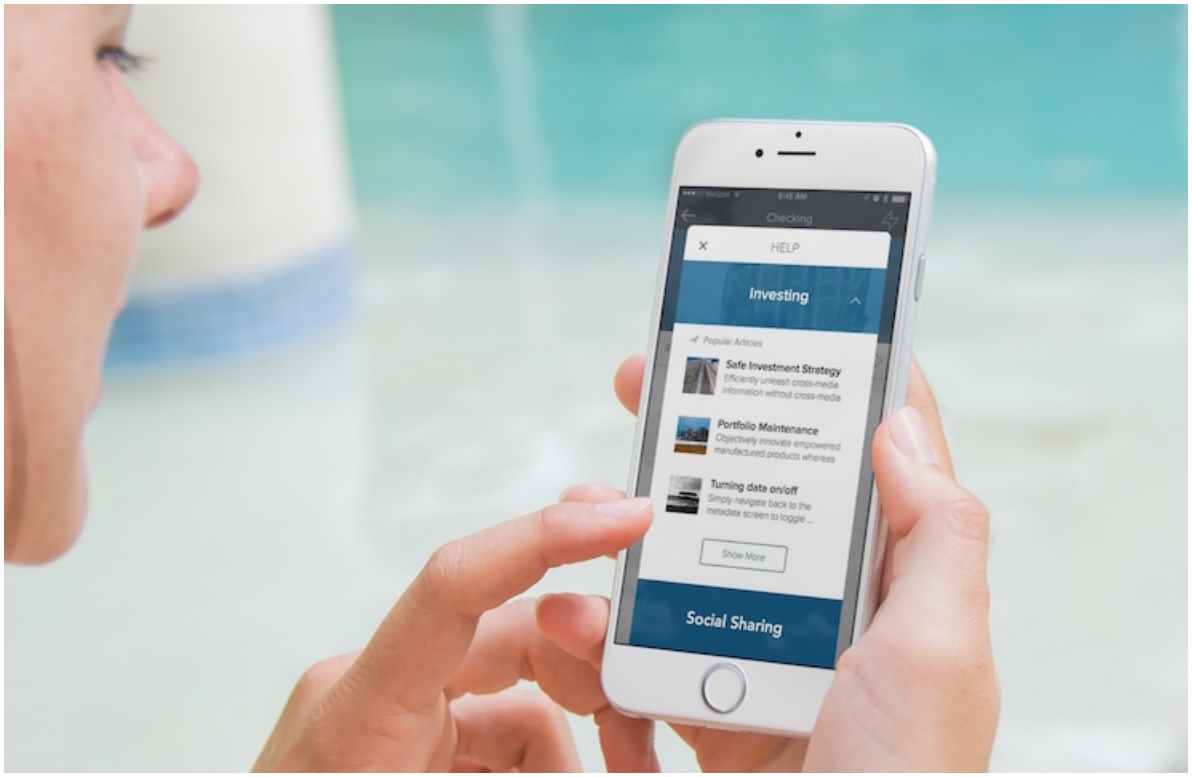
Embedded Service SDK for iOS Developer Guide

Version 220.1



@salesforcedocs

Last updated: September 6, 2019

CONTENTS

EMBEDDED SERVICE SDK FOR IOS DEVELOPER GUIDE

The Embedded Service SDK for Mobile Apps makes it easy to give customers access to powerful features right from within your native app. You can make these Service Cloud features feel organic to your app and have things up and running quickly using this SDK.

May 2019 Release (Version 220.1.0)

This documentation describes the Service SDK, which uses the following components.

Component	Version Number
Chat	3.2.0
Knowledge	3.4.0
Case Management	2.1.1
SOS	3.10.0
ServiceCore (common component used by all features)	4.1.2

Important: The Embedded Service SDK for Mobile Apps was previously named the Snap-ins SDK.

Release Notes
Check out the new features and known issues for the iOS Service SDK.

Service Cloud Setup for the Embedded Service SDK for Mobile Apps
Set up Service Cloud in your org before using the Service SDK.

Embedded Service SDK for Mobile Apps Setup
Set up the SDK to start using Service Cloud features in your mobile app.

iOS Tutorials & Examples
Get going quickly with these short introductory tutorials.

Using Chat with the Service SDK
Add the Chat experience to your mobile app.

Using Knowledge with the Service SDK
Add the Knowledge experience to your mobile app.

Using Case Management with the Service SDK
Add the Case Management experience to your mobile app.

Using SOS with the Service SDK
Add the SOS experience to your mobile app.

SDK Customizations with the Service SDK for iOS
Once you've played around with some of the SDK features, use this section to learn how to customize the Service SDK user interface so that it fits the look and feel of your app. This section also contains instructions for localizing strings in all supported languages.

Release Notes

Check out the new features and known issues for the iOS Service SDK.

To review the latest releases for the Service SDK for iOS, visit github.com/forcedotcom/ServiceSDK-iOS/releases.

Service Cloud Setup for the Embedded Service SDK for Mobile Apps

Set up Service Cloud in your org before using the Service SDK.

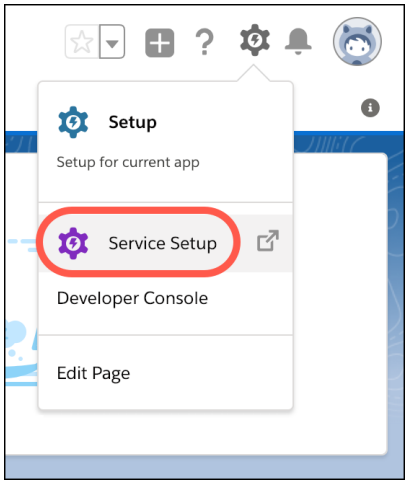
Org Setup for Chat in Lightning Experience with a Guided Flow

Use the guided setup flow in Lightning Experience to add chat to your org.

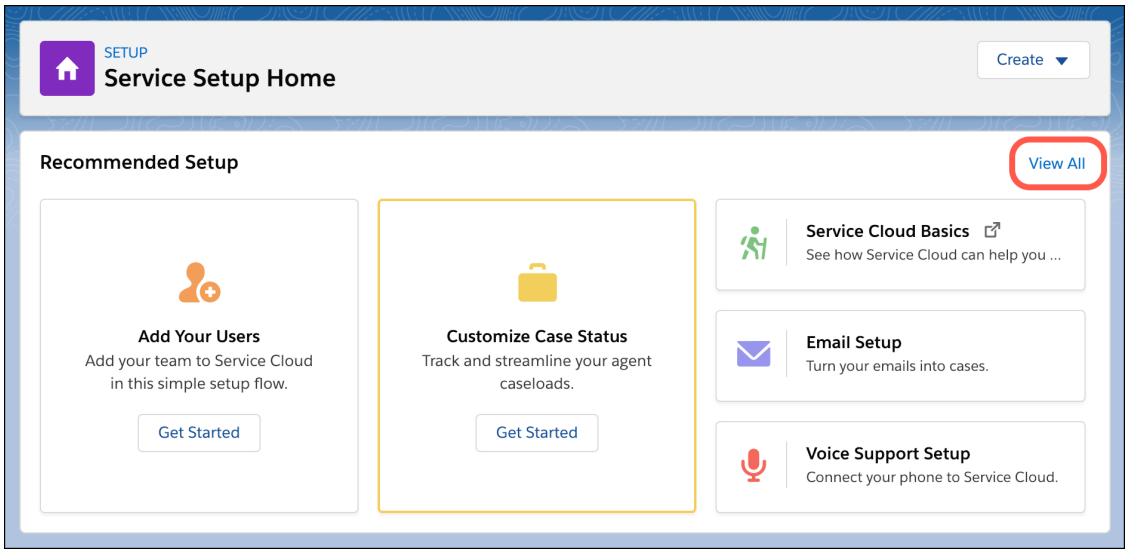
Note: If you're using Salesforce Classic, see Org Setup for Chat in Salesforce Classic.

These instructions walk you through a basic chat setup in Lightning Experience. To learn more about chat, check out the Web Chat Basics Trailhead module.

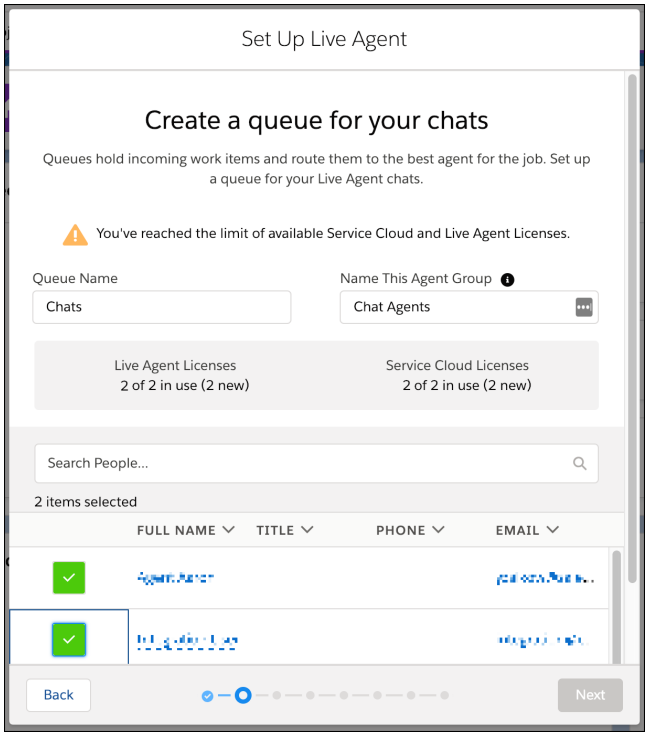
1. Click the Setup gear icon and select **Service Setup**.



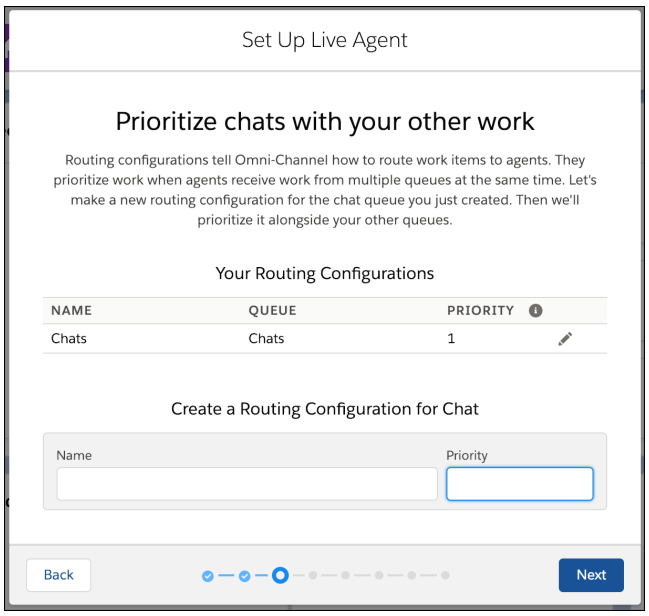
2. Under Recommended Setup, click **View All**.



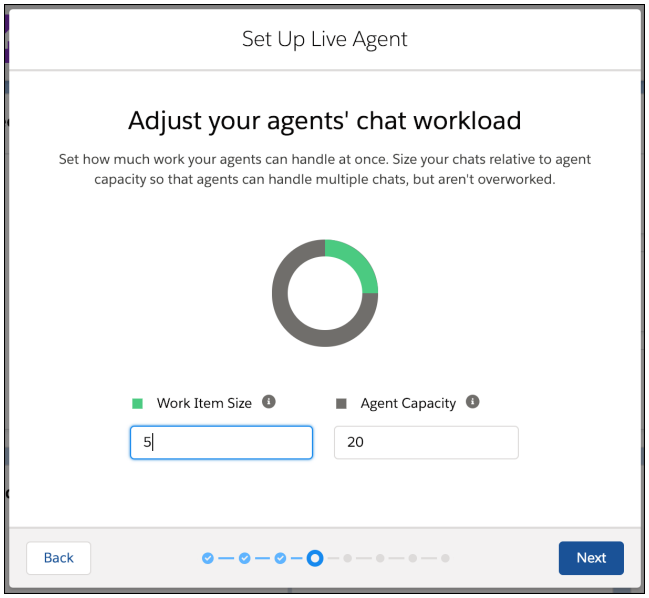
3. In the search box, enter `Chat`, and select **Chat with Customers**.
4. After you read the overview page, click **Start**.
5. Enter the name of your queue (for example, `Chats`) and agent group name (for example, `Chat Agents`). Then select the members for this group and click **Next**.



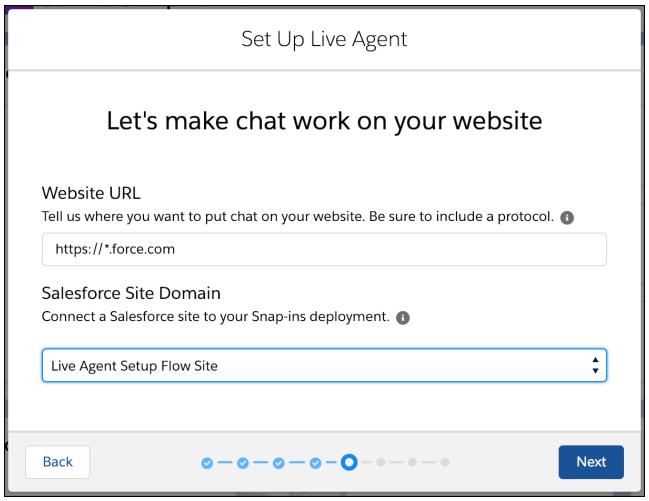
6. If you're asked to prioritize chats with your other work, enter the routing configuration name (for example, `Chats`) and give it a priority (for example, `1`).



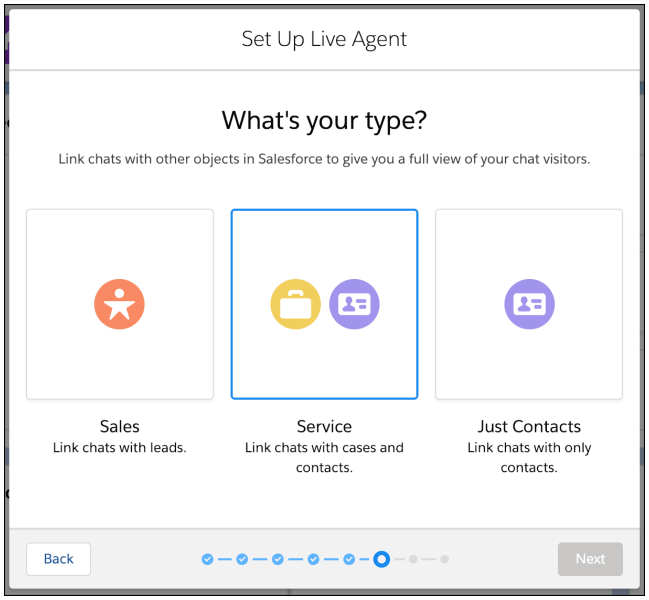
7. (Optional) Adjust the work item size and agent capacity.



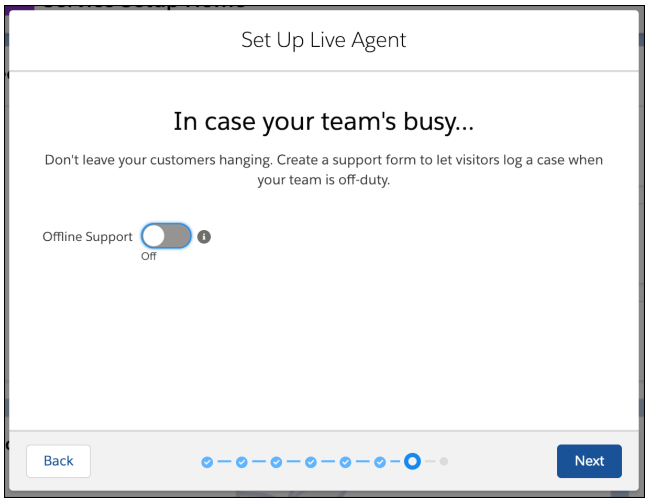
8. For the website URL, enter `https://*.force.com` or the URL for your site. Create or select a Salesforce site.



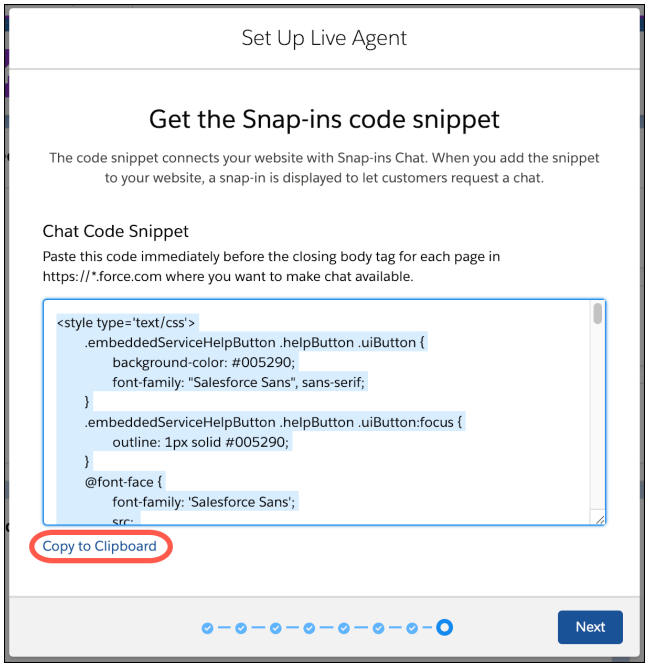
9. For the type of chat, select **Service**.



10. Choose whether you want to provide offline support for customers.



11. Copy the code snippet by clicking **Copy to Clipboard**, and paste it into a text editor. You need to extract a few pieces of information from this code snippet.



12. In the text editor, copy the following configuration information from the `embedded_svc.init` function.

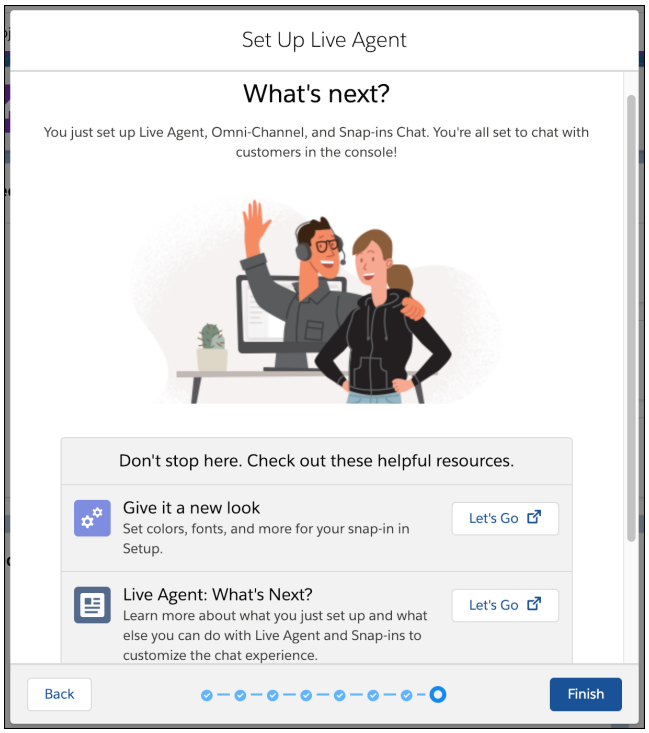
- (1) Chat Endpoint Hostname—This value is the hostname of the `baseLiveAgentURL` property. When copying the hostname, be sure not to include the protocol or the path. For instance, if the value for `baseLiveAgentURL` is `https://d.la2.salesforceliveagent.com/chat`, then the hostname is `d.la2.salesforceliveagent.com`.
- (2) Org ID—If you don't already know this value, it is the fourth argument in the `embedded_svc.init` function call.
- (3) Deployment ID—This value can be found in the `deploymentId` property.
- (4) Button ID—This value can be found in the `buttonId` property.

```
embedded_svc.init(
    'https://brave-bear-ssra2f-dev-ed.my.salesforce.com',
    'https://mainetown-developer-edition.na85.force.com/liveAgentSetupFlow',
    gslbBaseURL,
    '00S1E0000012GrT', 2
    'Chat_Agents',
    {
        baseLiveAgentContentURL: 'https://c.ph2.salesforceliveagent.com/content',
        deploymentId: '5722U000000Dt72', 3
        buttonId: '5731U000000E32v', 4
        baseLiveAgentURL: 'https://d.la2.salesforceliveagent.com/chat',
        eswLiveAgentDevName: 'Chat_Agents', 1
        isOfflineSupportEnabled: false
});
```

Give these four settings to your developer.

Note: If you don't copy this information now, you can copy it later using the instructions in Get Chat Settings from Your Org.

13. Go back to the guided setup flow and click **Finish**.



14. (Optional) If you want to build a chatbot to complement your chat experience, see Einstein Bots in Salesforce Help. In broad strokes, you must enable Einstein Bots, deploy the bot to your channel, and activate the bot. If you want to learn about building a more robust bot, see the Einstein Bots Developer Cookbook.

You're all set! Chat is now set up in your org. You can always fine-tune these settings from **Setup**. To learn more, see Chat in Salesforce Help.

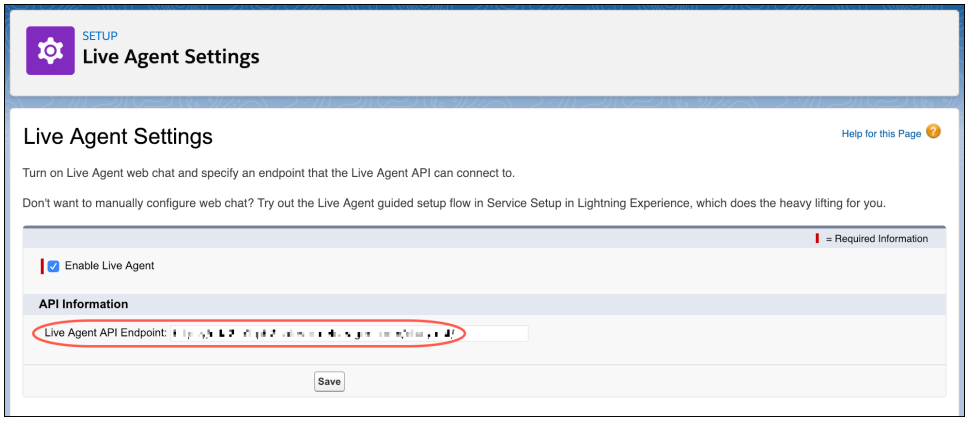
Get Chat Settings from Your Org

After you've set up chat in the console, supply your app developer with four values: the chat endpoint hostname, the organization ID, the deployment ID, and the button ID. You can get this information from your org's setup.

Note: If the endpoint for your server changes (due to an org migration, for example), the SDK automatically reroutes you to the correct server. However, to avoid unnecessary rerouting, you should still update the server endpoint when you notice it has changed inside your org's settings.

Chat Endpoint Hostname

The hostname for the Chat endpoint that your organization has been assigned. To get this value, from Setup, search for **Chat Settings** and copy the hostname from the **API Endpoint**.



Be sure not to include the protocol or the path. For instance, if the API Endpoint is:

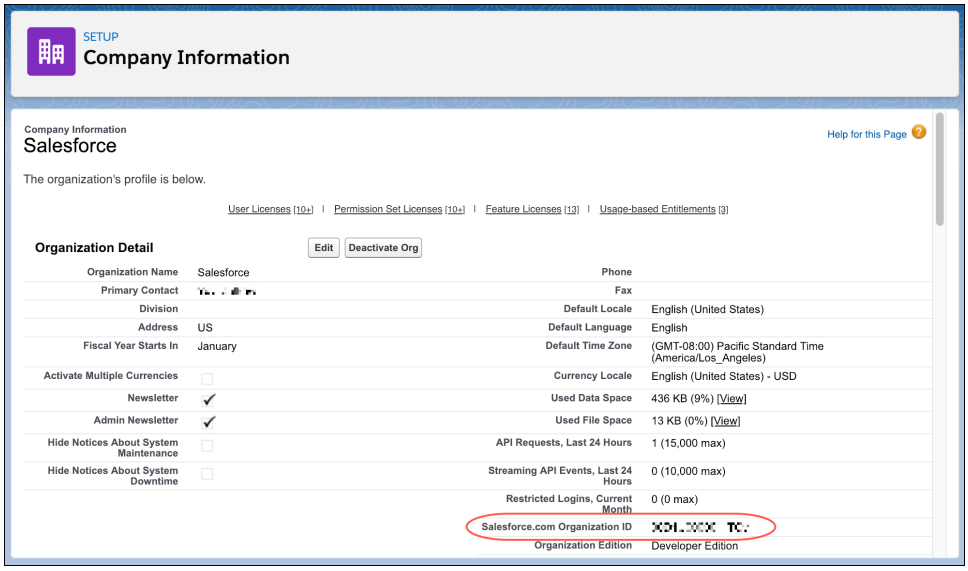
```
https://d.gla5.gus.salesforce.com/chat/rest/
```

The chat endpoint hostname is:

```
d.gla5.gus.salesforce.com
```

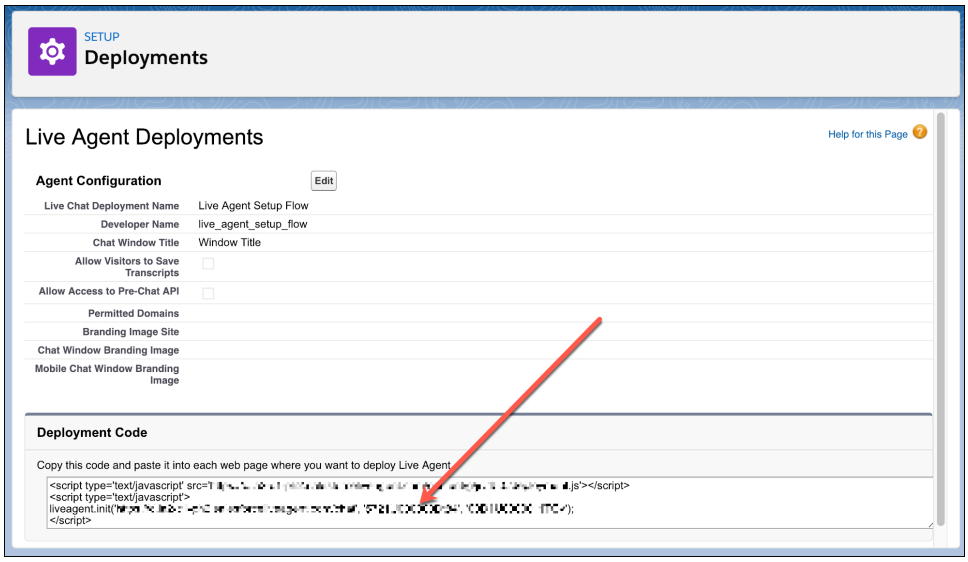
Org ID

The Salesforce org ID. To get this value, from Setup, search for **Company Information** and copy the **Salesforce Organization ID**.



Deployment ID

The unique ID of your Chat deployment. To get this value, from Setup, select **Chat** > **Deployments**. The script at the bottom of the page contains a call to the `liveagent.init` function with the **pod**, the **deploymentId**, and **orgId** as arguments. Copy the **deploymentId** value.



For instance, if the deployment code contains the following information:

```
<script type='text/javascript'
       src='https://d.gla3.gus.salesforce.com/content/g/js/44.0/deployment.js'></script>
<script type='text/javascript'>
liveagent.init('https://d.gla5.gus.salesforce.com/chat', '573B00000005KXz',
'00DB00000003Rxz');
</script>
```

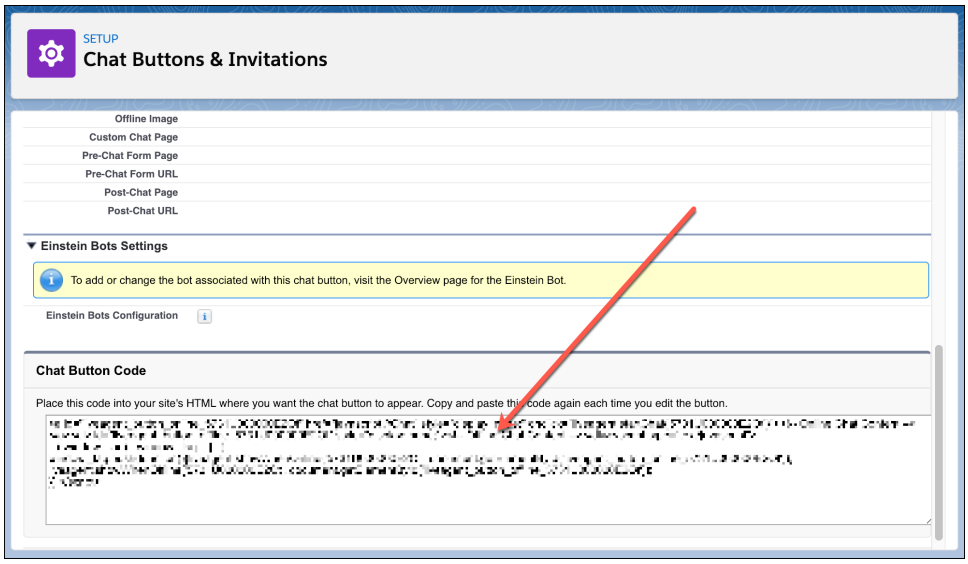
The deployment ID value is:

```
573B00000005KXz
```

Be sure not to use the org ID value (which is also in this deployment code) for the deployment ID.

Button ID

The unique button ID for your chat configuration. To get this value, from Setup, search for **Chat Buttons** and select **Chat Buttons & Invitations**. Copy the `id` for the button from the JavaScript snippet.



For instance, if your chat button code contains the following information:

```
<a id="liveagent_button_online_575C00000004h3m"
   href="javascript://Chat"
   style="display: none;"
   onclick="liveagent.startChat('575C00000004h3m')">
   <!-- Online Chat Content -->
</a>
<div id="liveagent_button_offline_575C00000004h3m"
     style="display: none;">
     <!-- Offline Chat Content -->
</div>
<script type="text/javascript">
  if (!window._laq) { window._laq = []; }
  window._laq.push(function() { liveagent.showWhenOnline('575C00000004h3m',
    document.getElementById('liveagent_button_online_575C00000004h3m'));
    liveagent.showWhenOffline('575C00000004h3m',
    document.getElementById('liveagent_button_offline_575C00000004h3m'));
  });
</script>
```

The button ID value is:

```
575C00000004h3m
```

Be sure to omit the `liveagent_button_online_` text from the ID when using it in the SDK.

Org Setup for Chat in Salesforce Classic

To use Chat in your mobile app, first set up Chat in your org.

Note: This topic shows you how to set up Chat in Salesforce Classic. If you're using Lightning Experience, see Org Setup for Chat in Lightning Experience with a Guided Flow.

1. Create a Chat implementation in Service Cloud, as described in Chat for Administrators (PDF). Your implementation needs a deployment and a chat button.

   Note: By default, a mobile chat session times out around two minutes after you leave the app or lose connectivity. To change this value, update the **Idle Connection Timeout Duration** field when setting up your chat deployment. Keep in mind that the actual timeout on the app can be up to 40 seconds longer than the specified value in this field. See Chat Deployment Settings.

2. (Optional) If you want to use Omni-Channel for routing, configure it as described in Omni-Channel for Administrators (PDF).

   Omni-Channel enables your agents to use the same widget for all real-time routing (for example, Chat, SOS, email, case management). However, you can use Chat without setting up Omni-Channel.

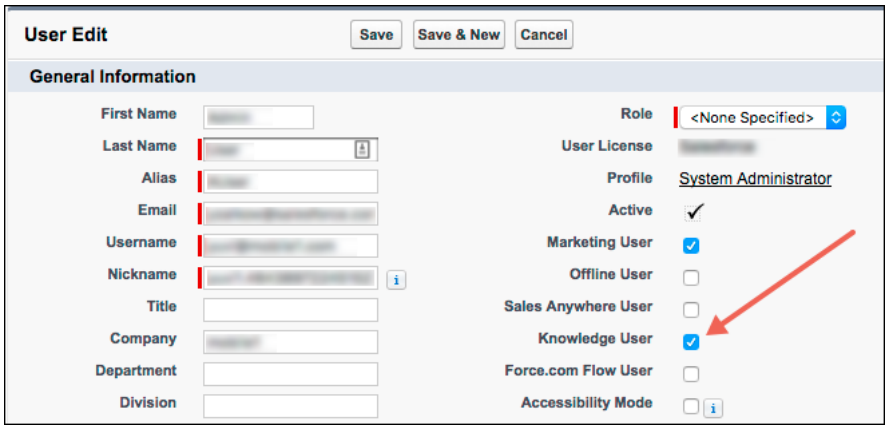
3. (Optional) If you want to build a chatbot to complement your chat experience, see Einstein Bots in Salesforce Help. In broad strokes, you must enable Einstein Bots, deploy the bot to your channel, and activate the bot. If you want to learn about building a more robust bot, see the Einstein Bots Developer Cookbook.

If you have trouble finding the settings that a developer requires to use this feature in the SDK, see Get Chat Settings from Your Org.

Cloud Setup for Knowledge

To use Knowledge in your mobile app, enable it in your org, create knowledge articles, and set up a community.

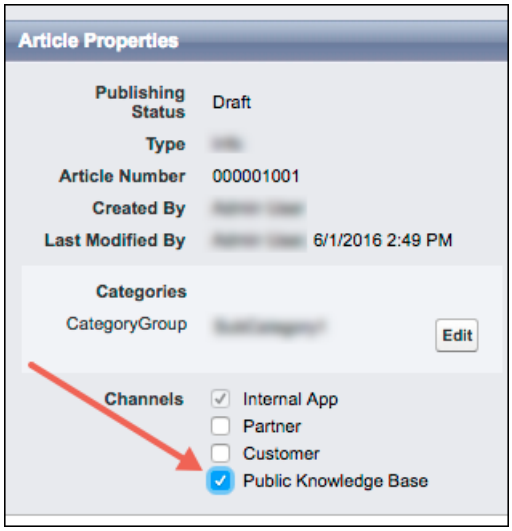
1. Enable Salesforce Knowledge and verify that you have Knowledge licenses. To learn more, see Enable Salesforce Knowledge in Salesforce Help.
2. In the user settings for those users you choose to administer the knowledge base, select **Knowledge User**.



3. Create Knowledge articles. When building out your knowledge base, make sure that you define the article types and associate articles with data categories within a category group.

   To learn more about setting up your Knowledge articles, check out: Salesforce Knowledge Documentation (HTML, PDF).

   When creating articles, ensure that they are accessible to the Public Knowledge Base channel.



4. Create a community. Your Salesforce org must have an available Community or Salesforce site. Your app developer needs the Community URL for the site to use the Knowledge or Case Management feature in the SDK.

   If you've never set up a Community, see Salesforce Communities Overview.

5. To show Knowledge articles from your app, enable guest user access for the **Article Types**, **Categories**, and **Fields** associated with your knowledge articles. Also ensure that **Guest Access to the Support API** is turned on.

   Note: If the Guest user profile isn't set up properly, your Knowledge categories and articles do not appear.

   For step-by-step instructions, see Guest User Access for Your Community.

