation. By default, Remote Expert Mobile draws a line with the color, opacity, and width, following these points as its path.

- `onAnnotationsCleared()`

  Called when an agent clears the annotations.ons.

You can implement these callbacks to control the display and clearing of annotations, or simply to record what the agent has sent:

```
AssistSDK.onAnnotationAdded = function(annotation, sourceName) {
   console.log("Annotation added by " + sourceName);
};

AssistSDK.onAnnotationsCleared = function() {
   console.log("Annotations cleared");
}
```

See **Annotations on page 37** for more details.

Zoom Callbacks

The application can receive notifications when the zoom window opens or closes (see **Zoom on page 37**):

```
AssistSDK.onZoomStarted = function() {
    ⋮
   pushDocumentButton.disabled = true;
});

AssistSDK.onZoomEnded = function() {
    ⋮
   pushDocumentButton.disabled = false;
});
```

You might want to use these callbacks to update the user interface to prevent user interaction which will not work.

The application will receive these callbacks whether the consumer or agent application opens or closes the zoom window.

Co-browsing Callbacks

As well as using the CSS class mechanism to customize its user interface (see **Customizing the Remote Expert Mobile popup Window on page 39**), there are two callback functions which the consumer web application can implement to define what happens when co browsing starts and ends:

```
AssistSDK.onCobrowseActive = function() {
```

```
    // Display indicator, log, etc.
}
AssistSDK.onCobrowseInactive = function() {
    // Remove indicator, log, etc.
}
```

If an application does not provide these callbacks, the SDK provides a default implementation, displaying a banner at the top of the browser window, stating ***This page is currently being shared***.

Agent Callbacks

The application can implement the following callbacks to receive notification when agents join and leave the co-browsing session::

- `onAgentJoinedSession(agent)`

  This callback indicates that an agent has answered the support call and joined the support session; this occurs before the agent either requests or initiates co-browsing. The callback allows the developer to pre-approve the agent into the co-browse, before the agent makes the request.

- `onAgentRequestedCobrowse(agent)`

  This callback notifies the developer that the agent has specifically requested to co-browse. There is no specific requirement for the application to allow or disallow co-browsing at this point, but it is an obvious point to do so.

- `onAgentJoinedCobrowse(agent)`

  This callback indicates when the Agent joins the co-browse session.

- `onAgentLeftCobrowse(agent)`

  This callback occurs when the agent leaves the co-browse session, and can no longer see the consumer's screen. Leaving the co-browse also resets the agent's co-browse permission; the agent may subsequently request co-browse access again.

- `onAgentLeftSession(agent)`

  This callback notifies the application that the agent has left the overall support session.

The `agent` parameter to all these callbacks is a *JavaScript* object which can be passed in to the `AssistSDK.allowCobrowseForAgent` or `AssistSDK.disallowCobrowseForAgent` functions. See **Allow and Disallow Co-browse for an Agent on the facing page**.

These callbacks allow the developer to maintain a list of agents that are in the co-browse and dynamically allow them in and out of the co-browsing session at any time. To do this the developer can hold on to the agent references that they receive during the `onAgentJoinedSession` callback, which will remain valid, and can then admit and eject agents during the co-browsing session on whatever basis the application determines.

The default implementation displays a dialog box on the consumer's device, asking whether to allow co-browsing or not. If the consumer allows co-browsing, it allows any agent into the co-browsing session whenever they request it. Implementing this interface can give the application more control over which agents are allowed into the co-browsing session, and when.

onEndSupport

When a Remote Expert Mobile session terminates (for example when the call ends, or the consumer application calls `AssistSDK.endSupport`), the application can receive notification in the `onEndSupport` function.

```
AssistSDK.onEndSupport = function() {
    ⋮
};
```

This callback provides a place for the application to reset its user interface to indicate that it is no longer in a support session. Remote Expert Mobile removes its own UI automatically, so the application only needs to restore any changes it has made itself.

onError

The application can handle error events that cause the failure of a Remote Expert Mobile session using the `onError` callback:

```
AssistSDK.onError = function(error) {
    ⋮
}
```

The `error` object received by the callback is a *JavaScript* object with properties `code` and `message`. The `message` property is a free-form text message. The following error codes may be received:

Error Code	Value	Received by:		Meaning
		Agent	Consumer	
`CONNECTION_LOST`	`0`	Yes	Yes	Failed to connect after retry count. No retry intervals specified, will not attempt to reconnect.
`PERMISSION`	`1`	Yes	Yes	Received a permission change message on a topic with no permissions. Error trying to leave a topic. The message will include topic ID and error message.
`SOCKET`	`2`	Yes	Yes	Low level socket error. The message will include the socket error code.
`CALL_FAIL`	`3`	Yes	Yes	Tried to share a document when co-browsing is not active. Tried to allow or disallow co-browsing for an agent when support is not active.
`POPUP`	`4`	No	Yes	Couldn't reconnect to popup.
`SESSION_IN_PROGRESS`	`5`	Yes	Yes	There is already a session in use.
`SESSION_CREATION_FAILURE`	`6`	No	Yes	Error connecting to server. The message will include the server URL.

Not all errors can be received by both parties.

Allow and Disallow Co-browse for an Agent

You may wish to remove a specific agent from the co-browsing session. To do this, call:

```
AssistSDK.disallowCobrowseForAgent(agent)
```

passing in the agent object received in the `onAgentRequestedCobrowse` callback (see **Agent Callbacks on the previous page**).

If the agent is already in the co-browse session, they are removed from it; if they are not in the co-browse session, they will not be admitted until the application calls

```
AssistSDK.allowCobrowseForAgent(agent);
```

When the application calls `allowCobrowseForAgent`, the specified agent joins the co-browse immediately.

Web-specific considerations

On the web, when the consumer navigates to a new support enabled page during a support session, the co-browse, and indeed the entire support session, is torn down and recreated on the new page. This means that any agents will re-join the session on each page without any permission to access the co-browse, and permission will need to be re-granted to the appropriate agents in order for the co-browse to continue without interruption.

Pausing and Resuming a Co-browsing Session

The application can temporarily pause a co-browse session with the agent by calling:

```
AssistSDK.pauseCobrowse();
```

While paused, the connection to the Remote Expert Mobile server remains open, but the co-browse session is disabled, disabling annotations, document sharing, and so on as a consequence. When the application wishes to resume the co-browsing session, it should call:

```
AssistSDK.resumeCobrowse();
```

When the application pauses a co-browse, Remote Expert Mobile notifies the Agent Console, which can present a notification or message to the agent to indicate what has happened.

Sharing Documents

As well as receiving shared documents from the agent (see **onPushRequest on page 31**), applications can use the Remote Expert Mobile SDK to share documents with the agent during a co-browsing session. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

Documents shared in this way appear the same as documents pushed by the agent: PDFs are full screen; images are in windows that can be dragged, re-sized, or moved.

Note Sharing a document does not actually send the document to the agent, but simply displays the document on the local device, so that both the consumer and the agent can see and co-browse the document.

The application shares a document by calling:

```
AssistSDK.shareDocument(document, onLoad, onError);
```

Where

- `document` is a PDF document or image to be shared, expressed as one of the following:
  - A string URL pointing to the PDF document or image to share
  - A JavaScript file or Blob object containing the PDF document or image
- `onLoad` is a callback function that takes no arguments, and is called when the document is successfully loaded.
- `onError` is a callback function that is called when an error occurs loading the document, it is passed the following arguments:
  - an error code
  - an error message.

The error codes are the same as the agent-side error codes for document push, and may be one of the following:

Error Code	Value	Received By:	
		Agent	Consumer
SHARED_DOCUMENT_ERROR_CONNECTION_ERROR	1	Yes	Yes
SHARED_DOCUMENT_ERROR_HTTP_ERROR	2	Yes	Yes
SHARED_DOCUMENT_ERROR_UNSUPPORTED_MIME_TYPE	3	Yes	Yes
SHARED_DOCUMENT_ERROR_FILE_PARSING_ERROR	4	No	Yes
SHARED_DOCUMENT_ERROR_NO_DATA_RECEIVED	5	Yes	No

Error Code	Value	Received By:	
		Agent	Consumer
SHARED_DOCUMENT_ERROR_CO_BROWSE_NOT_ACTIVE	6	No	Yes

Not all values are possible in either case, for example, the agent never receives error code 6, and the consumer never receives error code 5.

The error message is a text string describing the error; it is intended for debugging and logging, rather than for displaying to an end user.

Zoom

Either agent or consumer can open a zoom window on the consumer's device. While the zoom window is open, it displays a magnified version of the content of the consumer's screen where it is positioned.

The zoom window contains controls to change the magnification and to close the window. Either party can use these controls, or move the window about the consumer's screen by dragging it (so the agent can move the zoom window to a part of the consumer's screen they want to look at, or the consumer can move it to a part of the screen they want the agent to look at).

You can change the appearance of the zoom window in CSS by adding styles for the element with ID `assist-zoom-window`; for example:

```
#assist-zoom-window {
   border: 3px solid red;
}
```

to give the zoom window a red border for greater visibility.

Opening the Zoom Window

The application can open the zoom window by calling the `AssistSDK.startZoom` function:

```
zoom: function() {
   AssistSDK.startZoom();
    ⋮
}
```

You would normally assign the `zoom` function to the `onclick` handler of a button.

There is an equivalent `AssistSDK.endZoom` function, but you will not normally need to call this explicitly; normally, one of the users closes the zoom window with the its close button, and if it is open when the Remote Expert Mobile session ends, Remote Expert Mobile closes it automatically.

Note

Document sharing and zooming are mutually exclusive. If the zoom window is open when you call `AssistSDK.shareDocument`, it has no effect (apart from logging a message to the console). Similarly, if a shared document is open when you call `AssistSDK.startZoom`, it does nothing.

Annotations

By default the Remote Expert Mobile SDK displays any annotations which the application receives on an overlay, so that the consumer can see them together with their own screen. Normally an application needs to do nothing further, but if it needs to receive notifications when an annotation arrives or is removed, it can implement one of the annotation callbacks (see **Annotation Callbacks on page 33**).

Setting the z-index of the annotation layer

Elements in HTML pages may have a `z-index` property, which specifies the order to display them. Elements with a high `z-index` appear in front of elements with a lower `z-index`, potentially hiding the lower `z-index` elements.

Some sites may have a high `z-index` on some elements, leading to annotations appearing behind them. Using CSS, you can set the `z-index` value of the `glass-pane` so that it is high enough to overlay all the elements on the page:

```
#glass-pane {
   z-index:XXXXX !important; // Set to appropriate value
}
```

- Legitimate values for `z-index` are `auto`, `initial`, `inherit`, or a number (negative numbers are allowed), but if you need to set it, you will probably want to set it to a positive number in order to bring the annotation layer to the top. The other values seem to be less useful in this case.
- `z-index` only works on positioned elements (`position:absolute`, `position:relative`, or `position:fixed`).
- `!important` is necessary in order to override the `z-index` setting of other objects.

Form Filling

One of the main reasons for a consumer to ask for help, or for an agent to request a co-browse, is to enable the agent to help the consumer to complete a form which is displayed on their device. The agent can do this whenever a Remote Expert Mobile co-browse session is active, without further intervention from the application, but there are some constraints on how forms should be designed.

The Remote Expert Mobile SDK automatically detects form fields represented by `input` elements, and relays these forms to the agent so that the agent can fill in values for the user. You must provide each element with a unique label in the HTML, in one of the following ways:

- providing a `label` for the field and including the `for` attribute:
  ```
  <label for="otherloans_id">Other Loans: </label>
  <input id="otherloans_id" type="text"/>
  ```
- setting the `title` attribute of the `input` element:
  ```
  <input type="text" title="Other Loans"/>
  ```
- setting the `name` attribute of the `input` element:
  ```
  <input type="text" name="Other Loans"/>
  ```
- setting the `id` attribute of the `input` element:
  ```
  <input type="text" id="other_loans"/>
  ```
- setting the `value` attribute of the input element, *if* the `input` is of type `radio`:
  ```
  <input type="radio" name="bedrooms" value="studio"/>
  <input type="radio" name="bedrooms" value="one"/>
  <input type="radio" name="bedrooms" value="two"/>
  ```

The SDK looks for a label to present to the user in the order above; if it does not find a `<label>` element for the field, it will look for a `title` attribute; if it does not find a `title` attribute either, it will look for a `name` attribute; and so on.

The SDK automatically prevents the agent from performing form fill if the `type` is `password`.

Note While the SDK prevents these fields from being presented to the agent as fillable form data, it does not prevent them from being visible as part of the co-browse. You can hide them by adding the appropriate class or permission to the element (see **Excluding Elements from Co-browsing below**).

Excluding Elements from Co-browsing

When an agent is co-browsing a form, you may not want the agent to see every control on the form. Some may be irrelevant, and some may be private to the consumer.

To do this, add a CSS class (`assist-no-show`) to HTML elements, which instructs the Remote Expert Mobile SDK to mask those areas:

```
<div id="sensitive-details" class="assist-no-show">content</div>
```

By default, Remote Expert Mobile shows excluded elements as black boxes that occupy the same space on the page as the original element; you can specify the color of the box using the `color` attribute of the special `assist-no-show-agent-console` CSS class in your stylesheet (the `color` attribute is the only attribute of the `assist-no-show-agent-console` class that has any effect). The `color` attribute only affects the rendering of the boxes on the agent console, and does not affect the display of the elements on the consumer's pages. For example, the following CSS code makes elements marked with the `assist-no-show` class display as orange boxes in the agent console:

```
.assist-no-show-agent-console {
   color: orange;
}
```

You can make them not appear at all:

```
.assist-no-show-agent-console {
   color: transparent;
}
```

For more detailed control over element visibility, see **Permissions on page 42**.

Co-browsing Visual Indicator

The SDK provides a means to customize the visual indication displayed during screen sharing. The default implementation displays a banner at the top of the window. During screen sharing, the main window of the application has the CSS class `assist-cobrowsing` (in addition to any other CSS classes it may have). You can customize the visual indication by defining this class in your style-sheet and adding properties to it.

Customizing the Remote Expert Mobile popup Window

You can customize the colors, fonts, and images of the Remote Expert Mobile popup window by creating a CSS file with styles for the `body` tag, and for elements with the `#title`, `#logo`, and `#status` IDs. When you call `startSupport`, include the CSS file URL as the `popupCssUrl` member of the configuration object:

```
var config = {
   destination: "agent1",
   popupCssUrl: "/assistsample/css/popup.css"
};
 ⋮
AssistSDK.startSupport(config);
```

To customize the background of the window, specify background attributes for the `body` tag:

```
body {
   background-color: #0000FF;
   background-image: url('/assistsample/img/foo.jpg');
}
```

To customize the Remote Expert Mobile logo, specify a background image for the `#logo` ID, along with `width` and `height` attributes:

```
#logo {
   background-image: url('/assistsample/img/newlogo.png');
   width: 64px;
   height: 64px;
}
```

Customize fonts by specifying `font` attributes for the `#title` and `#status` IDs.

Popup window position

The default position of the Remote Expert Mobile popup window may obscure an important part of the consumer's screen. The application can control the position of the window by including the `popupInitialPosition` property in the configuration object passed to `startSupport`. The value should be an object containing two properties, `top` and `left`, which control the position (in pixels) of the popup window:

```
var config = {
   popupInitialPosition = {
      top: 200,
      left: 500
   },
};
 ⋮
AssistSDK.startSupport(config);
```

Note If you use negative numbers in for the `top` and `left` values, the Remote Expert Mobile popup window appears at `0,0` on the consumer's screen.

WebSocket Reconnection Control

When a co-browse session disconnects due to technical issues, the default behavior is to attempt to reconnect six times at increasing intervals. You can control this behavior by passing in one or both of the following when the application calls `startSupport` (see **Session Configuration on page 25**):

- Connection configuration
- A set of callbacks for connection events, allowing an application to perform its own reconnection handling, or to simply inform the user of the status of the current connection

Connection Configuration

You can use the optional `retryIntervals` property of the connection object to control reconnection behavior (see **Session Configuration on page 25**):

```
var configuration;
configuration.destination = 'agent1';
configuration.retryIntervals = [5.0,10.0,15.0];
 ⋮
AssistSDK.startSupport(configuration);
```

If the WebSocket connection to the Remote Expert Mobile server goes down, Remote Expert Mobile will try to re-establish the connection to the server the number of times specified in the array, with the specified time in seconds between them. In the above example, Remote Expert Mobile would try to reconnect 5 seconds after the initial disconnection; then, if that fails, it would try 10 seconds after that; then, if that fails, it would try 15 seconds after that; and if that reconnection attempt fails, it will give up and not try again.

Note If you do not specify `retryIntervals` in the connection object, Remote Expert Mobile will use its default values, which are `[1.0, 2.0, 4.0, 8.0, 16.0, 32.0]`. If you specify an empty array, Remote Expert Mobile will make no reconnection attempts.

Connection Callbacks

If the default reconnection behavior of Remote Expert Mobile is not what you want, even after specifying the retry intervals, you can implement a set of connection callbacks and pass them to Remote Expert Mobile in the `connectionStatusCallbacks` property of the configuration object:

```
var callbacks = {
   onDisconnect: function(error, connector) {},
   onConnect: function() {},
   onTerminated: function(error) {},
   willRetry: function(inSeconds, retryAttemptNumber, maxRetryAttempts,
   connector) {}
};
var config = {destination: 'agent1', connectionStatusCallbacks: callbacks};
 ⋮
AssistSDK.startSupport(config);
```

The `connectionStatusCallbacks` property is itself an object with the properties `onDisconnect`, `onConnect`, `onTerminated`, and `willRetry`. These must all be functions defined in the JavaScript.