If you have trouble finding the settings that a developer requires to use this feature in the SDK, see Get Knowledge Settings from Your Org.

Guest User Access for Your Community

Ensure that guest user access is set up correctly for your community. To show Knowledge articles from your app, enable guest user access for the **Article Types**, **Categories**, and **Fields** associated with your knowledge articles. To show Case Publisher, ensure that your **Quick Actions** are accessible.

Get Knowledge Settings from Your Org

After you've set up your knowledge base and your community, supply your app developer with the values for the community URL, data category group, and root data category. You can get this information from your org's setup.

Guest User Access for Your Community

Ensure that guest user access is set up correctly for your community. To show Knowledge articles from your app, enable guest user access for the **Article Types**, **Categories**, and **Fields** associated with your knowledge articles. To show Case Publisher, ensure that your **Quick Actions** are accessible.

These instructions describe how to enable guest user access for either a Community or a Salesforce site.

1. (Community sites only) If you are editing the settings for a **Community**:
   a. From Setup, select **Customize** > **Communities** > **All Communities**.
   b. For your chosen Community, make sure that the Status is "Active".

c. Select the **Workspaces** action.

d. From the Community Workspaces page, select **Administration**.

e. Select **Pages** > **Go to Force.com** to get to the **Site Detail** page.

2. (Salesforce sites only) If you are editing the settings for a **Salesforce site**:

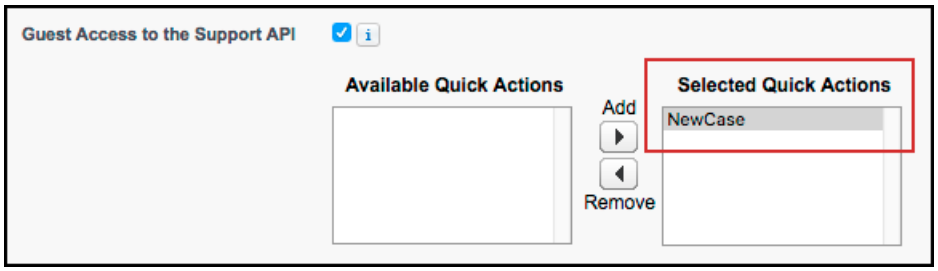
a. From Setup, select **Develop** > **Sites**.

b. Click the **Site Label** for your site to get to the **Site Detail** page.

3. From the **Site Detail** section, click **Edit**.

a. Ensure that **Guest Access to the Support API** is checked.

b. (For Case Management feature only) Ensure that the desired **Quick Actions** are selected. The global quick action determines which fields display when creating a case.



c. Click **Save**.

4. (For Knowledge feature only) From the **Site Detail** section, click **Public Access Settings**. This action displays the settings for the Guest user profile in your org.

a. Verify that the user has read access to the Article Type from the **Article Type Permissions** section.

b. Verify that the user has read access to the fields in the Article Type from the **Field-Level Security** section.

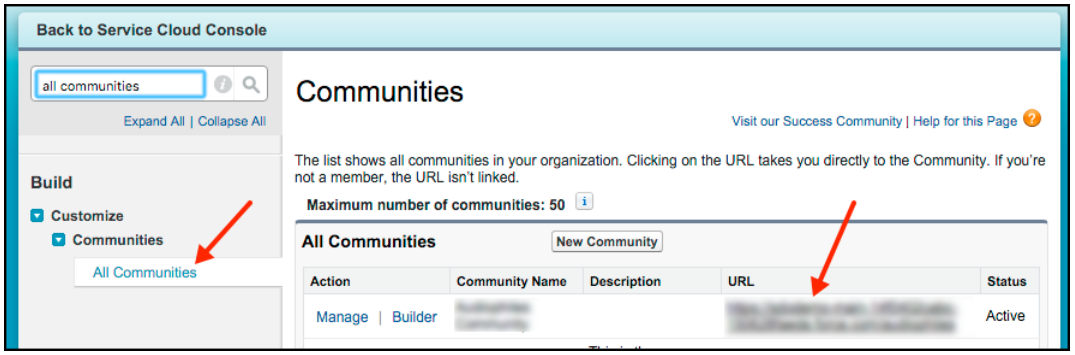
c. Verify that the user has visibility to the categories from the **Category Group Visibility Settings** section.

Get Knowledge Settings from Your Org

After you've set up your knowledge base and your community, supply your app developer with the values for the community URL, data category group, and root data category. You can get this information from your org's setup.

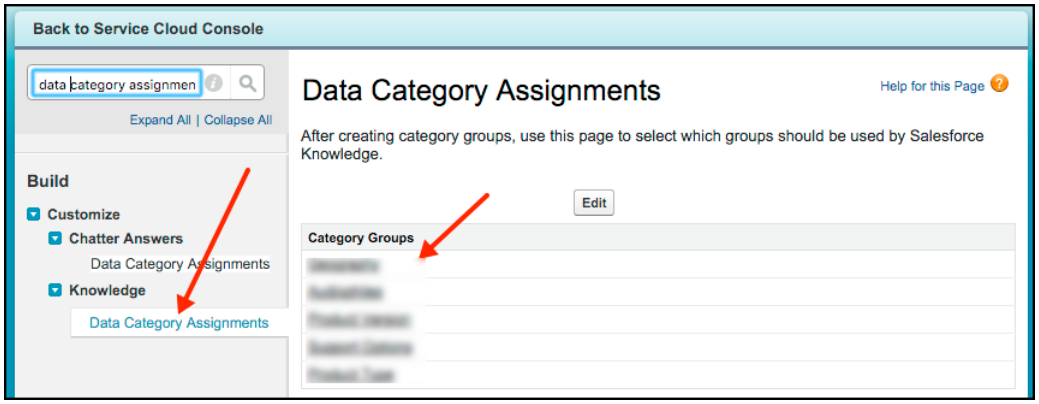
Community URL

From Setup, search for **All Communities**, and copy the URL for the desired community.



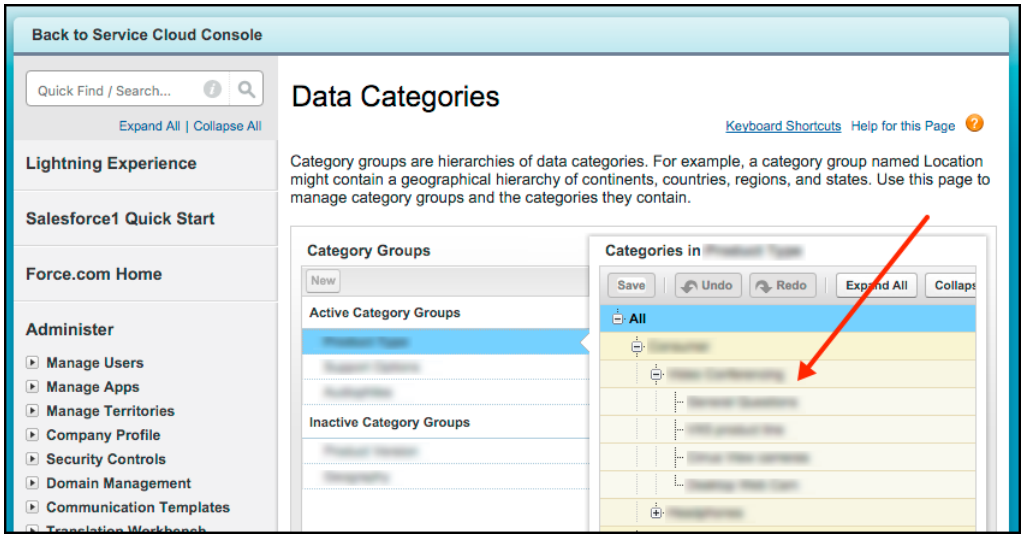
Data Category Group

From Setup, search for **Data Category Assignments** inside the Knowledge section, and copy the name of the desired data category group.



Data Category

From Setup, search for **Data Category Assignments** inside the Knowledge section, select the data category group, and copy the name for the desired root data category.



Cloud Setup for Case Management

To use Case Management in your mobile app, set up a community and create a quick action.

1. Create a community. Your Salesforce org must have an available Community or Salesforce site. Your app developer needs the Community URL for the site to use the Knowledge or Case Management feature in the SDK.

   If you've never set up a Community, see Salesforce Communities Overview.

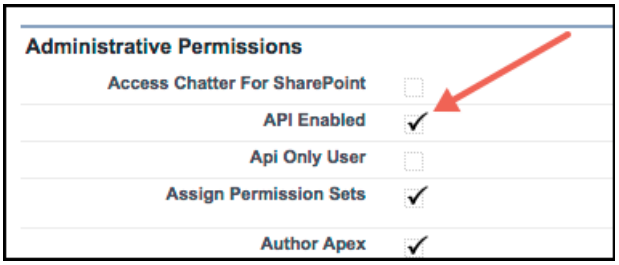
2. When setting up the site, add the **Quick Actions** that you'd like to use in your app for the Case Management functionality. You must specify a quick action to use Case Management. The global quick action determines which fields display when creating a case. To learn more about quick actions, see Create Global Quick Actions in Salesforce Help. Also ensure that **Guest Access to the Support API** is turned on.

   For step-by-step instructions, see Guest User Access for Your Community.

   Note: Be sure that your global action is accessible to the Guest user profile. Also note that the case publisher screen does not respect field-level security for guest users. If you want to specify different security levels for different users, use different quick actions.

3. If you'd like to let authenticated users manage a list of their existing cases, you need to perform a few additional setup steps.

   a. Make sure that the **User Profile** for the authenticated users has **API Enabled** checked. For an overview on user profiles, see Profiles in Salesforce Help.



   b. You'll need a list view for your cases in Service Cloud. To learn more about creating views, see Create a List View in Salesforce Help. Supply the **Case List Unique Name** for this view to your app developer.

      Note: If you use the built-in `My Cases` list view, keep in mind that it is filtered by the `Contact` field for community users and it is filtered by the `Created By` field for other user profiles. If you want a different behavior, create a new list view.

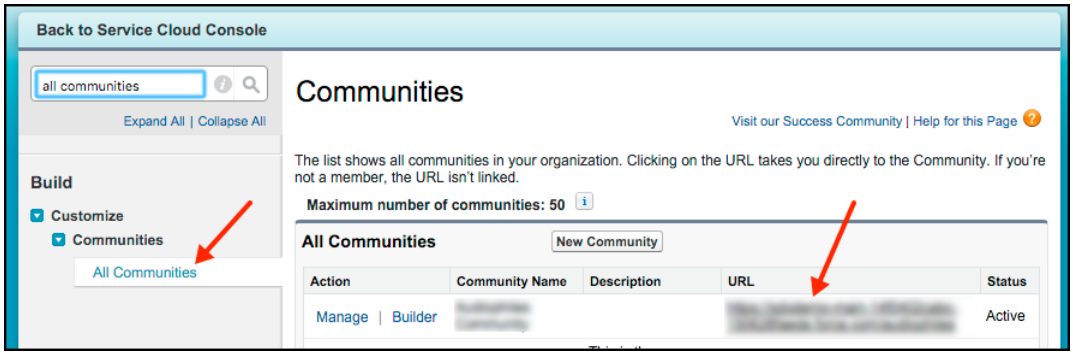
If you have trouble finding the settings that a developer requires to use this feature in the SDK, see Get Case Management Settings from Your Org.

Get Case Management Settings from Your Org

After you've configured your org, supply your app developer with the values for the community URL, the global action, and the case list. You can get this information from your org's setup.

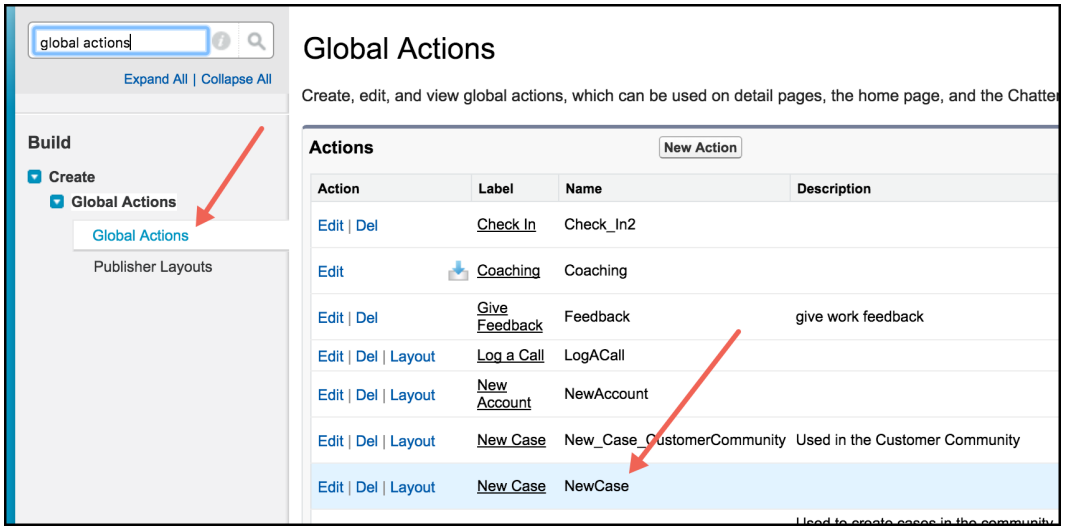
Community URL

From Setup, search for **All Communities**, and copy the URL for the desired community.



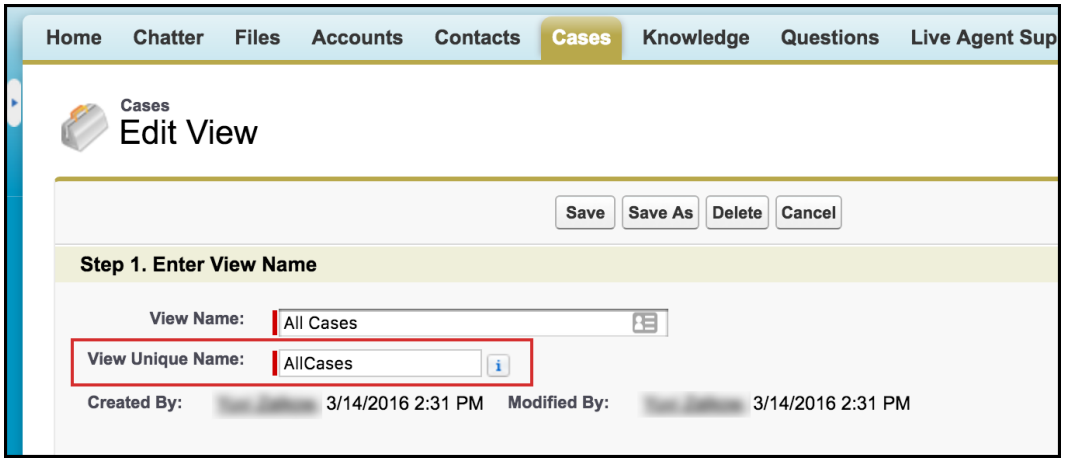
Global Action

From Setup, search for **Global Actions**, and copy the name of the desired quick action.



Case List Unique Name (for Authenticated Users Only)

To get this case list value, access the **Cases** tab in your org, pick the desired **View**, select **Go!** to see that view, and then select **Edit** to edit the view. From the edit window, you can see the **View Unique Name**. Use this value when you specify the `caseListName` in the SDK.



Console Setup for SOS

To use SOS in your mobile app, set up Omni-Channel and SOS for your console.

Note: If you intend to provide real-time support using *both* Chat and SOS, make sure that your agents know to go `Offline` before switching from the `Online` state of one feature to the `Online` state of the other.

To get SOS set up in your console, see Add SOS to Your Console. After setting up the SOS console, check out the other topics in this section to fine-tune your SOS configuration.

Add SOS to Your Console

Perform these steps to get SOS into your production environment.

Get SOS Settings from Your Org

After you've set up SOS in the console, supply your app developer with three values: the Chat endpoint, the organization ID, and the deployment ID. You can get this information from your org's setup.

Assign SOS Permissions

To allow an agent to use SOS, verify that the license and permissions settings are correct in Salesforce.

Automatic SOS Case Pop

With auto case pop, Service Cloud automatically creates a case when a new SOS session starts. Creating a case at the start of a session requires a trigger, a Visualforce page, and changes to the SOS session page layout.

Listen for SOS Console Events

Listen for SOS events from the Salesforce console to log activity, debug issues, and perform quality-of-service (QoS) analysis.

Record SOS Sessions

Enable SOS session recording to assure quality and let agents refer to session recordings.

SOS Reference ID

Provide an ID to give to support when there are issues with a session.

Multiple SOS Queues

Implement multiple SOS queues to route requests to specific agents or give specific requests a higher priority.

Add SOS to Your Console

Perform these steps to get SOS into your production environment.

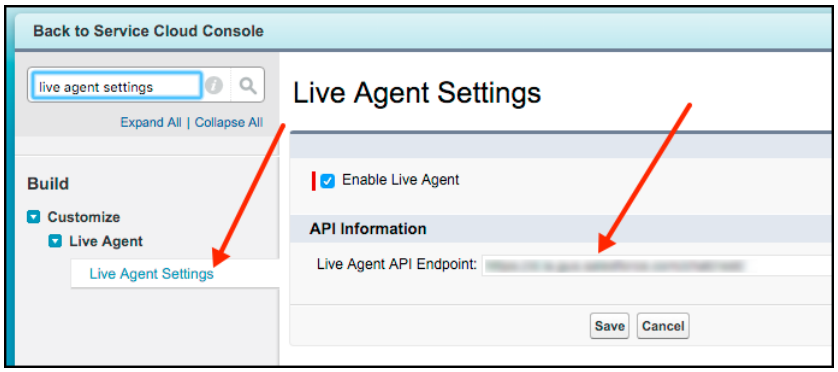
1. Configure Omni-Channel, as described in Omni-Channel for Administrators (PDF).
2. Set up SOS in Service Cloud, as described in Set Up SOS Video Chat and Screen-Sharing.
3. Be sure that you've assigned agent permissions to users, as described in Assign SOS Permissions.
4. Perform any additional customizations specified in Console Setup for SOS.

Get SOS Settings from Your Org

After you've set up SOS in the console, supply your app developer with three values: the Chat endpoint, the organization ID, and the deployment ID. You can get this information from your org's setup.

pod

The hostname for the Chat endpoint that your organization has been assigned. To get this value, from Setup, search for **Chat Settings** and copy the hostname from the **Chat API Endpoint**. Be sure not to include the protocol or the path — use only the hostname.



For instance, if your Chat API Endpoint is:

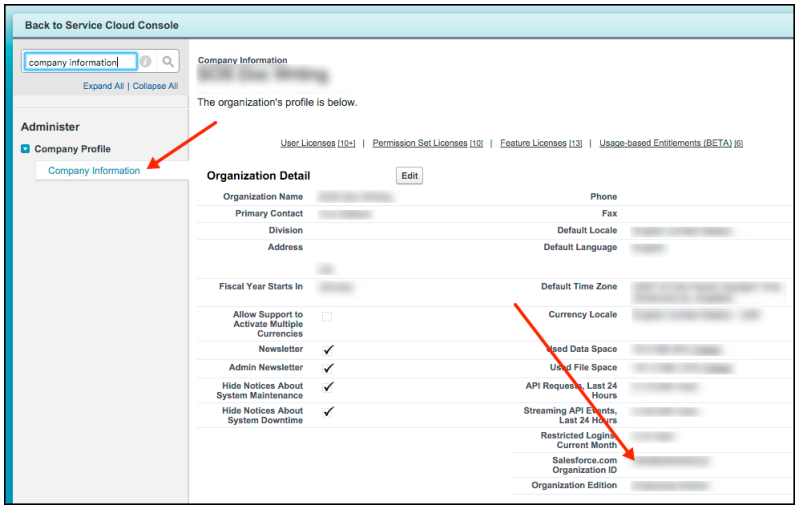
```
https://d.gla5.gus.salesforce.com/chat/rest/
```

Your pod hostname is:

```
d.gla5.gus.salesforce.com
```

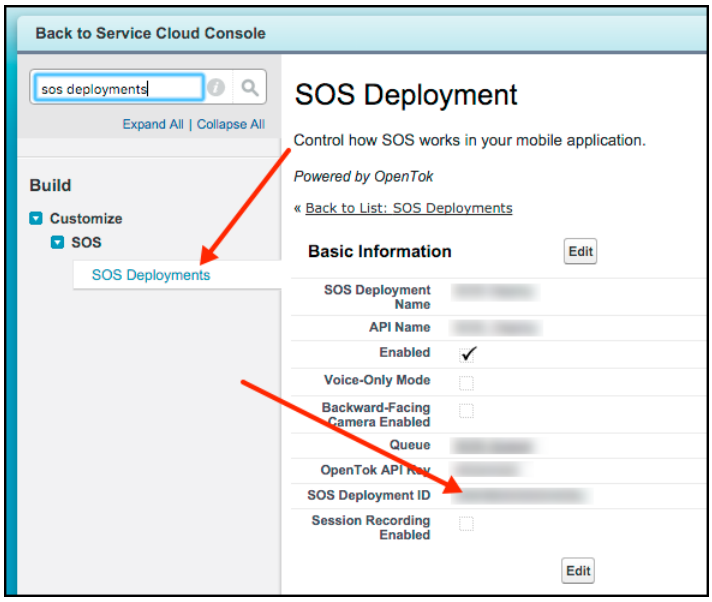
orgId

The Salesforce org ID. To get this value, from Setup, search for **Company Information** and copy the **Salesforce Organization ID**.



deploymentId

The unique ID of your SOS deployment. To get this value, from Setup, search for **SOS Deployments**, click the correct deployment and copy the **Deployment ID**.



Assign SOS Permissions

To allow an agent to use SOS, verify that the license and permissions settings are correct in Salesforce.

1. Assign an SOS user license.

   Assigning a license must be done for every user that requires access to SOS.

   a. From **Setup**, select **Manage Users** > **Users**.

   b. Click the name of the user. (Do not click **Edit**.)

   c. Select **Permission Set License Assignments** and then click **Edit Assignments**.

   d. Enable **SOS User**. If this option is not available, your org has not been assigned any SOS licenses.

   e. Click **Save**.

2. Enable the SOS license.

   Once licenses are assigned to users, enable them using a permission set. We recommend that you have a permission set specifically for SOS, because all users assigned to this permission set must have an SOS license. If you attempt to enable SOS for a permission set which contains users that do not have an SOS license, you'll receive an error.

   a. From **Setup**, select **Manage Users** > **Permission Sets**.

   b. If you do not have a permission set for SOS, click the **New** button. Give it a **Label** and click **Save**.

   c. If you already have a permission set, click the SOS permission set.

   d. Click **App Permissions** and then click the **Edit** button.

   e. Check **Enable** for the **Enable SOS Licenses** checkbox. You'll receive an error if any assigned users do not have the SOS license.

   f. Click **Save**.

3. Enable the service presence status.

You can enable the service presence status using either permission sets or profiles. If the presence status is only being used for SOS, it is easier to enable the presence status through the same permission set that enables the license. Using the same permission set guarantees that all agents who require the presence status have access to it. If the presence status is being used for multiple service channels, it is likely that the same permission set cannot be used, since all members of the permission set would require a SOS license. In this case, you may want to have multiple permissions sets, assign it to a profile, and use some combination of profiles and permission sets.

Service Permission via Permission Sets

a. From **Setup**, select **Manage Users** > **Permission Sets**.

b. Click an existing permission set associated with SOS, or create a new one.

c. Click **Service Presence Statuses Access** and then click the **Edit** button.

d. Add the service presence related to SOS to the **Enabled Service Presence Statuses**.

e. Click **Save**.

f. If necessary, click **Manage Assignments** to add agents to the permission set.

Service Permission via Profile

a. From **Setup**, select **Manage Users** > **Profiles**.

b. Click the name of the profile associated with SOS. (Do not click **Edit**.)

c. Click the **Edit** button for **Enabled Service Presence Status Access**.

d. Add the service presence related to SOS to the **Enabled Service Presence Statuses**.

e. Click **Save**.

4. Add agents to the queue.

All agents must be a member of at least 1 queue. You can determine which queues are used by SOS by looking at the SOS deployments. Agents can be added to a queue individually or in groups. These groups differ depending on the org — groups can be broken into: roles, public groups, partner users, and so on.

a. From **Setup**, select **Manage Users** > **Queues**.

b. Click **Edit** for the desired queue.

c. Scroll to the bottom of the page and find the **Queue Members** section. Add the required members.

d. Click **Save**.

Automatic SOS Case Pop

With auto case pop, Service Cloud automatically creates a case when a new SOS session starts. Creating a case at the start of a session requires a trigger, a Visualforce page, and changes to the SOS session page layout.

1. Create a trigger.

This trigger fires before a new SOS session object saves. The trigger creates a case and adds a reference to the case to the SOS session object. When the case is created, the owner is initially set to "Automated Process". This value changes to the owner of the SOS session object with the Visualforce page specified in the next step.

a. From **Setup**, search for **SOS Sessions**.

b. Select **Triggers** from the **SOS Sessions** section.

c. Click the **New** button.

d. Replace the **Apex Trigger** code with the code below. This code assumes that the email address is sent through the **SOS Custom Data** feature using the `Email__c` API Name. To learn more about custom data in SOS, see Using SOS with the Service SDK. Any data that can be used to identify a contact can be sent instead of the email, as long as the trigger is updated to reflect this information.

```
trigger SOSCreateCaseCustom on SOSSession (before insert) {
  List<SOSSession> sosSess = Trigger.new;
  for (SOSSession s : sosSess) {
    try {
      Case caseToAdd = new Case();
      caseToAdd.Subject = 'SOS Video Chat';
      if (s.ContactId != null) {
        caseToAdd.ContactId = s.ContactId;
      } else {
        List<Contact> contactInfo =
            [SELECT Id from Contact WHERE Email = :s.Email__c];
        if (!contactInfo.isEmpty()) {
          caseToAdd.ContactId = contactInfo[0].Id;
          s.ContactId = contactInfo[0].Id;
        }
      }
      insert caseToAdd; s.CaseId = caseToAdd.Id;
    }
    catch(Exception e){}
  }
}
```

e. Click **Save**.

2. Add a Visualforce page.

This Visualforce page changes the owner of the case and opens the case in a subtab. The page is added to the SOS session page layout in the final step.

a. From **Setup**, search for **Visualforce Pages**.

b. Click the **New** button.

c. Give the Visualforce page a name. For example, "SOS_Open_Case_Custom".

d. Replace the **Visualforce Markup** code with this code:

```
<apex:page sidebar="false" standardStylesheets="false">
  <apex:includeScript value="/soap/ajax/34.0/connection.js"/>
  <apex:includeScript value="/support/console/34.0/integration.js"/>

  <script type='text/javascript'>
    sforce.connection.sessionId = '{!$Api.Session_ID}';

    function escapeSoql (str) {
      return str.replace(/\\/g, '\\\\').replace(/'/g, "\\'");
    }

    document.addEventListener('DOMContentLoaded', function () {
      sforce.console.getEnclosingPrimaryTabObjectId(function(result) {
        if (!result || !result.success) {
          return;
```

```
        }

        var sosSessionId = result.id;
        var query =
            "SELECT CaseId, OwnerId FROM SOSSession WHERE Id = '" +
             escapeSoql(sosSessionId) + "'"
        var queryResult = sforce.connection.query(query);
        var record = queryResult.getArray('records');

        if (!record || !record[0]) {
          console.log('Can not determine session Id');
          return;
        }

        var caseId = record[0].CaseId;
        var ownerId = record[0].OwnerId;

        if (!ownerId) {
          console.log('No owner Id');
          return;
        }

        var caseUpdate = new sforce.SObject("Case");
        caseUpdate.Id = caseId;
        caseUpdate.OwnerId = ownerId;
        result = sforce.connection.update([caseUpdate]);

        if (!result[0].getBoolean("success")) {
          console.log('Unable to set owner', result, caseUpdate);
        }

        sforce.console.getEnclosingPrimaryTabId(function(result) {
          if (!result || !result.success) {
            return;
          }

          var query = "SELECT CaseNumber FROM Case WHERE Id = '" +
                      escapeSoql(caseId) + "'"
          var queryResult = sforce.connection.query(query);
          var record = queryResult.getArray('records');
          var caseNumber = record && record[0] &&
                           record[0].CaseNumber || 'Case';

          sforce.console.openSubtab(result.id, '/'+caseId,
                                    true, caseNumber);
        });
      });
    });
  </script>
</apex:page>
```

e. Click **Save**.

3. Update the SOS session page layout.

Now that the Visualforce page has been created, you can change the page layout of the SOS session. This change hides the Visualforce page in the layout.

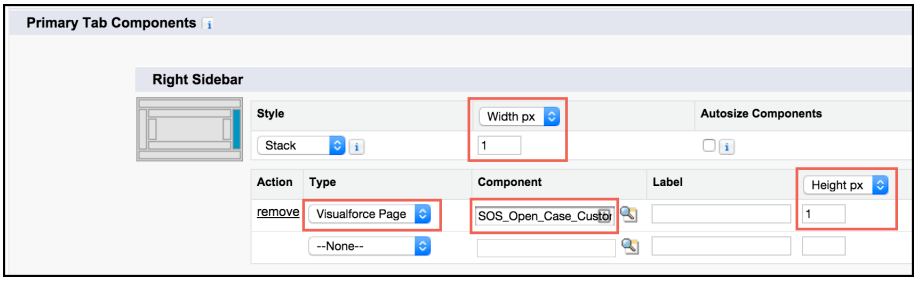
a. From **Setup**, search for **SOS Sessions**.

b. Select **Page Layouts** from the **SOS Sessions** section.

c. Click **Edit** for your active layout (probably **SOS Session Layout**).

d. From the top of the page, select the **Custom Console Components** link.

e. Under the **Primary Tab Components** section, add the following to one of the sidebars:



- Set **Style** to **Stack**.
- Set **Width px** to **1**.
- Set **Height px** to **1**. (Change **Height %** to **Height px** if necessary.)
- Set **Type** to **Visualforce Page**
- Set **Component** to the page created previously (for example, **SOS_Open_Case_Custom**).

f. Click **Save**.

Whenever an agent accepts an incoming call, a case automatically gets created.

Listen for SOS Console Events

Listen for SOS events from the Salesforce console to log activity, debug issues, and perform quality-of-service (QoS) analysis.

To detect events from the Salesforce console, use JavaScript in your Visualforce page. Call the `addEventListener` method, which is documented in the Salesforce Console Developer Guide. The method syntax is:

```
sforce.console.addEventListener(eventType: String, eventListener: Function);
```

Parameter	Type	Description
eventType	String	The event type. For SOS session state events, this value is `SFORCE_SOS:STATE_CHANGED`. For audio QoS, this value is `SFORCE_SOS:QOS_AUDIO`. For video QoS, this value is `SFORCE_SOS:QOS_VIDEO`.
eventListener	Function	This function is called when the registered event is emitted. You receive one JSON

Parameter	Type	Description
		`message` object within the payload passed to this function.

For more information about `SFORCE_SOS:STATE_CHANGED`, see SOS State Change Console Events.

For more information about `SFORCE_SOS:QOS_AUDIO` and `SFORCE_SOS:QOS_VIDEO`, see SOS Quality-of-Service Console Events.

SOS State Change Console Events

You can listen for SOS session state changes from the Salesforce console for logging and debugging purposes.

After you add an event listener for state changes to your console (see Listen for SOS Console Events), inspect your function's payload and handle the event.

```
sforce.console.addEventListener("SFORCE_SOS:STATE_CHANGED", function(payload) {
  // Handle event
});
```

Event Listener Payload

The JSON payload you receive within the event listener function follows this syntax.

```
{
  message: {
    sfdcSosSessionId: <SESSION_ID>,
    currentState: <CURRENT_STATE>,
    previousState: <PREVIOUS_STATE>,
    reason: <REASON_IF_APPLICABLE>
  }
}
```

`sfdcSosSessionId` (String)
The ID associated with the session that emitted the events.

`currentState` (String)
The current state of the session. See the Event States section.

`previousState` (String)
The previous state of the session. See the Event States section.

`reason` (Object or null)
Populated only when the current state is `ENDED`. Contains information about why the session was ended. See the End Reasons section.

This code sample illustrates how to handle an event. Subsequent sections describe how to interpret each part of the message object payload.

```
sforce.console.addEventListener("SFORCE_SOS:STATE_CHANGED",  function(event) {
  var stateChange = {};
  try {
    stateChange = JSON.parse(event.message);
  } catch (e) {
    // Error Parsing JSON Object
    throw new Error(e);
  }

  /*
    Use currentState vs previousState to determine how you reached
    the state you're in. Most useful in the case of the ENDED state
    where you want to know how it ended and if there were any errors.
  */

  var currentState = stateChange.currentState;
  var previousState = stateChange.previousState;

  // Handle Non ENDED state changes
  if (currentState !== 'ENDED') {
    logStateChange(currentState, previousState);
    return;
  }

  // Handle ENDED state change
  switch (stateChange.reason.name) {

    // Handle a session that was intentionally ended by customer or agent
    case 'ENDED_BY_CUSTOMER':
    case 'ENDED_BY_CONSOLE':
      logEndedSession(currentState, previousState, stateChange.reason.name);
      break;

    // Handle a session that ended in an error
    case 'ERROR':
      logEndedWithError(currentState, previousState, stateChange.reason.name,
stateChange.reason.error);
      break;
  }
});
```

Event States

The `currentState` and `previousState` fields can be one of the following states.

`LOADING_RESOURCES`
Widget has loaded. Fetching more resources from the server.

`INTERFACE_CHECK`
Applies to Internet Explorer browsers only. Attempting to install the Internet Explorer plug-in.

`JOINING`
Joining the audio/video session.

`INITIALIZING`
Starting to listen for updates from the audio/video session.

`AV_CONNECTION`
Connecting to the audio/video session, getting microphone or camera permissions from the browser, and starting to send the stream to the audio/video session.

`WAITING`
Waiting to receive the audio/video stream from the SDK.

`CONNECTED`
Session is fully established with both audio and video.

`HOLD`
Session has been put on hold by the customer, the agent, or both.

`PAUSED`
Session paused because the app was put into the background, the customer is typing into a masked field, or the customer accepted a phone call.

`ENDED`
The session has completed. See the End Reasons section for more information.

End Reasons

When the `currentState` is `ENDED`, you can inspect the `reason` object to find out why the session ended.

```
{
  message: {
    sfdcSosSessionId: <SESSION_ID>,
    currentState: <CURRENT_STATE>,
    previousState: <PREVIOUS_STATE>,

    reason: {
      name: <END_REASON>
      error: <ERROR_IF_APPLICABLE>
    }
  }
}
```

`name` (String)
The reason why the session ended. Can be `ENDED_BY_CUSTOMER`, `ENDED_BY_CONSOLE`, or `ERROR`.

`error` (Error object or null)
If there's an error, this field contains the error details. If there isn't an error, the value is `null`. See Errors section.

The following table describes various ways a session can end, with and without an error. See the Errors section for details about session failures.

Table 1: End Reason Scenarios

State	How a Session Can End Without an Error	How a Session Can End with an Error
LOADING_RESOURCES	The customer or agent manually ended the session prematurely.	An issue occurred while loading scripts from the server.

State	How a Session Can End Without an Error	How a Session Can End with an Error
INTERFACE_CHECK	The customer or agent manually ended the session prematurely. Possibly related to a user issue while installing the plug-in.	The agent encountered an issue while installing the OpenTok plug-in.
JOINING	The customer or agent manually ended the session prematurely.	The agent encountered an issue while joining the SOS session.
INITIALIZING	The customer or agent manually ended the session prematurely.	The agent encountered an issue while starting to listen for updates from the SOS session.
AV_CONNECTION	The customer or agent manually ended the session prematurely.	The agent encountered an issue while joining the session. The issue could be due to hardware permissions, misconfigured firewall rules, network performance, or an internal server error.
WAITING	The customer disconnected from the session without ending it, causing the session to end. Typical reasons include the customer lost network connectivity, the app crashed, or the customer closed the app.	The agent failed to connect to the customer's audio/video stream. The issue could be due to misconfigured firewall rules, network performance, an internal server error, or the customer was dropped from the audio/video session.
CONNECTED	The customer or agent manually ended the session from a normal state.	The session ended unexpectedly with a fatal error. This error could be due to network performance issues or an internal server error.
HOLD	The customer or agent manually ended the session from a normal state. The agent or customer could have ended the session after being in the hold state for too long.	The session ended unexpectedly with a fatal error. This error could be due to network performance or an internal server error.
PAUSED	The customer or agent manually ended the session from a normal state. The agent or customer could have ended the session after being in the paused state for too long.	The session ended unexpectedly with a fatal error. This error could be due to network performance or an internal server error.
ENDED	The session ended without issue.	The session experienced issues while disconnecting from the audio/video session or making a request to the audio/video server.

The following table shows some scenarios in which a session can end successfully, along with a sample payload.

Table 2: Successful End Reason Examples

Scenario	Sample Payload
Session ended by customer	`{ currentState: 'ENDED', previousState: 'CONNECTED', reason: { name: 'ENDED_BY_CUSTOMER', error: null } }`
Session ended by agent	`{ currentState: 'ENDED', previousState: 'CONNECTED', reason: { name: 'ENDED_BY_CONSOLE', error: null } }`
Session ended by agent while session is on hold	`{ currentState: 'ENDED', previousState: 'HOLD', reason: { name: 'ENDED_BY_CONSOLE', error: null } }`
Session ended by agent while customer app is in the background	`{ currentState: 'ENDED', previousState: 'PAUSED', reason: { name: 'ENDED_BY_CONSOLE', error: null } }`
Session ended by agent after app crash	`{ currentState: 'ENDED', previousState: 'WAITING', reason: { name: 'ENDED_BY_CONSOLE', error: null } }`

Errors

When a session ends with an error, inspect the `error` object for more information. The error syntax is:

```
{
  message: {
    sfdcSosSessionId: <SESSION_ID>,
    currentState: <CURRENT_STATE>,
    previous: <PREVIOUS_STATE>,
    reason: {
      name: 'ERROR',

      error: {
        code: <ERROR_CODE>,
        domain: <ERROR_DOMAIN>,
        message: <ERROR_MESSAGE>,
        name: <ERROR_NAME>,
        type: <ERROR_TYPE>,
        rawError: {
          code: <RAW_ERROR_CODE>,
          message: <RAW_ERROR_MESSAGE>,
          name: <RAW_ERROR_MESSAGE>
        }
      }
    }
  }
}
```

`code` (Number)
Error code used for grouping related errors. Some common error codes include: 1000 (SOS session timed out waiting to access camera or microphone); 1001 (audio/video request timed out), 1003 (failed to set agent name); 1006 (SOS session timed out waiting to access the camera or microphone); 1500 (permission to audio/video hardware denied). See OpenTok's Handling Exceptions documentation for more error conditions.

`domain` (String or null)
Describes the category of error when it's related to an OpenTok audio/video issue. Can be one of the following: `session`, `publisher`, or `subscriber` domains. A `session` error relates to an existing audio/video session. A `publisher` error describes an issue that the agent had when creating an audio/video stream. A `subscriber` error describes an issue that the agent had when receiving a customer's audio/video stream.

`message` (String)
A description of what caused the error.

`name` (String or null)
A unique name associated with the error.

`type` (String)
Specifies from where the error originated. Can be one of the following: `opentok` (the underlying WebRTC platform); `scrt` (Salesforce's real-time server); or `widget` (the Salesforce console widget).

`rawError` (Object)
The raw error returned by the server without parsing, grouping, or renaming.

The following table shows some scenarios in which a session can end in an error, along with a sample payload.

Table 3: End in Error Examples

Error Scenario	Sample Payload

Agent declines permissions prompt	<pre>{
	  currentState: 'ENDED',
	  previousState: 'AV_CONNECTION',
	  reason: {
	  name: 'ERROR',
	  error: {
	  code: 1500,
	  domain: 'publisher',
	  message: 'Permission to audio/video hardware
	  denied. You must grant permission for SOS to
	  access microphone and camera.',
	  name: 'OT_USER_MEDIA_ACCESS_DENIED',
	  type: 'opentok',
	  rawError: {
	  code: 1500,
	  message: 'ORIGINAL ERROR MESSAGE',
	  name: 'OT_USER_MEDIA_ACCESS_DENIED'
	  }
	  }
	  }
	}</pre>

Agent lets session time out without granting permissions	<pre>{
	  currentState: 'ENDED',
	  previousState: 'AV_CONNECTION',
	  reason: {
	  name: 'ERROR',
	  error: {
	  code: 1000,
	  domain: null,
	  message: 'SOS session timed out waiting to
	  access camera/microphone.',
	  name: null,
	  type: 'widget',
	  rawError: {
	  code: 1000,
	  message: 'SOS session timed out waiting to
	  access camera/microphone.'
	  }
	  }
	  }
	}</pre>

Error Scenario	Sample Payload

Session dies after OpenTok drops connection because of a timeout	<pre>{
	  currentState: 'ENDED',
	  previousState: 'AV_CONNECTION',
	  reason: {
	  name: 'ERROR',
	  error: {
	  code: 1006,
	  domain:session,
	  message: 'SOS session timed out waiting to
	  access camera/microphone.',
	  name: 'OT_SOCKET_CLOSE_ABNORMAL',
	  type: 'opentok',
	  rawError: {
	  code: 1006,
	  message: 'Unable to connect to the session.
	  Please ensure you have network
	  connectivity.',
	  name: 'OT_SOCKET_CLOSE_ABNORMAL'
	  }
	  }
	  }
	}</pre>

SOS Quality-of-Service Console Events

You can listen for SOS audio and video quality-of-service (QoS) events from the Salesforce console.

Note: The console allows you to track streaming issues on the other side of the conversation (from the client to the OpenTok media router). To track QoS issues on this side (from the agent to the media router), refer to the SOS SDK documentation on quality-of-service events: Using SOS with the Service SDK.

After you add an event listener for QoS to your console (see Listen for SOS Console Events), inspect your function's payload and handle the event.

```
sforce.console.addEventListener("SFORCE_SOS:QOS_AUDIO", function(payload) {
  // Handle audio QoS event
});
sforce.console.addEventListener("SFORCE_SOS:QOS_VIDEO", function(payload) {
  // Handle video QoS event
});
```

Audio QoS Event Listener Payload

This sample JSON payload is for the `SFORCE_SOS:QOS_AUDIO` event type.

```
{
  "message":"{
    "bytesReceived":131131,
```

```
    "packetsLost":3,
    "packetsReceived":1499,
    "timestamp":1502214189391,
    "sfdcSosSessionId":"0NXR000000000MS"
  }"
}
```

This payload specifies how many bytes were received, the number of packets lost, and the number of packets received for a 30-second span. If the session ends before 30 seconds, QoS data isn't logged.

Video QoS Event Listener Payload

This sample JSON payload is for the `SFORCE_SOS:QOS_VIDEO` event type.

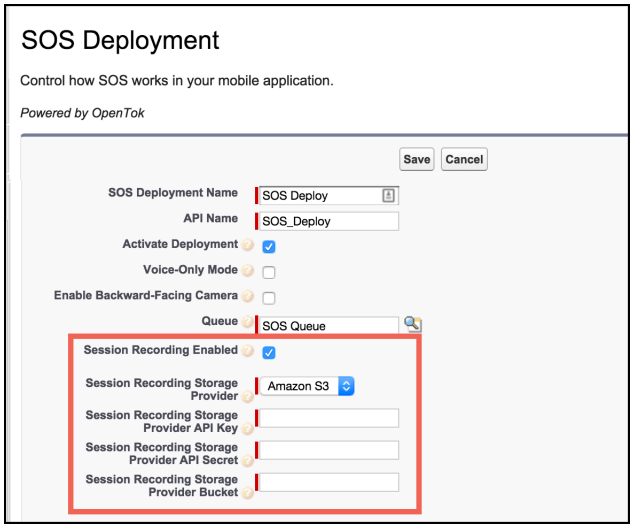
```
{
  "message":"{
    "bytesReceived":82253,
    "packetsLost":0,
    "packetsReceived":337,
    "timestamp":1502214189391,
    "size":"480x640",
    "timePerShareType":{
      "ss":"55.29",
      "ffc":"44.71",
      "bfc":"0.00"
    },
    "sfdcSosSessionId":"0NXR000000000MS"
  }"
}
```

This payload specifies how many bytes were received, the number of packets lost, and the number of packets received for a 30-second span. If the session ends before 30 seconds, QoS data isn't logged. This payload also describes the resolution size of the video and the percentage of time in each share type. The share types are `ss` for screen sharing, `ffc` for the front-facing camera, and `bfc` for the back-facing camera.

Record SOS Sessions

Enable SOS session recording to assure quality and let agents refer to session recordings.

1. From **Setup**, search for **SOS Deployments**.
2. Select your deployment.
3. Check the **Session Recording Enabled** checkbox. Specify your API key, secret, and bucket.



You can retrieve recorded sessions in the mp4 format from your Amazon S3 bucket.

Note: To configure your AWS environment, see Managing Access Permissions to Your Amazon S3 Resources.

SOS Reference ID

Provide an ID to give to support when there are issues with a session.

The SOS Reference ID (also referred to as the SOS Session ID) is a unique ID used to identify a session. It is 15 characters and starts with "0NX". If there is an issue with a session, this ID can be provided to Support to locate logs related to the session.

There are two ways to find the SOS Reference ID:

1. Add it to the SOS Session object
2. Add it to the fields displayed in the SOS Session list view.

Add to Session Object

If the SOS Reference ID is added to the SOS Session Object and SOS Session page layouts, the ID can be seen when viewing any SOS Session. To add the SOS Reference ID to the SOS Session object:

1. From **Setup**, search for **SOS Sessions**.
2. From **SOS Sessions**, select **Fields**. (Do not go to Fields under SOS Session Activities.)
3. Click **New** under **SOS Session Custom Fields & Relationships**.
4. Select **Formula**. Click **Next**.
5. Enter **SOS Reference Id** as the **Field Label**. **Field Name** auto populates.
6. Select **Text** as the **Formula Return Type**. Click **Next**.
7. In the **Simple Formula** text area, enter **Id**. Click **Next**.
8. Click **Next** again. (Permission to view the field can be removed on this page before clicking next.)
9. Click **Save**.

   We recommend that you add this field to all page layouts.

Add to Session List View

The SOS Session list view can be added as a navigation tab item to any console app. Using the SOS Session list view allows you to view the SOS Reference ID for multiple sessions on a single screen. To add the SOS Session list view to a console app:

1. From **Setup**, search for **Apps**.
2. Select **Edit** for the desired console app.
3. Under **Choose Navigation Tab Items**, move **SOS Sessions** to **Selected Items**.
4. Click **Save**.

The SOS Session list view may not display the SOS Reference ID by default. If so, a view can be edited or a new view can be created. To add the SOS Reference Id to the view:

1. Go to the SOS Session list view
2. Click either **Edit** or **Create New View**.
3. Now you can determine which fields are visible.
   - If the new field was added (as shown earlier) move **SOS Reference Id** to **Selected Fields**.
   - If the new field was not created, move both instances of **SOS Session Id** to **Selected Fields**. (The two SOS Session Id fields are different fields. One is the unique ID that starts with the characters "0NX"; the other is a number that increments for each session.)
4. Click **Save**.

Multiple SOS Queues

Implement multiple SOS queues to route requests to specific agents or give specific requests a higher priority.

Multiple queues can help out in the following situations:

- Giving paying customers a higher priority
- Having separate queues for different products
- Routing to agents with specific skill sets
- Giving agents a personal queue (great for training)
- Creating a training queue that has a lower priority or only gets requests from simple pages
- Grouping separate pages into different queues

You need two objects to make multiple queues work: a `Queue` and an `SOS Deployment` object. A third object, `Routing Configuration`, lets you use different priorities.

1. If a `Routing Configuration` is being used to achieve different priorities, create this object first. If you want all queues to have the same priority, the same routing configuration can be used.
2. Next, create the `Queue`. The queue references the routing config. An agent can be a member of multiple queues.
3. Create the `SOS Deployment` last. The deployment references the queue. An app may have access to several SOS deployment IDs, and then the app decides which queue the user should be sent to using the SOS deployment ID.

Embedded Service SDK for Mobile Apps Setup

Set up the SDK to start using Service Cloud features in your mobile app.

Requirements for the Service SDK for iOS

Before you set up the SDK, let's take care of a few pre-reqs.

Salesforce Org Requirements

The Service SDK can be used with both Lightning Experience and Salesforce Classic. However, the SOS agent widget currently only works in Salesforce Classic.

SDK Development Requirements

To develop using this SDK, you must have:

- iOS SDK version 10 or newer
- Xcode version 9 or newer

Mobile App Requirements

Any app that uses this SDK requires:

- iOS version 10 or newer

SOS Agent Requirements

The agents responding to SOS calls must have modern browsers and reasonably high-speed internet connectivity to handle the demands of real-time audio and video.

Hardware requirements:

- Webcam
- Microphone

Bandwidth requirements:

- 500 Kbps upstream
- 500 Kbps downstream

Important: Due to bandwidth limitations, 2G networks, such as GPRS and EDGE, are not supported.

Browser requirements:

- Chrome version 35 or newer
- Firefox version 30 or newer
- Internet Explorer version 10 or newer (plug-in required)

Note: Your browser must support Transport Layer Security (TLS) protocol version 1.1 or newer. If you are running Internet Explorer version 10, see this help topic on how to update the TLS version.

Operating System:

- OSX 10.5 or newer
- Windows 7 or newer

Accessibility with the Service SDK for iOS

The Service SDK is accessible to customers that use a screen reader. Depending on your needs, you can also change some settings to expand accessibility.

Disable the Minimized View in Chat

By default, a chat session starts out as a minimized, thumbnail view that you tap to open. This minimized view is not optimal for accessibility because a visually impaired person could have trouble locating the thumbnail. To improve accessibility, we suggest starting the session in the full-screen view. When creating the `SCSChatConfiguration` object, set `allowMinimization` and `defaultToMinimized` to `false`.

```
let config = SCSChatConfiguration(liveAgentPod: "YOUR_POD_NAME",
                                  orgId: "YOUR_ORG_ID",
                                  deploymentId: "YOUR_DEPLOYMENT_ID",
                                  buttonId: "YOUR_BUTTON_ID")

config?.allowMinimization = false
config?.defaultToMinimized = false
```

See Configure a Chat Session.

Contrast Ratio Considerations

By default, we brand the SDK using a 4.2 contrast ratio. You can customize the colors to increase this contrast ratio.

See Customize Colors with the Service SDK.

Dynamic Text Warning

When changing the iOS text size, the Apple Accessibility Inspector displays the following warning: “Dynamic Text font sizes are unsupported.” Dynamic text is supported, but it requires restarting the app after changing the font size.

Install the Service SDK for iOS

Before you can use the iOS SDK, install the SDK and configure your project.

1. Add the SDK frameworks.

   You can add the frameworks manually or add them using CocoaPods. CocoaPods is a popular dependency manager for Swift and Objective-C projects.

   - Add the Frameworks with CocoaPods
   - Add the Frameworks Manually

2. (Knowledge only) Add an App Transport Security (ATS) exception to `localhost` for serving cached knowledge base articles.

   a. Open the `Info.plist` file for your project.

   > Note: If you right-click the `plist` file from the project navigator, you can select **Open As** > **Source Code** from the context menu. The source code view is a quick way to add domains to your `plist` file.

   b. Add `NSAppTransportSecurity` to your `Info.plist` to allow insecure HTTP loads from `localhost` .

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSExceptionDomains</key>
  <dict>
    <key>localhost</key>
    <dict>
      <key>NSExceptionAllowsInsecureHTTPLoads</key>
      <true/>
    </dict>
  </dict>
  </dict>
```

3. (SOS only) If you're using SOS, iOS 10 requires descriptions for why the app needs to access the device's microphone and camera.

   Add string values for "Privacy - Microphone Usage Description" and "Privacy - Camera Usage Description" in your `Info.plist` file. To learn more about these properties, see Cocoa Keys in Apple's reference documentation.

   Sample values for these keys:

```
<key>NSMicrophoneUsageDescription</key>
<string>Used for an SOS chat with an agent.</string>
<key>NSCameraUsageDescription</key>
<string>Used for an SOS video chat with an agent.</string>
```

4. (Chat only) If you're using Chat, iOS 10 requires descriptions for why the app needs to access the device's camera and photo library.

   Add string values for "Privacy - Camera Usage Description" and "Privacy - Photo Library Usage Description" in your `Info.plist` file. To learn more about these properties, see Cocoa Keys in Apple's reference documentation.

   Sample values for these keys:

```
<key>NSCameraUsageDescription</key>
<string>Used when sending an image to an agent.</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>Used when sending an image to an agent.</string>
```

5. (SOS only) If you're using SOS, turn on **Background Modes** for your project.

To ensure that an SOS and Chat session remains active while the app is in the background, verify that your background settings are correct.

a. Open your Project Target Settings.

b. Select the **Capabilities** tab.

c. Set the **Background Modes** to **ON**.

d. From the **Background Modes** subcategories, check the **Audio, AirPlay, and Picture in Picture** item.

You're now ready to get started using the SDK!

Add the Frameworks with CocoaPods
Add the SDK frameworks using CocoaPods, a developer tool that automatically manages dependencies.

Add the Frameworks Manually
Add the SDK frameworks by manually embedding the appropriate frameworks.

Add the Frameworks with CocoaPods

Add the SDK frameworks using CocoaPods, a developer tool that automatically manages dependencies.

1. If you haven't already done so, install the CocoaPods gem and initialize the CocoaPods master repository.

```
sudo gem install cocoapods
pod setup
```

Note: The minimum supported version of CocoaPods is 1.0.1. If you're not sure what version you have, use `pod --version` to check the version number.

2. If you already have CocoaPods installed, update your pods to the latest version.

```
pod update
```

3. Change to the root directory of your application project.
4. Create or edit a file named `Podfile` that contains the Service SDK dependency.

   a. If you want to install the complete Service SDK, update your `Podfile` to include `ServiceSDK`.

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/goinstant/pods-specs-public'

# To use the SDK (with all components)
target '<your app target>' do
    pod 'ServiceSDK'
end
```

   b. If you want to install a single Service SDK component, create a similar `Podfile` to the one specified above, but only include the desired pod.

Feature	Pod name
Knowledge	`ServiceSDK/Knowledge`
Case Management	`ServiceSDK/Cases`

Feature	Pod name
Chat	`ServiceSDK/Chat`
SOS	`ServiceSDK/SOS`

For example, the following `Podfile` installs SOS.

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/goinstant/pods-specs-public'

# To use SOS
target '<your app target>' do
    pod 'ServiceSDK/SOS'
end
```

If you don't specify a version number, you automatically get the latest version of that component. If you want to add a specific version to your component, be sure to add the version number of the Service SDK and *not* the version number of the individual component.

For instance, if you want version 3.10.0 of SOS, specify 220.1.0 because version 220.1.0 of the Service SDK has version 3.10.0 of SOS.

```
source 'https://github.com/CocoaPods/Specs.git'
source 'https://github.com/goinstant/pods-specs-public'

# To use SOS (with version info)
target '<your app target>' do
    pod 'ServiceSDK/SOS', '220.1.0'
end
```

5. Run the CocoaPods installer.

   ```
   pod install
   ```

   This command generates a `.xcworkspace` file for you with all the dependencies included.

6. Open the `.xcworkspace` file that CocoaPods generated and continue with the installation process.

   Note: Be sure to open the `.xcworkspace` file (which includes all the dependencies) and *not* the `.xcodeproj` file.

Once you've added the SDK frameworks, proceed with the installation instructions on page 38.

Add the Frameworks Manually

Add the SDK frameworks by manually embedding the appropriate frameworks.

1. Download the SDK frameworks from the Service SDK download page.
2. Embed the relevant Service SDK frameworks into your project.

   You can find the framework files within the `Frameworks` folder. Specifically, the following frameworks are available for you to use:

Framework	Description	Required?
`ServiceCore`	Contains all the common components used by the Service SDK.	Yes
`ServiceKnowledge`	Contains access to the Knowledge features of the SDK.	Only if using Knowledge
`ServiceCases`	Contains access to the Case Management features of the SDK.	Only if using Case Management
`ServiceChat`	Contains access to the Chat features of the SDK.	Only if using Chat
`ServiceSOS`	Contains access to the SOS features of the SDK.	Only if using SOS

Add the relevant frameworks to the **Embedded Binaries** section of the **General** tab for your target app. Be sure to select **Copy items if needed** when embedding.

Once you've embedded the frameworks, you'll automatically see them appear in the **Linked Frameworks and Libraries** section as well. If you see two line items for each framework (which happens if you drag the frameworks into the project before embedding), delete the duplicates.

Once you've added the SDK frameworks, proceed with the installation instructions on page 38.

Authentication with the Service SDK for iOS

The Service SDK provides an authentication mechanism that allows your users to access user-specific information in Service Cloud. To authenticate, create an `SCSAuthenticationSettings` object and pass it to the SDK.

Authentication Settings

Authentication with the Service SDK uses an `SCSAuthenticationSettings` object. Create this object with a client ID and a dictionary containing authentication settings. This authentication settings dictionary must contain a URL for your org (`SCSOAuth2JSONKeyInstanceUrl`) and an access token (`SCSOAuth2JSONKeyAccessToken`). If your OAuth2 flow supports refresh tokens, include a refresh token (`SCSOAuth2JSONKeyRefreshToken`) to the authentication settings.

In Swift:

```
// Specify auth info
let myClientId: String = "CLIENT_ID_VALUE"
let authDictionary: [SCSOAuth2JSONKey: String] =
  [ .instanceUrl : "https://URL_FOR_YOUR_ORG.com",
    .accessToken : "ACCESS_TOKEN_VALUE" ]

// Create auth settings object
let authSettings = SCSAuthenticationSettings(oauth2: authDictionary,
                                             clientId: myClientId)
```

In Objective-C:

```
// Specify auth info
NSString *myClientId = @"CLIENT_ID_VALUE";
NSDictionary<SCSOAuth2JSONKey, NSString*> *authDictionary =
  @{ SCSOAuth2JSONKeyInstanceUrl : @"https://URL_FOR_YOUR_ORG.com",
     SCSOAuth2JSONKeyAccessToken : @"ACCESS_TOKEN_VALUE" };

// Create auth settings object
SCSAuthenticationSettings *authSettings =
  [[SCSAuthenticationSettings alloc] initWithOAuth2Dictionary: authDictionary
                                                     clientId: myClientId];
```

If you're using the Salesforce Mobile SDK, we provide a helper method that allows you to construct an `SCSAuthenticationSettings` object directly from the Mobile SDK user account. You can use this sample code after you've successfully logged in a user.

In Swift:

```
// Get user account info from the Salesforce Mobile SDK
let identity: SFUserAccountIdentity =
  SFUserAccountIdentity(userId: myUserId, orgId: myOrgId)
let account: SFUserAccount =
  SFUserAccountManager.sharedInstance().userAccount(forUserIdentity: identity)!

// Create auth settings object from SFUserAccount
let authSettings = SCSAuthenticationSettings(mobileSDK: account)
```

In Objective-C:

```
// Get user account info from the Salesforce Mobile SDK
SFUserAccountIdentity *identity =
  [SFUserAccountIdentityClass identityWithUserId:myUserId orgId:myOrgId];
SFUserAccount *account =
  [[SFUserAccountManagerClass sharedInstance] userAccountForUserIdentity:identity];

// Create auth settings object from SFUserAccount
SCSAuthenticationSettings *authSettings =
  [[SCSAuthenticationSettings alloc] initWithMobileSDKAccount:account];
```

Note: For developers who plan to use the Salesforce Mobile SDK for authentication, the Mobile SDK Developer's Guide contains authentication instructions. If you're using a Salesforce community, be sure to configure the login endpoint as described in the Salesforce Mobile SDK documentation (Configure the Login Endpoint). This documentation describes how to use the `SFDCOAuthLoginHost` property in your `info.plist` file to create a custom login URI.

If you plan to use remote push notification to alert the user when an event occurs in your org, call `registerForPushNotifications` on the `SCSAuthenticationSettings` object. To learn more, see Notifications with the Service SDK for iOS.

However you create an `SCSAuthenticationSettings` object, pass it to the Service SDK during the authentication flow.

Authentication Flow

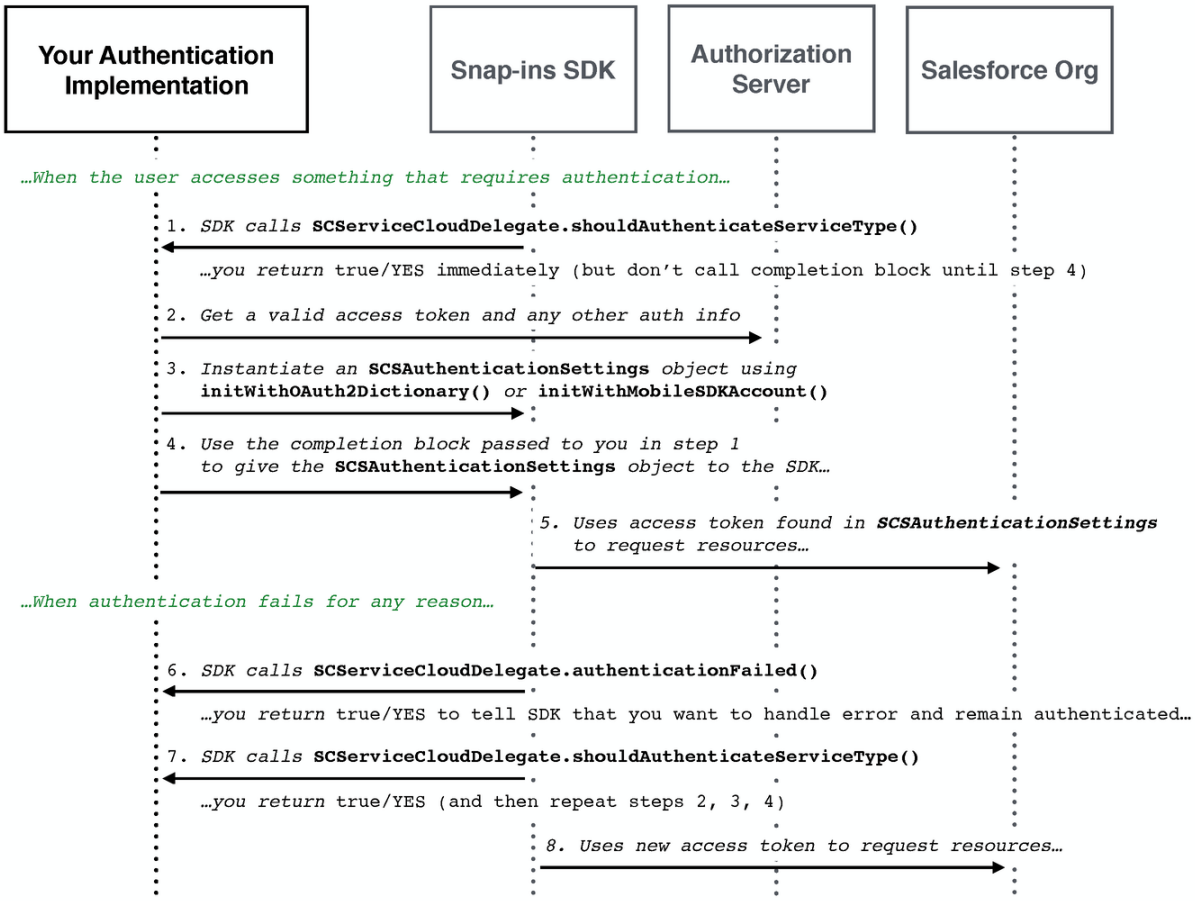
You can either authenticate on-demand when the SDK calls `serviceCloud(shouldAuthenticateServiceType:completion:)` in your `SCServiceCloudDelegate` implementation, or you can authenticate explicitly (that is, before the app attempts to show the relevant UI) using the

`setAuthenticationSettings(settings:forServiceType:completion:)` method in the `ServiceCloud` shared instance.

With on-demand authentication, you perform the authentication asynchronously, after the SDK calls your `serviceCloud(shouldAuthenticateServiceType:completion:)` delegate method. Once authenticated, pass the `SCSAuthenticationSettings` object to the completion block that you're given in the `serviceCloud(shouldAuthenticateServiceType:completion:)` method.

The following sequence diagram illustrates the basic authentication flow for on-demand authentication.

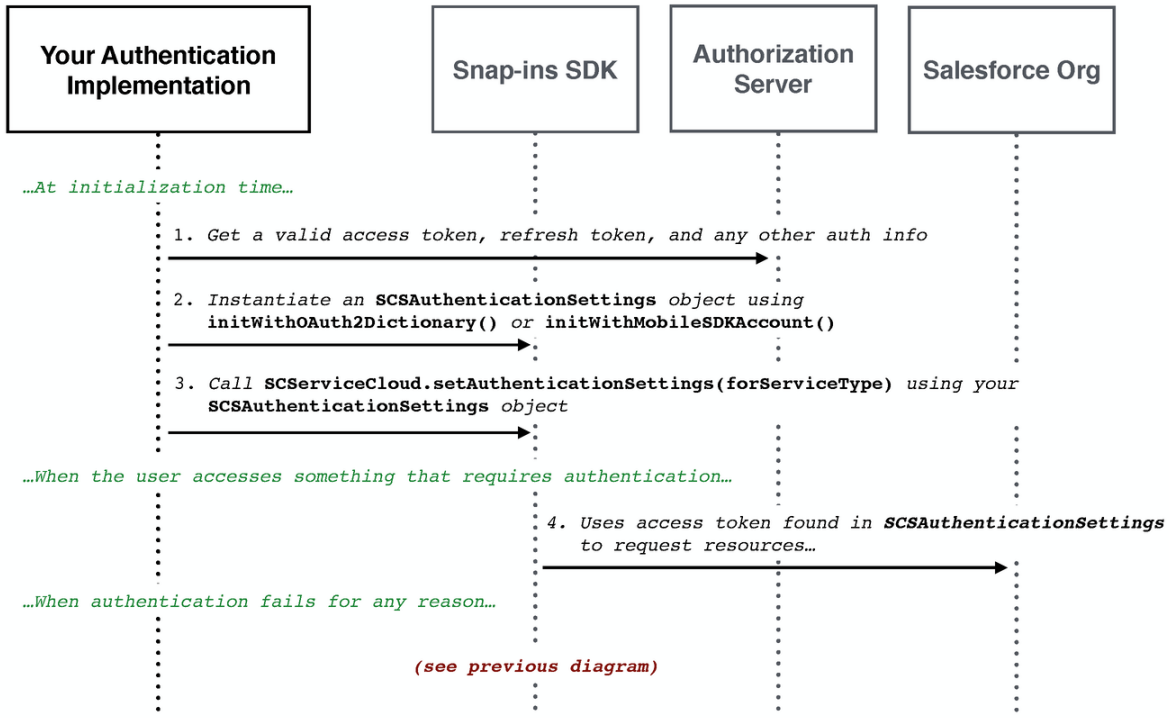
On-Demand Authentication



Alternatively, you can explicitly authenticate before any UI appears that requires authentication. Call the `setAuthenticationSettings(settings:forServiceType:completion:)` method in the `ServiceCloud` shared instance using the `SCSAuthenticationSettings` object.

The following sequence diagram illustrates the authentication flow for explicit authentication.

Explicit Authentication



To programmatically log out a user, call `setAuthenticationSettings(settings:forServiceType:completion:)` using `nil` for the `SCSAuthenticationSettings` argument.

Error Conditions

Implement `serviceCloud(authenticationFailed:forServiceType:)` in your `SCServiceCloudDelegate` object to handle error conditions. The SDK calls this method if the access token expires, and for any other scenario that results in an authentication failure. If you return `true` or `YES`, the SDK assumes that you want to proceed, and it subsequently calls `serviceCloud(shouldAuthenticateServiceType:completion:)` to give you a chance to send updated authentication information. If you return `false` or `NO`, the SDK goes back to the guest user state.

Sample Code

The following sample code illustrates how to implement an `SCServiceCloudDelegate` object to handle authentication.

```
class MyAuthHandler: NSObject, SCServiceCloudDelegate {

  override init() {
    super.init()

    // Subscribe to ServiceCloud events
    ServiceCloud.shared().delegate = self
  }
```

```
  /**
   Implementation of a `ServiceCloudDelegate` method that allows you to
   authenticate for a given service.
   */
  func serviceCloud(_ serviceCloud: ServiceCloud,
                    shouldAuthenticateServiceType service: SCServiceType,
                    completion: @escaping (SCSAuthenticationSettings?) -> Void) -> Bool {


    // Rather than scrutinize the service to see if we want to authenticate,
    // let's just assume that we always want to authenticate...

    // TO DO: Authenticate asynchronously
    let urlRequest = URLRequest.init(url: URL(string: "https://example.com/auth")!)
    URLSession.shared.dataTask(with: urlRequest) { (data, response, error) in

      // TO DO: Populate the `SCSAuthenticationSettings` object from the result.
      var authSettings: SCSAuthenticationSettings?