Note

- These callbacks need to be defined and added to the configuration explicitly as above. It is not enough to define them on the appropriate object, as it is with other callbacks.
- If you do not specify `retryIntervals` in the configuration object, Remote Expert Mobile will use its default reconnection behavior; if you specify `retryIntervals` , Remote Expert Mobile will use its default reconnection behavior using those values. You can turn off the default reconnection behavior, and take full control of reconnection, by specifying an empty list for `retryIntervals`.

When implementing your own reconnection logic, the most important notifications you receive are `onDisconnect` (called whenever the connection is lost) and `willRetry` (called when automatic reconnection is occurring, and there are more reconnection attempts to come). Both these methods include a `Connector` object in their arguments. You can use the `Connector` object to make a reconnection attempt, or to terminate all reconnection attempts.

Callback	Description
`onDisconnect`	Called for the initial WebSocket failure, and for every failed reconnection attempt (including the last one). This method is called regardless of whether `retryIntervals` is specified (that is, whether automatic reconnections are attempted or not). The `Connector` allows the implementing class to 'take control' of reconnecting, even if reconnection is automatic. For example, an application might decide to give up reconnection attempts even if more reconnection events would subsequently occur, or to try the next reconnection attempt immediately and not wait until the next retry interval has passed
`onConnect`	Called when a reconnection attempt succeeds. This may be useful to clear an error in the application, or for canceling reconnection attempts if the application is managing its own reconnections.
`willRetry`	Called under the following conditions: ■ when the WebSocket connection is lost, or ■ when a reconnection attempt fails *and* automatic reconnections are occurring (`retryIntervals` is a non-empty array or unspecified) *and* there are more automatic reconnection attempts to be made. Reconnection behavior can be overridden by using the `Connector` object. For example, a reconnect attempt could be made straight away.

Callback	Description
`onTerminated`	Called under the following conditions: ■ when all reconnection attempts have been made and failed, or ■ when either the `Connector.disconnect` or the `AssisstSDK.endSupport` function is called.

Example—make a reconnection attempt immediately on disconnection:

In this example, the default reconnection behavior has been disabled, and the application reconnection behavior is dependent on the reason for disconnection.

```
var onDisconnect = function(error, connector) {
   switch(error.code) {
      case -1:
         connector.terminate(error);
         break;
      default:
         connector.reconnect();
         break;
   }
}
```

Example—terminate reconnection attempts in response to user command:

In this example, the default reconnection behavior has not been disabled, but there is a UI control which the user can press to short-circuit the reconnection attempts. If the user has not terminated the connection attempts, automatic reconnection attempts continue.

```
var willRetry = function(retryInSeconds, retryAttemptNumber,
maximumRetryAttempts, connector) {
   if (userHasTerminatedConnection) {
      connector.terminate({code: -1, message: 'User has terminated
      connection'});
   }
}
```

Permissions

This section shows how to set and read permissions using the JavaScript SDK. See **Permissions on page 19** for details of how to use permissions to mask elements from an agent's view.

- ## Control element permissions

  Client applications assign permission markers to UI control elements by calling the `AssistSDK.setPermissionForElement` method:

  ```
  var element = document.getElementById('element_id');
  AssistSDK.setPermissionForElement('permission_X', element);
  ```

  or by setting it on the element in the HTML as a data attribute:

  ```
  <input type='button' id='id_hidden' data-assist-permission='permission_
  X'/>
  ```

  where `permission_X` is the *permission marker* to set on the control.

- ## Agent permissions

  The application can determine an agent's permissions from the `agent` object which it receives in the agent callbacks (see **Agent Callbacks on page 34**). If the application needs to examine this (for instance, to notify the consumer that a particular control will not be visible to the agent), use the `viewablePermissions` and `interactivePermissions` properties of the `agent` object. These properties are arrays of strings representing the permissions an agent has:

```
var permissions = agent.viewablePermissions();
var index = permissions.findIndex(function(element) {
   return element == 'permission_X';
});
if (index >= 0) {
    ⋮
}
```

Dynamic Web Element Masking

You can also mask page elements that are dynamically added and removed using AJAX. To do this, the application should call `setPermissionForElementWithId`, which allows the application to add a permission to an element which does not yet exist:

```
AssistSDK.setPermissionForElementWithId('permission_X', 'element_id');
```

When the application calls the above method, typically when the page is loaded, Remote Expert Mobile:

- Checks to see if the element exists on the page:
  - if it does, then the element is marked with the given permission.
  - otherwise, it stores the combination of permission and element ID.
- Listens for DOM change events, and when a new element is added:
  - if the element ID corresponds to one of the stored element IDs, Remote Expert Mobile adds the stored permission.

Note The list of permission markers and element IDs is cleared when the page is refreshed, so `setPermissionForElementWithId` does need to be called when the page is loaded.

The application can also call:

```
AssistSDK.setPermissionForElementInIframeWithId('permission_X', 'elementId',
iframe);
```

which does the same for an element within an iframe. The `iframe` parameter is the iframe element itself (acquired by calling `getElementById`, `createElement('iframe',...)`, or a similar function of the `Document` object).

Internationalization

The Remote Expert Mobile Web SDK keeps its assets in `assist_assets.war`. This file can be directly edited to add another language, using the following procedure:

1. Get a copy of `assist_assets.war` from `/opt/<version>/REAS/domain/deployment_backups`. It will be named `assist_assets.war-<datetime>`, where `<datetime>` is a timestamp in ISO 8601 format.
2. Unzip it and open the file at `sdk/web/shared/locales/assistIi18n.en.json` (this is the English language file).
3. Edit the entries so that the values are in the target language.
4. Save the file in the same directory as `assistIi18n.<lang>.json`, replacing `<lang>` with the 2 letter language code of the target language (e.g. `fr` for French).
5. Re-zip the file, maintaining the original file structure, and redeploy it to the server (update `assist_assets.war` with the new file -see the *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)*).

When calling `AssistSDK.startSupport`, provide a `locale` parameter in the configuration object. The value should be the 2 letter language code for the target language.

Self-signed Certificates and Internet Explorer

Using Internet Explorer and HTTPS to connect to a REM installation that has a self-signed certificate causes the call to fail—the REM popup window appears, but it is blank (that is, it has no content at all).

To resolve this, use HTTP instead, or add the self-signed certificate to the IE trust store to use HTTPS.

WebSocket Initiation

To prevent getting a `302` error reported to you by a WebSocket during handshaking, ensure that your deployment allows direct access to the WebSocket endpoint:

```
wss://<reas address>:<port>/assistserver/topic
```

CHAPTER 6

Remote Expert Mobile—CSDK for iOS (Objective C)

This is an overview of the tasks and development required to integrate RE Mobile with a pre-existing web application using the CSDK for iOS. Developers can easily embed voice, video and or Expert Assist sessions in their website or web application with Objective C.

Integration with an Existing Application

You can integrate Remote Expert Mobile with an existing iOS application. You need to create an XCode project. For simplicity, the examples following assume that the project was empty, but a similar approach will work if you want to add the Remote Expert Mobile SDK to an existing iOS application.

iOS and Xcode Supported Versions

The officially supported versions of Xcode and iOS for REM 11.6(1) are Xcode 6 and iOS 8—to use Xcode 7 and run on iOS 9, or Xcode 8 and run on iOS 10, rebuild the sample app using the instructions below, and re-submit to the app store.

Existing application binaries built with earlier versions of Xcode should continue to work without modification, although you may be prompted to trust the application/developer.

1. New or existing projects loaded into Xcode 7 or 8 require changes before they build and run:
2. Disable the generation of bitcode—`Enable Bitcode = NO`.

Add entries to your application's plist file to disable the new iOS 9 Application Transport Security feature—see the following for further information:
https://developer.apple.com/library/prerelease/ios/technotes/App-Transport-Security-Technote/

Note The iOS sample application works without the need for these changes.

Embedding the Remote Expert SDK Library

Remote Expert Mobile relies on the voice and video capabilities of REM Advanced SDK - unless the application only uses the co-browse mode of Remote Expert Mobile, you must install the Remote Expert library (see the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)* for details of how to install the Remote Expert into the XCode project).

By default, Remote Expert Mobile requests its own sessions from Remote Expert when the application starts a support session (see **Starting a Support Session on the facing page**). You do not need to start a session explicitly, unless you need to integrate the Remote Expert Mobile application with an existing authentication and authorization mechanism (see **Consumer Session Creation on page 15**).

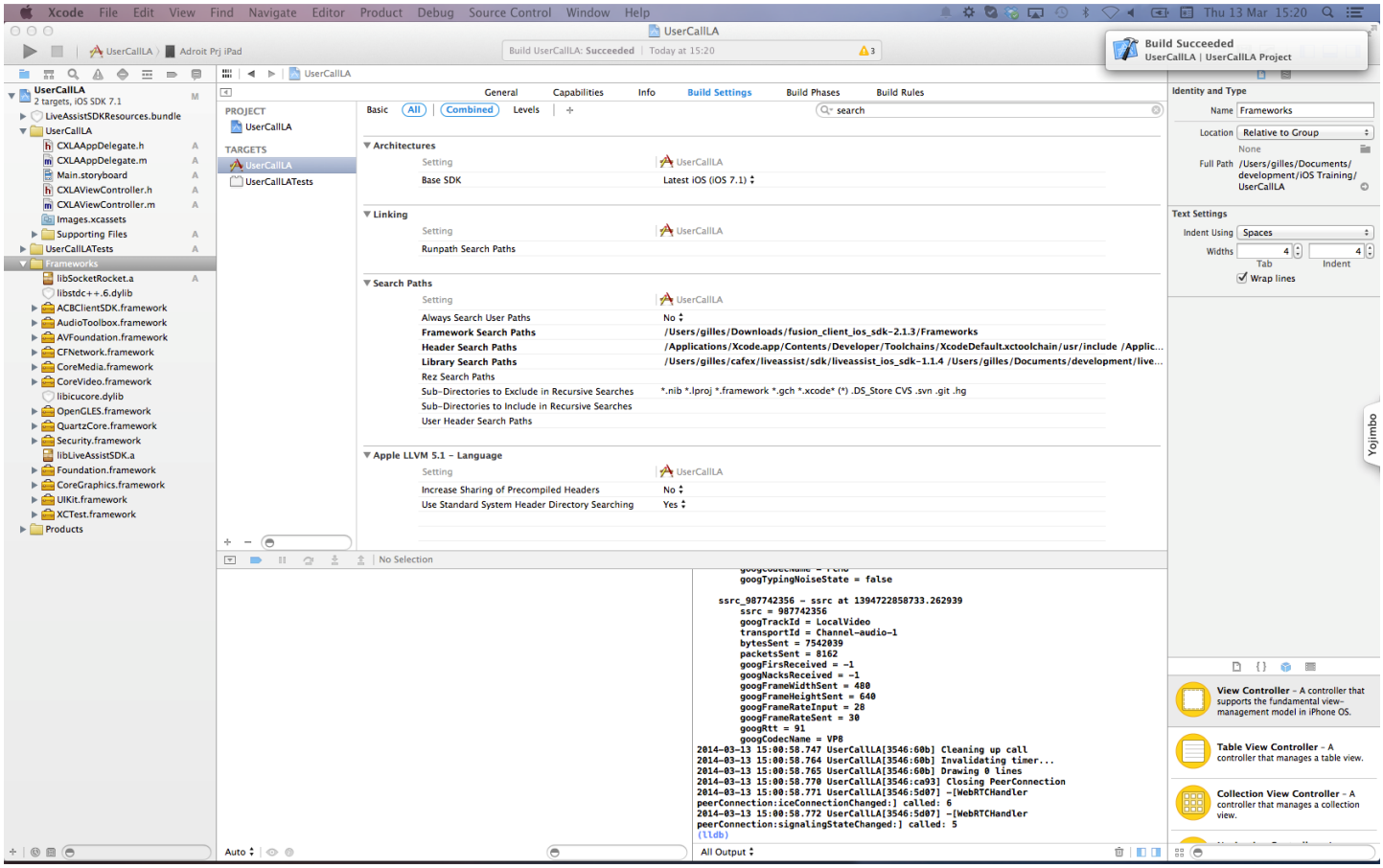
Embedding the Expert Assist Library

You need to link your code with the Remote Expert Mobile libraries and header files supplied in the `expert_assist_iOS_SDK-11.6.1.10000-7-ES3.zip` file, inside the `assist_ios_sdk` folder:

1. Copy the contents of the `assist_ios_sdk` folder of the `expert_assist_iOS_SDK-11.6.1.10000-7-ES3.zip` file somewhere suitable. This contains the headers and libraries for the application to link with.
2. In the **Target>Build Phases>LinkBinary With Libraries** section of your *XCode* project, add the `libAssistSDK.a` file.
3. In the **Target>Build Phases>LinkBinary With Libraries** section of your *XCode* project, add the following standard libraries:
   - `ImageIO.framework`
   - `MobileCoreServices.framework`
   - `WebKit.framework`
4. In **Build Settings**, ensure that **Header Search Paths** includes the location of `AssistSDK.h`, and that **Library Search Paths** includes the location of the `libAssistSDK.a` file.

Your project should look something like:

Figure 9:



Header Files

The public Remote Expert Mobile functions are defined in the Remote Expert Mobile header file, `AssistSDK.h`. You must include this in any compilation unit which makes use of the Remote Expert Mobile SDK:

```
#import <AssistSDK.h>
```

Starting a Support Session

The application starts a support session, normally in response to the user clicking on a **Help** or **Request Support** button, by making a call to the `AssistSDKstartSupport` class method:

```
[AssistSDK startSupport:@"support.test.com" destination:@"agent1"];
```

- The first argument (`server`) is an `NSString`. It can be the fully qualified hostname or IP address of the Remote Expert Mobile server (as above), or it can be a URL.

```
[AssistSDK startSupport:@"https://support.test.com:8443"
destination:@"agent1"];
```

Note If it is a URL, it should include only a scheme, hostname, and port. If the application is behind a reverse proxy, use a URL instead of just the server name.

- The second argument (`destination`) is also an `NSString`, and is the name of the agent or queue to contact.

```
#import <AssistSDK.h>

@interface WelcomeController ()
@end

@implementation WelcomeController
 ⋮
- (IBAction) startLiveAssist:(id)sender {
   NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
   NSString *address = [defaults stringForKey: @"serverAddress"];
   if ((address == nil) || ([address length] == 0)) {
      address = @"server.test.com";
   }
   // the single line to start the Live Assist SDK
   [AssistSDK startSupport: address destination:@"agent1"];
}
 ⋮
@end
```

Session Configuration

There is a more complex form of the `startSupport` method, which takes an `NSDictionary` as its second parameter. This parameter (`supportParameters`) can have a number of properties set on it to configure the session:

Property	Default Value/Behavior	Description
`destination`		Address of agent or queue (`NSString`)
`videoMode`	`@"full"`	Set whether video should be shown (`NSString`). Allowed values are: ■ `full` ■ `agentOnly` ■ `none`
`correlationId`		ID of the co-browsing session (`NSString`)
`acceptSelfSignedCerts`	`@NO`	`@YES` or `@NO` (`NSNumber`). Set to `@YES` to accept self-signed certificates in development environments. See **Accepting Self-Signed Certificates on page 71**.
`useCookies`	`@NO`	`@YES` or `@NO` (`NSNumber`). Set to `@YES` to send cookies set up to be sent to the Live Assist server on to the web socket connection.
`isAgentWindowOnTop`	`@NO`	`@YES` or `@NO` (`NSNumber`). Set to `@YES` to force the agent window to be topmost.
`hidingTags`		Set of numeric tags to use for obscuring

Property	Default Value/Behavior	Description
		content with black rectangles on the agent console (`NSSet`). See **Excluding Elements from Co-browsing on page 64**.
`maskingTags`		Set of numeric tags to use for masking content, making it appear as black or colored boxes on the agent console (`NSSet`). See **Excluding Elements from Co-browsing on page 64**.
`maskColor`		Color of boxes to be shown on agent console in place of masked content (`UIColor`).
`timeout`		Time in seconds to wait to establish communication with the Live Assist server (`NSNumber`).
`sessionToken`		REM Advanced SDK Web Gateway*Session Token* (if required) (`NSString`). See **Escalating a Call to Include Co-browse on page 52**.
`documentViewConstraints`		Instance of a class conforming to `ASDKDocumentViewConstraints`. See **Setting Shared Document View Constraints on page 61**
`addSharedDocCloseListener`	`@YES`	`@YES` or `@NO` (`NSNumber`). Set to `@NO` to remove any close link from a shared document, so that the agent and consumer cannot manually close the document.
`keepAnnotationsOnChange`	`@NO`	`@YES` or `@NO` (`NSNumber`). Set to `@YES` to keep annotations when the content behind them changes. Set to `@NO` to clear them when the content changes.
`screenShareRequestedDelegate`	Presents a `UIAlertView` to prompt the user to choose whether to accept screen sharing (but see the note at the end)	Instance of a delegate which conforms to `ASDKScreenShareRequestedDelegate`. Specifying this allows an application to choose whether to accept or reject screen sharing however it sees fit. See **ASDKScreenShareRequestedDelegate on page 57**.
`agentCobrowseDelegate`	Presents a `UIAlertView` to prompt the user to choose whether to accept screen-sharing (but see the note at the end)	Instance of a delegate which conforms to `ASDKAgentCobrowseDelegate`. Specifying this allows an application to receive notifications of agents joining and leaving the session. See **ASDKAgentCobrowseDelegate on page 56**.
`connectionDelegate`		Instance of a delegate which conforms to `ASDKConnectionStatusDelegate`. Specifying this allows an application to receive notifications of connection events. See **ASDKConnectionStatusDelegate on page 58**.