      // Call the completion block with the authentication settings (asynchronously)
      completion(authSettings)

    }.resume()

    // Tell the SDK that we do plan to authenticate
    return true
  }

  /**
   Implementation of a `ServiceCloudDelegate` method to handle authentication
   failure events.
   */
  func serviceCloud(_ serviceCloud: ServiceCloud,
                    authenticationFailed error: Error,
                    forServiceType service: SCServiceType) -> Bool {

    // For this example, let's not bother handling the error,
    // and just fall back to the guest user state...
    // TO DO: In your code, you should inspect this error.
    // If you want to handle the error, you could
    // return `true` and then you'd be called back in the
    // `shouldAuthenticateServiceType` method above.

    return false
  }
}
```

Notifications with the Service SDK for iOS

The Service SDK can display notifications for activity related to Chat and Case Management.

To learn more, see Notifications for Chat Activity and Push Notifications for Case Activity.

Prepare Your App for Submission

Before you can submit your app to the App Store, you need to strip development resources (such as unneeded architectures and header resources) from the dynamic libraries used by the Service SDK.

Xcode doesn't automatically strip unneeded architectures from dynamic libraries, nor remove some header and utility resources. Apps that don't do this clean up are rejected from the App Store. For example, you might receive the following error from iTunes Connect:

```
ERROR ITMS-90085:
No architectures in the binary. Lipo failed to detect any architectures in the bundle
executable.
```

You can resolve this problem by using the script provided in the `ServiceCore` framework that automatically strips unneeded architectures from the dynamic libraries and then re-signs them. To use this script:

1. Select `Build Phases` for your project target.
2. Create a `Run Script` phase to run the script.

   Access the `prepare-framework` script from within the `ServiceCore` framework in your project directory.

   For example, if the framework is in your main project directory, use:

   ```
   "$SRCROOT/ServiceCore.framework/prepare-framework"
   ```

   And if you've installed the SDK with CocoaPods, use:

   ```
   "$PODS_ROOT/ServiceSDK/Frameworks/ServiceCore.framework/prepare-framework"
   ```

Note: This build phase must occur **after** the link phase and all embed phases. If you're using CocoaPods, make sure to put this script after the "[CP] Embed Pods Frameworks" phase.

iOS Tutorials & Examples

Get going quickly with these short introductory tutorials.

In addition to these tutorials, check out our GitHub repository (github.com/forcedotcom/ServiceSDK-iOS) for sample apps.

Get Started with Chat
Get rolling quickly with chat sessions between your customers and your agents.

Get Started with Knowledge
It's easy to wire up your iOS app to your knowledge base articles.

Get Started with Case Publisher
Quickly build an app that lets you create a new case.

Get Started with SOS
See for yourself how easy and effective live video chat and screen sharing can be.

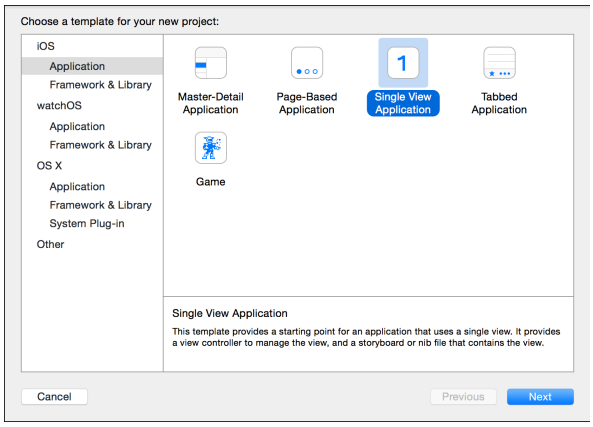
Get Started with Chat

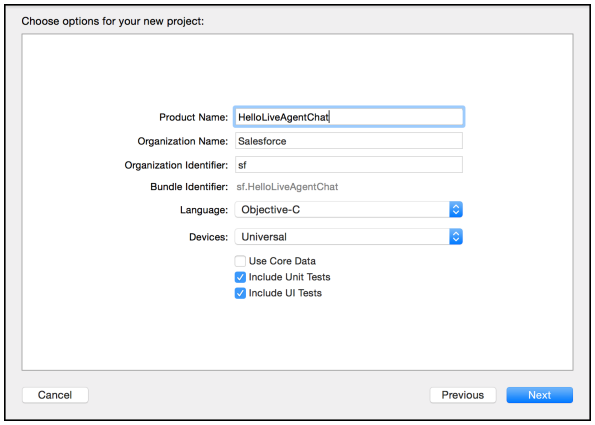
Get rolling quickly with chat sessions between your customers and your agents.

Before doing this tutorial, be sure that you've set up Service Cloud for Chat. See Org Setup for Chat in Lightning Experience with a Guided Flow for more information.

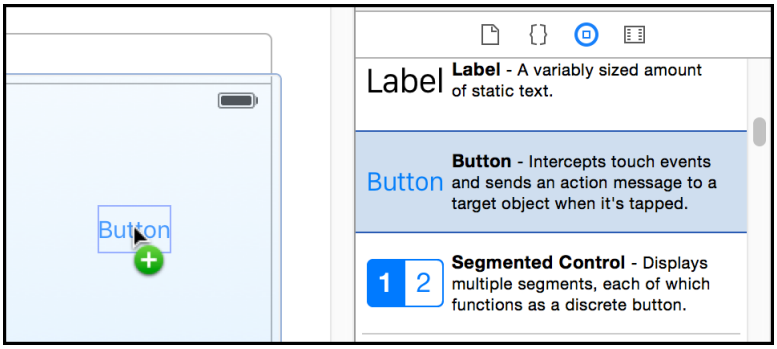
This tutorial shows you how to get Chat into your iOS app.

1. Create an Xcode project. For this example, let's make a Single View Application. Name it `HelloChat`.

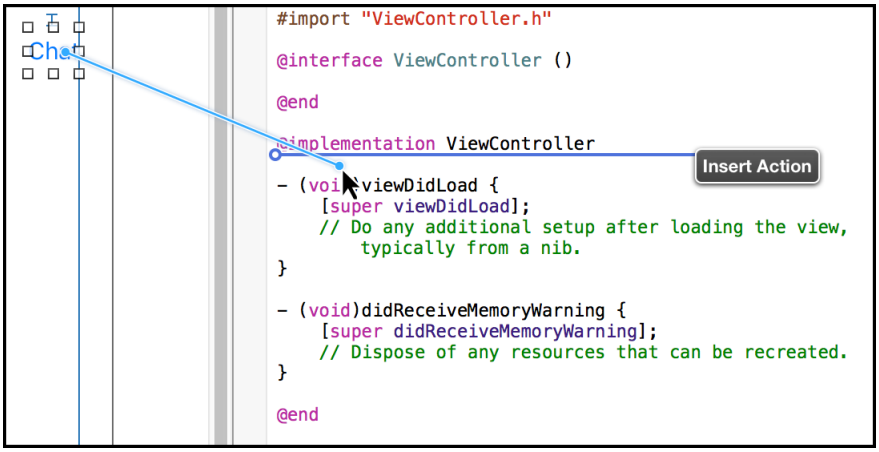


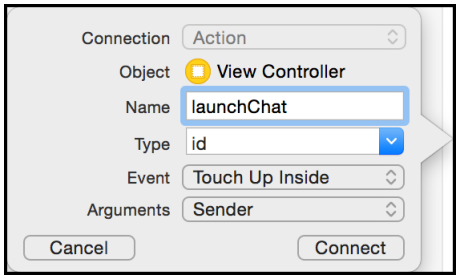


2. Install the SDK as described in Install the Service SDK for iOS.
3. Go to your storyboard and place a button somewhere on the view. Name it `Chat`.



4. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `launchChat`.





5. Import the SDK. Wherever you intend to use the Chat SDK, be sure to import the ServiceCore framework and the ServiceChat framework.

   In Swift:

```
import ServiceCore
import ServiceChat
```

   In Objective-C:

```
@import ServiceCore;
@import ServiceChat;
```

6. Launch a chat session from within the `launchChat` method.

   From the button action implementation, launch chat using the `showChat(with:showPrechat:)` method on `SCSChatInterface`.

   In Swift:

```
@IBAction func launchChat(sender: AnyObject) {

  let config = SCSChatConfiguration(liveAgentPod: "YOUR_POD_NAME",
                                    orgId: "YOUR_ORG_ID",
                                    deploymentId: "YOUR_DEPLOYMENT_ID",
                                    buttonId: "YOUR_BUTTON_ID")

  // Start the session
  ServiceCloud.shared().chatUI.showChat(with: config)
}
```

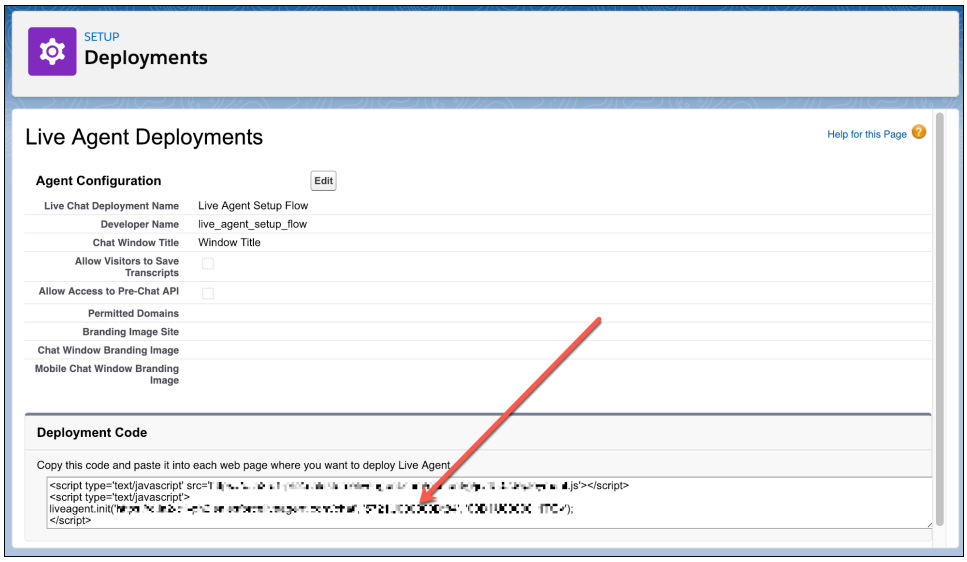
In Objective-C:

```
- (IBAction)launchChat:(id)sender {

  SCSChatConfiguration *config =
    [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"YOUR_POD_NAME"
                                                 orgId:@"YOUR_ORG_ID"
                                          deploymentId:@"YOUR_DEPLOYMENT_ID"
                                              buttonId:@"YOUR_BUTTON_ID"];

  // Start the session
  [[SCServiceCloud sharedInstance].chatUI showChatWithConfiguration:config];
}
```

Fill in the placeholder text for the Chat API endpoint, the org ID, the deployment ID, and the button ID.

Deployment ID

The unique ID of your Chat deployment. To get this value, from Setup, select **Chat** > **Deployments**. The script at the bottom of the page contains a call to the `liveagent.init` function with the **pod**, the **deploymentId**, and **orgId** as arguments. Copy the **deploymentId** value.



For instance, if the deployment code contains the following information:

```
<script type='text/javascript'

src='https://d.gla3.gus.salesforce.com/content/g/js/44.0/deployment.js'></script>
<script type='text/javascript'>
liveagent.init('https://d.gla5.gus.salesforce.com/chat', '573B00000005KXz',
'00DB00000003Rxz');
</script>
```

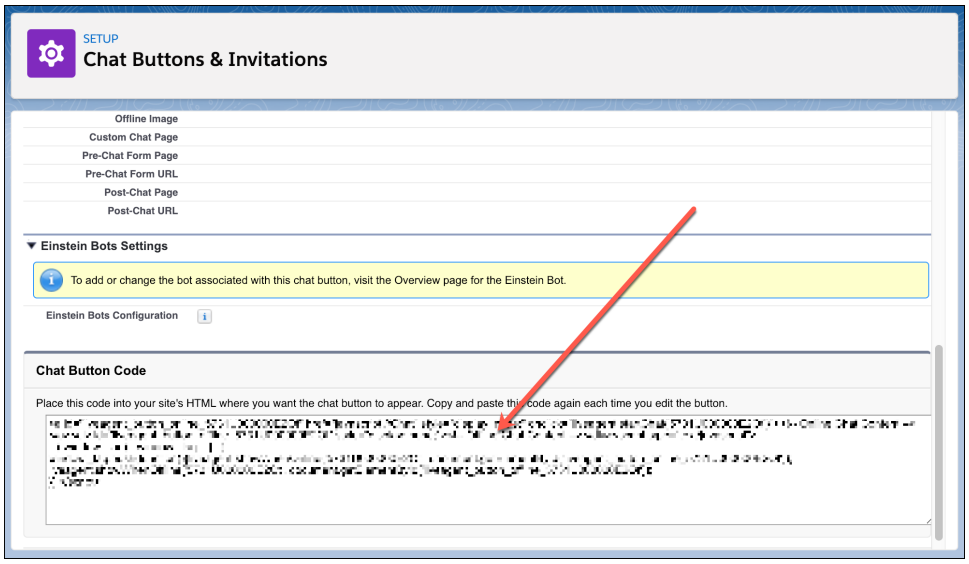
The deployment ID value is:

```
573B00000005KXz
```

Be sure not to use the org ID value (which is also in this deployment code) for the deployment ID.

Button ID

The unique button ID for your chat configuration. To get this value, from Setup, search for **Chat Buttons** and select **Chat Buttons & Invitations**. Copy the `id` for the button from the JavaScript snippet.



For instance, if your chat button code contains the following information:

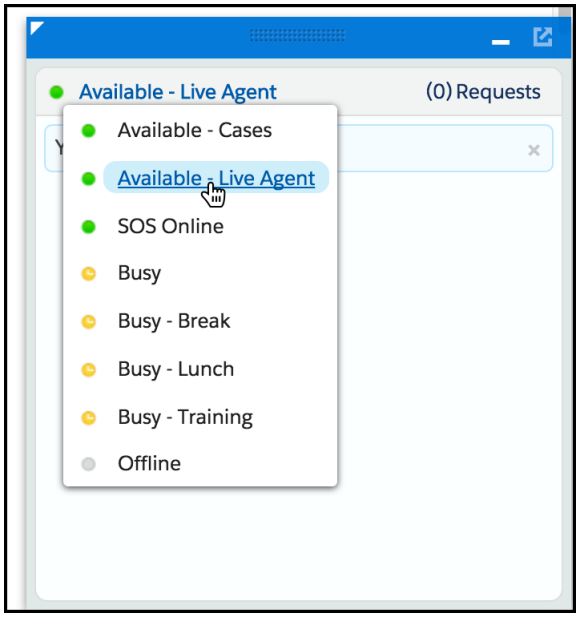
```
<a id="liveagent_button_online_575C00000004h3m"
   href="javascript://Chat"
   style="display: none;"
   onclick="liveagent.startChat('575C00000004h3m')">
   <!-- Online Chat Content -->
</a>
<div id="liveagent_button_offline_575C00000004h3m"
     style="display: none;">
     <!-- Offline Chat Content -->
</div>
<script type="text/javascript">
  if (!window._laq) { window._laq = []; }
  window._laq.push(function() { liveagent.showWhenOnline('575C00000004h3m',
    document.getElementById('liveagent_button_online_575C00000004h3m'));
    liveagent.showWhenOffline('575C00000004h3m',
    document.getElementById('liveagent_button_offline_575C00000004h3m'));
  });
</script>
```

The button ID value is:

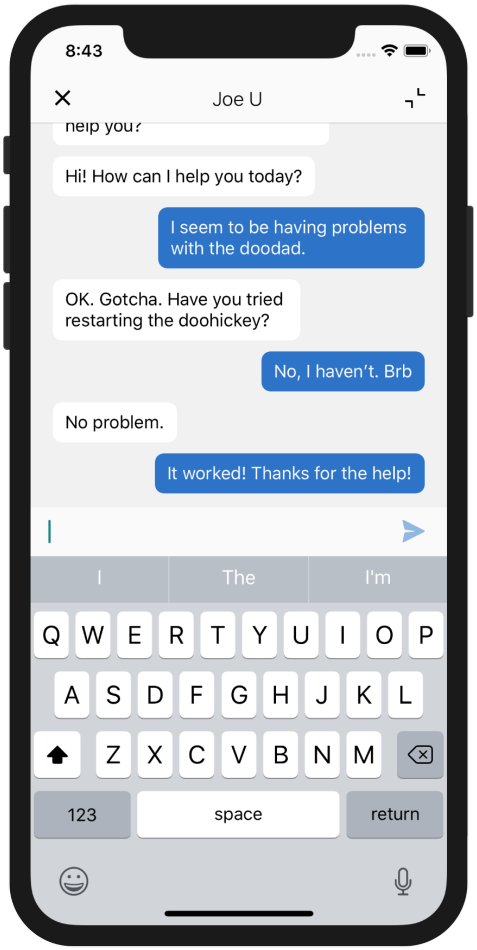
```
575C00000004h3m
```

Be sure to omit the `liveagent_button_online_` text from the ID when using it in the SDK.

7. Launch **Service Cloud Console**. From the **Omni-Channel** widget, ensure that an agent is online.



Now you can build and run the app. When you tap the **Chat** button, the app requests a chat session, which an agent can accept from the **Service Cloud Console**. From the console, an agent can real-time chat with a customer.



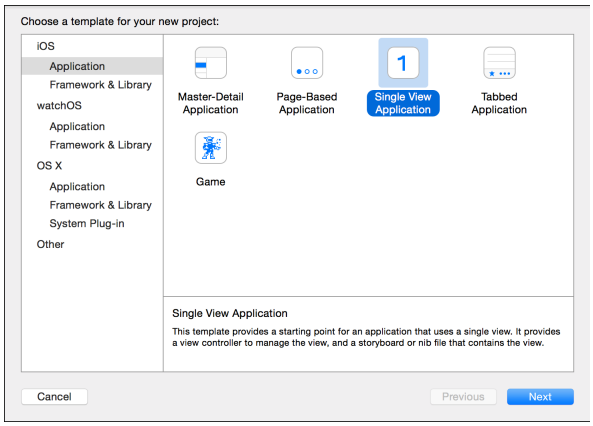
Get Started with Knowledge

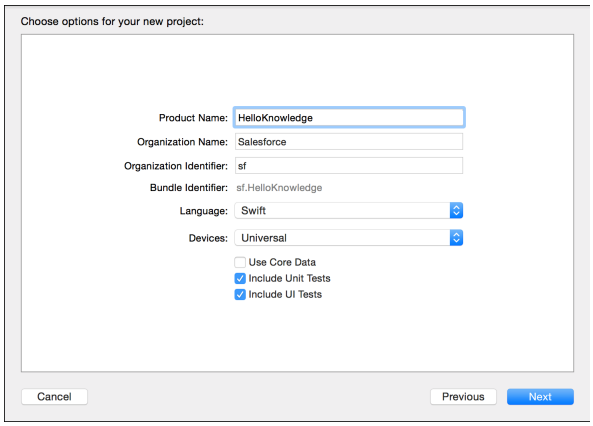
It's easy to wire up your iOS app to your knowledge base articles.

Before doing this tutorial, be sure that you've set up Service Cloud for Knowledge. See Cloud Setup for Knowledge for more information.

This tutorial shows you how to put a knowledge base into your iOS app.

1. Create an Xcode project. For this example, let's make a Single View Application. Name it `HelloKnowledge`.





2. Install the SDK as described in Install the Service SDK for iOS.
3. From your app delegate implementation, import the SDK.

   In Swift:

   ```
   import ServiceCore
   import ServiceKnowledge
   ```

   In Objective-C:

   ```
   @import ServiceCore;
   @import ServiceKnowledge;
   ```

4. Point the SDK to your org from the `applicationDidFinishLaunchingWithOptions` method of your app delegate implementation.

To connect your app to your organization, create an `SCSServiceConfiguration` object containing the community URL, the data category group, and the root data category. Pass this object to the `ServiceCloud` shared instance using `SCSServiceConfiguration(community:dataCategoryGroup:rootDataCategory:)`

In Swift:

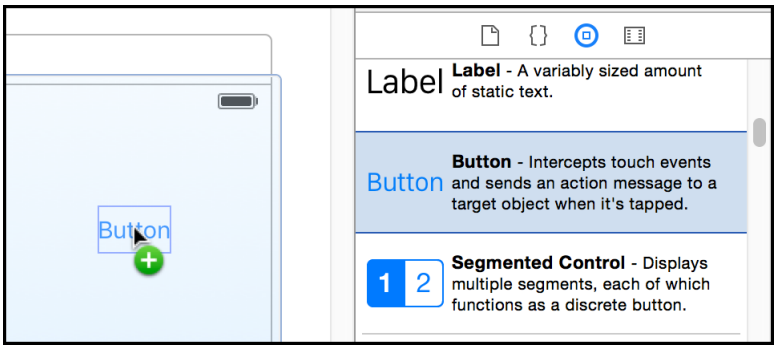
```
// Create configuration object with init params
let config = SCSServiceConfiguration(
               community: URL(string: "https://mycommunity.example.com")!,
               dataCategoryGroup: "Regions",
               rootDataCategory: "All")

// Perform any additional configuration here

// Pass configuration to shared instance
ServiceCloud.shared().serviceConfiguration = config
```

In Objective-C:

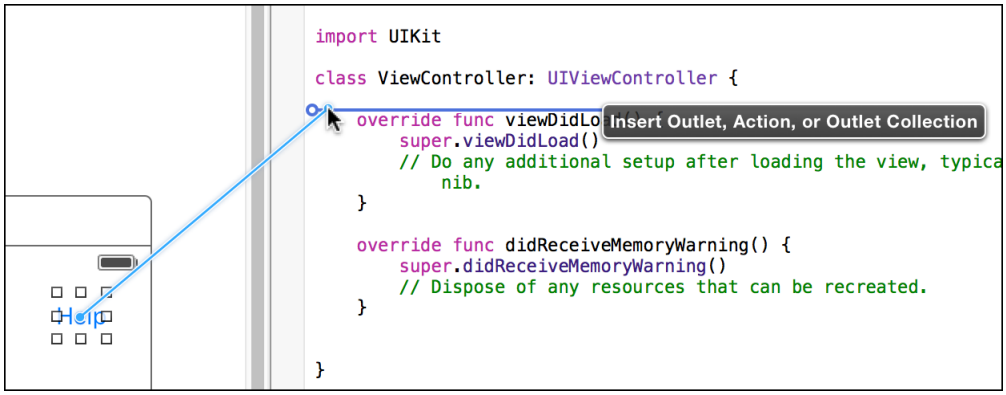
```
// Create configuration object with init params
SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
   initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]
   dataCategoryGroup:@"Regions"
    rootDataCategory:@"All"];

// Perform any additional configuration here

// Pass configuration to shared instance
[SCServiceCloud sharedInstance].serviceConfiguration = config;
```

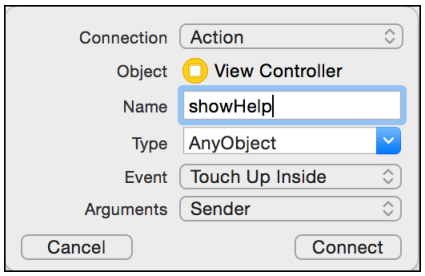
Note: You can get the required parameters for this method from your Salesforce org. If your Salesforce admin hasn't set up Knowledge in Service Cloud or you need more guidance, see Cloud Setup for Knowledge.

5. Go to your storyboard and place a button somewhere on the view. Name it `Help`.



6. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `showHelp`.





7. From your view controller implementation, import the SDK.

   In Swift:

```
import ServiceCore
import ServiceKnowledge
```

   In Objective-C:

```
@import ServiceCore;
@import ServiceKnowledge;
```

8. From within the button action handler, activate the Knowledge interface using the `setInterfaceVisible` method.

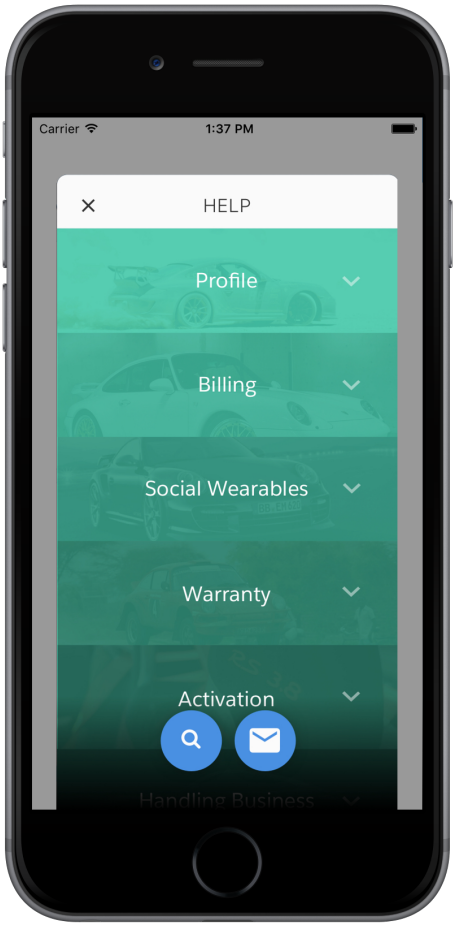
   In Swift:

```
ServiceCloud.shared().knowledge.setInterfaceVisible(true,
                                                    animated: true,
                                                    completion: nil)
```

   In Objective-C:

```
[[SCServiceCloud sharedInstance].knowledge setInterfaceVisible:YES
                                                      animated:YES
                                                    completion:nil];
```

And that's it! You can now build and run your app to see how it looks. Click the `Help` button to activate the interface.



You can customize the interface so it looks and feels just like your app. Check out SDK Customizations with the Service SDK for iOS for guidance in this area.

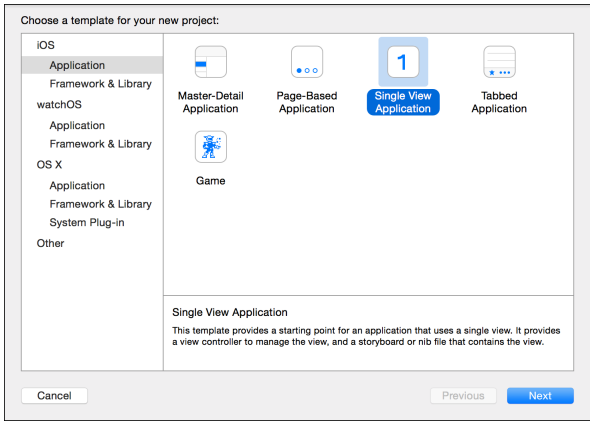
Get Started with Case Publisher

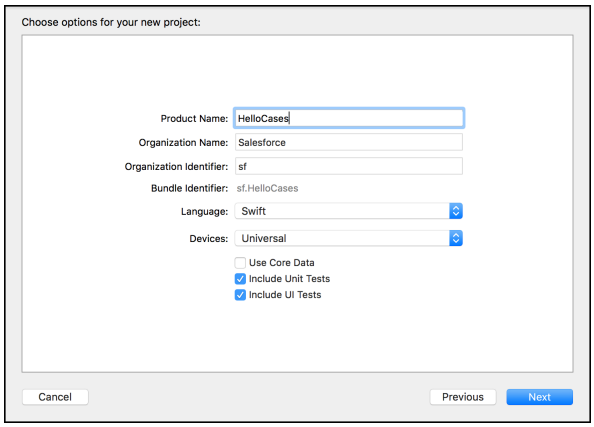
Quickly build an app that lets you create a new case.

Before doing this tutorial, be sure that you've set up Service Cloud for Case Management. See Cloud Setup for Case Management for more information.

This tutorial shows you how to connect your iOS app to the case management interface as a guest user. A guest user is able to publish new cases. However, if you'd like to manage existing cases, you'll need to authenticate a user from within your app. Authentication is discussed in Case Management as an Authenticated User.

1. Create an Xcode project. For this example, let's make a Single View Application. Name it `HelloCases`.





2. Install the SDK as described in Install the Service SDK for iOS.
3. From your app delegate implementation, import the SDK.

   In Swift:

   ```
   import ServiceCore
   import ServiceCases
   ```

   In Objective-C:

   ```
   @import ServiceCore;
   @import ServiceCases;
   ```

4. Point the SDK to your org using an `SCSServiceConfiguration` object.

   To connect your application to your organization, create an `SCSServiceConfiguration` object containing the community URL. Pass this object to the `ServiceCloud` shared instance using `SCSServiceConfiguration(community:)`.

   In Swift:

   ```
   // Create configuration object with your community URL
   let config = SCSServiceConfiguration(
                community: URL(string: "https://mycommunity.example.com")!)

   // Pass configuration to shared instance
   ServiceCloud.shared().serviceConfiguration = config
   ```

In Objective-C:

```
// Create configuration object with your community URL
SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
  initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]];

// Pass configuration to shared instance
[SCServiceCloud sharedInstance].serviceConfiguration = config;
```

You can get the community URL from your Salesforce org. From Setup, search for **All Communities**, and copy the URL for the desired community. For more help, see Cloud Setup for Case Management.

Note: If you plan to access Knowledge in addition to Case Management, use `SCSServiceConfiguration(community:dataCategoryGroup:rootDataCategory:)` instead. This constructor sets up data categories in addition to setting the community URL. See Quick Setup: Knowledge in the Service SDK in the Knowledge section for more info.

5. Assign a global action to the Case Management interface. The global action determines the fields shown when a user creates a case.

   To configure the fields shown when creating a case, specify the global action name in the `caseCreateActionName` property. This code snippet illustrates how to associate the case publisher feature with the **New Case** global action layout, which is one of the default actions provided in most orgs.

   In Swift:

   ```
   ServiceCloud.shared().cases.caseCreateActionName = "NewCase"
   ```

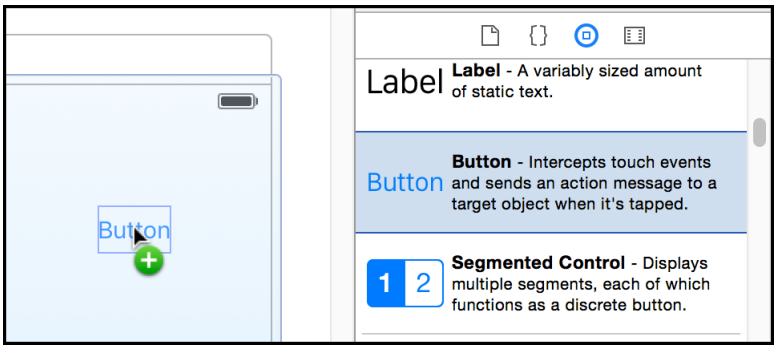
   In Objective-C:

   ```
   [SCServiceCloud sharedInstance].cases.caseCreateActionName = @"NewCase";
   ```

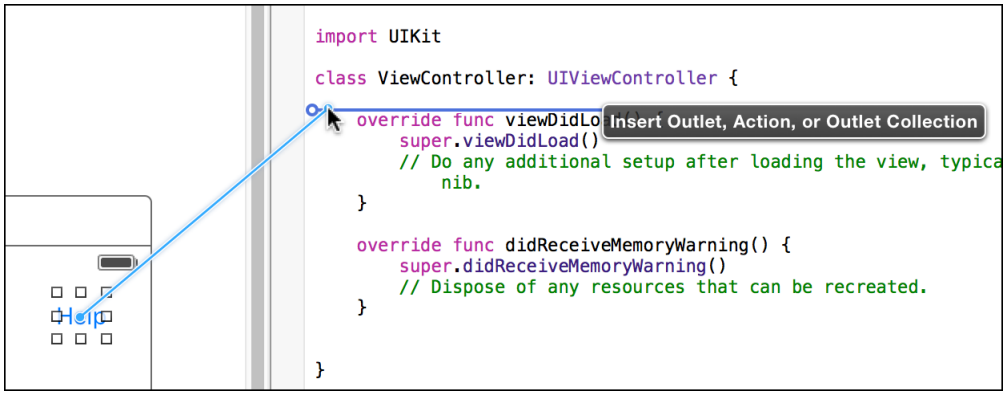
   You can get the global action name from your Salesforce org. From Setup, search for **Global Actions**, and copy the name of the desired quick action. For more help, see Cloud Setup for Case Management.

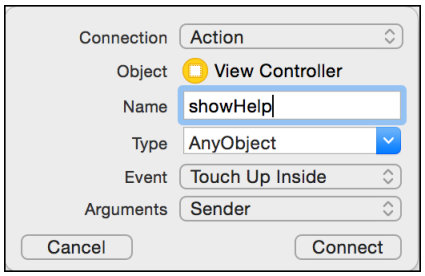
   Note: Be sure that your global action is accessible to the Guest user profile. Also note that the case publisher screen does not respect field-level security for guest users. If you want to specify different security levels for different users, use different quick actions.

6. Go to your storyboard and place a button somewhere on the view. Name it `Help`.



7. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `showHelp`.





8. From your view controller implementation, import the SDK.

   In Swift:

```
import ServiceCore
import ServiceCases
```

   In Objective-C:

```
@import ServiceCore;
@import ServiceCases;
```

9. From within the button action handler, activate the Case Management interface using the `setInterfaceVisible` method.

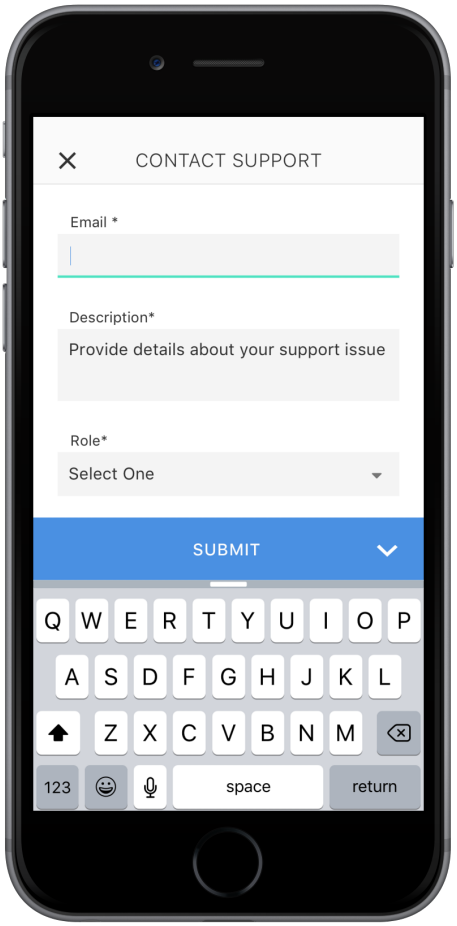
   In Swift:

```
ServiceCloud.shared().cases.setInterfaceVisible(true,
                                                animated: true,
                                                completion: nil)
```

   In Objective-C:

```
[[SCServiceCloud sharedInstance].cases setInterfaceVisible:YES
                                                  animated:YES
                                                completion:nil];
```

And that's it! You can now build and run your app to see how it looks. Click the `Help` button to activate the interface.



If you would like to give your users access to their existing case list, you'll need to authenticate the user first. To learn more about authentication, see Case Management as an Authenticated User. You can also customize the look and feel of the interface, as described in SDK Customizations with the Service SDK for iOS.

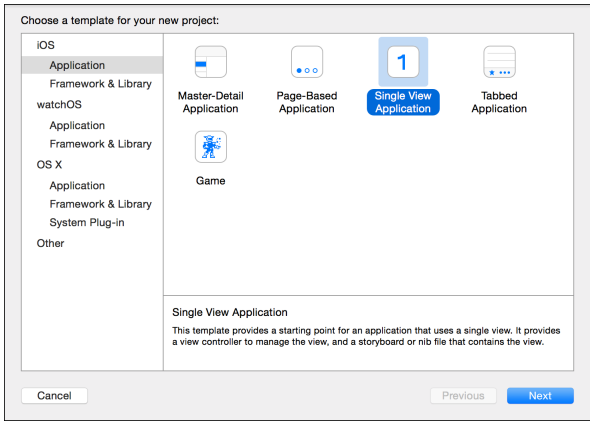
Get Started with SOS

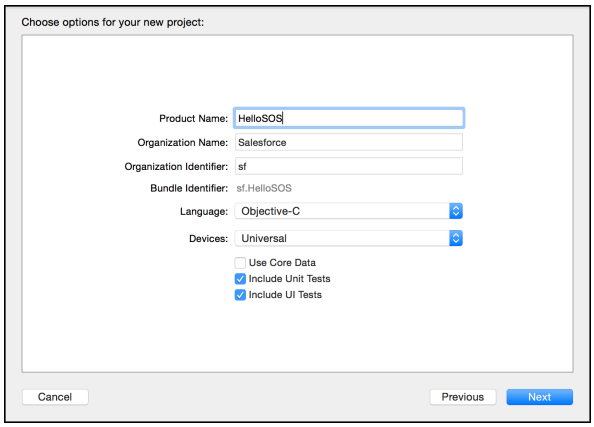
See for yourself how easy and effective live video chat and screen sharing can be.

Before doing this tutorial, be sure that you've set up Service Cloud for SOS. See Console Setup for SOS for more information.

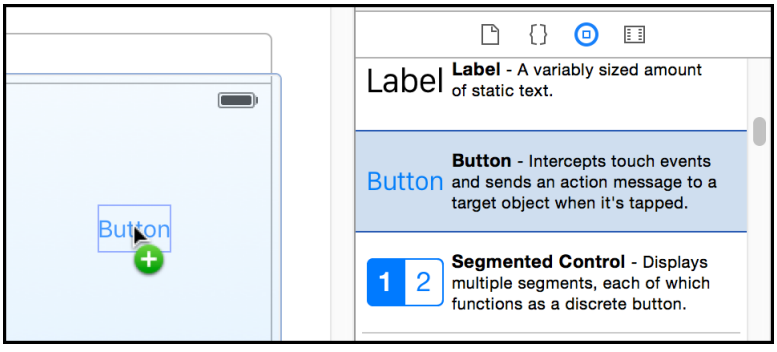
This tutorial shows you how to get SOS into your iOS app.

1. Create an Xcode project. For this example, let's make a Single View Application. Name it `HelloSOS`.

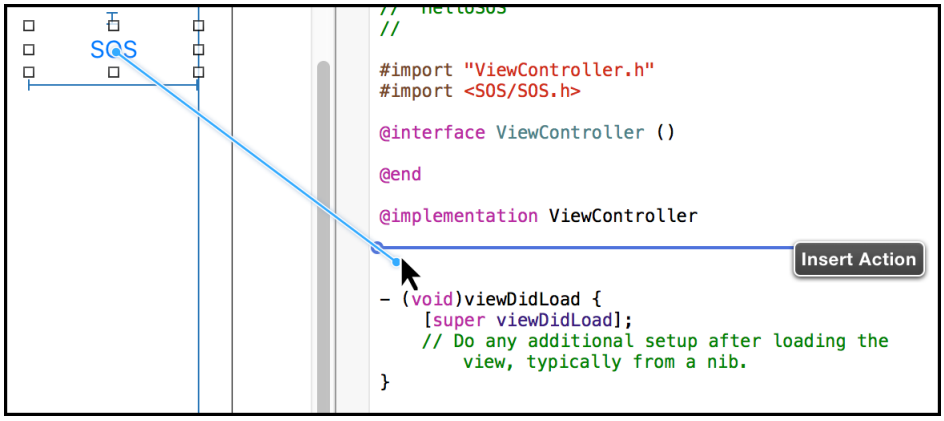


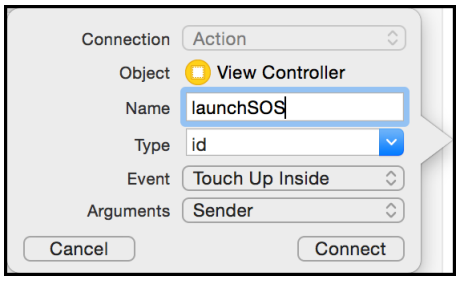


2. Install the SDK as described in Install the Service SDK for iOS.
3. Go to your storyboard and place a button somewhere on the view. Name it `SOS`.



4. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `launchSOS`.





5. Import the SDK. Wherever you intend to use the SOS SDK, be sure to import the Service Common framework and the SOS framework.

   In Swift:

```
import ServiceCore
import ServiceSOS
```

   In Objective-C:

```
@import ServiceCore;
@import ServiceSOS;
```

6. Launch an SOS session from within the `launchSOS` method.

   From the button action implementation, launch SOS using the `startSession` method on the `SOSSessionManager` shared instance.

   In Swift:

```
@IBAction func launchSOS(sender: AnyObject) {

  let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                           orgId: "YOUR-ORG-ID",
                           deploymentId: "YOUR-DEPLOYMENT-ID")

  ServiceCloud.shared().sos.startSession(with: options)
}
```

   In Objective-C:

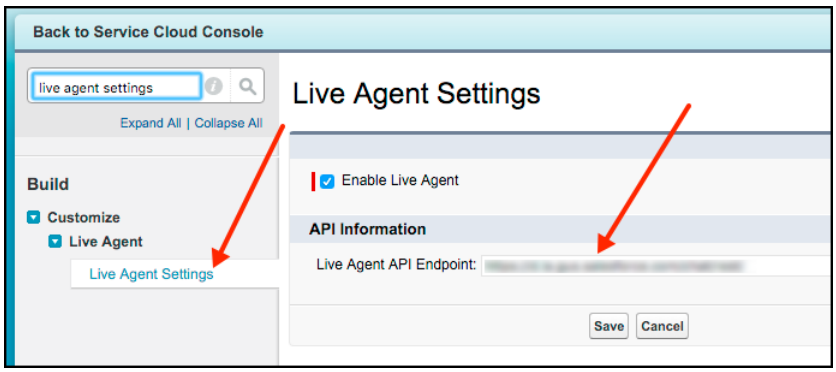
```
- (IBAction)launchSOS:(id)sender {

  SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
```

```
                                                    orgId:@"YOUR-ORG-ID"
                                              deploymentId:@"YOUR-DEPLOYMENT-ID"];

  [[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
}
```

Fill in the placeholder text for the Chat server, the org ID, and the deployment ID.

pod

The hostname for the Chat endpoint that your organization has been assigned. To get this value, from Setup, search for **Chat Settings** and copy the hostname from the **Chat API Endpoint**. Be sure not to include the protocol or the path — use only the hostname.



For instance, if your Chat API Endpoint is:

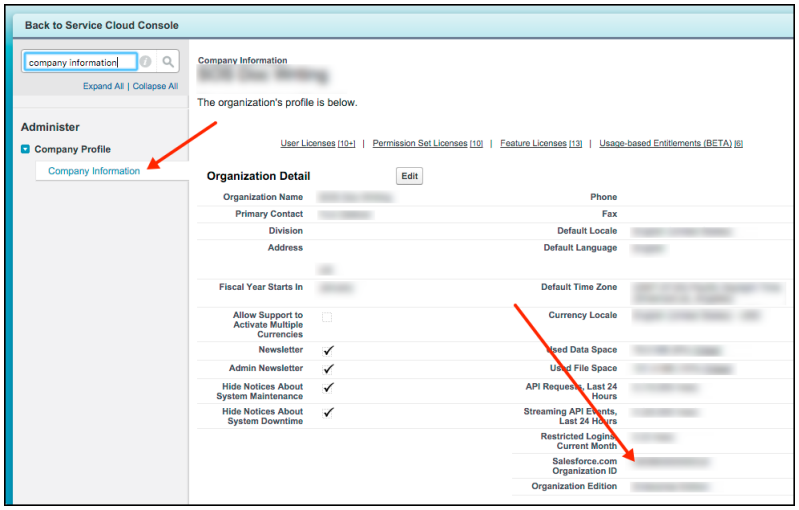
```
https://d.gla5.gus.salesforce.com/chat/rest/
```

Your pod hostname is:

```
d.gla5.gus.salesforce.com
```

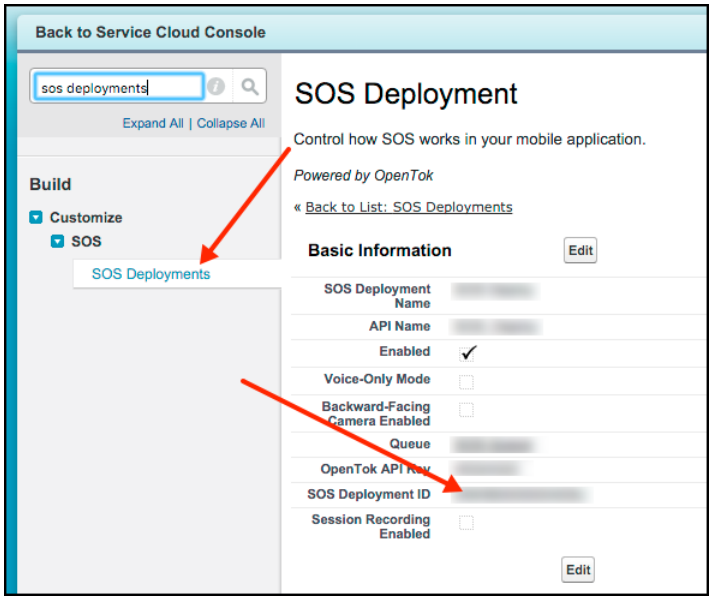
orgId

The Salesforce org ID. To get this value, from Setup, search for **Company Information** and copy the **Salesforce Organization ID**.

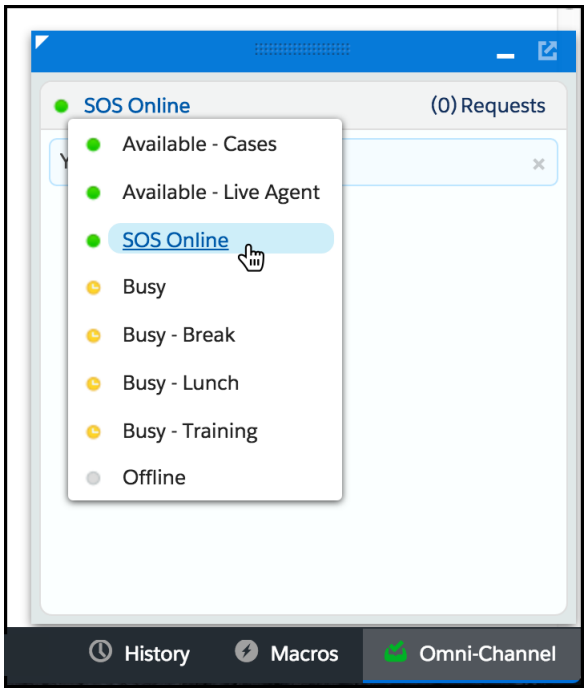


deploymentId

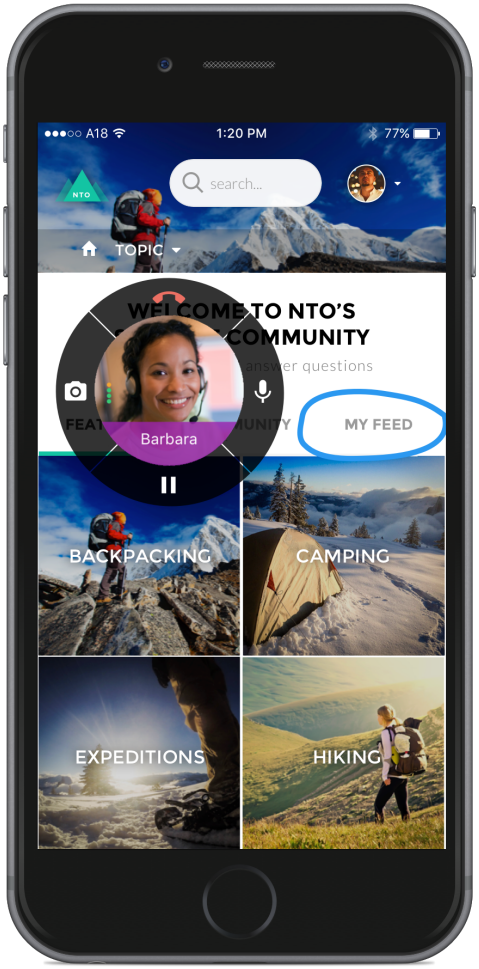
The unique ID of your SOS deployment. To get this value, from Setup, search for **SOS Deployments**, click the correct deployment and copy the **Deployment ID**.



7. Launch **Service Cloud Console**. From the **Omni-Channel** widget, ensure that an SOS agent is online.



Now you can build and run the app. When you tap the **SOS** button, the app requests an SOS session, which an agent can accept from the **Service Cloud Console**. From the console, you can chat with the customer, annotate things on their screen, and perform a two-way video session (if enabled).



Using Chat with the Service SDK

Add the Chat experience to your mobile app.

Chat in the Service SDK for iOS

Using Chat within the Service SDK, you can provide real-time chat sessions from within your native app.

Quick Setup: Chat in the Service SDK

To add Chat to your iOS app, create an `SCSChatConfiguration` object and pass it to the `showChat` method.

Use Einstein Bots with Chat

With Einstein Bots, you can complement your chat support experience with a smart, automated system that saves your agents time and keeps your customers happy. Once you've set up Einstein Bots in your org, the SDK automatically begins the chat experience using your bot. You can design your bot to transfer to an agent at any point.

Display Knowledge Article Previews in Chat

When agents send Knowledge article URLs in a chat session, users automatically see an article preview in the session. If they tap the article, the SDK opens the article details page.

Notifications for Chat Activity

If there's chat activity when the user is not viewing the chat session, you can present that information to them using the iOS notification system.

Configure a Chat Session

Before starting a chat session, you have several ways to configure the session using the `SCSChatConfiguration` object. These configuration settings allow you to specify pre-chat fields, determine whether a session starts minimized or full screen, and get updates about the user's queue position.

Listen for State Changes and Events

Implement `SCSChatSessionDelegate` to be notified about state changes made before, during, and after a chat session. This delegate also allows you to listen for error conditions so you can present alerts to the user when applicable.

Show Pre-Chat Fields to User

Before a chat session begins, you can request that the user enter pre-chat fields that are sent to the agent at the start of the session.

Create or Update Salesforce Records from a Chat Session

When a chat session begins, you can create or find records within your org and pass this information to the agent. Using this technique, your agent can immediately have all the context they need for an effective chat session.

Check Agent Availability

Before starting a session, you can check the availability of your chat agents and then provide your users with more accurate expectations.

Transfer File to Agent

Give users the ability to transfer files during a chat so they can share information about their issues.

Block Sensitive Data in a Chat Session

To block sending sensitive data to agents, specify a regular expression in your org's setup. When the regular expression matches text in the user's message, the matched text is replaced with customizable text before it leaves the device.

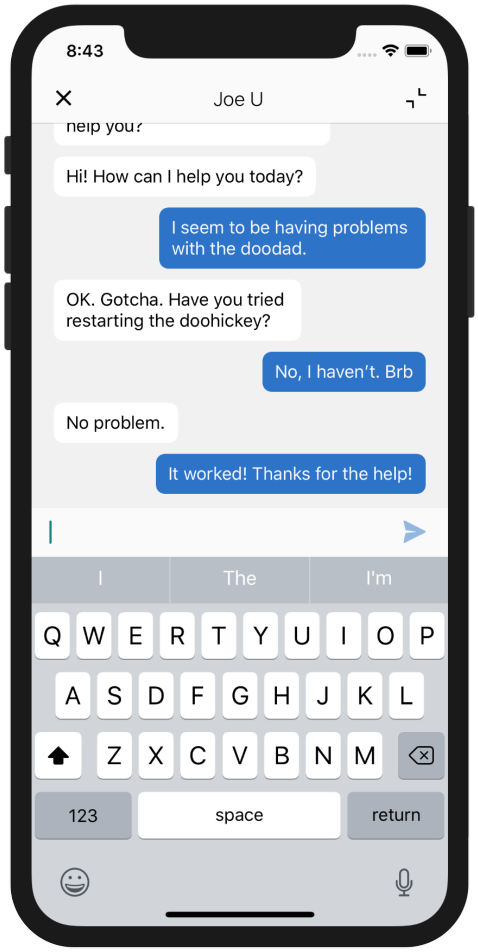
Build Your Own UI with the Chat Core API

With the Chat Core API, you can access the functionality of Chat without a UI. This API is useful if you want to build your own UI and not use the default.

Chat in the Service SDK for iOS

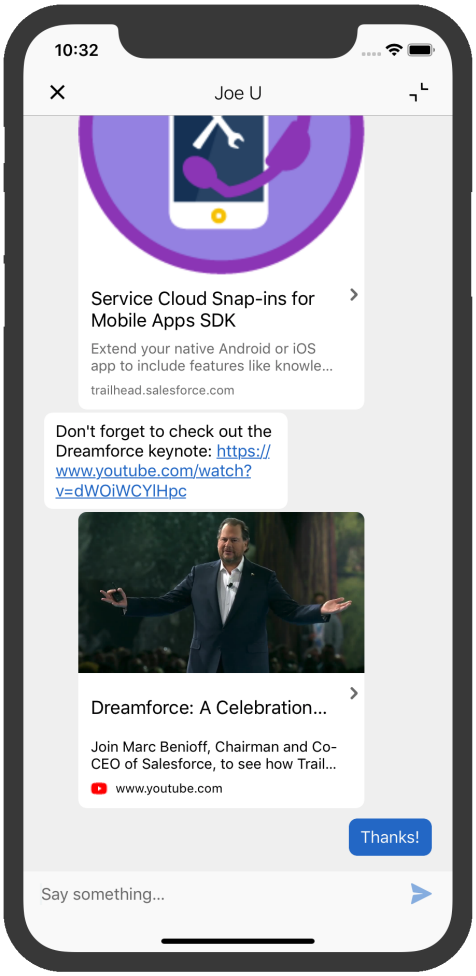
Using Chat within the Service SDK, you can provide real-time chat sessions from within your native app.

Once you've set up chat for Service Cloud, it takes just a few calls to the SDK to have your app ready to handle agent chat sessions.



This chat session can be minimized so that the user can continue to navigate from within the app while speaking with an agent. And if you've implemented Einstein Bots, the user first speaks with a chat bot on page 72 before optionally getting transferred to an agent.

When an agent sends links to your users, they see link previews right from within the chat session. The SDK tries to use Open Graph meta tags (`og:title`, `og:description`, `og:image`) to extract relevant information for the preview.



You can also customize the look and feel of the interface so that it fits naturally within your app. These customizations include the ability to fine-tune the colors, the fonts, the images, and the strings used throughout the interface.

Quick Setup: Chat in the Service SDK

To add Chat to your iOS app, create an `SCSChatConfiguration` object and pass it to the `showChat` method.

Before running through these steps, be sure you've already:

- Set up Service Cloud to work with Chat. To learn more, see Org Setup for Chat in Lightning Experience with a Guided Flow.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

Once you've reviewed these prerequisites, you're ready to begin.

These steps describe how to set up Chat with the default UI. If you prefer to build your own user interface, see Build Your Own UI with the Chat Core API.

1. Import the SDK. Wherever you intend to use the Chat SDK, be sure to import the ServiceCore framework and the ServiceChat framework.

In Swift:

```
import ServiceCore
import ServiceChat
```

In Objective-C:

```
@import ServiceCore;
@import ServiceChat;
```

2. Create an `SCSChatConfiguration` instance with information about your LiveAgent pod, your Salesforce org ID, the deployment ID, and the button ID.

   In Swift:

   ```
   let config = SCSChatConfiguration(liveAgentPod: "TO_DO_POD_NAME",
                                     // e.g. "d.gla5.gus.salesforce.com"
                                     orgId: "TO_DO_ORG_ID",
                                     // e.g. "00DB00000003Rxz"
                                     deploymentId: "TO_DO_DEPLOYMENT_ID",
                                     // e.g. "573B00000005KXz"
                                     buttonId: "TO_DO_BUTTON_ID")
                                     // e.g. "575C00000004h3m"
   ```

   In Objective-C:

   ```
   SCSChatConfiguration *config =
     [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"TO_DO_POD_NAME"
                                                  // e.g. "d.gla5.gus.salesforce.com"
                                                   orgId:@"TO_DO_ORG_ID"
                                                  // e.g. "00DB00000003Rxz"
                                          deploymentId:@"TO_DO_DEPLOYMENT_ID"
                                                  // e.g. "573B00000005KXz"
                                              buttonId:@"TO_DO_BUTTON_ID"];
                                                  // e.g. "575C00000004h3m"
   ```

   Note: You can get the required parameters for this method from your Salesforce org. See Get Chat Settings from Your Org. If your Salesforce admin hasn't already set up Chat in Service Cloud or you need more guidance, see Org Setup for Chat in Lightning Experience with a Guided Flow.

3. (Optional) Configure the visitor name, whether the user can minimize the chat session, and various other configuration settings.

   See Configure a Chat Session for more information.

4. (Optional) Customize the appearance with the configuration object.

   You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor`, `setFontDescriptor`, and `setImage`. You can also configure the strings used throughout the interface. See SDK Customizations with the Service SDK for iOS.

5. (Optional) Specify any pre-chat fields.

   You can specify both optional and required fields shown to the user before a chat session starts. You can also directly pass data to an agent without requiring any user input. These fields can be mapped directly to fields in a record in your org.

   See Show Pre-Chat Fields to User and Create or Update Salesforce Records from a Chat Session for more information.

6. To start a chat session, call the `showChat(with:showPrechat:)` method on `SCSChatInterface`.

In Swift:

```
ServiceCloud.shared().chatUI.showChat(with: config)
```

In Objective-C:

```
[[SCServiceCloud sharedInstance].chatUI showChatWithConfiguration:config];
```

You can provide an optional completion block to execute code when the session has been fully connected to all services. During a successful session initialization, the SDK calls the completion block at the point that the session is active and the user is waiting for an agent to join. If there is a failure, the SDK calls the completion block with the associated error.

For instructions on launching the interface from a web view, see Launch the Service SDK from a Web View in iOS.

7. Listen for events and handle error conditions.

   You can detect when a session ends by implementing the `session(didEnd:)` method on the `SCSChatSessionDelegate` delegate. Register this delegate using the `add(delegate:)` method on your `SCSChat` instance. In particular, we suggest that you handle the `.agent` reason (for when an agent ends a session) and the `.noAgentsAvailable` reason (for when there are no agents available). See Listen for State Changes and Events.

   Note: The SDK doesn't show an alert when a session fails to start, or when a session ends. It's your responsibility to listen to events and display an error when appropriate.

These steps embed the chat experience into your app.

Note: A chat session does not timeout due to user inactivity. However, if your app enters the background state while the user is in a chat session, that session will expire after 2-3 minutes due to the limitations of what an iOS app can do in the background. To prevent this timeout scenario, you can send a local notification to the user to bring them back to the app. To learn more, see Notifications for Chat Activity.

Example: To use this example code, create a Single View Application and Install the Service SDK for iOS.

Use the storyboard to add a button to the view. Add a `Touch Up Inside` action in your `UIViewController` implementation with the name `startChat`. In the view controller code:

- Implement the `SCSChatSessionDelegate` protocol so that you can be notified when there are errors or state changes.
- Specify `self` as a chat delegate.
- Start a chat session in the button action.
- Implement the `session(didEnd:)` method and show a dialog when appropriate.

In Swift:

```
import UIKit
import ServiceCore
import ServiceChat

class ViewController : UIViewController, SCSChatSessionDelegate {

  override func viewDidLoad() {
    super.viewDidLoad()

    // Add our chat delegate
    ServiceCloud.shared().chatCore.add(delegate: self)
  }

  @IBAction func startChat(_ sender: AnyObject) {
```

```
    // Create config object
    let config = SCSChatConfiguration(liveAgentPod: "YOUR-POD-NAME",
                                      orgId: "YOUR-ORG-ID",
                                      deploymentId: "YOUR-DEPLOYMENT-ID",
                                      buttonId: "YOUR-BUTTON-ID")

    // Start a session
    ServiceCloud.shared().chatUI.showChat(with: config!)
  }

  func session(_ session: SCSChatSession!, didEnd endEvent: SCSChatSessionEndEvent!)
{

    var description = ""

    // Here we'll handle the situation where the agent ends the session
    // and when there are no agents available...
    switch endEvent.reason {
    case .agent:
      description = "The agent has ended the session."
    case .noAgentsAvailable:
      description = "It looks like there are no agents available. Try again later."
    // TO DO: Handle other reasons
    default:
      description = "Session ended for an unknown reason."
    }

    let alert = UIAlertController(title: "Session Ended",
                                  message: description,
                                  preferredStyle: .alert)
    let okAction = UIAlertAction(title: "OK",
                                 style: .default,
                                 handler: nil)
    alert.addAction(okAction)
    self.present(alert, animated: true, completion: nil)
  }

  func session(_ session: SCSChatSession!, didError error: Error!, fatal: Bool) {
    // TO DO: Handle error condition
    NSLog("Chat error: \(error.localizedDescription)")
  }
}
```

In Objective-C:

```
#import "ViewController.h"
@import ServiceCore;
@import ServiceChat;

@interface ViewController : UIViewController <SCSChatSessionDelegate>

@end

@implementation ViewController
```

```
- (void)viewDidLoad {
  [super viewDidLoad];

  // Add our chat delegate
  [[SCServiceCloud sharedInstance].chatCore addDelegate:self];
}

- (IBAction)startChat:(id)sender {

  // Create config object
  SCSChatConfiguration *config =
  [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"YOUR-POD-NAME"
                                               orgId:@"YOUR-ORG-ID"
                                        deploymentId:@"YOUR-DEPLOYMENT-ID"
                                            buttonId:@"YOUR-BUTTON-ID"];
  // Start the session
  [[SCServiceCloud sharedInstance].chatUI showChatWithConfiguration:config];
}

- (void)session:(id<SCSChatSession>)session didEnd:(SCSChatSessionEndEvent *)endEvent
 {

  NSString *description = nil;

  // Here we'll handle the situation where the agent ends the session
  // and when there are no agents available...
  switch (endEvent.reason){
    case SCSChatEndReasonAgent:
      description = @"The agent has ended the session.";
      break;
    case SCSChatEndReasonNoAgentsAvailable:
      description = @"It looks like there are no agents available. Try again later.";

      break;
    // TO DO: Handle other reasons
    default:
      description = @"Session ended for an unknown reason.";
      break;
  }

  UIAlertController *alert = [UIAlertController
                              alertControllerWithTitle:@"Session Ended"
                              message:description
                              preferredStyle:UIAlertControllerStyleAlert];

  UIAlertAction* okAction = [UIAlertAction
                             actionWithTitle:@"OK"
                             style:UIAlertActionStyleDefault
                             handler:nil];

  [alert addAction:okAction];
  [self presentViewController:alert animated:YES completion:nil];
}
```

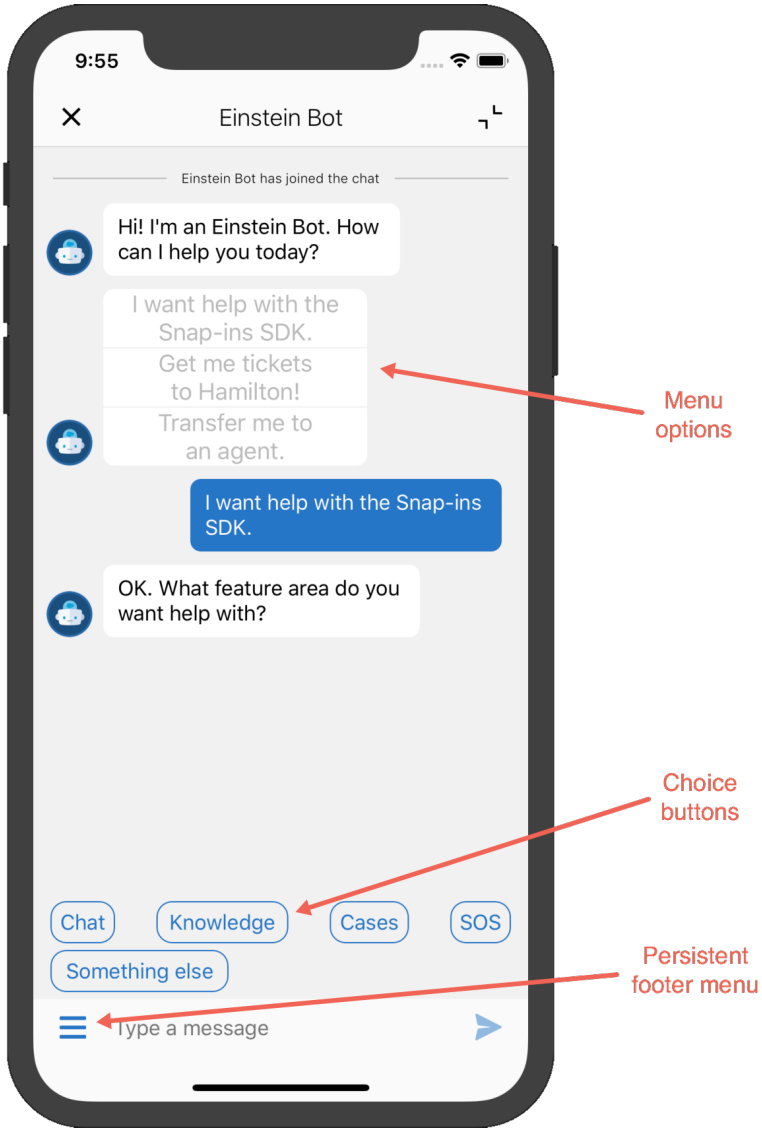
```
- (void)session:(id<SCSChatSession>)session didError:(NSError *)error fatal:(BOOL)fatal
 {
  // TO DO: Handle error condition
  NSLog(@"Chat error: \(error.localizedDescription)");
}

@end
```

Use Einstein Bots with Chat

With Einstein Bots, you can complement your chat support experience with a smart, automated system that saves your agents time and keeps your customers happy. Once you've set up Einstein Bots in your org, the SDK automatically begins the chat experience using your bot. You can design your bot to transfer to an agent at any point.

Before you can use Einstein Bots in your mobile app, enable and build a bot in your org. To learn more, see Einstein Bots in Salesforce Help. In broad strokes, you must enable Einstein Bots, deploy the bot to your channel, and activate the bot. If you want to learn about building a more robust bot, see the Einstein Bots Developer Cookbook.

Once you've set up your bot and assigned it to your chat button, a chat session automatically starts out as a bot. The menu options, choice buttons, and persistent footer menu that you designed for your bot all appear from within the mobile chat session. These features give your customers direct ways to get what they need—fast.

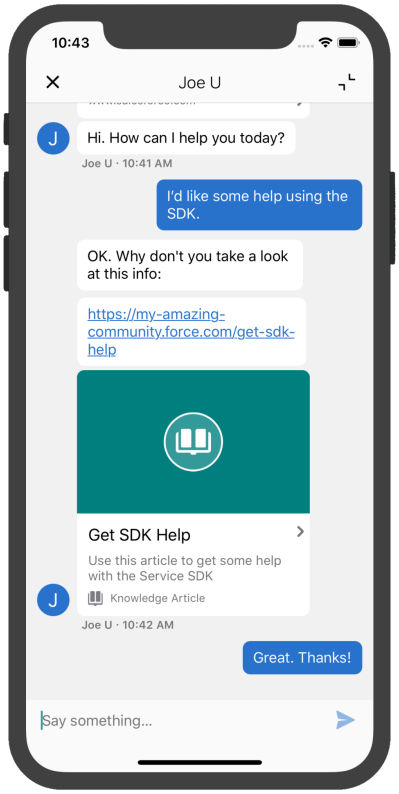


You do have a few ways you can fine-tune the bot from the SDK.

Feature Area	Details
Einstein Bot Avatar	Configure the bot avatar that displays during a session with a bot. To do this, use the `setImage` method with the `chatBotAvatar` enum value. To learn more, see Customize Images with the Service SDK.
Einstein Bot Footer Icon	Configure the "hamburger" icon in the text entry area that launches the persistent footer menu, which is always accessible to the user. To do this, use the `setImage` method with the `chatBotFooterMenu` enum value. To learn more, see Customize Images with the Service SDK.

Display Knowledge Article Previews in Chat

When agents send Knowledge article URLs in a chat session, users automatically see an article preview in the session. If they tap the article, the SDK opens the article details page.



If you have set up and configured the SDK for Knowledge, no coding is necessary for this feature to work.

Note: If the agent sends a link to an article that isn't part of your community's knowledge base, the SDK displays a generic URL preview.

Notifications for Chat Activity

If there's chat activity when the user is not viewing the chat session, you can present that information to them using the iOS notification system.

Notifications can be sent to the user when the user isn't viewing the chat session. A notification can appear when the app is in the background, or when the app is in the foreground but the chat session is minimized. The following activities can cause notifications:

- Agent has connected
- Agent sent a message
- Agent requested a file transfer
- Agent canceled a file transfer
- Agent ended a session
- Session will timeout soon

To ensure that the app can send these notifications while in the background, chat must be configured to allow background execution (`allowBackgroundExecution`) and to allow for background notifications (`allowBackgroundNotifications`). Both these settings are turned on by default. See Configure a Chat Session for details.

1. Import the `UserNotifications` framework in your `AppDelegate` class.

   In Swift:

   ```
   import UserNotifications
   ```

   In Objective-C:

   ```
   @import UserNotifications;
   ```

2. Register for local notifications in your `AppDelegate`'s `didFinishLaunchingWithOptions` method.

   In Swift:

   ```
   // Get the notification center object
   let center = UNUserNotificationCenter.current()

   // Register a delegate (see next step for delegate implementation)
   center.delegate = self

   // Request authorization
   center.requestAuthorization(options: [.alert,.sound],
                               completionHandler: { granted, error in
     // Enable or disable features based on authorization
   })

   // Create general category
   let generalCategory = UNNotificationCategory(identifier: "General", actions: [],
   intentIdentifiers: [], options: .customDismissAction)
   let categorySet: Set<UNNotificationCategory> = [generalCategory]

   // Set category
   center.setNotificationCategories(categorySet)
   ```

   In Objective-C:

   ```
   // Get the notification center object
   UNUserNotificationCenter* center = [UNUserNotificationCenter currentNotificationCenter];

   // Register a delegate (see next step for delegate implementation)
   [center setDelegate:self];

   // Request authorization
   [center requestAuthorizationWithOptions:(UNAuthorizationOptionAlert +
   UNAuthorizationOptionSound)
           completionHandler:^(BOOL granted, NSError * _Nullable error) {
     // Enable or disable features based on authorization
   }];

   // Create general category
   UNNotificationCategory* generalCategory =
     [UNNotificationCategory
       categoryWithIdentifier:@"GENERAL"
                      actions:@[]
            intentIdentifiers:@[]
                      options:UNNotificationCategoryOptionCustomDismissAction];
   ```

```
// Set category
[center setNotificationCategories:[NSSet setWithObjects:generalCategory, nil]];
```

3. Implement `UNUserNotificationCenterDelegate`. Handle the `didReceiveNotificationResponse` and `willPresentNotification` methods.

   iOS calls the `didReceiveNotificationResponse` method when your app is in the background. In this method, you can tell your app to enter the foreground and for chat to maximize. To perform this behavior, use the `handle(notification:)` method on the `SCSChatInterface` object, which is accessible from the `chatUI` property of the `ServiceCloud` shared instance.

   iOS calls the `willPresentNotification` method when your app is in the foreground. To determine whether to display the notification, use the `shouldDisplayNotificationInForeground` method (accessible from the `chatUI` property of the `ServiceCloud` shared instance). If the Chat UI is already showing the relevant information related to this event, the method returns `false`.

   In Swift:

```
/**
 This delegate method is executed when the application launches as a result
 of the user interacting with a notification when it is in the background.
 The result of passing the notification to Chat is that we will
 maximize if the notification was scheduled as a result of a chat event.
*/
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                            withCompletionHandler completionHandler:
                            @escaping () -> Void) {

  let chat = ServiceCloud.shared().chatUI!
  chat.handle(response.notification)
}

/**
 This delegate method is executed when a notification is received while
 the app is in the foreground. Check with Chat to see whether it's
 appropriate to display the notification (or if the notification
 relates to information already being shown to the user).
*/
func userNotificationCenter(_ center: UNUserNotificationCenter,
                            willPresent notification: UNNotification,
                            withCompletionHandler completionHandler:
                            @escaping (UNNotificationPresentationOptions) -> Void) {

  let chat = ServiceCloud.shared().chatUI!