Property	Default Value/Behavior	Description
`pushDelegate`		A delegate which conforms to the protocol `ASDKPushAuthorizationDelegate`. Specifying this allows an application to choose whether to accept or reject pushed content however it sees fit. See **ASDKPushAuthorizationDelegate on page 57**.
`uui`		The value specified is placed in the SIP User-to-User Interface header in hex-encoded form. **Note:** The UUI can only be used when **Anonymous Consumer Access** is set to `trusted` mode (see the *Remote Expert Mobile Design Guide, Release 11.6 (1)* for more information). The UUI is ignored if the session token is provided.
`retryIntervals`	`[@1.0f, @2.0f, @4.0f, @8.0f, @16.0f, @32.0f]`	Indicates the number of automatic reconnection attempts, and the time in seconds between each attempt. See **Connection Configuration on page 66**. If an empty array is specified, then no reconnection attempt is made.
`maxReconnectTimeouts`	`[@5.0f]`	Indicates the maximum times in seconds, until WebSocket reconnection attempts fail. An array of values is given that corresponds to the values in `retryIntervals` - as each value in `retryIntervals` is used, the relevant value is used from this array. See **Connection Configuration on page 66**. **Note:** If the length of the `retryIntervals` is greater than that of `maxReconnectionTimeouts`, then the last value of the `maxReconnectionTimeouts` array is used.
`initialConnectTimeout`	`@30.0f`	Indicates the maximum time in seconds until the initial WebSocket connection attempt fails. See **Connection Configuration on page 66**.

You need to configure the session in this way for the more advanced uses of Remote Expert Mobile (see **During a Co-browse Session on page 53**), but the simple case is also covered:

```
NSDictionary *config = [NSDictionary dictionaryWithObjectsAndKeys:
  @"agent1", @"destination", nil];
[AssistSDK startSupport: @"server.test.com"
  supportParameters: config];
```

Note The application will not receive calls to `assistSDKScreenShareRequested` in `ASDKScreenShareRequestedDelegate`, even if an implementation is also supplied. In this case, only the methods in `ASDKAgentCobrowseDelegate` will be called. Remote Expert Mobile will only display the default UI if neither `screenShareRequestedDelegate` not `agentCobrowseDelegate` are supplied by the application.

Using UUI

The value specified in the `uui` element is placed in the SIP `User-to-User` Interface header exactly as it is passed. The application must ensure that the encoding is correct.

```
NSString *uui = @"5465737420555549" // Hex encoded String "Test UUI"
NSDictionary *laConfig = @{@"destination":@"agent1", @"uui":uui};
[AssistSDK startSupport:server supportParameters:config];
```

Note Setting the UUI has no effect in co-browse only sessions; see **Co-browse only (with code) on page 10** or **Co-browse only (with correlation ID) on page 9**

Co-browse only mode—Expert Assist with no voice or video by using Correlation ID

Remote Expert Mobile can also be used in *co-browse only* mode, for occasions when the voice or video call is provided independently of the Client SDK and Remote Expert Mobile system, or when something like a chat session is used instead of a voice and video call.

If the client application does not want Remote Expert Mobile to place a call using the Remote Expert SDK, the application can provide a *correlation ID* that Remote Expert Mobile uses to correlate the consumer and agent side. The correlation ID provides a way to join agent and consumer sessions without prior knowledge of the domain-specific way that sessions are identified. This allows an application to use the features of Remote Expert Mobile (for example, co-browsing, document push, annotation, and remote control) without voice or video.

To create a Remote Expert Mobile session without voice and video, provide a correlation ID in the `supportParameters` which you pass to `startSupport`:

```
NSMutableDictionary* config = [[NSMutableDictionary alloc] init];
 ⋮
config[@"correlationId"] = @"correlation-123";
 ⋮
[AssistSDK startSupport: @"server.test.com" supportParameters: config];
```

In this case, you should *not* supply a `destination` in the `supportParameters`.

Note In a co-browse only session, you will need to explicitly call `endSupport` when the call ends (or when the session is no longer needed), as Remote Expert Mobile will no longer present its default UI to the user.

Note The correlation ID needs to be known to both parties in the call, and needs to be unique enough that the same correlation ID is not used by two support calls at the same time. The application developer must decide the mechanism by which this happens, but possible ways are for both parties to calculate a value from data about the call known to both of them, or that one side calculates it and communicates it to the other on the existing communication channel. There is also a REST service provided by Remote Expert Mobile which will create a correlation ID and associate it with a short code; see **Co-browse only (with correlation ID) on page 9** and **Co-browse only (with code) on page 10**.

Escalating a Call to Include Co-browse

In most cases, the application calls `startSupport` with an agent name, and allows Remote Expert Mobile to set up a call to the agent and implicitly add Remote Expert Mobile support to that call. However, there may be cases where a call to an agent already exists, and the application needs to add Remote Expert Mobile support capabilities.. To do this, you need to supply the *session token* and a *correlation ID* in the configuration object which you supply to `startSupport`; and the agent needs to connect to the same session. The Remote Expert Mobile server provides some support for doing this.

1. The application connects to a specific URL on the Remote Expert Mobile server, to request a *short code* (error handling omitted):

```
NSString *url = @"<reas address>/assistserver/shortcode/create"];
NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:
url];
[request setHTTPMethod: @"PUT"];


NSURLSessionConfiguration *sc= [NSURLSession
sessionWithConfiguration:defaultConfiguration];
NSURLSession *session = [NSURLSession sessionWithConfiguration:sc
delegate:nil delegateQueue:nil];
NSURLSessionDataTask *task = [session dataTaskWithRequest:request
   completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable
   response,
      NSError * _Nullable error) {
         NSError *jerror = nil;
         NSDictionary *dictionary = [NSJSONSerialization
         JSONObjectWithData:data
            options:0 error:&jerror];
         NSString *shortcode = dictionary[@"shortCode"];
   }];
   [task resume];
```

2. The application uses the short code in another call to a URL on the Remote Expert Mobile server, and receives a JSON object containing a session token and a correlation ID:

```
NSString url = @"<reas address>/assistserver/shortcode/consumer?appkey=";
url = [url stringByAppendingString: shortcode];
NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:
url];
[request setHTTPMethod: @"GET"];


NSURLSessionConfiguration *sc= [NSURLSession
sessionWithConfiguration:defaultConfiguration];
NSURLSession *session = [NSURLSession sessionWithConfiguration:sc
delegate:nil delegateQueue:nil];
NSURLSessionDataTask *task = [session dataTaskWithRequest:request
   completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable
   response,
```

```
NSError * _Nullable error) {
   NSError *jerror = nil;
   NSDictionary *dictionary = [NSJSONSerialization
   JSONObjectWithData:data
      options:0 error:&jerror];
   NSString *sessionToken = dictionary[@"session-token"];
   NSString *correlationId = dictionary[@"cid"];
}];
[task resume];
```

3. The application includes those values in the configuration object, and passes it to `startSupport`:

```
NSDictionary *configuration = [[NSMutableDictionary alloc] init];
 ⋮
configuration[@"sessionToken"] = sessionToken;
configuration[@"correlationId"] = correlationId;
 ⋮
[AssistSDK startSupport:@"<reas address>" supportParameters:
configuration];
```

   More configuration can be set in the configuration object.

4. The agent uses the same short code to get a JSON object containing the session token and correlation ID, which it then uses to connect to the same Remote Expert Mobile support session (see the *Remote Expert Mobile Agent Console DeveloperGuide, Release 11.6 (1)*). Informing the agent of the short code is a matter for the application. It could be something as simple as having it displayed on the consumer's screen and having the consumer read it to the agent on the existing call (this is how the sample application does it).

Note

- When escalating an existing call, the `destination` property should *not* be set on the configuration object; in this case, the destination is known implicitly from the existing call.
- The short code expires after 5 minutes, or when it has been used by both agent and consumer to connect to the same session.
- If you wish to define an audit name to identify the consumer in event logs (see the *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* for more details on event logging), include an `auditName` parameter in the URL which creates the short code:

  `/assistserver/shortcode/create?auditName=consumer`

Ending a Session

When voice and video is enabled, the default UI that Remote Expert Mobile adds allows the user to end the session; otherwise, the session will be ended when the underlying support call ends. However, it is also possible for the application to programmatically end the session using the `endSupport` function. When voice and video is disabled, for example when Remote Expert Mobile is being used in co-browse-only mode (see **Co-browse only mode—Expert Assist with no voice or video by using Correlation ID on page 51**), the application has to call `endSupport`, as Remote Expert Mobile no longer presents its default UI to the user.

During a Co-browse Session

While a co-browsing session is active (after the application has called `startSupport` successfully, and before either it calls `endSupport` or receives the `supportCallDidEnd` notification (see **AssistSDKDelegate on page 55**) to indicate that the agent has ended the support session), the application may:

- Accept an agent into, or expel the agent from, the co-browsing session
- Pause and resume the co-browsing session
- Receive a document from the agent
- Push a document to the agent
- Receive an *annotation* (a piece of text or drawing to show on the device's screen, overlaid on the application's view) from the agent
- Have a form on its screen wholly or partly filled in by the agent

Actions which are initiated by the application (such as pushing a document to the agent) require it to call one of the class methods on the `AssistSDK` object.

Actions initiated by the agent (such as annotating the consumer's screen) can in general be allowed to proceed without interference from the application, as the Remote Expert Mobile SDK manages them, overlaying the user's screen with its own user interface where necessary. However, the application can receive notifications of these events by providing an implementation of one of the various `Delegates`, and can take control of the operations if it wishes.

Application Delegation

An application can receive notification of certain actions that occur within Remote Expert Mobile and take control of the response to them. The following sections discuss what notifications the application can receive and what operations are available.

The application can add delegates conforming to the following protocols to the configuration when it calls `AssistSDKstartSupport`:

- `ASDKScreenShareRequestedDelegate`
- `ASDKAgentCobrowseDelegate`
- `ASDKConnectionStatusDelegate`
- `ASDKPushAuthorizationDelegate`

It can only add a single instance of each of these delegates to the configuration (see **Session Configuration on page 48**).

It can add delegates conforming to the following protocols using the `AssistSDKaddDelegate` method:

- `AssistSDKDelegate`
- `AssistSDKDocumentDelegate`
- `AssistSDKAnnotationDelegate`

`AssistSDK` supports multiple delegates for any of these protocols. Delegates registered with `AssistSDK` must conform to at least one of them. If they do not, then they are not added to the delegate set. The delegates are not ordered, and the order in which they receive messages is not defined. Multiple delegates can support the same or different protocols. The application manages the registered delegates by calling the `addDelegate` and `removeDelegate` class methods.

In the following code, the `TabViewController` is the `AssistSDKDelegate`; it uses `addDelegate` to register itself, and receives calls to two of its notifications:

```
@interface TabViewController : UITabBarController<AssistSDKDelegate>
@end

@implementation TabViewController {
}

- (void) start : (NSString*) server {
   [AssistSDK addDelegate:self];
   [AssistSDK startSupport:server destination:@"agent1"];
}
```

```
- (void) supportCallDidEnd{
    ⋮
}
- (void) assistSDKDidEncounterError:(NSNotification*) notification {
   [self reportError:[notification object]];
    ⋮
}

- (void) reportError:(NSError*) error {
    ⋮
}
@end
```

Each `NSNotification` name is the capitalized form of the method name without the trailing colon; for example `@selector(assistSDKDidDoSomething)` yields a name of `@"AssistSDKDidDoSomething"`. `NSNotification object` and `userInfo` properties vary as detailed below.

The SDK defines another delegate protocol (`AssistSDKConsumerDocumentDelegate`), which is passed to the document sharing methods (see **Sharing Documents on page 59**).

AssistSDKDelegate

Adopting `AssistSDKDelegate` and implementing its methods enables the application to receive these notifications:

- `assistSDKDidEncounterError: (NSNotification*) notification`

  Implement this method to receive notifications when Remote Expert Mobile encounters an error. The `object` of the `notification` parameter is an `NSError`. No keys are defined for the `userInfo` dictionary, but error reporters may add additional details that could be useful.

  Each reported `NSError` has its `code` attribute set to one of the constants provided in the supplied `ASDKErrorCodes.h` file. See **Error Codes on page 71**.

- `cobrowseActiveDidChangeTo: (BOOL) active`

  Implement this method to receive notification when co-browsing becomes active or inactive. The `active` parameter is `YES` if co-browsing has started, and `NO` if it has stopped. You could use this to display something other than the default indication in the user interface.

- `supportCallDidEnd`

  Implement this method to receive notification of when the support call ends, either by the application calling `endSupport`, or the agent hanging up the call.. The callback is triggered only when an REM Advanced SDK support call is made; it does not occur in co-browse only mode. The callback has no parameters.

Add this delegate using the `AssistSDK addDelegate` method.

Your application will probably want to adopt this delegate, at least for error reporting.

AssistSDKAnnotationDelegate

Adopt the `AssistSDKAnnotationDelegate` to receive notifications relating to annotations which the agent may send to the consumer. It has three optional methods:

- `assistSDKWillAddAnnotation: (NSNotification*) notification`

  Called when the application receives an annotation from an agent. The application can make changes to the annotation before Remote Expert Mobile displays it. See **Notification of Annotations Received on page 63**.

- `assistSDKDidAddAnnotation: (NSNotification*) notification`

  Called when the application receives an annotation from an agent, immediately before Remote Expert Mobile displays it. See **Notification of Annotations Received on page 63**.

- `assistSDKDidClearAnnotations: (NSNotification*) notification`

  Called when an agent clears the annotations. See **Notification of Annotations Cleared on page 63**.

You can use this in order to control the display and clearing of annotations which the application receives from the agent (see **Annotations on page 62**).

Add this delegate using the `AssistSDK addDelegate` method.

AssistSDKDocumentDelegate

Adopting `AssistSDKDocumentDelegate` enables an application to receive notification when the agent sends a document to the consumer:

- `onOpened: (ASDKSharedDocument*) document`

  Called when the document has been received and displayed.

- `onClosed: (ASDKSharedDocument*) document by: (AssistSDKDocumentCloseInitiator) whom`

  Called when the document is closed. `AssistSDKDocumentCloseInitiator` is an enumeration with the following members:

  — `AssitSDKDocumentClosedByUnknown` (sic.)
  — `AssistSDKDocumentClosedByAgent`
  — `AssistSDKDocumentClosedByConsumer`
  — `AssistSDKDocumentClosedBySupportEnded`

- `onError: (ASDKSharedDocument*) document reason: (NSString*) reasonStr`

  Called if there was a problem displaying the document.

Each callback also has an `ASDKSharedDocument` parameter, which has a `close` method, allowing the application to close the document programmatically. It also has an `idNumber` property, allowing received documents to be compared, and a `metadata` property (an `NSString`), which receives any additional information which the agent has associated with the document.

By default, the Remote Expert Mobile SDK displays the document. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

Add this delegate using the `AssistSDK addDelegate` method.

ASDKAgentCobrowseDelegate

Adopting the `ASDKAgentCobrowseDelegate` enables the application to receive the following notifications:

- `agentJoinedSession: (ASDKAgent*) agent`

  This callback indicates that an agent has answered the support call and joined the support session; this occurs before the agent either requests or initiates co-browsing. The callback allows the developer to pre-approve the agent into the co-browse, before the agent makes the request.

- `agentRequestedCobrowse: (ASDKAgent*) agent`

  This callback notifies the application that the agent has specifically requested to co-browse. There is no specific requirement for the application to allow or disallow co-browsing at this point, but it is an obvious point to do so.

- `agentJoinedCobrowse: (ASDKAgent*) agent`

  This callback occurs when the agent joins the co-browse session.

- `agentLeftCobrowse: (ASDKAgent*) agent`

  This callback occurs when the agent leaves the co-browse session, and can no longer see the consumer's screen. Leaving the co-browse also resets the agent's co-browse permission; the agent may subsequently request co-browse access again.

- `agentLeftSession: (ASDKAgent*) agent`

  This callback notifies the application that the agent has left the overall support session.

The default implementation displays a dialog box on the consumer's device, and if the consumer allows co-browsing, allows all agents into a co-browsing session when they request it. The application can override the default behavior to, for example, pre-approve the agent. This delegate also allows the application to store the agent value for later use (see **Allow and Disallow Co-browse for an Agent on page 59**).

This delegate is added in the configuration when the application calls the `AssistSDKstartSupport` method (see **Session Configuration on page 48**).

ASDKScreenShareRequestedDelegate

Adopting `ASDKScreenShareRequestedDelegate` allows the application to receive notifications when the agent asks to share the consumer's screen. It has a single method:

- `assistSDKScreenShareRequested: (void (^)(void)) allow deny: (void (^)(void)) deny`

  Called when the agent has requested to share the consumer's screen. The `allow` and `deny` parameters are functions which allow or reject the co-browse request. The application should call one of them.

By default, Remote Expert Mobile pops up a `UIAlertView` which presents the user with options to accept or reject the request. The application can implement this method to override this behavior:

```
-(void) assistSDKScreenShareRequested:
   (void (^) (void)) allow deny: (void (^) (void)) deny {

   if ([self allowScreenshare]) {
      allow();
   } else {
      deny();
   }
}
```

Pass an instance of a class which conforms to the `ASDKScreenShareRequestedDelegate` protocol to `startSupport` as the `screenShareRequestedDelegate` attribute of the `supportParams` (see **Session Configuration on page 48**).

ASDKPushAuthorizationDelegate

When the agent pushes a document to the consumer, Remote Expert Mobile's default action is to prompt the consumer if they want to view it; if the consumer accepts, it shows the document to the consumer.

An application can supply an `ASDKPushAuthorizationDelegate` in the `supportParameters` of the call to `startSupport` (see **Session Configuration on page 48**) in order to control whether the consumer sees the document. It has a single callback (which is mandatory):

- `displaySharedDocumentRequested: (ASDKSharedDocument*) document, allow: (void (^) (void)) allow deny: (void (^) (void)) deny`

The `document` parameter is an `ASDKSharedDocument` (see **AssistSDKDocumentDelegate on the previous page** for details). The `allow` and `deny` parameters are functions which the application calls to accept or reject sharing of the document:

```
- (void)displaySharedDocumentRequested:
   (ASDKSharefDocument*)sharedDocument
```

```
    allow:(void (^)(void))allow
    deny:(void (^)(void))deny {

    if ([self isSharingAuthorized]) {
       allow();
    } else {
       deny();
    }
}
```

In this case, sharing is allowed if the `isSharingAuthorized` method (not shown, but it could check a flag set in the user interface or some application configuration, or show a bespoke prompt to the user) returns true.

To always show the document without prompting:

```
- (void)displaySharedDocumentRequested:
   (ASDKSharefDocument*)sharedDocument
   allow:(void (^)(void))allow
   deny:(void (^)(void))deny {
      allow();
}
```

ASDKConnectionStatusDelegate

The `ASDKConnectionStatusDelegate` has the following methods, which supply notifications about the connection status of the WebSocket connection to the Remote Expert Mobile server:

- `onConnect`

  Received when the WebSocket becomes connected to the Remote Expert Mobile server, so that the application can send and receive messages.

Note The application does not need to wait to receive this notification before it can use the Remote Expert Mobile API methods.

- `onDisconnect: (NSError*)reason connector: (ASDKConnector*) connector)`

  Received when the WebSocket connection to the Remote Expert Mobile server has been lost, or has failed to reconnect.

- `onTerminated: (NSError*) reason`

  Received when the WebSocket connection with the Remote Expert Mobile server has terminated, and there will be no more reconnection attempts. If the termination is due to `endSupport` being called explicitly, the error code will be `ASDKAssistSupportEnded` (see **Error Codes on page 71**).

- `willRetry: (float)inSeconds attempt: (int) attempt of: (int) maxAttempts connector: (ASDKConnector*) connector)`

  Received when the Remote Expert Mobile SDK is preparing to retry a connection attempt (controlled by the connection configuration which may be supplied in the configuration passed to `startSupport`; see **Connection Configuration on page 66**).

These callbacks can be used to control the reconnection strategy (see **Connection Status Delegate on page 67**).

Add this delegate to the configuration when the application calls the `AssistSDK startSupport` method (see **Session Configuration on page 48**).

Allow and Disallow Co-browse for an Agent

You may wish to remove a specific agent from the co-browsing session. To do this, call:

```
[AssistSDK disallowCobrowseForAgent: agent];
```

passing in the `ASDKAgent` object received in one of the notifications on the `ASDKAgentCobrowseDelegate` (see **ASDKAgentCobrowseDelegate on page 56**).

If the agent is already in the co-browse session, they are removed from it; if they are not in the co-browse session, they will not be admitted until the application calls:

```
[AssistSDK allowCobrowseForAgent: agent];
```

When the application calls `allowCobrowseForAgent`, the specified agent joins the co-browse immediately.

Pausing and Resuming a Co-browsing Session

The application can temporarily pause a co-browse session with the agent by calling:

```
[AssistSDK pauseCobrowse];
```

While paused, the connection to the Remote Expert Mobile server remains open, but the co-browse session is disabled, disabling annotations, document sharing, and so on as a consequence. When the application wishes to resume the co-browsing session, it should call:

```
[AssistSDK resumeCobrowse];
```

When the application pauses a co-browse, Remote Expert Mobile notifies the Agent Console, which can present a notification or message to the agent to indicate what has happened.

Sharing Documents

As well as receiving shared documents from the agent (see **AssistSDKDocumentDelegate on page 56**), applications can use the Remote Expert Mobile SDK to share documents with the agent during a co-browsing session. Acceptable documents are PDFs and images.

Documents shared in this way are represented visually in the same way as documents that are pushed from the agent: PDFs are full screen, and images are in windows that can be dragged, re-sized, or moved.


Note

Sharing a document does not actually send the document to the agent, but simply displays the document on the local device, so that both the consumer and the agent can see and co-browse the document.

There are three class methods exposed by `AssistSDK` to handle document sharing:

- `(NSError*) shareDocumentUrl:(NSString*) documentUrl delegate: (id<AssistSDKConsumerDocumentDelegate>) consumerShareDelegate`
- `(NSError*) shareDocumentNSUrl:(NSURL*) documentUrl delegate: (id<AssistSDKConsumerDocumentDelegate>) consumerShareDelegate`

  Both the above methods allow sharing a document given its URL. Typically, this would be used to share a document on another machine.


Note

The close callback (`onClose`) is not called if a link is pushed by using either of these methods.

- `(NSError*) shareDocument: (NSData*) content mimeType:(NSString*) mimeType delegate:(id<AssistSDKConsumerDocumentDelegate>) consumerShareDelegate`

  This method allows sharing a data block containing the document's data. Typically, this would be used to share a document on the local machine. In this case you must supply the mime type (typically `@"application/pdf"` for PDFs, or something like `@"image/jpeg"` or `@"image/png"` for images).

Whichever method is used, the final parameter of the method is an optional instance of a class conforming to `AssistSDKConsumerDocumentDelegate` which receives callbacks. The delegate has two methods:

- `onError: (ASDKSharedDocument*) document reason: (NSString*) reasonStr`

  Called when an error overlay is displayed by Remote Expert Mobile due to the failure to successfully display the shared document for some reason.

- `onClosed: (ASDKSharedDocument*) document by: (AssistDocumentCloseInitiator) whom`

  Called when the document is closed. See **AssistSDKDocumentDelegate on page 56** for the `whom` parameter.

Note For a consumer-shared document, the `idNumber` property of the `ASDKSharedDocument` always has a value of `-1`.

Errors are handled in two ways:

- By returning a non-nil `NSError`
- By invoking the `onError` method of a specified `AssistSDKConsumerDocumentDelegate` delegate.

The three methods return a non-nil `NSError` if they are invoked when screen-sharing is not active. The delegate `onError` method is invoked for all other error cases, for example if:

- An invalid URL is specified;
- A document cannot be downloaded from the specified URL;
- An invalid mime type is specified.

Embedding Shared Documents

By default, Remote Expert Mobile displays shared documents on top of the iOS application. The Remote Expert Mobile iOS SDK allows an application to embed a shared document in its view hierarchy. For example, an application may want to present a text chat window on top of a shared document.

An application can embed shared documents as shown in the following steps:

1. Create an instance of the iOS SDK's `ASDKDefaultDocumentViewController` class:

   ```
   UIViewController *dvc = [[ASDKDefaultDocumentViewController alloc] init];
   ```

2. Add it to the view controller of a window in the application's view hierarchy, ensuring that it displays within that window:

   ```
   UIViewController *rootViewController =
   [UIApplication sharedApplication].keyWindow.rootViewController;
   dvc.view.frame = rootViewController.view.bounds;
   dvc.view.autoresizingMask =
   UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;
   [rootViewController addChildViewController:dvc];
   [rootViewController.view addSubview: dvc.view];
   [dvc didMoveToParentViewController:rootViewController];
   ```

3. Pass this class to `startSupport` in the configuration:

   ```
   NSMutableDictionary* config = [[NSMutableDictionary alloc] init];
    ⋮
   config[@"documentViewController"] = dvc;
    ⋮
   [AssistSDK startSupport:server supportParameters:config];
   ```

4. Re-arrange the view hierarchy when required:

```
// Make 'otherView' appear above shared documents.
UIView *otherView=......
[rootViewController.view bringSubviewToFront:otherView];
```

If the application supplies the `ASDKDefaultDocumentViewController` as shown above, then it is the application's responsibility to dismiss it when required. Typically, this would be done when the call ends:

```
- (void) supportCallDidEnd {
   [dvc willMoveToParentViewController:nil];
   [dvc.view removeFromSuperview];
   [dvc removeFromParentViewController];
}
```

Setting Shared Document View Constraints

The application may control which portion of the screen is used to display shared documents and images by setting the `documentViewConstraints` configuration property (see **Session Configuration on page 48**); it should set the property to an object of a class conforming to the `ASDKDocumentViewConstraints` protocol. The class has four methods: `leftMargin`, `rightMargin`, `topMargin`, and `bottomMargin`. Each of these methods takes no arguments, and returns a `float`. If required, you can create properties using the `@property` directive, and let the compiler generate the methods for you:

```
#import <Foundation/Foundation.h>
#import "ASDKDocumentViewConstraints.h"

@interface DocumentViewConstraints : NSObject<ASDKDocumentViewConstraints>

@property (nonatomic, assign) float leftMargin;
@property (nonatomic, assign) float rightMargin;
@property (nonatomic, assign) float topMargin;
@property (nonatomic, assign) float bottomMargin;

- (id)initWithLeftMargin:(float)leftMargin rightMargin:(float)rightMargin
topMargin:(float)topMargin bottomMargin:(float)bottomMargin;

@end

@implementation DocumentViewConstraints

- (id)initWithLeftMargin:(float)leftMargin rightMargin:(float)rightMargin
topMargin:(float)topMargin bottomMargin:(float)bottomMargin
{
   if (self = [super init]) {
      self.leftMargin = leftMargin;
      self.rightMargin = rightMargin;
      self.topMargin = topMargin;
      self.bottomMargin = bottomMargin;
   }
   return self;
}

@end
```

And to use it:

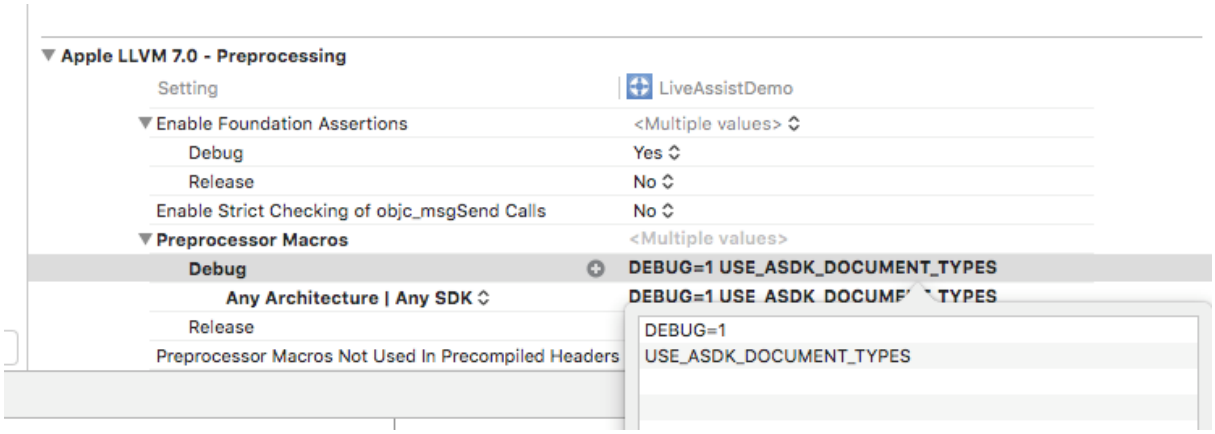
```
DocumentViewConstraints *constraints - [[DocumentViewContraints alloc]
   initWithLeftMargin:30 rightMargin:250 topMargin:250 bottomMargin:15];

NSMutableDictionary *config = [[NSMutableDictionary alloc] init];
 ⋮
config[@"documentViewConstraints"] = dictionary;
 ⋮
[AssistSDK startSupport:server supportParameters:config];
```

Pre-Processor Macros

The `DocType` enumeration normally contains the values `PDF`, `Image`, `Link`, and `Unknown`. If this is inconvenient (for instance, if one or more of these values is defined elsewhere), you can define the pre-processor macro `USE_ASDK_DOCUMENT_TYPES`, in which case `DocType` will contain `ASDKPDF`, `ASDKImage`, `ASDKLink`, and `ASDKUnknown`, respectively.

To use it:

1. Open up **Targets** in your project
2. Add your pre-processor to the pre-processing phase in the **Build Settings** tab:



Annotations

By default the Remote Expert Mobile SDK displays any annotations which the application receives on an overlay, so that the consumer can see them together with their own screen. Normally an application needs to do nothing further, but if it needs to receive notifications when an annotation arrives, it can define a class which conforms to the `AssistSDKAnnotationDelegate` protocol and add it to `AssistSDK` using `addDelegate`:

```
@interface AnnotationController : AssistSDKAnnotationDelegate
@end

@implementation AnnotationController {
}

- (void) start : (NSString*) server {
   [AssistSDK addDelegate:self];
}

 ⋮

@end
```

The `AssistSDKAnnotationDelegate` offers:

- Notification of new annotations received
- Notification of annotations cleared

See **AssistSDKAnnotationDelegate on page 55**.

Notification of Annotations Received

The two methods `assistSDKWillAddAnnotation` and `assistSDKDidAddAnnotation` are called when the agent sends an annotation to the consumer. The `notification` parameter for both methods has an `object` which is an `NSMutableDictionary` with the following members:

Key	Value Class	Description
`kASDKSVGPathKey`	`UIBezierPath`	The path which will be added as an annotation
`kASDKSVGLayerKey`	`CAShapeLayer`	The layer containing the annotation which will be displayed
`kASDKSVGPathStrokeKey`	`UIColor`	The color of the annotation
`kASDKSVGPathStrokeWidthKey`	`NSNumber`	A number giving the width of the line of the annotation
`kASDKSVGPathStrokeOpacityKey`	`NSNumber`	A number between 0.0 and 1.0 indicating the opacity of the annotation

The `userInfo` dictionary is used for ancillary data.

The `CAShapeLayer` is created and initialized from the other attributes between the calls to the `Will` and `Did` callbacks. Any values changed in the `Will` callback are used in the initialization. To change the color of the annotation to purple:

```
- (void) assistSdkWillAddAnnotation:(NSNotification*)notification {
   NSMutableDictionary* dic = [notification object];
   dic[@"kASDKSVGPathStrokeKey"] = [UIColor purpleColor];
}
```

To affect the display in the `Did` callback, the application must manipulate the `CAShapeLayer` directly:

```
- (void) assistSdkDidAddAnnotation:(NSNotification*)notification {

   NSMutableDictionary* dic = [notification object];
   CAShapeLayer* layer = dic[@"kASDKSVGLayerKey"];
   CGColor* purple = [[UIColor purpleColor] CGColor];

   [layer setStrokeColor:purple];
}
```

Notification of Annotations Cleared

To get notification when the agent clears the annotations, implement the `assistSDKDidClearAnnotations` method.

The `object` of the `notification` parameter is an array of `CAShapeLayer` instances which have been cleared. The dictionary used to create the layer is no longer available, but the layer could contain metadata placed on it in the `assistSDKDidAddAnnotation` callback:

```
- (void) assistDidAddAnnotation: (NSNotification*)notification {

   NSMutableDictionary* dic = [notification object];
   CAShapeLayer* layer = dic[@"kASDKSVGLayerKey"];
```

```
    [layer setValue:@"agent1" forKey:@"agentName"];
}

- (void) assistSDKDidClearAnnotations: (NSNotification*)notification {

    CAShapeLayer* layer = [notification object];
    NSString* agent = [layer valueForKey: @"agentName"];
    NSLog(@"Layer from %@ cleared", agent);
}
```

Form Filling

One of the main reasons for a consumer to ask for help, or for an agent to request a co-browse, is to enable the agent to help the consumer to complete a form which is displayed on their device. The agent can do this whenever a Remote Expert Mobile co-browse session is active, without further intervention from the application, but there are some constraints on how forms should be designed.

The Remote Expert Mobile SDK automatically detects form fields represented by `UIButton`, `UISlider`, `UISwitch`, `UIStepper`, or `UIDatePicker` controls, and relays these forms to the agent so that the agent can fill in values for the user. You must provide each element with a unique `Label` attribute, either in the *Interface Builder* or programmatically (if you are adding the controls programmatically).

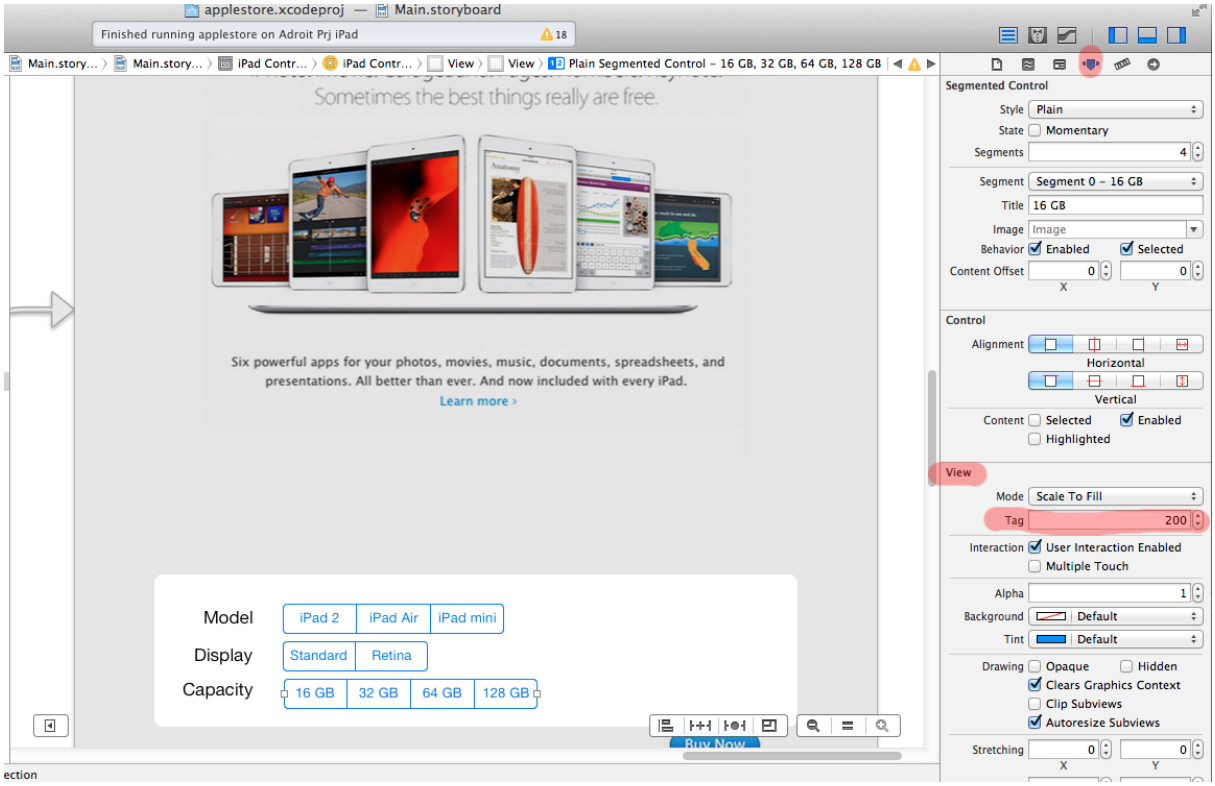
The SDK automatically prevents the agent from filling in the field if the `secureTextEntry` attribute is set to `true`, or **Secure** is specified in the Interface Builder.