  // Show we display this notification?
  if (chat.shouldDisplayNotificationInForeground()) {

    // Display notification as an alert
    completionHandler(.alert)
  }
}
```

In Objective-C:

```
/**
 This delegate method is executed when the application launches as a result
 of the user interacting with a notification when it is in the background.
 The result of passing the notification to Chat is that we will
 maximize if the notification was scheduled as a result of a chat event.
*/
- (void)userNotificationCenter:(UNUserNotificationCenter *)center
didReceiveNotificationResponse:(UNNotificationResponse *)response
         withCompletionHandler:(void(^)(void))completionHandler {

  SCSChatInterface *chat = [SCServiceCloud sharedInstance].chatUI;
  [chat handleNotification:response.notification];
}

/**
 This delegate method is executed when a notification is received while
 the app is in the foreground. Check with Chat to see whether it's
 appropriate to display the notification (or if the notification
 relates to information already being shown to the user).
*/
- (void)userNotificationCenter:(UNUserNotificationCenter *)center
       willPresentNotification:(UNNotification *)notification
         withCompletionHandler:
           (void (^)(UNNotificationPresentationOptions options))completionHandler {

  SCSChatInterface *chat = [SCServiceCloud sharedInstance].chatUI;

  // Show we display this notification?
  if ([chat shouldDisplayNotificationInForeground]) {

    // Display notification as an alert
    completionHandler(UNNotificationPresentationOptionAlert);
  }
}
```

Configure a Chat Session

Before starting a chat session, you have several ways to configure the session using the `SCSChatConfiguration` object. These configuration settings allow you to specify pre-chat fields, determine whether a session starts minimized or full screen, and get updates about the user's queue position.

When you start a chat session, you specify an `SCSChatConfiguration` object as one of the arguments. This object contains all the configuration settings necessary for Chat to start a session. To create an `SCSChatConfiguration` object, you specify information about your org and deployment.

In Swift:

```
let config = SCSChatConfiguration(liveAgentPod: "TO_DO_POD_NAME",
                                  // e.g. "d.gla5.gus.salesforce.com"
                                  orgId: "TO_DO_ORG_ID",
                                  // e.g. "00DB00000003Rxz"
                                  deploymentId: "TO_DO_DEPLOYMENT_ID",
```

```
                                    // e.g. "573B00000005KXz"
                                    buttonId: "TO_DO_BUTTON_ID")
                                    // e.g. "575C00000004h3m"
```

In Objective-C:

```
SCSChatConfiguration *config =
  [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"TO_DO_POD_NAME"
                                                // e.g. "d.gla5.gus.salesforce.com"
                                                 orgId:@"TO_DO_ORG_ID"
                                                // e.g. "00DB00000003Rxz"
                                          deploymentId:@"TO_DO_DEPLOYMENT_ID"
                                                // e.g. "573B00000005KXz"
                                              buttonId:@"TO_DO_BUTTON_ID"];
                                                // e.g. "575C00000004h3m"
```

Note: You can get the required parameters for this method from your Salesforce org. See Get Chat Settings from Your Org. If your Salesforce admin hasn't already set up Chat in Service Cloud or you need more guidance, see Org Setup for Chat in Lightning Experience with a Guided Flow.

However, there are other options you can set using `SCSChatConfiguration` at configuration time.

The following features are available for configuration:

Table 4: Session Display Configuration Settings

Property Name	Description	Type & Default Value
`allowMinimization`	Indicates whether the user is allowed to minimize the chat session view.	`Bool`: `true`/`YES`
`allowURLPreview`	Indicates whether the user is shown URL previews when the agent types a URL in the chat feed.	`Bool`: `true`/`YES`
`defaultToMinimized`	Indicates whether the chat session starts out as a minimized thumbnail view.	`Bool`: `true`/`YES`

Table 5: Background Configuration Settings

Property Name	Description	Type & Default Value
`allowBackgroundExecution`	Indicates whether to allow extended background execution to support active chat sessions. When `true`, active chat sessions can remain in the background for more than three minutes. See `allowBackgroundNotifications` for related functionality.	`Bool`: `true`/`YES`
`allowBackgroundNotifications`	Indicates whether the session posts local notifications based on chat activity. Requires that `allowBackgroundExecution` is also set to `true`. To learn more, see Notifications for Chat Activity.	`Bool`: `true`/`YES`

Table 6: Pre-Chat Configuration Settings

Property Name	Description	Type & Default Value
`prechatEntities`	Pre-chat fields are always sent to the agent at the start of the session. But if you want to fill in fields of a particular record, instantiate an `SCSPrechatEntity` for each Salesforce object (for example, `Case` or `Contact`) and instantiate an `SCSPrechatEntityField` for each field association within that Salesforce object (for example, `Subject` or `LastName`). To learn more, see Create or Update Salesforce Records from a Chat Session.	`SCSPrechatEntity` array: `nil`
`prechatFields`	You can specify both optional and required fields shown to the user before a chat session starts. You can also directly pass data to an agent without requiring any user input. To create pre-chat fields, add `SCSPrechatObject` instances to the `prechatFields` property on the `SCSChatConfiguration` object. To learn more, see Show Pre-Chat Fields to User.	`SCSPrechatObject` array: `nil`

Table 7: Other Configuration Settings

Property Name	Description	Type & Default Value
`queueUpdatesEnabled`	Determines whether the framework receives and displays updates about the session queue position. If `true`, the queue position is shown in the UI during the `Queued` state. You can also subscribe to queue position events using `session(didUpdateQueuePosition:)` on `SCSChatSessionDelegate`. Use the `add(delegate:)` method on `SCSChat` to register your delegate. The queue position is 0 if the agent capacity is greater than or equal to the number of customer requests. Otherwise, the position value represents how far the customer is from getting served by an agent. `q = max(n - c, 0)` Where: • q is the queue position • n is the position of the customer compared to all waiting customers • c is the total capacity of all agents For example, if the total capacity is 10, the first 10 waiting visitors have a position of 0, the 11th has a position of 1, the 12th has a position of 2, and so on.	`BOOL`: `true/YES`

Property Name	Description	Type & Default Value
remoteLoggingEnabled	Indicates whether session logs are sent for collection. (Logs sent remotely don't collect personal information. Unique IDs are created for tying logs to sessions and those IDs can't be correlated back to specific users.)	`Bool`: `true`/`YES`
`visitorName`	Name of the chat visitor. This value is used by the Service Cloud console and displayed to the agent.	`String`: "`Visitor`"

Once you've fully configured the `SCSChatConfiguration` object, you can start the session using the `startSession` method.

Example: The following example configures a session with one pre-chat field and a visitor name "Jane Doe".

In Swift:

```
let config = SCSChatConfiguration(liveAgentPod: "YOUR_POD_NAME",
                                  orgId: "YOUR_ORG_ID",
                                  deploymentId: "YOUR_DEPLOYMENT_ID",
                                  buttonId: "YOUR_BUTTON_ID")

// Set the visitor name
config?.visitorName = "Jane Doe"

// Add a required email field (with an email keyboard and no auto-correction)
let emailField = SCSPrechatTextInputObject(label: "Email")
emailField?.isRequired = true
emailField?.keyboardType = .emailAddress
emailField?.autocorrectionType = .no
config?.prechatFields.add(emailField)
```

In Objective-C:

```
SCSChatConfiguration *config =
    [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"YOUR_POD_NAME"
                                                 orgId:@"YOUR_ORG_ID"
                                          deploymentId:@"YOUR_DEPLOYMENT_ID"
                                              buttonId:@"YOUR_BUTTON_ID"];

// Set the visitor name
config.visitorName = @"Jane Doe";

// Add a required email field (with an email keyboard and no auto-correction)
SCSPrechatTextInputObject* emailField = [[SCSPrechatTextInputObject alloc]
                                        initWithLabel:@"Email"];
emailField.required = YES;
emailField.keyboardType = UIKeyboardTypeEmailAddress;
emailField.autocorrectionType = UITextAutocorrectionTypeNo;
[config.prechatFields addObject:emailField];
```

Listen for State Changes and Events

Implement `SCSChatSessionDelegate` to be notified about state changes made before, during, and after a chat session. This delegate also allows you to listen for error conditions so you can present alerts to the user when applicable.

Listening to State Changes

A chat session can be in one of the following states:

Inactive
No active session.

Connecting
A connection with chat servers is being established.

Queued
A connection has been established, and is now in the queue for next available agent.

Connected
Connected with an agent .

Ending
Session is cleaning up the connection at the end of a session.

These states are defined in `SCSChatSessionState`.

Throughout a session, your application might want to know the current state. You can monitor state changes by implementing `SCSChatSessionDelegate`. Use the `add(delegate:)` method on `SCSChat` to register your delegate. Use the `session(didTransitionFrom:to:)` method to listen for state changes.

Handling Session Termination and Error Conditions

The SDK doesn't present UI alerts for session termination or error conditions so you'll need to listen for these events and decide what to show your users. There are two `SCSChatSessionDelegate` methods for this purpose:

1. To track session termination, use the `session(didEnd:)` method. Inspect the reason (`SCSChatSessionEndEvent.type`) to determine why the session stopped. Use the `SCSChatSession` object to get the session ID and other session information.
2. You can track errors with the `session(didError:fatal:)` method. Compare the error code to `SCSChatErrorCode` to determine what kind of error occurred.

Example: This sample code does the following:

- Implements the `SCSChatSessionDelegate` protocol.
- Implements the `session(didTransitionFrom:to:)` method to listen for state changes.
- Implements the `session(didEnd:)` method and logs a few possible end reasons.
- Implements the `session(didError:fatal:)` method to listen for errors.

```
import UIKit
import ServiceCore
import ServiceChat

class MyChatSessionDelegateImplementation: NSObject, SCSChatSessionDelegate {

  // TO DO: Register this delegate using
  //        ServiceCloud.shared().chatCore.add(delegate: myDelegate)
```

```
  /**
   Delegate method to handle state change.
   */
  func session(_ session: SCSChatSession!,
               didTransitionFrom previous: SCSChatSessionState,
               to current: SCSChatSessionState) {

    NSLog("Chat state changed...")

    switch current {
      case .connecting:
        NSLog("Chat now connecting...")
      case .connected:
        NSLog("Chat connected...")
      // TO DO: Handle other reasons
      default:
        break
      }
  }

  /**
   Delegate method for session stop event.
   */
  func session(_ session: SCSChatSession!, didEnd endEvent: SCSChatSessionEndEvent!)
{

    var reason = "Unknown"

    switch endEvent.reason {
      case .agent:
        reason = "The agent has ended the session."
      case .noAgentsAvailable:
        reason = "No agents were available."
      default:
        // TO DO: Handle other reasons
        break
    }

    NSLog("\nChat End Session. Reason: \(reason)")

    // You can access the session ID from the SCSChatSession object
    let sessionId = session.sessionId
  }

  /**
   Delegate method for error conditions.
   */
  func session(_ session: SCSChatSession!, didError error: Error!, fatal: Bool) {
    // TO DO: Handle error condition
    NSLog("Chat error: \(error.localizedDescription)")
  }
}
```

Show Pre-Chat Fields to User

Before a chat session begins, you can request that the user enter pre-chat fields that are sent to the agent at the start of the session.

To create pre-chat fields, add `SCSPrechatObject` instances to the `prechatFields` property on the `SCSChatConfiguration` object.

1. Create an `SCSChatConfiguration` object.

   In Swift:

   ```
   let config = SCSChatConfiguration(liveAgentPod: "TO_DO_POD_NAME",
                                     // e.g. "d.gla5.gus.salesforce.com"
                                     orgId: "TO_DO_ORG_ID",
                                     // e.g. "00DB00000003Rxz"
                                     deploymentId: "TO_DO_DEPLOYMENT_ID",
                                     // e.g. "573B00000005KXz"
                                     buttonId: "TO_DO_BUTTON_ID")
                                     // e.g. "575C00000004h3m"
   ```

   In Objective-C:

   ```
   SCSChatConfiguration *config =
     [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"TO_DO_POD_NAME"
                                                // e.g. "d.gla5.gus.salesforce.com"
                                                 orgId:@"TO_DO_ORG_ID"
                                                // e.g. "00DB00000003Rxz"
                                           deploymentId:@"TO_DO_DEPLOYMENT_ID"
                                                // e.g. "573B00000005KXz"
                                               buttonId:@"TO_DO_BUTTON_ID"];
                                                // e.g. "575C00000004h3m"
   ```

   See Configure a Chat Session on how to configure a chat session.

2. Create `SCSPrechatObject` objects for the pre-chat fields you want to specify in your app.

   There are several types of pre-chat fields:

   - `SCSPrechatObject` does not require user input and can be used to send custom data directly to the agent.
   - `SCSPrechatTextInputObject` (a subclass of `SCSPrechatObject`) takes user input from a text field.
   - `SCSPrechatPickerObject` (a subclass of `SCSPrechatObject`) provides the user with a dropdown list of options.

   Each type has different properties you can configure.

   a. Create objects that don't require user input using `SCSPrechatObject`.

      In Swift:

      ```
      let customData = SCSPrechatObject(label: "CustomEmailField",
                                        value: "lauren@example.com")
      ```

      In Objective-C:

      ```
      SCSPrechatObject* customData = [[SCSPrechatObject alloc]
                                        initWithLabel:@"CustomEmailField"
                                        value:@"lauren@example.com"];
      ```

      When using `SCSPrechatObject` to send data without user input, specify both the label and the value. The `SCSPrechatObject` base class contains the following properties:

- `label`—name of the pre-chat field shown to agent.
- `value`—value of the pre-chat field; only use this property if you don't intend for the user to fill in this field.
- `displayLabel`—optional display name of the pre-chat field shown to the user if different than the label.
- `transcriptFields`—optional array of field identifiers on the `LiveAgentChatTranscript` object in Salesforce.
- `displayToAgent`—indicates whether this pre-chat detail is shown to an agent accepting the chat session; defaults to `true`.

b. Create text input objects using `SCSPrechatTextInputObject`.

In Swift:

```
// Create a text field
let myPrechatField = SCSPrechatTextInputObject(label: "Full Name")
```

In Objective-C:

```
// Create a text field
SCSPrechatTextInputObject* myPrechatField = [[SCSPrechatTextInputObject alloc]
  initWithLabel:@"Full Name"];
```

When using a `SCSPrechatTextInputObject`, you can control several other properties:

- `required`—indicates whether the field is required.
- `keyboardType`—provides access to other standard keyboards (such as `UIKeyboardTypeEmailAddress`).
- `autocapitalizationType`—controls how text capitalization works.
- `autocorrectionType`—controls auto-correction behavior.
- `maxLength`—specifies the maximum length of the field
- `displayLabel`—optional display name of the pre-chat field shown to the user if different than the label.
- `transcriptFields`—optional array of field identifiers on the `LiveAgentChatTranscript` object in Salesforce.
- `displayToAgent`—indicates whether this pre-chat detail is shown to an agent accepting the chat session; defaults to `true`.

c. Create dropdown lists using `SCSPrechatPickerObject`.

In Swift:

```
// Create dropdown choices
let statusOptions = NSMutableArray()
statusOptions.add(SCSPrechatPickerOption(label:"New Issue", value:"New"))
statusOptions.add(SCSPrechatPickerOption(label:"Fixed Issue", value:"Fixed"))

// Create a dropdown list with choices
let statusPickerField = SCSPrechatPickerObject(
      label: "Issue Type",
      options: statusOptions as! [SCSPrechatPickerOption])
```

In Objective-C:

```
// Create a dropdown list with two choices
NSMutableArray *statusOptions = [[NSMutableArray alloc] init];
[statusOptions addObject: [[SCSPrechatPickerOption alloc]
                          initWithLabel:@"New Issue" value:@"New"]];
[statusOptions addObject: [[SCSPrechatPickerOption alloc]
                          initWithLabel:@"Fixed Issue" value:@"Fixed"]];
```

```
// Create a dropdown list with choices
SCSPrechatPickerObject *picker = [[SCSPrechatPickerObject alloc]
                                  initWithLabel:@"Issue Type" options:statusOptions];
```

When using a `SCSPrechatPickerObject`, you can access these properties:

- `required`—indicates whether the field is required.
- `options`—specifies items in the dropdown list. This property is an array of `SCSPrechatPickerOption` objects.
- `displayLabel`—optional display name of the pre-chat field shown to the user if different than the label.
- `transcriptFields`—optional array of field identifiers on the `LiveAgentChatTranscript` object in Salesforce.
- `displayToAgent`—indicates whether this pre-chat detail is shown to an agent accepting the chat session; defaults to `true`.

3. (Optional) Create `SCSPrechatEntity` objects to associate pre-chat fields with fields from a record in your org.

   Pre-chat fields are always sent to the agent at the start of the session. But if you want to fill in fields of a particular record, instantiate an `SCSPrechatEntity` for each Salesforce object (for example, `Case` or `Contact`) and instantiate an `SCSPrechatEntityField` for each field association within that Salesforce object (for example, `Subject` or `LastName`). To learn more, see Create or Update Salesforce Records from a Chat Session.

4. Update the config object's `prechatFields` property with an array of your pre-chat objects.

   In Swift:

   ```
   // Add the array of pre-chat fields to the config object
   config?.prechatFields = myPrechatFields
   ```

   In Objective-C:

   ```
   // Add the array of pre-chat fields to the config object
   config.prechatFields = myPrechatFields;
   ```

5. Show the pre-chat form and start a chat session by calling `showChat(with:showPrechat:)` and specify `true` for whether to show the pre-chat form.

   a. If you want to show the pre-chat form and then send those results to the agent when starting a session, call `showChat(with:showPrechat:)`.

      In Swift:

      ```
      // Show the pre-chat form and start a session
      ServiceCloud.shared().chatUI.showChat(with: config, showPrechat: true)
      ```

      In Objective-C:

      ```
      // Show the pre-chat form and start a session
      [[SCServiceCloud sharedInstance].chatUI showChatWithConfiguration:config showPrechat:
       YES];
      ```

   b. If you want to programmatically change the pre-chat data before passing it to the org or agent, call the `showPrechat(withFields:modal:completion:)` method first. From within the completion block of this method, update the config object's `prechatFields` property and then call `showChat(with:showPrechat:)` and specify `false` for `showPrechat`.

For example, the following Swift code asks the user for their first and last name. After the user completes the pre-chat form, the code creates a new field by concatenating the two name fields. When we start the chat session, only this new field is sent to the agent.

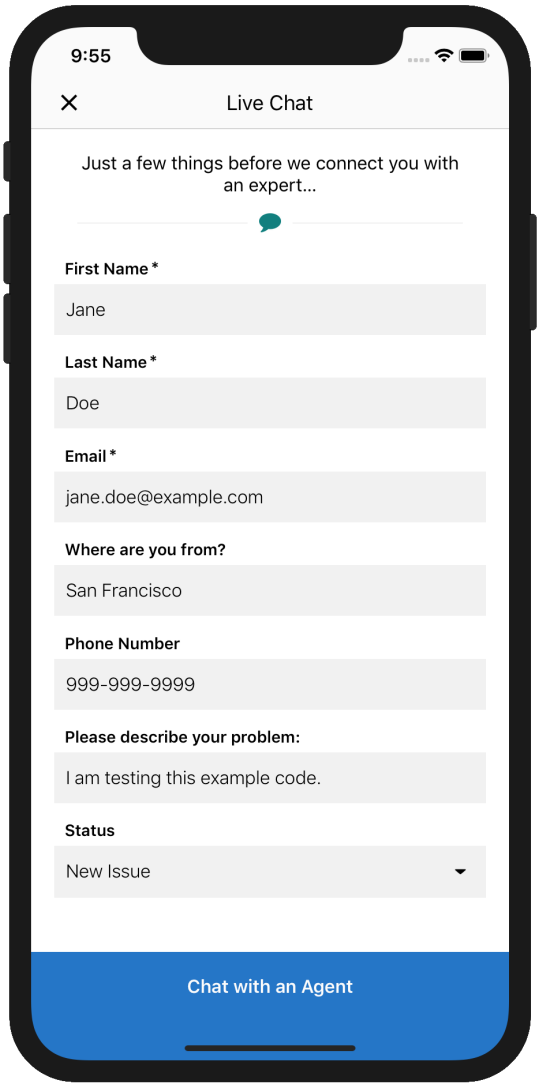
```
// Create two text fields
let firstNameField = SCSPrechatTextInputObject(label: "First Name")
let lastNameField = SCSPrechatTextInputObject(label: "Last Name")
let prechatInputFields = [firstNameField, lastNameField] as? [SCSPrechatObject]

// Show the pre-chat form...
ServiceCloud.shared().chatUI.showPrechat(withFields: prechatInputFields, completion:
 {
  (prechatResultFields, completed) in

  // If the pre-chat form completed successfully...
  if (completed && prechatResultFields != nil && prechatResultFields!.count >= 2) {

    // Create the full name from the values of the two pre-chat name fields
   let fullName = prechatResultFields![0].value + " " + prechatResultFields![1].value


    // Create a full name field
    let fullNameField = SCSPrechatObject(label: "Full Name", value: fullName)

    // Add this new field to the config object
    // (We DON'T include the original pre-chat fields because we don't need to send
them...)
    config!.prechatFields = [fullNameField]

    // And now start a chat session (without showing pre-chat)
    ServiceCloud.shared().chatUI.showChat(with: config!)

  } else {
    // TO DO: Handle the scenario where the user cancels out of the pre-chat form
  }
})
```

Example: This code sample builds a set of pre-chat fields that are shown to the user.

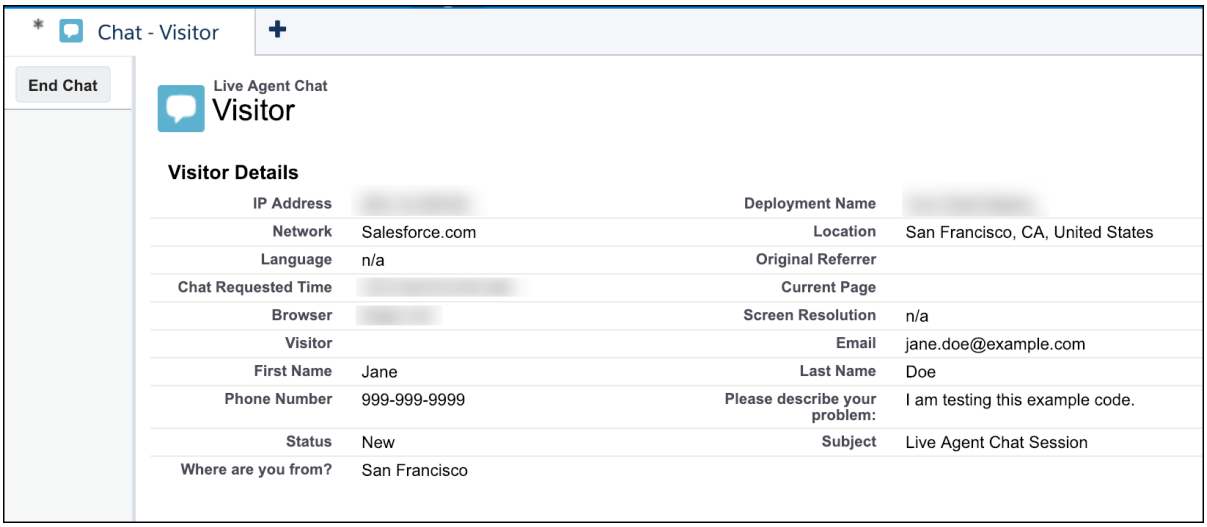
```
let config = SCSChatConfiguration(liveAgentPod: "YOUR-POD-NAME",
                                  orgId: "YOUR-ORG-ID",
                                  deploymentId: "YOUR-DEPLOYMENT-ID",
                                  buttonId: "YOUR-BUTTON-ID")

// Add some required fields
let firstNameField = SCSPrechatTextInputObject(label: "First Name")
firstNameField!.isRequired = true
let lastNameField = SCSPrechatTextInputObject(label: "Last Name")
lastNameField!.isRequired = true
let emailField = SCSPrechatTextInputObject(label: "Email")
emailField!.isRequired = true
emailField!.keyboardType = .emailAddress
emailField!.autocorrectionType = .no
```

```
// Add some optional fields
let originField = SCSPrechatTextInputObject(label: "Where are you from?")
originField!.isRequired = false
let phoneField = SCSPrechatTextInputObject(label: "Phone Number")
phoneField!.isRequired = false
phoneField!.keyboardType = .phonePad
let descriptionField = SCSPrechatTextInputObject(label: "Please describe your problem:")
descriptionField!.isRequired = false

// Add a picklist field
let statusOptions = NSMutableArray()
statusOptions.add(SCSPrechatPickerOption(label:"New Issue", value:"New"))
statusOptions.add(SCSPrechatPickerOption(label:"Fixed Issue", value:"Fixed"))
let statusPickerField = SCSPrechatPickerObject(label: "Status",
options: statusOptions as NSArray as! [SCSPrechatPickerOption])
statusPickerField!.isRequired = false

// Add hidden field containing the subject
let subjectField = SCSPrechatObject(label: "Subject", value: "Chat Session")

// Create an array of all pre-chat fields
let prechatFields: [SCSPrechatObject] = [firstNameField!, lastNameField!, emailField!,

                                         originField!, phoneField!,
descriptionField!,
                                         statusPickerField!, subjectField]

// Add the array of pre-chat fields to the config object
config?.prechatFields = prechatFields

// And NOW show the chat UI
ServiceCloud.shared().chatUI.showChat(with: config!, showPrechat: true)
```

With this code, the user sees the following pre-chat UI in their mobile app.



And the agent sees the following UI from the console.



Create or Update Salesforce Records from a Chat Session

When a chat session begins, you can create or find records within your org and pass this information to the agent. Using this technique, your agent can immediately have all the context they need for an effective chat session.

Overview

Before reading these instructions, review Show Pre-Chat Fields to User to understand how to create pre-chat fields.

Pre-chat fields are always sent to the agent at the start of the session. But if you want to fill in fields of a particular record, instantiate an `SCSPrechatEntity` for each Salesforce object (for example, `Case` or `Contact`) and instantiate an `SCSPrechatEntityField` for each field association within that Salesforce object (for example, `Subject` or `LastName`).

Each pre-chat entity must map to a pre-chat object (`SCSPrechatObject`, `SCSPrechatTextInputObject`, or `SCSPrechatPickerObject`). The label string in this pre-chat object must be identical to the label used in your `SCSPrechatEntityField` object.

Use the config object's `prechatFields` property for the array of your pre-chat objects and the `prechatEntities` property for the array of your entity objects.

Note: Case creation does not currently work for Omni-Channel routing without a setup change to your org. To resolve this problem, raise a ticket with Salesforce to ensure that Omni-Channel is enabled to create a Case in your org.

Basic Flow

This sample code shows how to pass the first and last name to a contact record in your org. This example doesn't involve user input, but you can use `SCSPrechatTextInputObject` instead of `SCSPrechatObject` to allow user input.

In Swift:

```
// Create pre-chat fields
let firstNameField = SCSPrechatObject(label: "First Name", value: "Jane")
let lastNameField = SCSPrechatObject(label: "Last Name", value: "Doe")

// Create entity fields
let firstNameEntityField =
  SCSPrechatEntityField(fieldName: "FirstName", label: "First Name")
firstNameEntityField.doFind = true
firstNameEntityField.doCreate = true
let lastNameEntityField =
  SCSPrechatEntityField(fieldName: "LastName", label: "Last Name")
lastNameEntityField.doFind = true
lastNameEntityField.doCreate = true

// Create an entity object
let contactEntity =
  SCSPrechatEntity(entityName: "Contact")
contactEntity.showOnCreate = true

// Add entity fields to entity object
contactEntity.entityFieldsMaps.add(firstNameEntityField)
contactEntity.entityFieldsMaps.add(lastNameEntityField)

// Update config object with the pre-chat fields
config!.prechatFields = [firstNameField, lastNameField]
```

```
// Update config object with the entity mappings
config!.prechatEntities = [contactEntity]
```

In Objective-C:

```
// Create pre-chat fields
SCSPrechatObject* firstNameField = [[SCSPrechatObject alloc]
                                    initWithLabel:@"First Name"
                                    value:@"Banana"];
SCSPrechatObject* lastNameField = [[SCSPrechatObject alloc]
                                   initWithLabel:@"Last Name"
                                   value:@"Town"];

// Create entity fields
SCSPrechatEntityField* firstNameEntityField =
  [[SCSPrechatEntityField alloc]
  initWithFieldName:@"FirstName" label:@"First
Name"];
firstNameEntityField.doFind = YES;
firstNameEntityField.doCreate = YES;
SCSPrechatEntityField* lastNameEntityField =
  [[SCSPrechatEntityField alloc]
  initWithFieldName:@"LastName" label:@"Last Name"];
lastNameEntityField.doFind = YES;
lastNameEntityField.doCreate = YES;

// Create an entity object
SCSPrechatEntity* contactEntity = [[SCSPrechatEntity alloc]
                                   initWithEntityName:@"Contact"];
contactEntity.showOnCreate = YES;

// Add entity fields to entity object
[contactEntity.entityFieldsMaps addObject:firstNameEntityField];
[contactEntity.entityFieldsMaps addObject:lastNameEntityField];

// Update config object with the pre-chat fields
NSMutableArray<SCSPrechatObject *> *preChatFields = [NSMutableArray new];
[preChatFields addObject:firstNameField];
[preChatFields addObject:lastNameField];
config.prechatFields = preChatFields;

// Update config object with the entity mappings
NSMutableArray<SCSPrechatEntity *> *prechatEntities = [NSMutableArray new];
[prechatEntities addObject:contactEntity];
config.prechatEntities = prechatEntities;
```

Entity Configuration Settings

The `SCSPrechatEntity` and `SCSPrechatEntityField` classes give you various configuration settings for mapping fields. For example, if a field doesn't exist, you can have the SDK create that field. The following code sample illustrates some basic building blocks when creating an `SCSPrechatEntity` object.

In Swift:

```
// Create an entity
let entity = SCSPrechatEntity(entityName: "Contact")
entity.saveToTranscript = "ContactId"    // Save this entity to Transcript.ContactId
entity.linkToEntityName = "Case"
entity.linkToEntityField = "ContactId"   // Link this entity to Case.ContactId

// Add an entity field map to our entity
let entityField = SCSPrechatEntityField(fieldName: "FirstName", label: "First Name")
entityField.doFind = true                // Attempt to search for that field
entityField.isExactMatch = true          // Must be an exact match
entityField.doCreate = true              // Create if not found
entity.entityFieldsMaps.add(entityField) // Add field to entity map
```

In Objective-C:

```
// Create an entity
SCSPrechatEntity* entity = [[SCSPrechatEntity alloc] initWithEntityName:@"Contact"];
entity.saveToTranscript = @"ContactId";    // Save this entity to Transcript.ContactId
entity.linkToEntityName = @"Case";
entity.linkToEntityField = @"ContactId";   // Link this entity to Case.ContactId

// Add an entity field map to our entity
SCSPrechatEntityField* entityField = [[SCSPrechatEntityField alloc]
  initWithFieldName:@"FirstName" label:@"First Name"];
entityField.doFind = YES;        // Attempt to search for that field
entityField.isExactMatch = YES;  // Must be an exact match
entityField.doCreate = YES;      // Create if not found
[entity.entityFieldsMaps
  addObject:entityField];        // Add field to entity map
```

See the reference documentation for `SCSPrechatEntity` and `SCSPrechatEntityField`. Also refer to Chat REST API Data Types for the `Entity` and `EntityFieldsMaps` data types, which define the underlying functionality of these SDK objects.

Example: This code sample adds `FirstName`, `LastName`, `Email` to a `Contact` record and a `Subject` field to a `Case` record.