Note While the SDK prevents these fields from being presented to the agent as fillable form data, it does not prevent them from being visible as part of the co-browse. If desired, they may be hidden by adding the appropriate tag or permission to the control (see **Excluding Elements from Co-browsing below**).

Excluding Elements from Co-browsing

When an agent is co-browsing a form, you may not want the agent to see every control on the form. Some may be irrelevant, and some may be private to the consumer.

In the Remote Expert MobileiOS SDK, you can mark the UI element with a specific tag value. Do this in XCode by opening the *Attribute Inspector*, then opening the **View** panel for the UI elements to exclude and entering them in the **Tag** field:



The tag value is an arbitrary numeric value. The application can hide or mask all controls with that tag value by passing it in the `hidingTags` or `maskingTags` members of the `supportParameters` when it calls `startSupport` (see **Session Configuration on page 48**).

Use of a unique tag value to obscure elements

In this scenario, the same tag value is used to mark all the elements that need to be obscured; these elements appear on the agent console as black rectangles. To do this, submit the single tag value as an argument of the `hidingTags` member of the `supportParameters`:

```
NSSet *tags = [NSSet setWithObjects: [NSNumber numberWithInteger:100], nil];
NSMutableDictionary *config= [[NSMutableDictionary alloc] init];
 ⋮
config[@"hidingTags"] = tags;
 ⋮
[AssistSDK startSupport: @"server.test.com" supportParameters:config];
```

Use of multiple tag values to hide elements

Some applications already use the `UIView` tag value, and require each UI component to use a unique value. Add the pre-existing tag values of those elements which you want to hide to an `NSSet` which you add to the configuration argument when calling the `startSupport` method:

```
NSSet *tags = [NSSet setWithObjects: [NSNumber numberWithInteger:100],
   [NSNumber numberWithInteger:200], nil];
NSMutableDictionary *config= [[NSMutableDictionary alloc] init];
 ⋮
config[@"hidingTags"] = tags;
 ⋮
[AssistSDK startSupport: @"server.test.com" supportParameters: config];
```

Masking elements

In addition to obscuring elements with black boxes, you can also mask elements using the `maskingTags` configuration property (see **Session Configuration on page 48**); in this case the masked elements appear as solid rectangles in the agent console. By default, the rectangles appear black in the agent console, but this color can be changed using the `maskColor` configuration property. Like `hidingTags`, the value of the `maskingTags` property must be an object of type `NSSet`, which contains objects of type `NSNumber` that are constructed from integers. The value of the `maskColor` property must be an object of type `UIColor`:

```
NSSet *tags = [NSSet setWithObjects: [NSNumber numberWithInteger:150],
   [NSNumber numberWithInteger:151], nil];
NSMutableDictionary *config= [[NSMutableDictionary alloc] init];
 ⋮
config[@"maskingTags"] = tags;
config[@"maskColor"] = [UIColor redColor];
 ⋮
[AssistSDK startSupport: @"server.test.com" supportParameters: config];
```

Note The sub-elements of any hidden or masked element are also implicitly hidden or masked.

For more detailed control over element visibility, see **Permissions on page 69**.

Note If you are using a `UIWebView` or `WKWebView`, and using HTML elements on that web view, you will not be able to exclude individual HTML elements from being seen by the agent. You can exclude the whole web view using the above techniques, but the methods described in the Web section of this guide for masking individual elements will not work.

Snapshots

iOS devices have a feature called *Snapshot*, which allows the user to take a screenshot and share it. If the consumer takes a screenshot and shares it (using the **Use Photo** button, or similar), the agent receives a notification from the Remote Expert Mobile Agent SDK (see the *Remote Expert Mobile Agent Console DeveloperGuide, Release 11.6 (1)*).

Note This is not strictly a Remote Expert Mobile iOS SDK feature, as doing it needs no interaction with the Remote Expert Mobile iOS SDK (apart from calling `startSupport`). However, developers should be aware of it.

WebSocket Reconnection Control

When a co-browse session disconnects due to technical issues, the default behavior is to attempt to reconnect six times at increasing intervals. You can control this behavior by passing in one or both of the following when the application calls `startSupport` (see **Session Configuration on page 48**):

- Connection configuration
- An instance of a delegate which conforms to `ASDKConnectionStatusDelegate`, allowing an application to perform its own reconnection handling, or to simply inform the user of the status of the current connection.

Connection Configuration

You can use the optional configuration items - `retryIntervals`, `maxReconnectTimeouts`, and `initialConnectTimeout` - to control connection and reconnection behavior (see **Session**

Configuration on page 48):

```
NSMutableDictionary config = [[NSMutableDictionary alloc] init];
 ⋮
config[@"retryIntervals"] = @[@5.0f,@10.0f,@15.0f];
config[@"maxReconnectTimeouts"] = @[@0.1f,@1.0f,@10.0f];
config[@"initialConnectTimeout"] = [NSNumber numberWithFloat:30.0f];
 ⋮
[AssistSDK startSupport: @"server.test.com" supportParameters: config];
```

If the WebSocket connection to the Remote Expert Mobile server goes down, Remote Expert Mobile will try to re-establish the connection to the server the number of times specified in the array, with the specified time in seconds between them. The above example sets a timeout for the initial connection of 30 seconds; if the connection is lost, it will try to reconnect 3 times, at intervals of 5, 10, and 15 seconds. The first reconnection attempt will time out after 0.1 seconds, the second after 1 second, and the third after 10 seconds.


Note

- If you do not specify `retryIntervals` in the `supportParameters`, Remote Expert Mobile will use its default values, which are `[@1.0f, @2.0f, @4.0f, @8.0f, @16.0f, @32.0f]`. If you specify an empty array, Remote Expert Mobile will make no reconnection attempts.
- Reconnection applies only to the case where an existing connection is lost. If the initial connection attempt fails, it is not retried automatically.

Connection Status Delegate

If the default reconnection behavior of Remote Expert Mobile is not what you want, even after changing the configuration, you can supply an instance of a class which conforms to `ASDKConnectionStatusDelegate` to implement your own reconnection logic, and include it in the `supportParameters` passed to `startSupport` - see Session Configuration on page 48 and ASDKConnectionStatusDelegate on page 58 for details.


Note

If you do not specify `retryIntervals` or `maxReconnectionTimeouts` in `supportParameters`, Remote Expert Mobile will use its default reconnection behavior; if you specify `retryIntervals` and `maxReconnectionTimeouts` in `supportParameters`, Remote Expert Mobile will use its default reconnection behavior using those values. You can turn off the default reconnection behavior, and take full control of reconnection, by specifying an empty list for `retryIntervals`.

When implementing your own reconnection logic, the most important notifications you will receive are `onDisconnect` (called whenever the connection is lost) and `willRetry` (called when automatic reconnection is occurring, and there are more reconnection attempts to come). Both these methods include a `ASDKConnector` object in their parameters. You can use the `ASDKConnector` object to make a reconnection attempt (by calling `reconnect`), or to terminate all reconnection attempts (by calling `terminate`). The `ASDKConnector` object remains valid after the call has ended, so the application can also hold onto it for use elsewhere.

Method	Description
`onDisconnect`	Called for the initial WebSocket failure, and for every failed reconnection attempt (including the last one). This method is called regardless of whether `retryIntervals` is specified (that is, whether automatic reconnection is attempted or not). The `ASDKConnector` class allows the implementing class to 'take control' of reconnecting, even if reconnection is automatic. For example, an application might decide to give up reconnection attempts even if more automatic reconnection events would subsequently occur, or to try the next reconnection attempt immediately and not wait until the next retry interval has passed.

Method	Description
	Note: The only error this method will receive is a transportation error (i.e. the network has gone down).
`willRetry`	Called under the following conditions: ■ when the WebSocket connection is lost; or ■ when a reconnection attempt fails *and* automatic reconnections are occurring (`retryIntervals` is a non-empty array) *and* there are more automatic reconnection attempts to be made. This method is called after the `onDisconnect` method. Reconnection behavior can be overridden by using the `ASDKConnector`. For example, a reconnect attempt could be made straight away.
`onConnect`	Called when a reconnection attempt succeeds. This may be useful to clear an error in the application, or for canceling reconnection attempts if the application is managing its own reconnection.
`onTerminated`	Called under the following conditions: ■ when all reconnection attempts have been made (and failed); or ■ when either the `[ASDKConnector disconnect]` method or `[AssistSDK endSupport]` method is called.

Example—make a reconnection attempt immediately on disconnection:

In this example, the default reconnection behavior has been disabled, but the user is able choose the reconnection behavior by setting the `reconnection_type` flag. In one case, it makes a reconnection attempt immediately; in another, it terminates all reconnection attempts; otherwise, it starts a timer to reconnect in 5 seconds:

```
ASDKConnector* reconnector;
int reconnection_type;

- (void) onDisconnect:(NSError *) reason connector:(ASDKConnector*)connector {
   switch (reconnection_type) {
      case RECONNECT_IMMEDIATELY:
         [connector reconnect:2.0f];
         break;
      case DO_NOT_RECONNECT:
         [connector terminate:reason];
         break;
      default:
         // Start timer for reconnection
         reconnector = connector;
         [NSTimer sceduledTimerWithTimeInterval:5.0f target:self
            selector:@selector(reconnect) userInfo:nil repeats:NO];
         break;
   }
}

- (void) reconnect {
   [reconnector reconnect:5.0f];
}
```

Example—terminate reconnection attempts in response to user command:

In this example, the default reconnection behavior has not been disabled, but there is a UI control which the user can use to change the maximum number of reconnection attempts. When that maximum number has been reached, reconnection attempts are terminated:

```
- (void) willRetry:(float) inSeconds attempt:(int) attempt of:(int)
maxAttempts connector:(ASDKConnector*) connector {
   if (attempt > userMaxAttempts) {
      [connector terminate: [[NSError alloc] init @"UserAction" code: -1]];
   }
}
```

Cookies

By default, Remote Expert Mobile does not include cookies on the Web Socket connection that it opens to the Remote Expert Mobile server. If the Web Socket needs to include all the appropriate cookies for the Web Socket URL, you can enable them by providing a `useCookies` configuration parameter set to `@YES` (see **Session Configuration on page 48**).

Remote Expert Mobile uses the cookies stored in the `NSHTTPCookieStorage` class provided by iOS, so any cookies which should apply to the Web Socket must be in the `NSHTTPCookieStorage` singleton *before* invoking `startSupport`. Remote Expert Mobile uses the `cookiesForURL` method of `NSHTTPCookieStorage` to obtain the collection of applicable cookies for the Web Socket.

Permissions

This section shows how to set and read permissions using the iOS SDK. See **Permissions on page 19** for details of how to use permissions to mask elements from an agent's view.

- ## Control element permissions

  Client applications assign permission markers to UI control elements by calling the `setPermission` class method of the `AssistSDK`:

  ```
  [AssistSDK setPermission: @"X" forView: control)];
  ```

  where `X` is the *permission marker* to be set on the control.

- ## Agent permissions

  The application can determine an agent's permissions from the `ASDKAgent` object it receives in the methods of the `ASDKAgentCobrowseDelegate` (see **ASDKAgentCobrowseDelegate on page 56**). If the application needs to examine this, but if it does (for instance, to notify the consumer that a particular control will not be visible to the agent), use the `viewable` and `interactive` values in the agent's `agentPermissions` dictionary.

  ```
  NSSet* viewable = agent.agentPermissions[@"viewable"];
  ```

Internationalization

Currently, there is no internationalization support in the Remote Expert Mobile SDK. We suggest that the application should supply its own implementations of those delegates which result in a default UI, and use them to show its own (internationalized) UI:

```
- (void) cobrowseActiveDidChangeTo: (BOOL)active
{
   if (active) {
      [self displayCobrowseNotification];
   } else {
      [self removeCobrowseNotification];
   }
}
```

```
- (void) displayCobrowseNotification
{
   NSString message = NSLocalizedString(@"cobrowse-message", nil);
    ⋮
}

- (void) removeCobrowseNotification
{
    ⋮
}
```

where we assume that the internationalized text for the co-browse message in the current language exists. The application is responsible for displaying the internationalized string to the user in whatever way it likes.

The following delegates show a default UI, and an internationalized application needs to implement them:

- `AssistSDKDelegate`
- `ASDKScreenShareRequestedDelegate`
- `ASDKPushAuthorizationDelegate`
- `ASDKAgentCobrowseDelegate`

Integrating an iOS Application with the Advanced SDK

When you call the `AssistSDK startSupport` method and provide a `destination`, but no `correlationId` or `sessionToken`, in the `supportParameters`, Remote Expert Mobile automatically starts a voice and video call and a co-browse session with the agent, and automatically ends the call when the application calls `AssistSDK endSupport`. If you want more control over the voice and video call than this, then you need to start the call using the REM Advanced SDK:

```
- (void) initialize
{
   NSString* sessionId = [self getSessionId];
   ACBUC* uc = [ACBUC ucWithConfiguration:sessionId delegate:self];
   [uc startSession];
}

- (void) ucDidStartSession:(ACBUC *)uc
{
   ACBClientPhone* phone = uc.phone;
    ⋮
   ACBClientCall* call = [phone createCallToAddress:calleeAddress
   withAudio:ACBMediaDirectionSendAndReceive
   withVideo:ACBMediaDirectionSendAndReceive delegate:self];
    ⋮
}

- (void) call: (ACBClientCall*)call didChangeStatus:(ACBClientCallStatus)
status
{
    ⋮
   if (status == ACBClientCallStatusInCall)
   {
      // Escalate call to co-bowse
```

```
        }
    }
```

After the call is connected, you need to escalate that call to co-browse (see **Escalating a Call to Include Co-browse on page 52**). In order to control the call while it is in progress, the application saves the `ACBClientCall` object returned by `createCallToAddress`, so that it can use it when needed. For convenience of illustration, `self` represents the two delegates which receive information on the progress of session creation and call setup.

See the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)*(available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) for more details of how to set up a call, and in particular, of how to obtain a session token to use as the sessionId. It also gives details of what call control features are available and how to use them.

Error Codes

Code	Value	Meaning
`ASDKERRCalleeNotFound`	`20101`	The dialed number could not be found
`ASDKERRCalleeBusy`	`20102`	The callee was unable to answer the call.
`ASDKERRCallCreationFailed`	`20103`	The creation of the call failed
`ASDKERRCallTimeout`	`20104`	The callee did not answer the call within the network's timeout
`ASDKERRCallFailed`	`20105`	The call failed to complete
`ASDKERRSessionFailure`	`20106`	Failed to establish a Client SDK session for voice and video
`ASDKERRCameraNotAuthorized`	`20107`	The user did not give permission to use the device's camera
`ASDKERRMicrophoneNotAuthorized`	`20108`	The user did not give permission to use the device's microphone
`ASDKERRAssistSessionCreationFailure`	`30101`	The Assist session failed to be created
`ASDKERRAssistTransportFailure`	`30102`	Network error
`ASDKERRAssistSessionInProgress`	`30103`	There is already a session in use
`ASDKERRAssistConsumerDocument ShareFailedNotScreenSharing`	`40101`	Attempt to share a screen when screen sharing is not active
`ASDKAssistSupportEnded`	`50101`	The `endSupport` method was explicitly called to end co-browsing (as opposed to co-browsing being terminated in any other way).

Alerts and System Dialog Boxes

Due to limitations imposed by iOS, Remote Expert Mobile cannot replicate any iOS-generated dialog boxes, such as Alert boxes, the iOS keyboard, or menus generated by HTML (for example, the popup menu generated by the HTML `<select>` element), to the agent. Instead, consider alternative implementations, such as JavaScript and CSS.

Accepting Self-Signed Certificates

By default, self-signed security certificates are rejected by the iOS SDK. If you wish to accept self signed certificates, set the `acceptSelfSignedCerts` configuration parameter to `@YES` (see **Session Configuration on page 48**).

We recommend that you restrict this mode to debug builds. By default, XCode uses the debug mode for a local run, but uses the release mode for anything pushed to the App Store or an enterprise server. Doing this automatically restricts acceptance of self-signed certificates:

```
NSMutableDictionary *config = [[NSMutableDictionary alloc] init];
 ⋮
#ifdef DEBUG
   config[@"acceptSelfSignedCerts"] = @YES;
#endif
 ⋮
[AssistSDK startSupport:server supportParameters:config];
```

Password Fields

When entering a password into a text field on an iOS device, it briefly displays the character that has been entered before masking it. As a user's device screen is being replicated and displayed to an agent, the agent may be able to see the password as it is being entered. Consequently, we recommend that you mask fields that can contain sensitive information using the built-in masking capabilities provided by the Remote Expert Mobile SDK (see **Excluding Elements from Co-browsing on page 64**).