```
let config = SCSChatConfiguration(liveAgentPod: "YOUR_POD_NAME",
                                  orgId: "YOUR_ORG_ID",
                                  deploymentId: "YOUR_DEPLOYMENT_ID",
                                  buttonId: "YOUR_BUTTON_ID")

// Create some basic pre-chat fields (with user input)
let firstNameField = SCSPrechatTextInputObject(label: "First Name")
firstNameField!.isRequired = true
let lastNameField = SCSPrechatTextInputObject(label: "Last Name")
lastNameField!.isRequired = true
let emailField = SCSPrechatTextInputObject(label: "Email")
emailField!.isRequired = true
emailField!.keyboardType = .emailAddress
emailField!.autocorrectionType = .no

// Create a pre-chat field without user input
let subjectField = SCSPrechatObject(label: "Subject", value: "Chat Session")
```

```
// Create an entity mapping for a Contact record type
let contactEntity = SCSPrechatEntity(entityName: "Contact")
contactEntity.saveToTranscript = "Contact"
contactEntity.linkToEntityName = "Case"
contactEntity.linkToEntityField = "ContactId"

// Add some field mappings to our Contact entity
let firstNameEntityField = SCSPrechatEntityField(fieldName: "FirstName", label: "First
 Name")
firstNameEntityField.doFind = true
firstNameEntityField.isExactMatch = true
firstNameEntityField.doCreate = true
contactEntity.entityFieldsMaps.add(firstNameEntityField)
let lastNameEntityField = SCSPrechatEntityField(fieldName: "LastName", label: "Last
Name")
lastNameEntityField.doFind = true
lastNameEntityField.isExactMatch = true
lastNameEntityField.doCreate = true
contactEntity.entityFieldsMaps.add(lastNameEntityField)
let emailEntityField = SCSPrechatEntityField(fieldName: "Email", label: "Email")
emailEntityField.doFind = true
emailEntityField.isExactMatch = true
emailEntityField.doCreate = true
contactEntity.entityFieldsMaps.add(emailEntityField)

// Create an entity mapping for a Case record type
let caseEntity = SCSPrechatEntity(entityName: "Case")
caseEntity.saveToTranscript = "Case"
caseEntity.showOnCreate = true

// Add one field mappings to our Case entity
let subjectEntityField = SCSPrechatEntityField(fieldName: "Subject", label: "Subject")
subjectEntityField.doCreate = true
caseEntity.entityFieldsMaps.add(subjectEntityField)

// Update config object with the pre-chat fields
config!.prechatFields =
  [firstNameField, lastNameField, emailField, subjectField] as? [SCSPrechatObject]

// Update config object with the entity mappings
config!.prechatEntities = [contactEntity, caseEntity]

// Start the session!
ServiceCloud.shared().chatUI.showChat(with: config!, showPrechat: true)
```

Check Agent Availability

Before starting a session, you can check the availability of your chat agents and then provide your users with more accurate expectations.

To check whether agents are available, call the `determineAvailabilityWithConfiguration` method on the `chatCore` property, similar to how you start a chat session.

In Swift:

```
let config = SCSChatConfiguration(liveAgentPod: "YOUR-POD-NAME",
                                  orgId: "YOUR-ORG-ID",
                                  deploymentId: "YOUR-DEPLOYMENT-ID",
                                  buttonId: "YOUR-BUTTON-ID")

ServiceCloud.shared().chat.determineAvailability(with: config,
                                    completion: { (error: Error?, available: Bool) in

  if (error != nil) {
    // Handle error
  }
  else if (available) {
    // Enable chat button
  }
  else {
    // Disable button or warn user that no agents are available
  }

})
```

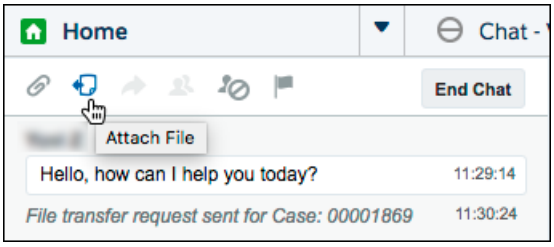
In Objective-C:

```
SCSChatConfiguration *config =
  [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"YOUR-POD-NAME"
                                               orgId:@"YOUR-ORG-ID"
                                        deploymentId:@"YOUR-DEPLOYMENT-ID"
                                            buttonId:@"YOUR-BUTTON-ID"];

[[SCServiceCloud sharedInstance].chat
                 determineAvailabilityWithConfiguration:config
                                             completion:^(NSError *error, BOOL available)
 {

  if (error != nil) {
    // Handle error
  }
  else if (available) {
    // Enable chat button
  }
  else {
    // Disable button or warn user that no agents are available
  }

});
```

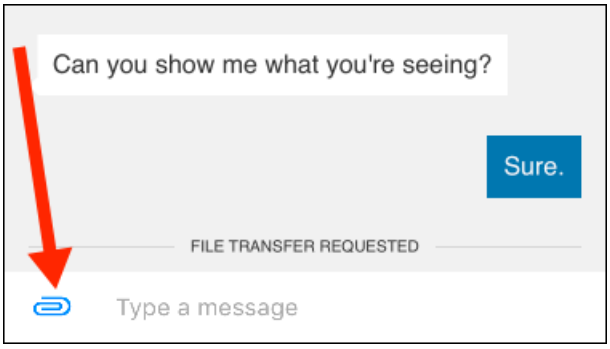
Transfer File to Agent

Give users the ability to transfer files during a chat so they can share information about their issues.

The agent can request that the user transfer a file by clicking the **Attach File** button from the Service Cloud Console.



The user sees a **FILE TRANSFER REQUESTED** message in the app and can then send a file using the paperclip button.



No coding is necessary in your app to make this behavior work.

See Transfer Files During a Chat in Salesforce Help for details about setting up this functionality in the Service Cloud Console.

Note: If your app crashes when a user attempts to perform a file transfer, check that you've enabled the device privacy permissions for the camera and the photo library. An app will crash if these permissions are not set in Xcode. See Install the Service SDK for iOS.

Block Sensitive Data in a Chat Session

To block sending sensitive data to agents, specify a regular expression in your org's setup. When the regular expression matches text in the user's message, the matched text is replaced with customizable text before it leaves the device.

To learn more, see Block Sensitive Data in Chats.

Build Your Own UI with the Chat Core API

With the Chat Core API, you can access the functionality of Chat without a UI. This API is useful if you want to build your own UI and not use the default.

Before running through these steps, be sure you've already:

- Set up Service Cloud to work with Chat. To learn more, see Org Setup for Chat in Lightning Experience with a Guided Flow.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

Once you've reviewed these prerequisites, you're ready to begin.

These steps describe how to use the Chat Core API with your own custom UI. To use the default UI, see Quick Setup: Chat in the Service SDK.

1. Import the SDK. Wherever you intend to use the Chat SDK, be sure to import the ServiceCore framework and the ServiceChat framework.

In Swift:

```
import ServiceCore
import ServiceChat
```

In Objective-C:

```
@import ServiceCore;
@import ServiceChat;
```

2. Create an `SCSChatConfiguration` object.

In Swift:

```
let config = SCSChatConfiguration(liveAgentPod: "TO_DO_POD_NAME",
                                  // e.g. "d.gla5.gus.salesforce.com"
                                  orgId: "TO_DO_ORG_ID",
                                  // e.g. "00DB00000003Rxz"
                                  deploymentId: "TO_DO_DEPLOYMENT_ID",
                                  // e.g. "573B00000005KXz"
                                  buttonId: "TO_DO_BUTTON_ID")
                                  // e.g. "575C00000004h3m"
```

In Objective-C:

```
SCSChatConfiguration *config =
  [[SCSChatConfiguration alloc] initWithLiveAgentPod:@"TO_DO_POD_NAME"
                                                // e.g. "d.gla5.gus.salesforce.com"
                                                 orgId:@"TO_DO_ORG_ID"
                                                // e.g. "00DB00000003Rxz"
                                           deploymentId:@"TO_DO_DEPLOYMENT_ID"
                                                // e.g. "573B00000005KXz"
                                               buttonId:@"TO_DO_BUTTON_ID"];
                                                // e.g. "575C00000004h3m"
```

See Configure a Chat Session on how to configure a chat session.

3. Implement `SCSChatSessionDelegate` and handle the relevant session-related methods.

Using this delegate, you can:

- Detect state transitions with `session(didTransitionFrom:to:)`.
- Detect the end of the session with `session(didEnd:)`.
- Detect error conditions with `session(didError:fatal:)`.
- Detect queue position changes with `session(didUpdateQueuePosition:)`.

Pass your implementation to the `SCSChat` instance.

In Swift:

```
ServiceCloud.shared().chatCore.add(delegate: mySessionDelegate)
```

In Objective-C:

```
[[SCServiceCloud sharedInstance].chatCore addDelegate:mySessionDelegate];
```

To learn more, see Listen for State Changes and Events.

4. Implement `SCSChatEventDelegate` and handle the relevant event-related methods.

Using this delegate, you can:

- Detect when an agent joins with `session(agentJoined:)`.
- Detect when an agent leaves with `session(agentLeftConference:)`.
- Detect when an outgoing message is sent with `session(processedOutgoingMessage:)`.
- Detect when the delivery status of a message has been updated with `session(didUpdateOutgoingMessageDeliveryStatus:)`.
- Detect when an incoming message arrives with `session(didReceiveMessage:)`.
- Detects when a URL is found in an message with `session(didReceiveURL:)`.
- Detect when a chat bot menu arrives with `session(didReceiveChatBotMenu:)`.
- Detect when a chat bot menu item is selected with `session(didSelectMenuItem:)`.
- Detect when the agent starts and finishes typing.
- Detect events related to the file transfer process.
- Detects when the user is transferred to an agent from a chat bot.

Pass your implementation to the `SCSChat` instance.

In Swift:

```
ServiceCloud.shared().chatCore.addEvent(delegate: myEventDelegate)
```

In Objective-C:

```
[[SCServiceCloud sharedInstance].chatCore addEventDelegate:myEventDelegate];
```

5. Start the session using `startSession(with:)` on `SCSChat`.

   In Swift:

   ```
   ServiceCloud.shared().chatCore.startSession(config)
   ```

   In Objective-C:

   ```
   [[SCServiceCloud sharedInstance].chatCore startSessionWithConfiguration:config];
   ```

6. Send activity to the `SCSChatSession` object found in `SCSChat`.

   You can access the session object either from the `SCSChat.session` property, or from any of your delegate event methods that the SDK calls.

   With this session object, you can:

   - Send sneak peek data about the user's message to the agent with `sendSneakPeek`.
   - Send a message to the agent with `sendMessage`.
   - Get or set the user's typing status with `userTyping`.
   - Get information about the actors in the chat session with `actors`.
   - Get the history of all events from the chat session with `allEvents`.
   - Get the current queue position when the user is waiting for an agent with `queuePosition`.

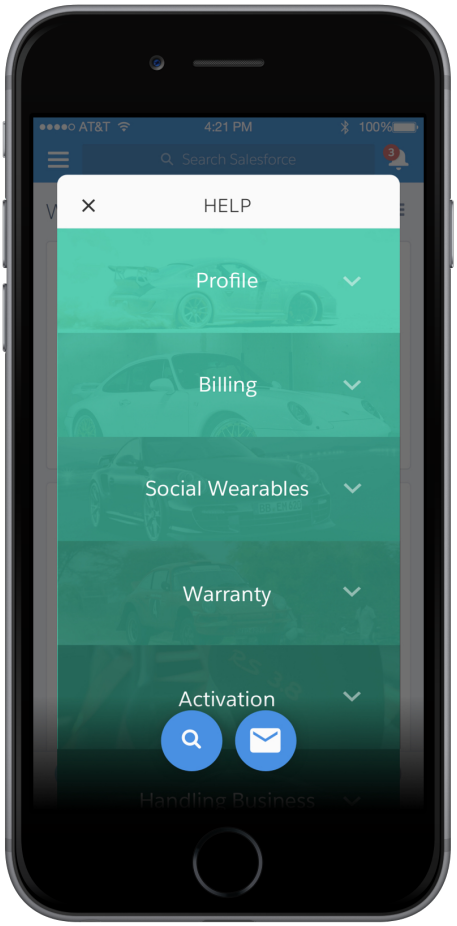
Using Knowledge with the Service SDK

Add the Knowledge experience to your mobile app.

Knowledge in the Service SDK for iOS

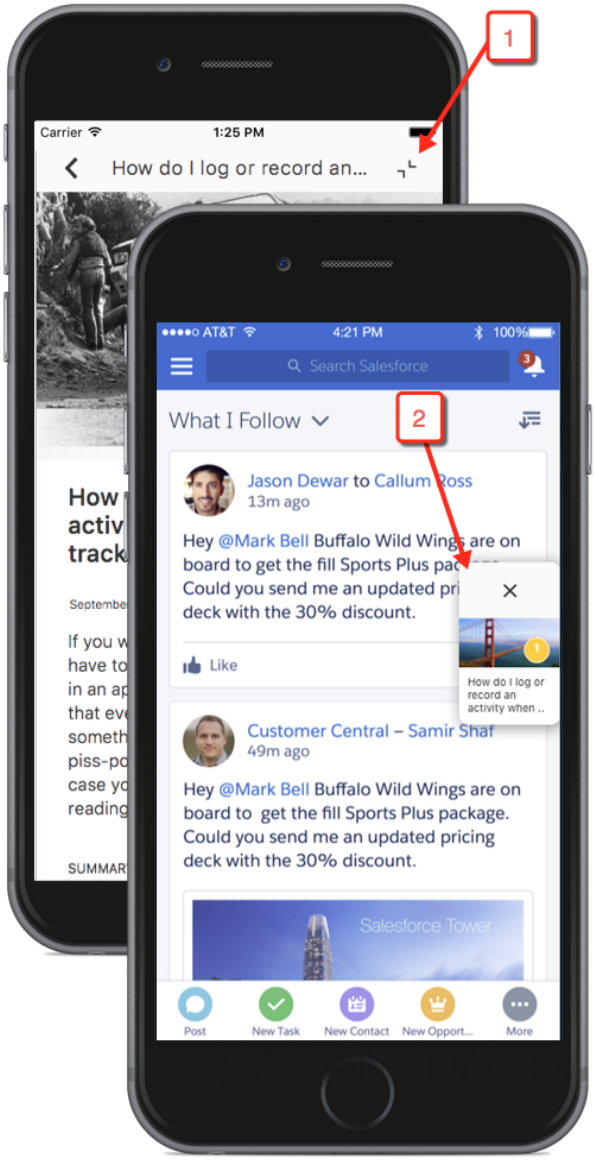
The Knowledge feature in the SDK gives you access to your org's knowledge base directly from within your app.

Once you point your app to your community URL with the right category group and root data category, you can display your knowledge base to your users.



By default, knowledge appears as a floating dialog on top of your app's existing content, though you can customize the presentation if you'd like. From the knowledge home, a user can navigate through articles that are organized by category. Articles are also searchable from within the app. By default, a user can create or manage cases using an action button from within the Knowledge interface.

When a user views an article, they can minimize it using the minimize button at the top right of the article (1) so that they can continue to navigate your app. The user can drag this thumbnail (2) to any part of the screen to improve visibility of the currently showing view. Tapping the X closes the article. Tap on any other part of the thumbnail to make it full screen again.



You can also customize the look and feel of the interface so that it fits naturally within your app. These customizations include the ability to fine-tune the colors, the fonts, the images, and the strings used throughout the interface.

Quick Setup: Knowledge in the Service SDK

To set up Knowledge in your iOS app, point the shared instance to your community, customize the look and feel, and show the interface.

Before starting, make sure that you've already:

- Set up Service Cloud to work with Knowledge. To learn more, see Cloud Setup for Knowledge.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

1. Import the SDK. Wherever you intend to use the Knowledge SDK, be sure to import the Service Common framework and the Knowledge framework.

In Swift:

```
import ServiceCore
import ServiceKnowledge
```

In Objective-C:

```
@import ServiceCore;
@import ServiceKnowledge;
```

2. Point the SDK to your org using an `SCSServiceConfiguration` object.

   To connect your app to your organization, create an `SCSServiceConfiguration` object containing the community URL, the data category group, and the root data category. Pass this object to the `ServiceCloud` shared instance using `SCSServiceConfiguration(community:dataCategoryGroup:rootDataCategory:)`.

   In Swift:

   ```
   // Create configuration object with init params
   let config = SCSServiceConfiguration(
                  community: URL(string: "https://mycommunity.example.com")!,
                  dataCategoryGroup: "Regions",
                  rootDataCategory: "All")

   // Perform any additional configuration here

   // Pass configuration to shared instance
   ServiceCloud.shared().serviceConfiguration = config
   ```

   In Objective-C:

   ```
   // Create configuration object with init params
   SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
      initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]
      dataCategoryGroup:@"Regions"
       rootDataCategory:@"All"];

   // Perform any additional configuration here

   // Pass configuration to shared instance
   [SCServiceCloud sharedInstance].serviceConfiguration = config;
   ```

   Note: You can get the required parameters for this method from your Salesforce org. If your Salesforce admin hasn't set up Knowledge in Service Cloud or you need more guidance, see Cloud Setup for Knowledge.

3. (Optional) Customize the appearance and behavior of the interface.

   You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor`, `setFontDescriptor`, and `setImage`.

   You can customize the action buttons used throughout the UI. You can override the look and the behavior of existing buttons, and you can create buttons associated with new actions.

   There are many different ways to customize the interface. See SDK Customizations with the Service SDK for iOS.

4. (Optional) Implement any of the Service SDK delegates.

`SCServiceCloudDelegate`
Access to general Service SDK events (for example, `willDisplayViewController`, `didDisplayViewController`, `shouldShowActionWithName`).

`SCKnowledgeInterfaceDelegate`
Access to Knowledge interface events (for example, `imageForArticle`, `imageForDataCategory`). See Customize Images with the Service SDK for an example of using this delegate.

`SCAppearanceConfigurationDelegate`
Access to appearance-related events (for example, `appearanceConfigurationWillApplyUpdates`, `appearanceConfigurationDidApplyUpdates`).

5. Show the interface from your view controller using `setInterfaceVisible`.

   You can show the interface as soon as the view controller loads, or start it from a UI action.

   In Swift:

```
ServiceCloud.shared().knowledge.setInterfaceVisible(true,
                                                    animated: true,
                                                    completion: nil)
```

   In Objective-C:

```
[[SCServiceCloud sharedInstance].knowledge setInterfaceVisible:YES
                                                      animated:YES
                                                    completion:nil];
```

   By default, the interface appears as a floating dialog. Alternatively, you can present the interface using a custom presentation. You can even manually control and configure the Knowledge view controllers yourself.See Customize the Presentation and View Controllers for Knowledge for more info.

   For instructions on launching the interface from a web view, see Launch the Service SDK from a Web View in iOS.

If you run into issues accessing your community, check out Can't Access My Knowledge Base.

Example: **Swift Example**

To use this example code, create a Single View Application and Install the Service SDK for iOS.

Set up the Knowledge interface within the `AppDelegate` implementation.

```
import UIKit
import ServiceCore
import ServiceKnowledge

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?

  func application(_ application: UIApplication,
    didFinishLaunchingWithOptions
    launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    // Create configuration object with init params
    let config = SCSServiceConfiguration(
      community: URL(string: "https://mycommunity.example.com")!,
```

```
      dataCategoryGroup: "Regions",
      rootDataCategory: "All")

    // Pass configuration to shared instance
    ServiceCloud.shared().serviceConfiguration = config

    return true
  }
}
```

Using the storyboard, add a button to the view. Then add a `Touch Up Inside` action in your `UIViewController` implementation with the name `showHelp`. When the button is clicked, make the Knowledge interface visible.

```
import UIKit
import ServiceCore
import ServiceKnowledge

class ViewController: UIViewController {

  @IBAction func showHelp(_ sender: AnyObject) {

    ServiceCloud.shared().knowledge.setInterfaceVisible(true,
       animated: true,
       completion: nil)
  }
}
```

Knowledge as an Authenticated User

In some scenarios, you may want knowledge base access for logged-in users only. You might even have different knowledge bases for different user profiles. For these scenarios, you can use the authenticated Knowledge feature.

Before starting, make sure that you've already:

- Set up Service Cloud to work with Knowledge. To learn more, see Cloud Setup for Knowledge.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

When you activate the Knowledge interface for authenticated users, they see Knowledge content assigned to their user profile. If you do not want to authenticate users and prefer to let them see Knowledge content accessible to guest users, see Quick Setup: Knowledge in the Service SDK for instructions on accessing a public knowledge base.

Note: When using Knowledge with authenticated users, be sure that your knowledge article types are visible (set to "Read") for the desired user profile and that the knowledge articles belong to a channel that is accessible to that user. For more information, see Knowledge Article Access and Create and Edit Articles in Salesforce Help.

1. Follow authentication instructions for this SDK: Authentication with the Service SDK for iOS.
2. Review the steps in Quick Setup: Knowledge in the Service SDK. The basic steps for setting up and displaying the interface still apply for authenticated users.

After you authenticate users, they see only Knowledge content that is accessible to their user profile.

Customize the Presentation and View Controllers for Knowledge

The simplest way to show and hide the Knowledge interface is by calling the `setInterfaceVisible` method. Alternatively, you can present the interface using a custom presentation. You can even manually control and configure the Knowledge view controllers yourself.

Activating Interface Using the Default Presentation

Use the `setInterfaceVisible` method to show the Knowledge interface using the default presentation. This method shows the interface as a floating dialog on top of your app's existing content. When the user drills into a detail screen, the interface automatically transitions to a full screen mode.

In Swift:

```
ServiceCloud.shared().knowledge.setInterfaceVisible(true,
                                                    animated: true,
                                                    completion: nil)
```

In Objective-C:

```
[[SCServiceCloud sharedInstance].knowledge setInterfaceVisible:YES
                                                      animated:YES
                                                    completion:nil];
```

Activating Interface Using a Custom Transitioning Delegate

You can also present the interface using a custom transitioning animation and custom presentation. Implement a `UIViewControllerTransitioningDelegate`.

1. Supply the `ServiceCloud` shared instance with your `SCServiceCloudDelegate` implementation.

   In Swift:

   ```
   ServiceCloud.shared().delegate = mySCServiceCloudDelegate
   ```

   In Objective-C:

   ```
   [SCServiceCloud sharedInstance].delegate = mySCServiceCloudDelegate;
   ```

2. Implement the `serviceCloud(transitioningDelegateForPresentedController:presenting:)` method in your delegate and return a custom `UIViewControllerTransitioningDelegate` from this method.

   In Swift:

   ```
   func serviceCloud(_ serviceCloud: ServiceCloud,
                       transitioningDelegateForPresentedController
                         presentedController: UIViewController,
                       presenting presentingController: UIViewController)
                       -> UIViewControllerTransitioningDelegate? {

     // TO DO: Put your logic here and then return your transitioning delegate...

     return myTransitioningDelegate
   }
   ```

In Objective-C:

```
- (NSObject<UIViewControllerTransitioningDelegate> *)
                                  serviceCloud:(SCServiceCloud *)serviceCloud
        transitioningDelegateForViewController:(UIViewController *)controller {

  // TO DO: Put your logic here and then return your transitioning delegate...

  return myTransitioningDelegate;
}
```

Showing or Customizing the View Controllers

Instead of having the SDK manage the flow from one view to the next, you can instantiate any of the view controllers and display it manually. When instantiating a view controller, be sure to implement the associated delegate and pass that delegate to the view controller (using the `delegate` property). The delegates allow you to override the default behavior for the views. For instance, you can filter the categories shown to the user based on your own run-time logic.

If you don't want to manually instantiate a view controller but you still want the ability to control its behavior, implement the `serviceCloud(willDisplay controller:animated:)` and `serviceCloud(didDisplay controller:animated:)` methods of `SCServiceCloudDelegate`. You can access the view controllers from those methods.

View Controller Description	View Controller	Delegate
Knowledge Home Screen — to show a list of categories	`SCSKnowledgeHomeViewController`	`SCSKnowledgeHomeViewControllerDelegate`
Category List View — to show a list of articles for a category	`SCSCategoryViewController`	`SCSCategoryViewControllerDelegate`
Article Query List — to show a list of articles based on a query (used for case deflection when creating a case)	`SCSArticleQueryListViewController`	`SCSArticleQueryListViewControllerDelegate`
Article View — to show an individual article	`SCSArticleViewController`	`SCSArticleViewControllerDelegate`

By default, `SCSKnowledgeHomeViewController` and `SCSCategoryViewController` show three articles per category, sorted by the view score. (You can see more than three articles by tapping the **more** button in the UI.) However, you can change what gets shown in these views. For instance, you can change the number of articles that are shown, and you can sort articles by date. To configure a view, set the `articleQueryTemplate` property using an `SCSArticleQuery` object. To learn about building queries, see Article Fetching and Caching. To change the view back to the default query, set the property to `nil`.

You can manually display a specific article by instantiating an `SCSArticleViewController` instance, specifying the article using the `article` property, and then presenting the view yourself. This technique is useful if you want to display specific articles within your app and you want the view within your own view hierarchy. To learn more about this technique, see Article Fetching and Caching.

Article Fetching and Caching

By default, the SDK fetches knowledge articles as they are needed. These articles are then cached locally for faster access. However, using methods in `SCSKnowledgeManager`, you can pre-fetch articles to support offline access and other use cases.

When the SDK displays an article, it first requests the latest article data from the server. If the content is stored in the cache, it's shown immediately — even before the network request completes. If the information stored on the server is newer than what is located in the cache, the database updates with the latest article content and the article view controller updates accordingly. In this way, article content is quickly available even when the device is offline or has a spotty network connection, but the SDK also ensures that content is up to date.

This default behavior may be all you need out of a knowledge base. However, you can explicitly fetch articles for one of several common use cases:

- Faster access to the most commonly viewed articles.
- Custom presentation of specific content.
- Offline access to some or all of your knowledge base.

There are several different ways to fetch knowledge articles. You can fetch all articles in a category. You can fetch articles based on a query. You can sort or limit the search results. And of course, you can also fetch specific articles by article ID.

The following classes are associated with article caching:

`KnowledgeManager`
Manager class for interacting with Knowledge at the data level, including article caching functionality.

`CategoryGroup`
Class that represents a category group.

`Category`
Class that represents a category or subcategory.

`MutableArticleQuery`
Class used to build your article query. The immutable parent class is `SCSArticleQuery`.

`Article`
Class that contains information about an article.

Fetching Categories

Since data categories are at the heart of knowledge articles, the categories for your org need to be downloaded before any interactions with Knowledge can be made. To see if categories have been cached, check the value for `hasFetchedCategories`. If it returns `NO`, call `fetchAllCategories`. When downloading individual articles, you don't need to make this call, but you won't be able to access content that requires information about data categories without performing this fetch.

In Swift:

```
let knowledgeManager = KnowledgeManager.default()
if (!knowledgeManager.hasFetchedCategories()) {

  knowledgeManager.fetchAllCategories(completionHandler: {
    (categoryGroups: [CategoryGroup]?, error: Error?) in

      // TO DO: Get articles from each category using queryArticlesInCategory

  })
}
```

In Objective-C:

```
SCSKnowledgeManager *knowledgeManager = [SCSKnowledgeManager defaultManager];
if (!knowledgeManager.hasFetchedCategories) {
```

```
  [knowledgeManager fetchAllCategoriesWithCompletionHandler:^
                   (NSArray<SCSCategoryGroup *> * _Nullable categoryGroups,
                   NSError * _Nullable error) {

    // TO DO: Get articles from each category using queryArticlesInCategory

  }];
}
```

Note: If your interface supports authenticated users, keep in mind that the user account information does not change for a pre-existing instance of `SCSKnowledgeManager`. After you authenticate a user, call the static `defaultManager` method on `KnowledgeManager` to get a new instance containing new user account information.

Querying for Articles (from Server, or Locally)

To fetch articles from the server for a given query, call `fetchArticles(with:)`. To fetch local (already cached) articles for a given query, call `articles(matching:)`. These methods take a `MutableArticleQuery` instance, which is an essential part of your fetch request. If you already fetched categories, you can drive this query based on a category; you can also directly query for an article based on its ID, or using another type of query.

When you create a `MutableArticleQuery` instance, you can determine the type of query using the following properties:

Table 8: Query Types

Query Type	Property Name / Type	Notes
Articles matching article ID	`articleId`: `String`	The 18-character article ID. This property cannot be used with `searchTerm`, `queryMethod`, `sortOrder`, or `sortByField`.
Articles matching search term	`searchTerm`: `String`	This property cannot be used with `articleId`, `sortOrder`, or `sortByField`.
Articles within a set of categories	`categories`: array of `Category` objects	You must use the `queryMethod` filter described in the Query Filters table if you use this query type.

You can sort the query results with the following properties:

Table 9: Query Sort Properties

Query Sort Property	Property Name / Type / Default	Description
Sort using a specific sort order	`sortOrder`: `SCArticleSortOrder` = `.descending`	Whether to sort by ascending order or descending order.
Sort using a specific field type	`sortByField`: `SCArticleSortByField` = `.lastPublishedDate`	Which field you want to sort by: `.title`, `.lastPublishedDate`, or `.viewScore`.

You can filter the query results with the following properties:

Table 10: Query Filters

Query Filter	Property Name / Type / Default	Description
Number of articles to fetch	`pageSize`: `UInt` = 20	The number of articles to retrieve. The server does not provide more than 100 articles at a time.
Filter selector for category (used with the `categories` property)	`queryMethod`: `SCQueryMethod` = `.below`	Whether you want the query to operate on just the specified categories (`.at`), the categories and all their parent categories (`.above`), the categories and all their subcategories (`.below`). Typically, you'll want to use `.below` to capture the specified category and its children. Be sure to have something specified in the `categories` property when using this filter.

Before performing a query, you can check whether the query is valid using the `valid` property. A query is invalid when conflicting property values are specified. The following situations cause an invalid query: when `searchTerm` and `articleId` are both populated, when `searchTerm` and `sortByField` are both populated, when `searchTerm` and `sortOrder` are both populated, when `articleId` and `categories` are both populated, when `articleId` and `sortByField` are both populated, when `articleId` and `sortOrder` are both populated, and when`articleId` and `queryMethod` are both populated.

This code shows an example that queries using a search term with a limit of five articles per page.

In Swift:

```
let query = MutableArticleQuery()
query.searchTerm = "login issues"
query.pageSize = 5

knowledgeManager.fetchArticles(with: query, completion: {
  (articles: [Article], error: Error?) in

    // TO DO: Download articles using downloadContentWithOptions

})
```

In Objective-C:

```
SCSMutableArticleQuery *query = [SCSMutableArticleQuery new];
query.searchTerm = @"login issues";
query.pageSize = 5;

[knowledgeManager fetchArticlesWithQuery:query completion:^
                 (NSArray<SCSArticle *> * _Nonnull articles,
                 NSError * _Nullable error) {

  // TO DO: Download articles using downloadContentWithOptions
```

```
}];
```

Downloading Content

Once you've fetched the articles, you'll need to download them before displaying. Use the `downloadContent(withOptions:)` method in the `Article` class to perform this function. This method caches the HTML content. It also fetches the article images, if you specify it to do so with the `Options` parameter.

In Swift:

```
let article = // once you have an article

let options: Int = SCSArticleDownloadOption.refetchArticleContent.rawValue |
                   SCSArticleDownloadOption.images.rawValue
article.downloadContent(withOptions: SCSArticleDownloadOption(rawValue:options)!,
        completion: { (error: Error?) in

  // TO DO: Handle completion

})
```

In Objective-C:

```
SCSArticle *article = // once you have an article

[article downloadContentWithOptions:
  (SCSArticleDownloadOptionRefetchArticleContent|SCSArticleDownloadOptionImages)
  completion:^(NSError * _Nullable error) {

  // TO DO: Handle completion

}]
```

At this point, the article is downloaded and available for offline access.

If you're not sure if an article has already been downloaded, use the `isArticleContentDownloaded` and `isAssociatedContentDownloaded` methods to check before downloading.

Displaying Content

You've got several ways to present knowledge content:

1. Use the default presentation. If you use the `setInterfaceVisible` method, the SDK automatically displays the content, and it uses cached content before trying to get content online.
2. You can display an article using the `showArticle` method and specifying the article you want to show. As with the default presentation, this technique presents the article in a floating window that can be minimized.
3. You can manually display a specific article by instantiating an `SCSArticleViewController` instance, specifying the article using the `article` property, and then presenting the view yourself. This technique is useful if you want to display specific articles within your app and you want the view within your own view hierarchy. When using this technique, you can use the `SCSArticleViewControllerDelegate` class to listen for events.

In Swift:

```
// Get Knowledge Manager instance
let knowledgeManager = KnowledgeManager.default()

// Create query for a specific article
let query = MutableArticleQuery()
query.articleId = "TO_DO:QUERY_ID"

// Fetch article
knowledgeManager.fetchArticles(with: query, completion: {
    (articles: [Article], error: Error?) in

  if (error != nil) {
    // TO DO: Handle error
  }
  else if (articles.count == 0) {
    // TO DO: Handle no results
  }
  else {
    let article: Article = articles[0]

    // Download article
    let options: Int = SCSArticleDownloadOption.refetchArticleContent.rawValue |
                       SCSArticleDownloadOption.images.rawValue
    article.downloadContent(withOptions: SCSArticleDownloadOption(rawValue:options)!,
        completion: { (error: Error?) in

      if (error != nil) {
        // TO DO: Handle error
      } else {

        // Display view with article
        let articleVC = SCSArticleViewController()
        articleVC.article = article
        self.present(articleVC, animated:true, completion: nil)
      }
    })
  }
})
```

In Objective-C:

```
// Get Knowledge Manager instance
SCSKnowledgeManager *knowledgeManager = [SCSKnowledgeManager defaultManager];

// Create query for a specific article
SCSMutableArticleQuery *query = [SCSMutableArticleQuery new];
query.articleId = @"TO_DO:QUERY_ID";

// Fetch article
[knowledgeManager fetchArticlesWithQuery:query completion:^
    (NSArray<SCSArticle *> * _Nonnull articles,
    NSError * _Nullable error) {
```

```
  if (error != nil) {
    // TO DO: Handle error
  }
  else if ([articles count] == 0) {
    // TO DO: Handle no results
  }
  else {
    SCSArticle *article = articles[0];

    // Download article
    [article downloadContentWithOptions:
        (SCSArticleDownloadOptionRefetchArticleContent|SCSArticleDownloadOptionImages)
        completion:^ (NSError* error) {

      if (error != nil) {
        // TO DO: Handle error
      } else {

        // Display view with article
        SCSArticleViewController *articleVC = [[SCSArticleViewController alloc] init];
        articleVC.article = article;
        [self presentViewController:articleVC animated:YES completion:nil];
      }
    }];
  }
}];
```

Example: This code shows an example of how to download the top three articles in each category.

In Swift:

```
let knowledgeManager = KnowledgeManager.default()

// Create an article download block
let articleDownloadBlock = {

  // Get category group
  let categoryGroup = knowledgeManager.categoryGroup(withName: "MY-CATEGORY-GROUP")

  // Get root category from category group
  let rootCategory = categoryGroup?.category(withName: "MY-CATEGORY")

  if (rootCategory != nil) {

    // Iterate through all categories in root category
    for category in (rootCategory!.childCategories) {

      // Build a query...
      let query = MutableArticleQuery()

      // ... for the current category
      query.categories = [category]

      // ... containing the top 3 articles
      query.pageSize = 3
```

```
      // And then fetch articles with that query
      knowledgeManager.fetchArticles(with: query, completion:
        { (articles: [Article], error: Error?) in

          // TO DO: Check for error

          // For each article object fetched
          for article in articles {

            // Fetch the contents
            let options: Int =
              SCSArticleDownloadOption.refetchArticleContent.rawValue |
              SCSArticleDownloadOption.images.rawValue
            article.downloadContent(
              withOptions: SCSArticleDownloadOption(rawValue: options)!,
              completion: nil)
          }
      })
    }
  }
}

// If we haven't fetched the categories
if (!knowledgeManager.hasFetchedCategories()) {

  // Then first fetch the categories
  knowledgeManager.fetchAllCategories(completionHandler:
    { (categoryGroups: [CategoryGroup]?, error: Error?) in

      // TO DO: Check for error

      // And then download the articles
      articleDownloadBlock();
  })
}
else {
  // Download the articles
  articleDownloadBlock();
}
```

In Objective-C:

```
SCSKnowledgeManager *knowledgeManager = [SCSKnowledgeManager defaultManager];

// Create an article download block
dispatch_block_t articleDownloadBlock = ^{

  // Get category group
  SCSCategoryGroup *categoryGroup =
    [knowledgeManager categoryGroupWithName:@"MY-CATEGORY-GROUP"];

  // Get root category from category group
  SCSCategory *rootCategory =
    [categoryGroup categoryWithName:@"MY-CATEGORY"];
```

```
  if (rootCategory != nil) {

    // Iterate through all categories in root category
    for (SCSCategory *category in rootCategory.childCategories) {

      // Build a query...
      SCSMutableArticleQuery *query = [SCSMutableArticleQuery new];

      // ... for the current category
      query.categories = [NSArray arrayWithObjects: category, nil];

      // ... containing the top 3 articles
      query.pageSize = 3;

      // And then fetch articles with that query
      [knowledgeManager fetchArticlesWithQuery:query completion:^
                        (NSArray<SCSArticle *> * _Nonnull articles,
                        NSError * _Nullable error) {

        // TO DO: Check for error

        // For each article object fetched
        for (SCSArticle *article in articles) {

          // Fetch the contents
          [article downloadContentWithOptions:
            (SCSArticleDownloadOptionRefetchArticleContent|
            SCSArticleDownloadOptionImages)
            completion:nil];
        }
      }];
    }
  }
};

// If we haven't fetched the categories
if (!knowledgeManager.hasFetchedCategories) {

  // Then first fetch the categories
  [knowledgeManager fetchAllCategoriesWithCompletionHandler:^
                    (NSArray<SCSCategoryGroup *> * _Nullable categoryGroups,
                    NSError * _Nullable error) {

        // TO DO: Check for error

        // And then download the articles
        articleDownloadBlock();

  }];
} else {

  // Download the articles
```

```
  articleDownloadBlock();
}
```

Customize Knowledge Articles with JavaScript or CSS

Create a richer experience for your users by injecting custom JavaScript or CSS into your knowledge articles. For example, change the style sheet for all your articles, or add introductory content to a subset of articles.

You can perform JavaScript and CSS injection globally or on a per-article basis.

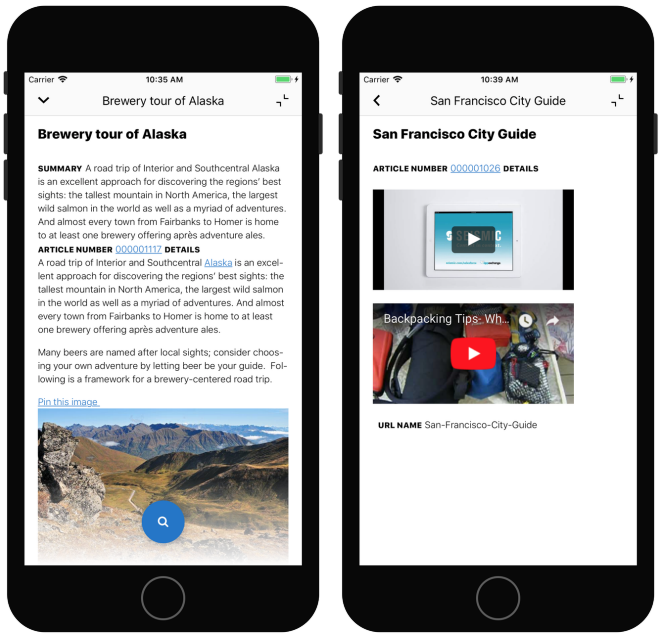
To update the global CSS or JavaScript, use the `globalArticleCSS` property and the `globalArticleJavascript` property on the `SCAppearanceConfiguration` object. To learn more about using this appearance object, see SDK Customizations with the Service SDK for iOS.

To update the CSS or JavaScript for a particular article, use the `additionalCSSForArticle` method and the `additionalJavascriptForArticle` method on `SCSArticleViewControllerDelegate`. To learn about view controllers and their delegates, see Customize the Presentation and View Controllers for Knowledge.

To learn about the HTML structure for a knowledge article, see the reference documentation for `SCSArticleViewController`.

Example: This example injects custom CSS to all articles and injects custom JavaScript only to articles whose title contains "San Francisco." You can apply these concepts to your own use cases.

Let's run this example on the following articles.



We want to make 2 changes to the articles.

- A global CSS change to give all articles a vivid green background and thick blue border.
- A JavaScript change to insert a bright pink paragraph at the top of all articles whose title includes "San Francisco."

After the example runs, you see the following changes.

To implement the green background color and thick blue border to all articles, add the following CSS to the `globalArticleCSS` property on the `SCAppearanceConfiguration` object.