IPv6 Support

With the release of iOS 9, Apple have made IPv6 support mandatory for App Store submissions. OS X 10.11 (El Capitan) provides the ability to create an IPv6 only Wi-Fi hotspot, which mobile devices can connect to during testing.

The Apple Developer document, UnderstandingandPreparingfortheIPv6Transition, is a useful resource explaining how to set up such a network, and things to consider when ensuring your application operates in an IPv6-only network .

Use the following steps to test a Remote Expert Mobile-enabled application in an IPv6-enabled network:

1. Set up an IPv6-only Wi-Fi hotspot using the built-in utility, as described above.
2. Connect to this network from a computer, and obtain the IPv6 address of your Remote Expert Mobile cluster by running the following command:
   - Linux: `ping6 FQDN-address-of-cluster`
   - Windows: `ping -6 FQDN-address-of-cluster`
3. Connect your Apple mobile device to the hotspot.
4. Have your application pass in an IPv6 address or URL (as returned from `ping6`), or the FQDN of your Remote Expert Mobile cluster, as the `server` parameter of your call to `startSupport`.

Note An IPv6 server address or URL passed as a `server` argument must be surrounded by `[ ]` regardless of whether the server is a URL, or just an address, or whether a port is specified. For example `[fe80::7aca:39ff:feb4:1002]:8080`. This does not apply when using an FQDN.

CHAPTER 7

Remote Expert Mobile—CSDK for Android (Java)

This is an overview of the tasks and development required to integrate RE Mobile with a pre-existing web application using the CSDK for Android. Developers can easily embed voice, video and or Expert Assist sessions in their website or web application with Java.

Integration with an Existing Application

You can integrate Remote Expert Mobile with an existing Android application. You need to create an Android project in whatever way your development environment specifies, and put any existing code into it. For simplicity, the examples in the text assume that the project was empty, but a similar approach works if you want to add the Remote Expert Mobile SDK to an existing Android application.

Adding the SDK Libraries and Assets

Once you have an Android project, with its standard directory structure, you need to add the appropriate files from the Remote Expert Mobile Android SDK zip file (`expert_assist_android_SDK-11.6.1.10000-7-ES3.zip`, which contains the following component parts that developers need to integrate into their application in order to use Remote Expert Mobile:

Component	Description
`assets`	A folder containing assets needed by the SDK. Copy the contents of this folder into the matching `assets` directory of the application
`libs`	The artifacts needed to integrate an application with Remote Expert Mobile. Copy the `assist-android-sdk1.x.x.jar` file, `fusionclient-android-sdkx.x.x.jar` file, and `armeabi-v7a` folder into the application's `libs` build directory.
`res`	A folder containing resources needed by the SDK for the user interface. Copy the contents of this folder into the matching `res` directory of the application build directory.

AndroidManifest Entries

The Remote Expert Mobile SDK requires entries within the application's `AndroidManifest.xml`. You need to add these if they are not there already:

- Under the root `<manifest>` element, add the following lines to enable the corresponding features:

```
<uses-feature android:name="android.feature.CAMERA"
   android:required="true" android:glEsVersion="0x00020000"/>
<uses-feature android:name="android.hardware.camera.autofocus"/>
```

- Again under the root `<manifest>` element, add the following entries to enable these permissions:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_
STORAGE"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_
SETTINGS"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
```

- Under the `<application>` element, add the following service entry:

```
<service android:name="com.alicecallsbob.assist.sdk.core.AssistService"/>
```

- You *may* want to add, under the `<application>` element, the following meta-data entries:

```
<meta-data android:name="assist-host" android:value=""/>
<meta-data android:name="assist-port" android:value="8080"/>
<meta-data android:name="assist-secure" android:value="false"/>
```

  These entries act as defaults for values which Remote Expert Mobile needs to be present, but which you normally set using the application's shared preferences, or by providing a `serverHost` value to the configuration object when starting a Remote Expert Mobile session (see **The AssistConfigBuilder on page 77**).

The Application Object

In order to take advantage of Remote Expert Mobile functions, subclass the `AssistApplicationImpl` object, and make that class the main application class of your application:

```
package com.example;
import com.alicecallsbob.assist.sdk.core;

public class SampleAssistApplication extends AssistApplicationImpl
{
    ⋮
}
```

And in the `AndroidManifest.xml`:

```
<application android:label="@string/assist_app"
   android:name="com.example.SampleAssistApplication"
   android:icon="@drawable/launcher_icon"
   android:theme="@android:style/Theme.Holo.Light">
    ⋮
</application>
```

Implementing AssistApplication

If you are integrating Remote Expert Mobile with an existing application, you may have an existing application class which extends an `Application` class from another library. In this case you will not be able to extend `AssistApplicationImpl`. In these circumstances, you should make your existing application class implement the `AssistApplication` interface. To follow this pattern, the application should construct and terminate an `AssistCoreImpl` object within the lifecycle of your application:

```
import android.app.Application;
import com.alicecallsbob.assist.sdk.core.AssistApplication;
import com.alicecallsbob.assist.sdk.core.AssistCore;
import com.alicecallsbob.assist.sdk.core.AssistCoreImpl;

public class SampleAssistApplication
   extends Application implements AssistApplication
{
   private AssistCore assistCore;

   @Override
   public void onCreate()
   {
      super.onCreate();
      assistCore = new AssistCoreImpl(this);
   }

   @Override
   public void onTerminate()
   {
      assistCore.terminate();
      assistCore = null;
   }
```

```
    @Override
    public AssistCore getAssistCore()
    {
       return assistCore;
    }
}
```

Starting a Support Session

The application typically starts a Remote Expert Mobile support session in one of its `Activity` classes; often, though not necessarily, its main `Activity`. This may be in response to user interaction (user presses a **Help** button), or automatically by the program in certain circumstances (the user has entered syntactically incorrect data into a form field, for instance). You do this using the `Assist.startSupport` static call.

When you call `startSupport`, you provide an `AssistConfig` configuration item, the `Application` object for the current application, and an `AssistListener` object which receives notification when the support session ends.

The `AssistListener` interface contains a single non-deprecated method, `onSupportEnded`, which offers an opportunity for the application to tidy up its user interface and generally do whatever house-keeping it requires.

It also contains a deprecated method, `onSupportError` - *we recommend that this should be given no more than a skeleton implementation*; to receive notifications of support errors, implement an `AssistErrorListener`, and include it in the `AssistConfig` object (see **AssistErrorListener on page 84**).

Note

An `AssistListener` implementation ***must*** be supplied, even if there is nothing for `onSupportEnded` to do.

The application's `Application` object must implement `AssistApplication` or extend `AssistApplicationImpl` (see **The Application Object on the previous page**). If it does not, `startSupport` throws an `Exception`.

The `AssistConfig` object supplies the configuration for the Remote Expert Mobile session. The recommended way to construct it is to use the `AssistConfigBuilder` class (see **The AssistConfigBuilder on the facing page**).

```
private class AssistListenerImpl implements AssistListener
{
   void onSupportEnded(boolean endInitiatedLocally)
   {
      if (!endInitiatedLocally)
      {
         // Display "Agent ended session"
      }
   }
   void onSupportError(AssistError errorType, String message)
   {
   }
}
 ⋮
AssistConfig config = new AssistConfigBuilder(getApplicationContext()).
   setAgentName("agent1").
   setServerHost("127.0.0.1").build();
```

```
Assist.startSupport(config, getApplication(), new AssistListenerImpl());
```

The AssistConfigBuilder

The `com.alicecallsbob.assist.sdk.config.impl.AssistConfigBuilder` supports the following properties:

Method	Default Value/Behavior	Description
`setServerHost`	Value of the `assist-host` key in the application's shared preferences or meta data.	Base URL of Remote Expert Mobile server and REM Advanced SDK Gateway, including only the hostname, or IP address. If not set using `setServerHost`, this value must be set in the application's shared preferences or metadata for Remote Expert Mobile to work.
`setAgentName`	`agent1`	Username of agent (or agent group), if that agent (or agent group) is local to the web gateway; otherwise full SIP URI of agent or queue
`setConnectSecurely`	`false`	Whether to connect using HTTPS (rather than HTTP) to the supplied server host
`setMediaMode`	voice and video to and from agent	Set whether to show video, and from which parties. The `AssistMediaMode` class provides the enum to control this parameter. Values include the following: ■ voice/video in both directions ■ voice in both directions, but video from the agent only ■ voice only.
`setCorrelationId`	generated	ID of the co-browsing session
`setServerPort`	`8080`	Port for the http(s) connection to Remote Expert Mobile server
`setSessionToken`		Web gateway session token (if required)
`setHostnameVerifier`		A `HostnameVerifier` object to validate connections made by the SDK to secure URLs (including pushed content, such as documents)
`setTrustManager`		A `TrustMananger` object to validate connections made by the SDK to secure URLs (including pushed content, such as documents)
`setSharedDocumentAuthHandler`	Shows an `AlertDialog` prompting the user to choose whether to accept or reject a doc-	Supply an `AssistSharedDocumentAuth` implementation that is notified when the agent shares a document with the consumer. Allows the application to accept or reject a document.

Method	Default Value/Behavior	Description
	ument	
`setSharedDocumentReceivedListener`		Supply an `AssistSharedDocumentReceivedListener` implementation that is notified about successful and unsuccessful incoming shared documents. Also handles document closure notifications, and provides an API for programmatically closing currently opened documents.
`setSharedDocumentViewConstraints`	All margins `0`	An `AssistSharedDocumentViewConstraints` object with the desired left, top, right and bottom margins within which shared documents will be displayed or constrained.
`setErrorListener`		An implementation of `AssistErrorListener` used for listening to errors from the Remote Expert Mobile SDK. See **AssistErrorListener on page 84** for error codes.
`setAssistAnnotationListener`		Supply an `AssistAnnotationListener` implementation that is notified when an annotation is drawn on the consumer application, and when annotations are cleared. It also allows the client to receive an `AnnotationContext` that they can use to manually clear annotations.
`setCobrowseAuthListener`	Shows an `AlertDialog` prompting the user to choose whether to accept or reject the co browse request	Supply an `AssistCobrowseAuthListener` implementation that is notified when the agent requests co-browse with the consumer. Allows the application to accept or reject the co-browse request.
`setUUI`		The value set is placed in the SIP User to User Interface header. **Note**: The UUI can only be used when **Anonymous Consumer Access** is set to `trusted` mode. See the *Remote Expert Mobile Design Guide, Release 11.6 (1)* for further information.
`setCobrowseListener`		An implementation of the `AssistCobrowseListener` that overrides the default visual indicator when the application screen is shared.
`setAgentCobrowseListener`		An implementation of the `AssistAgentCobrowseListener` which receives notifications when an agent joins, leaves, or requests to join the session.

Method	Default Value/Behavior	Description
`setConnectionStatusListener`		An implementation of the `ConnectionStatusListener` which will receive notifications about the state of the connection to the Remote Expert Mobile server.
`setConnectionProfile`		A `ConnectionProfile` object, containing retry intervals and timeouts for establishing and maintaining the connection to the Remote Expert Mobile server.

Using UUI

The value specified in the `uui` element is placed in the SIP `User-to-User` Interface header exactly as it is passed. The application must ensure that the encoding is correct.

```
String uui = "5465737420555549"; // Hex encoded String "Test UUI"
AssistConfigBuilder builder = new AssistConfigBuiler();
builder.setAgentName("agent1");
builder.setUUI(uui); // Set other desired properties on builder
Assist.startSupport(builder.build(), getApplication(), assistListener);
```

Note

This starts a call with voice and video. For co-browse only sessions, see Co-browse only (with code) on page 10 or Co-browse only (with correlation ID) on page 9.

Starting a Co-browse only Session

If the built application does not require voice and video functionality, you can exclude the following libraries:

- `fusionclient-android-sdk2.x.x.jar` file
- `andarmeabi-v7a`

This has the advantage of reducing the size of the application.

To create a Remote Expert Mobile session without voice and video, provide a *correlation ID* (using `setCorrelationId`), but without specifying an agent name (using `setAgentName(..)`) when configuring the `AssistConfigBuilder`.

The application can then call `Assist.startSupport` as described in Starting a Support Session on page 76. It is not necessary to call `setMediaMode()` on the `AssistConfigBuilder` when carrying out a Remote Expert Mobile-only session. It *will* be necessary to explicitly call `AssistSDK.endSupport` to end the session.

Note

The correlation ID needs to be known to both parties in the call, and needs to be unique enough that the same correlation ID is not used by two support calls at the same time. The application developer must decide the mechanism by which this happens, but possible ways are for both parties to calculate a value from data about the call known to both of them, or that one side calculates it and communicates it to the other on the existing communication channel. There is also a REST service provided by Remote Expert Mobile which will create a correlation ID and associate it with a short code; see Co-browse only (with correlation ID) on page 9 and Co-browse only (with code) on page 10.

Note The Remote Expert Mobile SDK throws an `Exception` if a voice and video call is attempted without the `fusionclient-android-sdk2.x.x.jar` and `armeabi-v7a` libraries present in the application, and will show the following message in the logs:

RuntimeException—CSDK library not found, cannot create call. Please check the classpath and/or refer to the documentation for more information.

To remedy this, either add the libraries to the application, or create a Remote Expert Mobile session without voice and video.

Escalating an existing call to include co-browse

In most cases, the application calls `startSupport` with an agent name, and allows Remote Expert Mobile to set up a call to the agent and implicitly add Remote Expert Mobile support to that call. However, there may be cases where a call to an agent already exists, and the application needs to add Remote Expert Mobile support capabilities. To do this, you need to supply the *session token* and a *correlation ID* in the `AssistConfig` which you supply to `startSupport`; and the agent needs to connect to the same session. The Remote Expert Mobile server provides some support for doing this:

1. The application connects to a specific URL on the Remote Expert Mobile server, to request a *short code* (error handling omitted):

```
JsonObjectRequest request = new JsonObjectRequest(Request.Method.PUT,
   assistServerAddr + "/assistserver/shortcode/create",
   null, new ShortCodeResponseListener(), new ErrorListener());
queue.add(request);
```

   The Volley `RequestQueue` object should already exist.

2. The application uses the short code in another call to a URL on the Remote Expert Mobile server, and receives a JSON object containing a session token and a correlation ID:

```
class ShortCodeResponseListener extends Response.Listener<JSONObject>
{
   public void onResponse(JSONObject response)
   {
      String shortCode = response.getString("sortCode");
      JsonObjectRequest request = new JsonObjectRequest
      (Request.Method.GET,
         assistServerAddr + "/assistserver/shortcode/consumer?appkey=" +
         shortCode,
         null, new SessionResponseListener(), new ErrorListener());
      queue.add(request);
   }
}
```

3. The application passes those values to the `AssistConfigBuilder`, and passes the `AssistConfig` object to `startSupport`:

```
class SessionResponseListener extends Response.Listener<JSONObject>
{
   public void onResponse(JSONObject response)
   {
      String correlationId = response.getString("cid");
      String sessionToken = response.getString("session-token");
      builder.setCorrelationId(correlationId).
         setSessionToken(sessionToken);
       ⋮
      Assist.startSupport(builder.build, getApplication(),
```

```
                new EndSupportListener());
        }
    }
```

More configuration can be set in the `AssistConfigBuilder`. In particular, you may want to set an `AssistCobrowseAuthListener` in order to validate the agent when the session has started and they request to co-browse.

4. The agent uses the same short code to get a JSON object containing the session token and correlation ID, which it then uses to connect to the same Remote Expert Mobile support session (see the *Remote Expert Mobile Agent Console DeveloperGuide, Release 11.6 (1)*). Informing the agent of the short code is a matter for the application. It could be something as simple as having it displayed on the consumer's screen and having the consumer read it to the agent on the existing call (this is how the sample application does it).

Note The short code will expire after 5 minutes, or when it has been used by both agent and consumer to connect to the same session.

Ending the Support Session

The agent may end the support session, or the application may end the session by calling `Assist.endSupport()`. In both cases, the Remote Expert Mobile SDK removes the user interface elements which it added, and calls the `onSupportEnded` notification; the application only needs to restore changes which it itself has made.

Building with ProGuard

The Remote Expert Mobile SDK includes a file, `proguard-project.txt`, that can be used or merged with an existing ProGuard configuration file for an application; copy the `proguard-project.txt` to the root of the application project, alongside the `project.properties` file.

Typically, the `project.properties` of the application should specify both the local and parent ProGuard configuration files. For example:

```
proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt
```

You can modify the `proguard-project.txt` configuration to add additional ProGuard rules as necessary.

Note If you are building with only co-browsing support (because your application does not support voice and video calling; see **Starting a Co-browse only Session on page 79**), you should ignore the CSDK files which would otherwise be referenced in the build.

During a Co-browsing Session

While a co-browsing session is active (after the application has called `startSupport` successfully, and before either it calls `endSupport` or receives the `onSupportEnded` notification to indicate that the agent has ended the support session), the application may:

- Accept an agent into, or expel the agent from, the co-browsing session
- Pause and resume the co-browsing session
- Receive a document from the agent
- Push a document to the agent
- Receive an *annotation* (a piece of text or drawing to show on the device's screen, overlaid on the

application's view) from the agent

- Have a form on its screen wholly or partly filled in by the agent

Actions which are initiated by the application (such as pushing a document to the agent) require it to call one of the methods on the `Assist` object.

Actions initiated by the agent (such as annotating the consumer's screen) can in general be allowed to proceed without interference from the application, as the Remote Expert Mobile SDK manages them, overlaying the user's screen with its own user interface where necessary. However, the application can receive notifications of these events by implementing of one of the various `Listener` interfaces (see **Receiving Notifications below**).

Receiving Notifications

You can receive notifications of events of interest on the various `Listener` interfaces which you can set in the `AssistConfig` object which is passed to the `Assist.startSupport` method. In most cases you can ignore these interfaces, but if you need to receive any of these notifications, you need to implement the appropriate interface in some class, and pass an instance of that class in the configuration (see **The AssistConfigBuilder on page 77**).

AssistAgentCobrowseListener

Implementing the `com.alicecallsbob.assist.sdk.core.AssistAgentCobrowseListener` interface enables the application to receive the following notifications:

- `agentJoinedSession(agent)`

  This callback indicates that an agent has answered the support call and joined the support session; this occurs before the agent either requests or initiates co-browsing. The callback allows the developer to pre-approve the agent into the co-browse, before the agent makes the request.

- `agentRequestedCobrowse(agent)`

  This callback notifies the application that the agent has specifically requested to co-browse. There is no specific requirement for the application to allow or disallow co-browsing at this point, but it is an obvious point to do so.

- `agentJoinedCobrowse(agent)`

  This callback occurs when the agent joins the co-browse session.

- `agentLeftCobrowse(agent)`

  This callback occurs when the agent leaves the co-browse session, and can no longer see the consumer’s screen. Leaving the co-browse also resets the agent's co-browse permission; the agent may subsequently request co-browse access again.

- `agentLeftSession(agent)`

  This callback notifies the application that the agent has left the overall support session.

The default implementation displays a dialog box on the consumer's device, asking whether to allow co-browsing or not. If the consumer allows co-browsing, it allows any agent into the co-browsing session whenever they request it. Implementing this interface can give the application more control over which agents are allowed into the co-browsing session, and when.

AssistSharedDocumentReceivedListener

The `com.alicecallsbob.assist.sdk.config.impl.AssistSharedDocumentReceivedListener` interface allows the application to receive notifications when the agent sends a document to the consumer. It contains three methods:

- `onDocumentReceived(AssistSharedDocument)`

  Indicates that the document has been received and displayed. The `com.alicecallsbob.assist.sdk.core.AssistSharedDocument` object gives access to the document's ID, its metadata (extra information supplied by the agent in the form of a `String`), and a `close` method.

- `onError(AssistSharedDocument)`

  Indicates that the SDK cannot display the document; by default, it displays a error overlay to inform the user.

- `onDocumentClosed(AssistSharedDocumentClosedEvent)`

  Indicates that a document has been closed. You can retrieve the `AssistSharedDocuement` object, and an indication of the source of the document, using methods on the `com.alicecallsbob.assist.sdk.core.AssistSharedDocumentClosedEvent`.

By default, the Remote Expert Mobile SDK displays the document. Acceptable document types are: PDF, and the image formats GIF, PNG, and JPG/JPEG.

AssistAnnotationListener

The `com.alicecallsbob.assist.sdk.config.impl.AssistAnnotationListener` allows the client application to receive notifications relating to annotations sent by the agent:

- `newAnnotation(annotation, sourceName)`

  Notification that an annotation has been received from an agent.

- `annotationsCleared()`

  Called when an agent clears the annotations.

- `annotationContextInitialised(AnnotationContext)`

  Receives the `com.alicecallsbob.assist.sdk.core.AnnotationContext` for a support session when it is ready to be used.

- `annotationContextEnded(AnnotationContext)`

  Notification that the `AnnotationContext` is no longer valid and should not be used.

You can implement this interface in order to control of the display and clearing of annotations which the application receives from the agent. See **Annotations on page 88** for details of how you might use these notifications.

AssistCobrowseListener

The `com.alicecallsbob.assist.sdk.config.impl.AssistCobrowseListener` contains two notifications, `onCobrowseActive` and `onCobrowseInactive`, which allow the application to receive notifications when co-browsing starts and stops. You might implement this to customize the display of a notification to the user. See **Co-browsing Visual Indicator on page 92** for details.

AssistCobrowseAuthListener

The `com.alicecallsbob.assist.sdk.config.impl.AssistCobrowseAuthListener` interface gives the application notification when an agent has requested to co-browse the consumer's screen. It has a single method:

- `onCobrowseRequested(AssistCobrowseAuthEvent)`

  The agent has requested a co-browse with the consumer. The `com.alicecallsbob.assist.sdk.config.impl.AssistCobrowseAuthEvent` object has two methods, `acceptCobrowse`, and `rejectCobrowse`. The listener implementation should call one of them.

By default, Remote Expert Mobile displays a dialog box when an agent requests co-browsing, allowing the user to accept or reject the co-browse. You might implement this interface to avoid asking the user each time, and instead use the application's shared preferences setting to decide whether to accept or reject the co-browse.

ConnectionStatusListener

The `com.alicecallsbob.assist.sdk.transportConnectionStatusListener` has the following methods, which supply notifications about the connection status of the WebSocket connection to the Remote Expert Mobile server:

- `onConnect()`

  Received when the WebSocket becomes connected to the Remote Expert Mobile server, so that the application can send and receive messages.

Note The application does not need to wait to receive this notification before it can use the Remote Expert Mobile API methods.

- `onDisconnect(reason, connector)`

  Received when the WebSocket connection to the Remote Expert Mobile server has been lost, or has failed to reconnect. The `reason` is a `com.alicecallsbob.assist.sdk.transport.websocket.WebSocketException`.

- `onTerminated(reason)`

  Received when the WebSocket connection with the Remote Expert Mobile server has terminated, and there will be no more reconnection attempts. The `reason` is a `WebSocketException`.

- `willRetry(retryInSeconds, retryAttemptNumber, maximumRetryAttempts, connector)`

  `retryInSeconds` is a `double`, while `retryAttemptNumber` and `maximumRetryAttempts` are `ints`. The notification indicates that the Remote Expert Mobile SDK is preparing to retry an automatic reconnection attempt (controlled by the `ConnectionProfile` object which may be supplied in the configuration passed to `startSupport` - see **Connection Configuration on page 93**).

There are two reasons for implementing your own `ConnectionStatusListener`:

- to log the connection attempts and other messages to somewhere other than the default Android log.
- to implement your own reconnection strategy which is not covered by setting the `ConnectionProfile`. See **Using the ConnectionStatusListener interface on page 94** for details of how you might use an implementation of this interface.

AssistErrorListener

The `com.alicecallsbob.assist.sdk.core.AssistErrorListener` has a single notification method, `onSupportError(AssistErrorEvent)`, which the application receives when an error occurs. The `com.alicecallsbob.assist.sdk.core.ErrorEvent` has a single method, `getError()`, which returns one of the possible `AssistErrorListener.AssistError` error codes:

Code	Meaning
`CALL_ERROR`	Error in the call. This may be due to no media, a network failure, or some other reason
`CALL_SESSION_NOT_STARTED`	Attempt to co-browse before the session has started
`CALL_DIAL_FAILED`	Unable to even try to dial (for example, the number to dial might be invalid)

Code	Meaning
CALL_CREATION_FAILED	Unable to create a Call object
CALL_TIMEOUT	The dial operation failed because the callee did not answer the call inside the network's timeout
CALL_FAILED	Tried to share a document when co-browsing is not active. Tried to allow or disallow co-browsing for an agent when support is not active
CALLEE_NOT_FOUND	The dialed number is formally valid, but could not be found. It might not be in use, or not accessible on the network.
SESSION_IN_PROGRESS	There is already a session in use
SESSION_CREATION_FAILURE	Error connecting to Remote Expert Mobile server

Allow and Disallow Co-browse for an Agent

You may wish to remove a specific agent from the co-browsing session. To do this, call:

```
Assist.disallowCobrowseForAgent(agent);
```

passing in the Agent object received in one of the notifications on the AssistAgentCobrowseListener interface (see **AssistAgentCobrowseListener on page 82**).

If the agent is already in the co-browse session, they are removed from it; if they are not in the co-browse session, they will not be admitted until the application calls:

```
Assist.allowCobrowseForAgent(agent);
```

When the application calls allowCobrowseForAgent, the specified agent joins the co-browse session immediately.

Pausing and Resuming a Co-browse Session

The application can temporarily pause a co-browse session with the agent by calling:

```
Assist.pauseCobrowse();
```

While paused, the connection to the Remote Expert Mobile server remains open, but the co-browse session is disabled, disabling annotations, document sharing, and so on as a consequence. When the application wishes to resume the co-browsing session, it should call:

```
Assist.resumeCobrowse();
```

When the application pauses a co-browse, Remote Expert Mobile notifies the Agent Console, which can present a notification or message to the agent to indicate what has happened.

Sharing Documents

As well as receiving shared documents from the agent (see **AssistSharedDocumentReceivedListener on page 82**), applications can use the Remote Expert Mobile SDK to share documents with the agent during a co-browsing session. Acceptable documents are PDFs and images.

Documents shared in this way are represented visually in the same way as documents that are pushed from the agent: PDFs are full screen, and images are in windows that can be dragged, re-sized, or moved.

Note

Sharing a document does not actually send the document to the agent, but simply displays the document on the local device, so that both the consumer and the agent can see and co-browse the document.

There are two methods exposed by the `AssistDocumentShareManager` object to handle document sharing:

- `void shareDocumentUrl(URL, contentType, listener)`

  This method allows sharing a document given its URL. Typically, this would be used to share a document on another machine.

- `void shareDocument(stream, contentType, listener)`

  This method allows sharing an `InputStream` containing the document's data. Typically, this would be used to share a document on the local machine.

To obtain the `AssistDocumentShareManager`, call `Assist.getDocumentShareManager()`.

In each method, the application must supply the content type of the document. This is typically `application/pdf` in the case of PDFs, or something like `image/jpeg` in the case of images (or similar, for example `image/png`).

The application can also provide a `AssistSharedDocumentListener` implementation to receive feedback on the state of the document; this argument can be `null` if not required. The listener provides the following callbacks, similar to that of the `AssistAgentSharedDocumentReceivedListener` API:

- `onOpened(AssistSharedDocument)`

  Called when the document is successfully opened, passing in a document object.

- `onError(AssistSharedDocument)`

  Called when an error overlay is displayed by Remote Expert Mobile due to the failure to successfully display the shared document for some reason.

- `onDocumentClosed(AssistSharedDocumentClosedEvent)`

  Called when the document is closed (the event object provides information about the source of the event, such as `CONSUMER`, `AGENT`, or `END_SUPPORT` to indicate where the source of the close originated from).

Note The `AssistSharedDocumentReceivedListener` and the `AssistSharedDocumentListener` interfaces share a superclass, and therefore have some method signatures in common. This means that all documents represented by an `AssistSharedDocument`, whether pushed by the agent to the consumer or pushed by the consumer to the agent, have a `getId()` method. IDs only exist for documents pushed by the agent and received by the `AssistSharedDocumentReceivedListener` interface; all IDs returned by `getId()` on `AssistSharedDocument` objects received in this interface are `-1`.

An `AssistNotInCobrowseException` is thrown when calling these methods if the consumer is not currently in a co-browse session - you cannot share a document with an agent if you are not actually co-browsing.

An `AssistInvalidContentTypeException` is thrown if the app provides an invalid content type (that is, it is not a PDF or image).

Note This is simply a check on the `contentType` argument provided to the API - if they provide a valid content type but invalid data (for example, they try to load a .DOCX file, but specify the content type as a PDF) then the methods will not throw this exception; instead, an error overlay will be displayed and the `onError` method is fired instead.

The major distinction between errors that are presented by throwing an `Exception` or the `onError` callback on the listener, is that the `Exceptions` handle anything that can be verified as demonstrably wrong synchronously (such as not being in a co-browse); `onError` is for errors that occur during the asynchronous loading of the document (for example, if they provide seemingly valid arguments, such as a well-formed URL, but it turns out that URL returns a 404 ‘Not found’ error message, then Remote Expert Mobile can not report that synchronously).

Example:

```
class DocListener extends AssistSharedDocumentListener
{
   @Override
   public void onOpened(AssistSharedDocument document)
   {
   }
   @Override
   public void onError(AssistSharedDocument document)
   {
      Toast.makeText(this, "Error opening document", Toat.LENGTH_SHORT).show();
   }
   @Override
   public void onDocumentClosed(AssistSharedDocumentClosedEvent event)
   {
      if (event.getSource() == AssistSharedDocumentClosedSource.AGENT)
      {
         Toast.makeText(this, "Agent closed the document", Toast.LENGTH_
         SHORT).show();
      }
   }
}


try
{
   AssistDocumentShareManager sm = Assist.getDocumentShareManager();
   FileInputStream fis = new FileInputStream("document.pdf");
   sm.shareDocument(fis, "application/pdf", new DocListener());
}
catch (AssistNotInCobrowseException e)
{
   Toast.makeText(this,
   "Error: can't share document when not in cobrowse", Toast.LENGTH_SHORT).show
   ();
}
catch (AssistInvalidContentTypeException e)
{
   Toast.makeText(this,
   "Error: document must be an image or pdf", Toast.LENGTH_SHORT).show();
}
```

Disabling document close buttons

To disable the close buttons in shared document views in the Remote Expert Mobile SDK, edit the following files in the layout resource directory accompanying the SDK, to remove the View UI component which has the `assist_doc_close` ID:

- `assist_closeable_document.xml`
- `assist_closeable_popup.xml`
- `assist_closeable_single_page_document.xml`
- `assist_error_popup.xml`

For instance, in the `assist_closable_document.xml`, remove or comment out the `Button` element which has the `assist_doc_close` ID:

```
<com.alicecallsbob.assist.sdk.overlay.view.impl.AssistRelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@color/document_popup_background">
   <FrameLayout
      android:id="@+id/assist_popup_content"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
   <!--
   <Button
      android:id="@+id/assist_doc_close"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/close_popup_text"
      android:textColor="#FFFFFF"
      android:layout_alignTop="@id/assist_popup_content"
      android:layout_alignRight="@id/assist_popup_content"/>
   -->
</com.alicecallsbob.assist.sdk.overlay.view.impl.AssistRelativeLayout>
```

and similarly for the other files.

Annotations

By default, the Remote Expert Mobile SDK displays any annotations which the application receives on an overlay, so that the consumer can see them together with their own screen. Normally an application needs to do nothing further, but if it needs to receive notifications when an annotation arrives, it can implement the `AssistAnnotationListener` interface, and supply the implementation in the `AssistConfig` object (see **The AssistConfigBuilder on page 77**).

The `AssistAnnotationListener` offers:

- Notification of new annotations received
- Notification of annotations cleared
- An `AnnotationContext` for each support session, which can be used to manually clear annotations, or to turn automatic clearing on or off.

Notification of new annotations

Implement the following method on the `AssistAnnotationListener`:

```
@Override
public void newAnnotation(Annotation annotation, String sourceName)
{
    ⋮
}
```

The received `Annotation` object contains the following methods:

- `Path getPath()`

  This is the standard Android library `Path` object that is used to draw the new `Annotation`
- `int getStrokeColour()`

- `int getStrokeWidth()`
- `float getStrokeOpacity()`

The `sourceName` is the name of the source that created the annotation, for example the agent's name.

Notification of annotations cleared

Implement the following method on the `AssistAnnotationListener`:

```
@Override
public void annotationsCleared()
{
    ⋮
}
```

This method is called every time the annotations are cleared on the screen, regardless of how it is triggered. For example, the method is called if the agent clears annotations, or if the application calls the `AnnotationContext.clearAnnotations` method (see **Annotation Context below**).

Annotation Context

You receive an `AnnotationContext` in the `annotationContextInitialised` and `annotationContextEnded` methods. You can use the `AnnotationContext` to take control of clearing annotations manually. Remote Expert Mobile creates a new context for each support session, lasting as long as the support session does. The `AnnotationContext` throws an `IllegalStateException` if the application calls any of its methods after it receives the `annotationContextEnded` callback.

The application can turn automatic clearing of annotations off or on by calling the `setClearAnnotationsOnScreenChange` method, passing `false` to turn them off and `true` to turn them on. Automatic clearing can be set at any time during the support session. Typically, applications set it to `false` to enable other views to draw over the support screen without clearing the annotations:

```
private AnnotationContext annotationContext;

@Override
public void annotationContextInitialised(AnnotationContext annotationContext)
{
   // turn off automatic annotation clearing
   annotationContext.setClearAnnotationsOnScreenChange(false);
   // store AnnotationContext to be able to clear annotations later
   this.annotationContext = annotationContext;
}

@Override
public void annotationContextEnded(AnnotationContext annotationContext)
{
   // discard the annotationContext at this point
   if (annotationContext.equals(this.annotationContext))
   {
      this.annotationContext = null;
   }
}

public void clearAnnotations()
{
   if (annotationContext != null)
   {
```

```
      annotationContext.clearAnnotations();
   }
}
```

Example: Placing the agent's name at the end of the annotation

This example describes how to display a label at the end of an annotation, showing the name of the agent who made it.

- Implement the `annotationContextInitialised` method to disable automatic clearing of annotations, and `annotationsCleared` and `annotationContextEnded` to do nothing.
- Implement `newAnnotation` to get the path information from the annotation:

  ```
  public void newAnnotation(Annotation annotation, String sourceName)
  {
     Path path = annotation.getPath();
      ⋮
  }
  ```

  You can obtain the `Path` using `getPath()`; you can then get the end point and tangent of the annotation from the `Path` object. You can use the end point and tangent to place a label at the end of the line and facing in the direction of the annotation.
- Use the standard Android library class `PathMeasure` to obtain the end point and tangent of the annotation `Path`:

  ```
  final PathMeasure pm = new PathMeasure(path, false);
  final float length = pm.getLength();
  final float[] endPoint = new float[2];
  final float[] tangent = new float[2];
  pm.getPosTan(length, endPoint, tangent);
  ```

  The call to `getPosTan` causes the `endPoint[]` and `tangent[]` to be populated with the end point X and Y co-ordinates and tangent of the `Path` at the end of the annotation:

```
private Bitmap annotationBitmap;

@Override
public void annotationContextInitialised(AnnotationContext annotationContext)
{
   annotationContext.setClearAnnotationsOnScreenChange(false);
}

@Override
public void annotationContextEnded(AnnotationContext annotationContext)
{
}

@Override
public void annotationsCleared()
{
}

@Override
public void newAnnotation(Annotation annotation, String sourceName)
{
   Path path = annotation.getPath();
   PathMeasure pm = new PathMeasure(path, false);
```

```
    float length = pm.getLength();
    float[] endPoint = new float[2];
    float[] tangent = new float[2];
    pm.getPosTan(length, endPoint, tangent);

    Paint paint = new Paint();
    paint.setColor(annotation.getStrokeColour());
    paint.setStrokeWidth(annotation.getStrokeWidth());
    paint.setStrokeOpacity(annotation.getStrokeOpacity());

    annotationBitmap = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(annotationBitmap);
    canvas.drawPath(path, paint);
    canvas.drawPosText(sourceName, endPoint, paint);
}
```

The `annotationBitmap` now has the annotation and the agent's name, and is available for use outside the listener (if a getter is provided). The `strokeColour`, `strokeWidth`, and `strokeOpacity` found on the `Annotation` object are used to keep the label consistent with the annotation.

Form Filling

One of the main reasons for a consumer to ask for help, or for an agent to request a co-browse, is to enable the agent to help the consumer to complete a form which is displayed on their device. The agent can do this whenever a Remote Expert Mobile co-browse session is active, without further intervention from the application, but there are some constraints on how forms should be designed.

The Remote Expert Mobile SDK automatically detects form fields represented by `EditText`, `RadioButton`, `CheckBox`, or `Spinner View` types, and relays these forms to the agent so that the agent can fill in values for the user. You must provide these form fields with unique labels in the `layout.xml`, in one of the following ways:

- setting the `android:contentDescription` attribute inside the xml element which defines the field itself:

```
<EditText
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:inputType="number"
android:ems="10"
android:contentDescription="Other Loans"
/>
```

- providing a label for the field and including the `android:labelFor` attribute inside the label's xml element:

```
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Other Loans: "
android:layout_gravity="center_vertical"
android:labelFor="@+id/field"
/>
```

For `View` instances where the `inputType` attribute may be specified, the SDK automatically prevents the agent from performing form fill on any password-related `inputType`. These include `textPassword`, `textWebPassword`, `textVisiblePassword`, and `numberPasswordinputTypes`. In order to successfully exclude these fields, Remote Expert Mobile SDK expects them to be the only `inputType` present on the given `View`.

Note While the SDK prevents these fields from being presented to the agent as fillable form data, it does not prevent them from being visible as part of the co-browse. You can hide them by adding the appropriate tag or permission to the `View` (see **Excluding elements from a view below**).

Using a custom Adapter with a Spinner

If you use a custom `Adapter` to provide the data for a `Spinner`, you must implement the `Adapter.getItem(position)` method to return some object. The `Spinner` does not display in the agent's form editor if `getItem(position)` simply returns `null`.

Most applications implement `getItem(position)` to return a `String`; if it returns an object other than a string, you should override that object's `toString()` method to return the string value that you want to be displayed on the device. This ensures that the agent sees the correct values for the spinner in the form editor; if you do not override `toString()`, the agent will see the object ID for each spinner item, rather than the value that is displayed on the device.

Excluding elements from a view

When an agent is co-browsing a form, you may not want the agent to see every control on the form. Some may be irrelevant, and some may be private to the consumer.

You can hide an element by assigning a special tag to it. To exclude an area of the screen from the screenshare, add this tag to the `View` object representing the area:

```
view.setTag(Assist.PRIVATE_VIEW_TAG, true);
```

For more detailed control over element visibility, see **Permissions on page 96**.

Note If you are using a `WebView`, and using HTML elements on that `WebView`, you will not be able to exclude individual HTML elements from being seen by the agent. You can exclude the whole `WebView` using the above techniques, but the methods described in the *Remote Expert Mobile Developer Guide, Release 11.6 (1)* for masking individual elements will not work.

Co-browsing Visual Indicator

The SDK provides a means to customize the visual indication displayed during screen sharing. The default implementation displays a notification icon in the notification bar during the application screen-share, and a *toast* when the sharing begins and ends. To use your own icon, place it in the `res/drawable` folder, and name it `launcher_icon.png`.

If you want to change this behavior, and not simply use your own icon, provide an implementation of the `AssistCobrowseListener` to the `AssistConfigBuilder`:

```
public class CobrowsingListener implements AssistCobrowseListener
{
   @Override
   public void onCobrowseActive()
   {
      ..implementation code
   }
   @Override
   public void onCobrowseInactive()
   {
      ..implementation code
   }
}
```

See the **AssistCobrowseListener on page 83** for details.

WebSocket Reconnection Control

When a co-browse session disconnects due to technical issues, the default behavior is to attempt to reconnect six times at increasing intervals. You can control this behavior by passing in one or both of the following when the application calls `startSupport` (see **The AssistConfigBuilder on page 77**):

- Connection configuration
- An implementation of `ConnectionStatusListener` used to listen for connection events, allowing an application to perform its own reconnection handling, or to inform the user of the status of the current connection.

Connection Configuration

You can set the connection configuration by specifying a `ConnectionProfile` in the `AssistConfig` object which you pass in when the application calls `startSupport`. You can use a builder for the `ConnectionProfile`, just as you can for the `AssistConfig` itself:

```
ConnectionProfile profile = new ConnectionProfile.Builder().
   setInitialConnectionTimeout(30.0).
   setRetryIntervals(5.0, 10.0, 15.0, 20.0).
   setReconnectTimeouts(0.1, 1.0, 10.0).build();
AssistConfig config = new AssistConfigBuilder(getApplicationContext()).
    ⋮
   setConnectionProfile(profile).build();
Assist.startSupport(config, ...);
```

The above example sets a timeout for the initial connection of 30 seconds; if the connection is lost, it will try to reconnect 3 times, at intervals of 5, 10, and 15 seconds. The first reconnection attempt will time out after 0.1 seconds, the second after 1 second, and the third after 10 seconds.

The `ConnectionProfile` settings have default values, so there is no need to specify all (or any) of them. Default values and behavior is as follows:

Item	Default	Description
`initialConnectionTimeout`	`30.0`	A `Double` value which defines the maximum time in seconds the initial WebSocket connection attempt tries to connect.
`retryIntervals`	`[1.0, 2.0, 4.0, 8.0, 16.0,32.0]`	An array of `Double` values which defines the number of automatic reconnection attempts, and the time in seconds between each attempt. If you specify an empty array, Remote Expert Mobile does not try to reconnect automatically.
`reconnectTimeouts`	`[5.0]`	An array of `Double` values which define the maximum time in seconds for each WebSocket reconnection to try to reconnect before failing. The array of values corresponds to the values in `retryIntervals` - as each value in `retryIntervals` is used, the corresponding value is used from this array. If the length of the `retryIntervals` is greater than that of `reconnectionTimeouts`, then Remote Expert Mobile uses the last value of the `reconnectionTimeouts` array for all remaining reconnection attempts.

Note Reconnection applies only to the case where Remote Expert Mobile loses an existing connection; if the initial connection attempt fails, Remote Expert Mobile does not retry automatically.

Using the ConnectionStatusListener interface

If the default reconnection behavior of Remote Expert Mobile is not what you want, even after changing the configuration, you can implement the `ConnectionStatusListener` interface to implement your own reconnection logic, and include it in the `AssistConfig` object which you pass to `startSupport` - see **Starting a Support Session on page 76** and **ConnectionStatusListener on page 84** for details.

Note If you do not specify a `ConnectionProfile` in the `AssistConfig`, Remote Expert Mobile will use its default reconnection behavior; if you specify `retryIntervals` and `reconnectionTimeouts` in the `ConnectionProfile`, Remote Expert Mobile will use its default reconnection behavior using those values. You can turn off the default reconnection behavior, and take full control of reconnection, by specifying an empty list for `retryIntervals`.

When implementing your own reconnection logic, the most important notifications you will receive are `onDisconnect` (called whenever the connection is lost) and `willRetry` (called when automatic reconnection is occurring, and there are more reconnection attempts to come). Both these methods include a `Connector` object in their arguments. You can use the `Connector` object to make a reconnection attempt, or to terminate all reconnection attempts.

Method	Description
`onDisconnect`	Called for the initial WebSocket failure, and for every failed reconnection attempt (including the last one). This method is called regardless of whether `retryIntervals` is specified (that is, whether automatic reconnection is attempted or not). The `Connector` class allows the implementing class to take control of reconnecting, even if reconnection is automatic. For example, an application might decide to give up reconnection attempts even if more automatic reconnection events would subsequently occur, or to try the next reconnection attempt immediately and not wait until the next retry interval has passed.
`willRetry`	Called under the following conditions: ■ when the WebSocket connection is lost; or ■ when a reconnection attempt fails *and* automatic reconnections are occurring (`retryIntervals` is a non-empty array) *and* there are more automatic reconnection attempts to be made. This method is called after the `onDisconnect` method. The application can override reconnection behavior by using the `Connector`. For example, a reconnect attempt could be made straight away.
`onConnect`	Called when a reconnection attempt succeeds. This may be useful to clear an error in the application, or for canceling reconnection attempts if the application is managing its own reconnection.
`onTerminated`	Called under the following conditions: ■ when all reconnection attempts have been made (and failed); or ■ when either the `Connector.disconnect` method or `Assist.endSupport` method is called.

Example—make a reconnection attempt immediately on disconnection:

In this example, the default reconnection behavior has been disabled, and the application reconnection behavior is dependent on the reason for disconnection.

```
void onDisconnect(WebSocketException reason, Connector connector)
{
   switch(reason.getCode())
   {
      case ERR_ONE:
         // Try to reconnect with a timeout of 2 seconds
         connector.reconnect(2.0);
         break;
      case ERR_TWO:
         // Terminate connection
         connector.terminate(new Exception("Cannot reconnect"));
         break;
      default:
         // Try to reconnect with a timeout of 5 seconds
         connector.reconnect(5.0);
         break;
   }
}
```

Example—terminate reconnection attempts in response to user command:

In this example, the default reconnection behavior has not been disabled, but there is a UI control which the user can press to short-circuit the reconnection attempts. If the user has not terminated the connection attempts, automatic reconnection attempts continue.

```
void willRetry(double retryInSeconds, int retryAttemptNumber, int
maximumRetryAttempts, Connector connector)
{
   if (userHasTerminatedConnection)
   {
      connector.terminate(new Exception("User has terminated connection"));
   }
}
```

Cookies

The Remote Expert Mobile SDK uses the following separate cookie handling and storage mechanisms:

- For any HTTP or HTTPS connections originating from `WebView` objects controlled by Remote Expert Mobile, the SDK uses any cookies set using the global `WebView` `CookieManager`. You can access the global `CookieManager` using:

  ```
  android.webkit.CookieManager.getInstance();
  ```

- For HTTP or HTTPS connections originating from the SDK outside a `WebView`, the SDK uses cookies set using the default system `java.net.CookieHandler` object. You can access this using:

  ```
  java.net.CookieHandler.getDefault();
  ```

  You can set a new default `CookieHandler` using:

  ```
  java.net.CookieHandler.setDefault(bespokeCookieHandler);
  ```

Remote Expert Mobile does not set any cookies itself.

Note Remote Expert Mobile does not supply a default `CookieHandler` - this is the responsibility of the application.

Permissions

This section shows how to set and read permissions using the Android SDK. See **Permissions on page 19** for details of how to use permissions to mask elements from an agent's view.

- ## Control element permissions

  Client applications assign permission markers to UI control elements by calling the `Assist.setPermissionForView` method:

  ```
  Assist.setPermissionForView("X", control);
  ```

  where `X` is the *permission marker* to be set on the control.

- ## Agent permissions

  The application can determine an agent's permissions from the `Agent` object which it receives in the methods of the `AssistAgentCobrowseListener` interface (see **AssistAgentCobrowseListener on page 82**). If the application needs to examine this (for instance, to notify the consumer that a particular control will not be visible to the agent), use the `Agent.getViewablePermissions` and `Agent.getInteractivePermissions` methods.

  ```
  Set<String> viewable = agent.getViewablePermissions();
  ```

Internationalization

The strings which the Remote Expert Mobile Android SDK displays are in the `res/values/assist_strings.xml` file. These strings can be internationalized in the usual way for Android applications, by providing a separate set of string resources in a `res/values-<lang>/assist_strings.xml` file, where `<lang>` is the language code.

Integrating an Android Application with the Advanced SDK

When you call the `Assist.startSupport` method and provide a `destination`, but no `correlationId` or `sessionToken`, in the `AssistConfig`, Remote Expert Mobile automatically starts a voice and video call and a co-browse session with the agent, and automatically ends the call when the application calls `Assist.endSupport`. If you want more control over the voice and video call than this, then you should start the call using the REM Advanced SDK:

```
private Call call;
 ⋮
void init()
{
   String sessionToken = getSessionToken();
   UC uc = UCFactory.createUc(context, sessionToken, this);
    ⋮
   uc.startSession();
}

public void onSessionStarted()
{
   Phone phone = uc.getPhone();
    ⋮
   call = phone.createCall(destination, true, true, this);
```

```
    ⋮
}

public void onStatusChanged(Call call, CallStatusInfo status)
{
    ⋮
   if (status == CallStatus.IN_CALL)
   {
      // Escalate call to co-browse
   }
    ⋮
}
 ⋮
```

After the call is connected, you need to escalate that call to co-browse (see **Escalating an existing call to include co-browse on page 80**). In order to control the call while it is in progress, the application saves the `Call` object returned by `phone.createCall`, so that it can use it when needed. For convenience of illustration, the code above implements the `CallListener` and `UCListener` interfaces which receive information on the progress of session creation and call setup.

See the *Remote Expert Mobile Advanced Developer Guide, Release 11.6 (1)*(available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) for more details of how to set up a call, and in particular, of how to obtain a session token to use as the sessionId. It also gives details of what call control features are available and how to use them.

Voice and Video Volume Control

To allow the volume keys on the device to control the volume of the voice and video functionality, use the following call in the `onCreate()` method in any `Activity` or `Fragment` that makes use of the voice and video functionality:

```
setVolumeControlStream(AudioManager.STREAM_VOICE_CALL);
```

System Dialog Boxes

Due to limitations imposed by Android, Remote Expert Mobile cannot show to the agent any dialog boxes generated by Android itself on the consumer device, such as the following items:

- Alert boxes
- Keyboard
- Menus generated by HTML (for example, the popup menu generated by the HTML `<select>` element).

You should consider whether the agent needs to see those elements, and consider alternative implementations on the consumer side, such as JavaScript and CSS.

Password Fields

When entering a password into a text field on an Android device, it briefly displays the character that has been entered before replacing it with a dot. As a user's device screen is being replicated and displayed to an agent, the agent may be able to see the password as it is being entered. Consequently, we recommend that you mask fields that can contain sensitive information using the built-in masking capabilities provided by the Remote Expert Mobile SDK.

See **Excluding elements from a view on page 92**.

HTML 5 Canvas Drawing is not Co-browsed to Agent

Due to a limitation of hardware accelerated rendering of Android Web Views, HTML5 `<canvas>` elements loaded in a `WebView` are not replicated to the agent. However, the canvas content is replicated if a `WebView` is configured to be software rendered.

Before loading the `WebView`, you can set the layer type for the `WebView` to `software`:

```
WebView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

Guidance for hardware and software rendering can be found in the Android documentation.

Media is Transmitted when the App is in the Background

When your application goes into the background, Android continues to transmit any media (voice, video, and screenshare). This is Android defined behavior, but may not be what you want. You may want to pause the co-browse session whenever the application goes into the background.

Other References

Cisco DevNet

https://developer.cisco.com/site/devnet/home/index.gsp

Internet Engineering Task Force (IETF®) Working Group

http://tools.ietf.org/wg/rtcweb/

W3C WebRTC Working Group

http://www.w3.org/2011/04/webrtc/

WebRTC Open Project

http://www.webrtc.org

Acronym List

Item	Description
CODEC	“Coder-decoder" encodes a data stream or signal for transmission and decodes it for playback in voice over IP and video conferencing applications.
CSDK	Remote Expert Mobile Client SDKs. Includes three distinct SDKs for iOS, Android and web/JavaScript developers.
G.711	PCMU/A 8-bit audio codec used for base telephony applications
G.729a	Low-bitrate audio codec for VoIP applications
H.264	Video codec. H.264 is the dominant video compression technology, or codec, in industry that was developed by the International Telecommunications Union (as H.264 and MPEG-4 Part 10, Advanced Video Coding, or AVC). Cisco is open-sourcing its H.264 codec (Open H.264) and providing a binary software module that can be downloaded for free from the Internet. Cisco covers MPEG LA licensing costs for this module.
Opus	Low bit rate, high definition audio codec for VoIP applications. Opus is unmatched for interactive speech and music transmission over the Internet, but is also intended for storage and streaming applications. It is standardized by the Internet Engineering Task Force (IETF) as RFC 6716 which incorporated technology from Skype's SILK codec and Xiph.Org's CELT codec (www.opus-codec.org)
REAS	Remote Expert Mobile Application Server
REMB	Remote Expert Mobile Media Broker
UC	Unified Communications
VP8	Video codec—VP8 is a video compression format owned by Google. Google remains a staunch supporter of VP8 after buying On2 Technologies in 2010; Google then released VP8 software under a BSD-like license, as well as the VP8 bitstream specification under an irrevocable license, and free of royalties. VP8 is roughly equivalent in processor usage, bandwidth, and quality to H.264.
WebRTC	Web Real Time Communications for communications without plug-ins