```
// Create appearance configuration instance
let appearance = SCAppearanceConfiguration()

// Customize the CSS with an ugly green background and a blue border...
appearance.globalArticleCSS = "body { background: #00ff00; border: thick solid #0000ff;
 }"

// Save configuration instance
ServiceCloud.shared().appearanceConfiguration = appearance
```

To add an introductory paragraph to specific articles, we need access to the right view controller and implement the right delegate method.

1. To check when an `SCSArticleViewController` will show, implement the `willDisplay` method of `SCServiceCloudDelegate`.
2. To inspect the new article and add JavaScript, implement the `additionalJavascriptForArticle` method of `SCSArticleViewControllerDelegate`.
3. If the article title contains "San Francisco," add JavaScript.

This code performs all these tasks.

```
class MyDelegate: NSObject, SCServiceCloudDelegate, SCSArticleViewControllerDelegate
{

  // The custom JavaScript that we'll add to selected articles
  let customJavaScript = """
    window.onload = function() {
      var testNode = document.createElement('p');
      testNode.innerText = 'San Francisco Article, added by JavaScript';
      testNode.style.color = '#000000';
      testNode.style.background = '#f76b95';
```

```
      testNode.style.padding = '5pt';
      testNode.style.textAlign = 'center';
      document.body.insertBefore(testNode, document.body.firstChild);
    }
  """

  override init() {
    // Assign us as the ServiceCloud delegate
    ServiceCloud.shared().delegate = self
  }

  // Called when a new view controller will display
  func serviceCloud(_ serviceCloud: ServiceCloud,
                    willDisplay controller: UIViewController,
                    animated: Bool) {

    // Are we about to show the article view controller?
    if let articleController = controller as? SCSArticleViewController {

      // If so, then assign us as the delegate
      articleController.delegate = self
    }
  }

  // Called before an article shows
  func articleController(_ controller: SCSArticleViewController,
                         additionalJavascriptFor article: Article) -> String? {

    var additionalJS: String? = nil

    // TO DO: Inspect article to see if we really want to add custom JavaScript...
    // For example, let's only add JS to articles
    // that have 'San Francisco' in the title:
    if article.title.contains("San Francisco") {
      additionalJS = customJavaScript
    }

    return additionalJS
  }
}
```

And that's it! This app now injects custom CSS and JavaScript into knowledge articles.

Disable Case Management from Knowledge Interface

By default, Case Management is enabled when a user accesses your Knowledge interface. A user can create a case with an action button at the bottom of the view. However, you can remove this action button by implementing a protocol method on `SCServiceCloudDelegate`.

To disable the Case Management button, follow the instructions in Customize Action Buttons with the Service SDK. Use the `casePublisher` enumerated type when determining which button to disable.

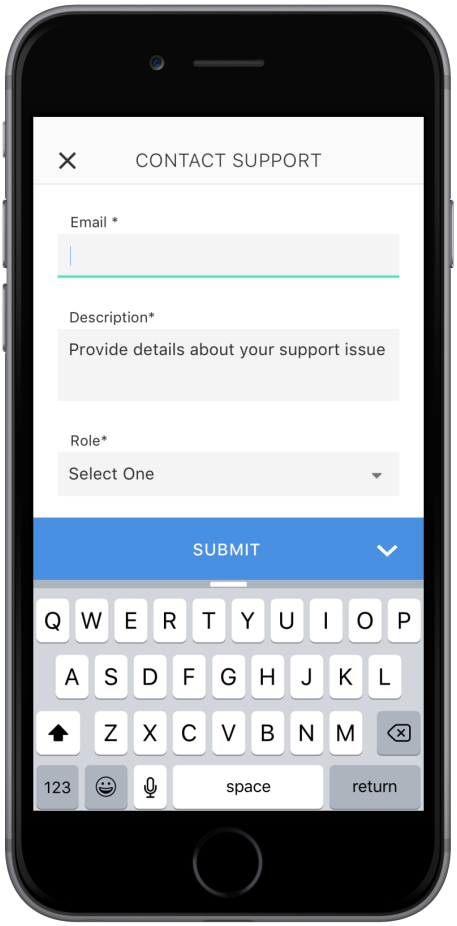
Using Case Management with the Service SDK

Add the Case Management experience to your mobile app.

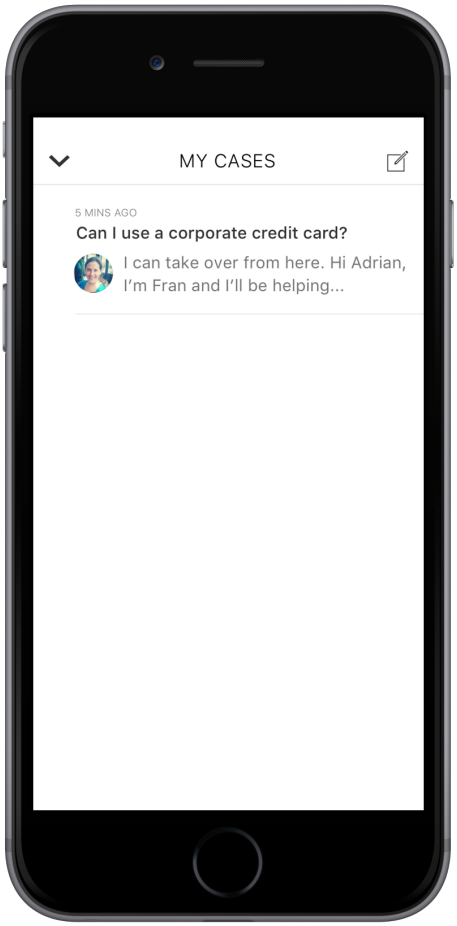
Case Management in the Service SDK for iOS

The Case Management feature in the SDK allows your users to create and manage cases.

Once you point your app to your community URL, you can display the case management interface to your users. If you don't authenticate the user, they can still create new cases using the guest user profile. A user creates a new case with the **Case Publisher** screen:



If you authenticate the user, they can also view and manage their list of cases. In the default 'guest user' flow, launching the interface causes the **Case Publisher** screen to appear. From this screen, a user can create a case as an anonymous guest user. In the default 'authenticated user' flow, launching the interface causes the **Case List** screen to appear. From this screen, a user can inspect an existing case (which launches the **Case Details** screen), or create a new case (which launches the **Case Publisher** screen).



If you'd prefer, you can manually control the Case Management view controllers.

For authenticated users, you can set up notifications so they are notified when there's a new post associated with one of their existing cases. You can even set it up so that the **Case Details** screen automatically appears with the latest case activity.

You can also customize the look and feel of the interface so that it fits naturally within your app. These customizations include the ability to fine-tune the colors, the fonts, the images, and the strings used throughout the interface.

Quick Setup: Case Publisher as a Guest User

To set up Case Publisher in your iOS app, point the shared instance to your community, specify a global action, customize the look and feel, and show the interface.

Before running through these steps, be sure you've already:

- Set up Service Cloud to work with Case Management. To learn more, see Cloud Setup for Case Management.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

These instructions allow you to set up Case Publisher as a guest user. This functionality allows a user to publish a new case. However, a guest user cannot manage existing cases. To manage cases, you'll need to authenticate the user, which requires a few more steps. To learn more about the authenticated user setup, see Case Management as an Authenticated User.

1. Import the SDK. Wherever you intend to use the Case Management SDK, be sure to import the Service Common framework and the Case Management framework.

In Swift:

```
import ServiceCore
import ServiceCases
```

In Objective-C:

```
@import ServiceCore;
@import ServiceCases;
```

2. Point the SDK to your org using an `SCSServiceConfiguration` object.

   To connect your application to your organization, create an `SCSServiceConfiguration` object containing the community URL. Pass this object to the `ServiceCloud` shared instance using `SCSServiceConfiguration(community:)`.

   In Swift:

```
// Create configuration object with your community URL
let config = SCSServiceConfiguration(
             community: URL(string: "https://mycommunity.example.com")!)

// Pass configuration to shared instance
ServiceCloud.shared().serviceConfiguration = config
```

   In Objective-C:

```
// Create configuration object with your community URL
SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
  initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]];

// Pass configuration to shared instance
[SCServiceCloud sharedInstance].serviceConfiguration = config;
```

   You can get the community URL from your Salesforce org. From Setup, search for **All Communities**, and copy the URL for the desired community. For more help, see Cloud Setup for Case Management.

   Note: If you plan to access Knowledge in addition to Case Management, use `SCSServiceConfiguration(community:dataCategoryGroup:rootDataCategory:)` instead. This constructor sets up data categories in addition to setting the community URL. See Quick Setup: Knowledge in the Service SDK in the Knowledge section for more info.

3. Assign a global action to the Case Management interface. The global action determines the fields shown when a user creates a case.

   To configure the fields shown when creating a case, specify the global action name in the `caseCreateActionName` property. This code snippet illustrates how to associate the case publisher feature with the **New Case** global action layout, which is one of the default actions provided in most orgs.

   In Swift:

```
ServiceCloud.shared().cases.caseCreateActionName = "NewCase"
```

   In Objective-C:

```
[SCServiceCloud sharedInstance].cases.caseCreateActionName = @"NewCase";
```

   You can get the global action name from your Salesforce org. From Setup, search for **Global Actions**, and copy the name of the desired quick action. For more help, see Cloud Setup for Case Management.

Note: Be sure that your global action is accessible to the Guest user profile. Also note that the case publisher screen does not respect field-level security for guest users. If you want to specify different security levels for different users, use different quick actions.

4. (Optional) Customize the appearance with the configuration object.

   You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor`, `setFontDescriptor`, and `setImage`. You can also configure the strings used throughout the interface. See SDK Customizations with the Service SDK for iOS.

5. (Optional) Implement any of the Service SDK delegates.

   `SCServiceCloudDelegate`
   Access to general Service SDK events (for example, `willDisplayViewController`, `didDisplayViewController`, `shouldShowActionWithName`).

   `SCAppearanceConfigurationDelegate`
   Access to appearance-related events (for example, `appearanceConfigurationWillApplyUpdates`, `appearanceConfigurationDidApplyUpdates`).

6. Show the interface from your view controller using `setInterfaceVisible`.

   You can show the interface as soon as the view controller loads, or start it from a UI action.

   Note: If you show the interface as a guest user, the case publisher screen appears. If you choose to authenticate first (see Case Management as an Authenticated User), then the case list screen is the first thing to appear.

   In Swift:

```
ServiceCloud.shared().cases.setInterfaceVisible(true,
                                                animated: true,
                                                completion: nil)
```

   In Objective-C:

```
[[SCServiceCloud sharedInstance].cases setInterfaceVisible:YES
                                                  animated:YES
                                                completion:nil];
```

   By default, the interface appears as a floating dialog. Alternatively, you can present the interface using a custom presentation. You can even manually control the Case Management view controllers yourself. See Customize the Presentation and View Controllers for Case Management for more info.

   For instructions on launching the interface from a web view, see Launch the Service SDK from a Web View in iOS.

Example: **Swift Example**

To use this example code, create a Single View Application and Install the Service SDK for iOS.

Set up the Case Management interface within the `AppDelegate` implementation.

```
import UIKit
import ServiceCore
import ServiceCases

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?
```

```
  func application(_ application: UIApplication,
    didFinishLaunchingWithOptions
    launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    // Point the SDK to your community
    let config =
      SCSServiceConfiguration(community:
      URL(string: "https://mycommunity.example.com")!)
    ServiceCloud.shared().serviceConfiguration = config

    // Assign global action to use for case layout
    ServiceCloud.shared().cases.caseCreateActionName = "NewCase"

    return true
  }
}
```

Using the storyboard, add a button to the view. Then add a `Touch Up Inside` action in your `UIViewController` implementation with the name `showHelp`. When the button is clicked, make the Case Management interface visible.

```
import UIKit
import ServiceCore
import ServiceKnowledge

class ViewController: UIViewController {

  @IBAction func showHelp(_ sender: AnyObject) {

    ServiceCloud.shared().cases.setInterfaceVisible(true,
       animated: true,
       completion: nil)
  }
}
```

Case Management as an Authenticated User

To manage existing cases, a user must authenticate with your org. Once authenticated, the user can create and manage cases from your app.

Before running through these steps, be sure you've already:

- Set up Service Cloud to work with Case Management. To learn more, see Cloud Setup for Case Management.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

These instructions allow you to set up case management as an authenticated user. When you activate the Case Management interface for an authenticated user, a list of their existing cases appears initially. From there, they can inspect an existing case, or they can create a new case. If you do not want to authenticate users, and you prefer to let them create cases as a guest user, see Quick Setup: Case Publisher as a Guest User.

1. Follow authentication instructions for this SDK: Authentication with the Service SDK for iOS.
2. Review the steps in Quick Setup: Case Publisher as a Guest User. The basic steps for setting up and displaying the interface still apply for authenticated users.

3. Assign a Cases view name to allow authenticated users to see a list of all their existing cases.

   To show a list of cases, specify the unique name for your preferred Cases view from your Salesforce org. To get this case list value, access the **Cases** tab in your org, pick the desired **View**, select **Go!** to see that view, and then select **Edit** to edit the view. From the edit window, you can see the **View Unique Name**. Use this value when you specify the `caseListName` in the SDK. For more help, see Cloud Setup for Case Management.

   In Swift:

   ```
   ServiceCloud.shared().cases.caseListName = "AllOpenCases"
   ```

   In Objective-C:

   ```
   [SCServiceCloud sharedInstance].cases.caseListName = @"AllOpenCases";
   ```

4. (Optional) Incorporate notifications whenever there is a new text post on an existing case and check the `caseUnreadCount` property to determine the number of unread cases.

   You can send push notifications from your org when activity associated with a user's case occurs. After you've set up notifications in your org, handle the notification from your app. To learn more, see Push Notifications for Case Activity.

   The number of unread cases is automatically listed as a numerical badge on the Case List action button within the UI. If you want to know the number of cases with unread messages, call the `caseUnreadCount` property on the `cases` instance. This value is only accurate after the user views the Case List screen. Before that, the value is 0.

After you authenticate users, they have access to the full Case Management functionality, with easy access to information about their existing cases.

Customize the Presentation and View Controllers for Case Management

The simplest way to show and hide the Case Management interface is by calling the `setInterfaceVisible` method. Alternatively, you can present the interface using a custom presentation. You can even manually control the Case Management view controllers yourself.

In the default 'guest user' flow, launching the interface causes the **Case Publisher** screen to appear. From this screen, a user can create a case as an anonymous guest user. In the default 'authenticated user' flow, launching the interface causes the **Case List** screen to appear. From this screen, a user can inspect an existing case (which launches the **Case Details** screen), or create a new case (which launches the **Case Publisher** screen). If you don't want to use a default user flow, you can manually show the Case Management view controllers.

Note: The fields in the Case Publisher screen are determined by the global action specified in your org. These fields are also shown at the top of the Case Details screen. See the global action step in Quick Setup: Case Publisher as a Guest User for more info.

Activating Interface Using the Default Presentation

Use the `setInterfaceVisible` method to show the Case Management interface using the default presentation.

In Swift:

```
ServiceCloud.shared().cases.setInterfaceVisible(true,
                                                animated: true,
                                                completion: nil)
```

In Objective-C:

```
[[SCServiceCloud sharedInstance].cases setInterfaceVisible:YES
                                                  animated:YES
                                                completion:nil];
```

If you just want to display the default Case Management experience, that's all you need to do. But if you'd like to access the associated view controllers, or their delegates, implement `SCServiceCloudDelegate` and supply that implementation to `ServiceCloud`.

In Swift:

```
ServiceCloud.shared().delegate = self
```

In Objective-C:

```
[SCServiceCloud sharedInstance].delegate = self;
```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud(willDisplay controller:animated:)` method to find out if it's a view controller you're interested in, and if so, assign the view controller delegate.

In Swift:

```
func serviceCloud(_ serviceCloud: ServiceCloud,
                    willDisplay controller: UIViewController,
                    animated: Bool) {

  // Case Publisher View
  if (controller is SCSCasePublisherViewController) {
    let publisherController = controller as! SCSCasePublisherViewController

    // TO DO: Implement SCSCasePublisherViewControllerDelegate
    publisherController.delegate = self

  // Case Detail View
  } else if (controller is SCSCaseDetailViewController) {
    let detailController = controller as! SCSCaseDetailViewController

    // TO DO: Implement SCSCaseDetailViewControllerDelegate
    detailController.delegate = self

  // Case List View
  } else if (controller is SCSCaseListViewController) {
    let listController = controller as! SCSCaseListViewController

    // TO DO: Implement SCSCaseListViewControllerDelegate
    listController.delegate = self
  }

}
```

In Objective-C:

```
-(void)serviceCloud:(SCServiceCloud *)serviceCloud
       willDisplayViewController:(UIViewController *)controller
                        animated:(BOOL)animated {

  // Case Publisher View
  if ([controller isKindOfClass:[SCSCasePublisherViewController class]]) {
```

```
    SCSCasePublisherViewController *casePublisherController =
      (SCSCasePublisherViewController *)controller;

    // TO DO: Implement SCSCasePublisherViewControllerDelegate
    casePublisherController.delegate = self;

  // Case Detail View
  } else if ([controller isKindOfClass:[SCSCaseDetailViewController class]]) {
    SCSCaseDetailViewController *caseDetailController =
      (SCSCaseDetailViewController *)controller;

    // TO DO: Implement SCSCaseDetailViewControllerDelegate
    caseDetailController.delegate = self;

  // Case List View
  } else if ([controller isKindOfClass:[SCSCaseListViewController class]]) {
    SCSCaseListViewController *caseListController =
      (SCSCaseDetailViewController *)controller;

    // TO DO: Implement SCSCaseListViewControllerDelegate
    caseListController.delegate = self;
  }
}
```

You now have access to the view controllers and their delegates.

Activating Interface Using a Custom Transitioning Delegate

You can also present the interface using a custom transitioning animation and custom presentation. Implement a `UIViewControllerTransitioningDelegate`.

1. Supply the `ServiceCloud` shared instance with your `SCServiceCloudDelegate` implementation.

   In Swift:

   ```
   ServiceCloud.shared().delegate = mySCServiceCloudDelegate
   ```

   In Objective-C:

   ```
   [SCServiceCloud sharedInstance].delegate = mySCServiceCloudDelegate;
   ```

2. Implement the `serviceCloud(transitioningDelegateForPresentedController:presenting:)` method in your delegate and return a custom `UIViewControllerTransitioningDelegate` from this method.

   In Swift:

   ```
   func serviceCloud(_ serviceCloud: ServiceCloud,
                       transitioningDelegateForPresentedController
                         presentedController: UIViewController,
                       presenting presentingController: UIViewController)
                       -> UIViewControllerTransitioningDelegate? {

     // TO DO: Put your logic here and then return your transitioning delegate...

     return myTransitioningDelegate
   }
   ```

In Objective-C:

```
- (NSObject<UIViewControllerTransitioningDelegate> *)
                                  serviceCloud:(SCServiceCloud *)serviceCloud
        transitioningDelegateForViewController:(UIViewController *)controller {

  // TO DO: Put your logic here and then return your transitioning delegate...

  return myTransitioningDelegate;
}
```

Showing or Customizing the View Controllers

Instead of having the SDK manage the flow from one view to the next, you can instantiate any of the view controllers and display it manually. When instantiating a view controller, be sure to implement the associated delegate and pass that delegate to the view controller (using the `delegate` property). The delegates allow you to override the default behavior for the views.

If you don't want to manually instantiate a view controller but you still want the ability to control its behavior, implement the `serviceCloud(willDisplay controller:animated:)` and `serviceCloud(didDisplay controller:animated:)` methods of `SCServiceCloudDelegate`. You can access the view controllers from those methods.

Feature	View Controller	Delegate
Case Publisher — for creating new cases	`SCSCasePublisherViewController`	`SCSCasePublisherViewControllerDelegate`
Case List — for viewing a list of a user's cases	`SCSCaseListViewController`	`SCSCaseListViewControllerDelegate`
Case Details — for inspecting the details of one case	`SCSCaseDetailViewController`	`SCSCaseDetailViewControllerDelegate`

Note: If you manually display the **Case List** view controller (`SCSCaseListViewController`), you'll also need to manually display the **Case Detail** view controller (`SCSCaseDetailViewController`). When a user selects a case from the case list, present your **Case Detail** view controller from the `caseList(selectedCaseWithId:)` method in your `SCSCaseListViewControllerDelegate` implementation. If you don't do this, nothing will happen when a user selects a specific case in the case list!

If you choose to manually launch the Case Management view controllers, you can't take advantage of the notification-handling mechanism provided by the SDK (using `showInterface(for:)`). See Notifications with the Service SDK for iOS.

For a use case that involves custom view controllers, see Send Custom Data Using Hidden Fields.

Send Custom Data Using Hidden Fields

You can hide specific Case-related fields in your Case Management views. This behavior is useful if you want to pass information to Service Cloud that does not require user input and that the user shouldn't see. To make this happen, implement the view controller delegates and specify the hidden fields.

Before getting started, you'll need to be sure that your global action layout (that you specified with the `caseCreateActionName` property as described in Quick Setup: Case Publisher as a Guest User) contains the fields you want to hide. This layout is used for both the Case Publisher screen (where users can fill in values for the fields) and the Case Details screen (where users can view the field values). To learn more about quick actions, see Create Global Quick Actions in Salesforce Help.

Once you have the correct fields in your layout, there are two approaches to hiding specific fields:

1. Use the default Case Management view controllers, but implement the view controller delegates.
2. Instantiate (and display) the view controllers yourself, and also implement the view controller delegates.

The first method is useful if you don't want to manually display the views yourself —you simply want to hide specific fields. The second method is useful if you want more control over how and when to display the views. Details about each of the two methods are described later in this section.

Regardless of which method you choose, you'll need to implement the Case Management view controller delegates.

Implementing the Delegates

To affect which fields are shown in your Case Publisher view, implement the `SCSCasePublisherViewControllerDelegate`. Use `casePublisher(fieldsToHideFromCaseFields:)` to specify which fields to hide. This method passes you an array of available fields to hide. This array contains the unique API name for each field (which is not necessarily the label text for the field). From this array, return a set of fields to hide.

In Swift:

```
func casePublisher(_ publisher: SCSCasePublisherViewController,
                   fieldsToHideFromCaseFields availableFields: [String])
                   -> Set<String> {

  let hideFieldSet: Set = ["MyHiddenField", "MyOtherHiddenField"]
  return hideFieldSet
}
```

In Objective-C:

```
- (NSSet *)casePublisher:(SCSCasePublisherViewController *)publisher
           fieldsToHideFromCaseFields:(NSArray *)availableFields {

  NSSet *hideFieldsSet = [NSSet setWithObjects: @"MyHiddenField",
                                                @"MyOtherHiddenField", nil];
  return hideFieldsSet;
}
```

You'll also need to implement the `casePublisher(valuesForHiddenFields:)` method, where you specify what values to use for each hidden field. This method passes you the set of hidden fields (that you specified earlier), and you'll need to provide a dictionary associating each hidden field with a value.

In Swift:

```
func casePublisher(_ publisher: SCSCasePublisherViewController,
                   valuesForHiddenFields hiddenFields: Set<String>)
                   -> [String : Any] {

  let hideValues: [String: String] = [
    "MyHiddenField" : "The value for my hidden field.",
    "MyHiddenField" : "Another value for hidden field."
  ]

  return hideValues
}
```

In Objective-C:

```
- (NSDictionary *)casePublisher:(SCSCasePublisherViewController *)publisher
                  valuesForHiddenFields:(NSSet *)hiddenFields {

  NSDictionary *hideValues = @{
    @"MyHiddenField": @"The value for my hidden field.",
    @"MyHiddenField": @"Another value for hidden field."
  };

  return hideValues;
}
```

Note: Be sure that you specify valid values for hidden fields! If you specify an invalid value, case submission will fail and it will be unclear to the user why it happened.

If you support authenticated users and you hide fields from Case Publisher, you should also hide those fields from the Case Details view. To do this, use a similar approach and implement `SCSCaseDetailViewControllerDelegate`. Use `caseDetail(fieldsToHideFromCaseFields:)` to specify which fields to hide.

In Swift:

```
func caseDetail(_ caseDetailController: SCSCaseDetailViewController,
                  fieldsToHideFromCaseFields availableFields: [String])
                  -> Set<String> {
  let hideFieldSet: Set = ["MyHiddenField", "MyOtherHiddenField"]
  return hideFieldSet
}
```

In Objective-C:

```
- (NSSet *)caseDetail:(SCSCaseDetailViewController *)caseDetailController
           fieldsToHideFromCaseFields:(NSArray *)availableFields {

  NSSet *hideFieldsSet = [NSSet setWithObjects:@"MyHiddenField",
                                               @"MyOtherHiddenField", nil];
  return hideFieldsSet;
}
```

There is no method in this delegate to specify values for hidden fields (as there is in the Case Publisher view controller delegate), because the Case Details view only shows a read-only version of these fields.

Once you've implemented your delegates, you can wire them up with one of two methods.

Method 1: Using the Default View Controllers

If you want to use the default view controllers, you'll need to find out when the view controllers are going to be shown so that you can associate your delegate implementation with the right view controller. To do this, implement `SCServiceCloudDelegate` and supply that implementation to `ServiceCloud`.

In Swift:

```
ServiceCloud.shared().delegate = self
```

In Objective-C:

```
[SCServiceCloud sharedInstance].delegate = self;
```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud(willDisplay controller:animated:)` method to find out if it's a view controller you're interested in, and if so, assign the view controller delegate.

In Swift:

```
func serviceCloud(_ serviceCloud: ServiceCloud,
                    willDisplay controller: UIViewController,
                    animated: Bool) {

  if (controller is SCSCasePublisherViewController) {
    let publisherController = controller as! SCSCasePublisherViewController
    publisherController.delegate = self

  } else if (controller is SCSCaseDetailViewController) {
    let detailController = controller as! SCSCaseDetailViewController
    detailController.delegate = self
  }
}
```

In Objective-C:

```
-(void)serviceCloud:(SCServiceCloud *)serviceCloud
       willDisplayViewController:(UIViewController *)controller
                        animated:(BOOL)animated {

  if ([controller isKindOfClass:[SCSCasePublisherViewController class]]) {
    SCSCasePublisherViewController *casePublisherController =
      (SCSCasePublisherViewController *)controller;
    casePublisherController.delegate = self;

  } else if ([controller isKindOfClass:[SCSCaseDetailViewController class]]) {
    SCSCaseDetailViewController *caseDetailController =
      (SCSCaseDetailViewController *)controller;
    caseDetailController.delegate = self;
  }
}
```

Once you've assigned your delegates, you'll no longer see the hidden fields.

Method 2: Instantiating the View Controllers

You can also instantiate all the view controllers yourself and display them manually. You'll still need to implement the delegates using this method.

Feature	View Controller	Delegate
Case Publisher — for creating new cases	`SCSCasePublisherViewController`	`SCSCasePublisherViewControllerDelegate`
Case List — for viewing a list of a user's cases	`SCSCaseListViewController`	`SCSCaseListViewControllerDelegate`
Case Details — for inspecting the details of one case	`SCSCaseDetailViewController`	`SCSCaseDetailViewControllerDelegate`

Note: If you manually display the **Case List** view controller (`SCSCaseListViewController`), you'll also need to manually display the **Case Detail** view controller (`SCSCaseDetailViewController`). When a user selects a case from the case list, present your **Case Detail** view controller from the `caseList(selectedCaseWithId:)` method in your `SCSCaseListViewControllerDelegate` implementation. If you don't do this, nothing will happen when a user selects a specific case in the case list!

Once you've instantiated a view controller, assign your delegate using the `delegate` property on that controller. Keep in mind that you'll need to display the Case Detail view controller from the `caseList(selectedCaseWithId:)` method in your `SCSCaseListViewControllerDelegate` implementation.

In Swift:

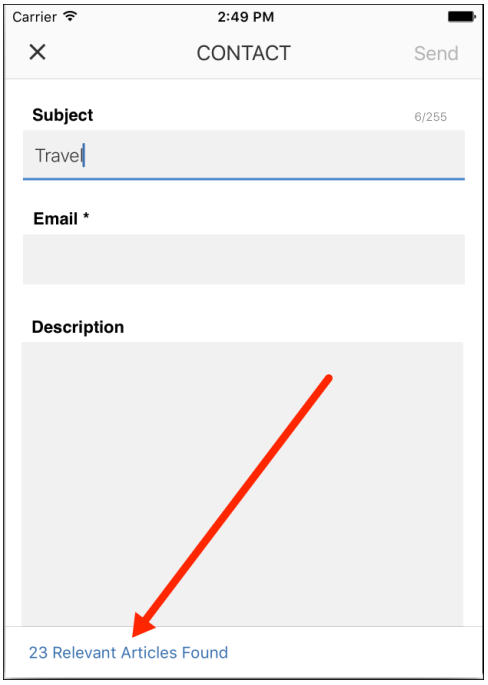
```
func caseList(_ caseList: SCSCaseListViewController,
                selectedCaseWithId caseId: String) {

  let controller = SCSCaseDetailViewController(caseId: caseId)
  controller.delegate = self
  caseList.navigationController!.pushViewController(controller, animated: true)
}
```

In Objective-C:

```
- (void)caseList:(SCSCaseListViewController*)caseList
        selectedCaseWithId:(NSString*)caseId {

  SCSCaseDetailViewController *controller =
    [[SCSCaseDetailViewController alloc] initWithCaseId:caseId];
  controller.delegate = self;
  [caseList.navigationController pushViewController:controller animated:YES];
}
```

Configure Case Deflection

When a user enters information about a new case – if you have a knowledge base available to that user – the SDK automatically searches that knowledge base for relevant articles and offers them to the user. You have the ability to turn this feature on and off, as well as control which case publisher fields are used to search content.



By default, this feature is enabled and it searches the Subject and Description fields for relevant articles using those search terms. You can change this behavior using methods in the `SCSCasePublisherViewControllerDelegate` class. This is the delegate class for `SCSCasePublisherViewController`. You can point the `SCSCasePublisherViewController` instance to your delegate implementation in one of two ways:

1. If you are displaying the Case Publisher using the default presentation (that is, using the `setInterfaceVisible` method), refer to the **Activating Interface Using the Default Presentation** section of Customize the Presentation and View Controllers for Case Management.
2. If you are displaying the Case Publisher by manually presenting the view controller, refer to the **Showing or Customizing the View Controllers** section of Customize the Presentation and View Controllers for Case Management.

You can turn case deflection on and off with the `shouldEnableCaseDeflection(forPublisher:)` method. You can control which fields are used for searching knowledge base articles using the `casePublisher(fieldsForCaseDeflection:)` method.

In Swift:

```
func casePublisher(_ publisher: SCSCasePublisherViewController,
                   fieldsForCaseDeflection availableFields: [String])
                   -> Set<String> {

  // Create the complete set of fields we want to use to search knowledge base
  let deflectionSearch: Set<String> = ["CustomSubject__c","CustomDescription__c"]

  // Now build the list of fields that are confirmed to be within
  // the existing case publisher layout
  var deflectionSearchFieldsInLayout = [String]()
  for field: String in deflectionSearch {
    if (availableFields.contains(field)) {
      deflectionSearchFieldsInLayout.append(field)
    }
  }
```

```
  // Return the list of fields
  return Set(deflectionSearchFieldsInLayout)
}
```

In Objective-C:

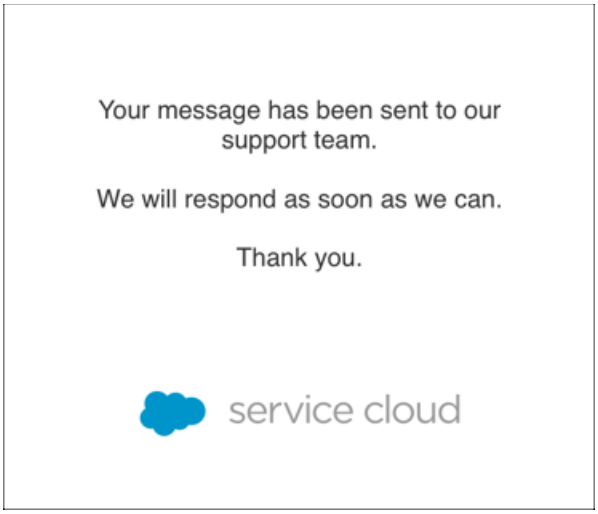
```
- (NSSet<NSString *> *)casePublisher:(SCSCasePublisherViewController *)publisher
             fieldsForCaseDeflection:(NSArray<NSString *> *)availableFields {

    // Create the complete set of fields we want to use to search knowledge base
    NSSet *deflectionSearch =
           [NSSet setWithObjects:@"CustomSubject__c",@"CustomDescription__c", nil];

    // Now build the list of fields that are confirmed to be within
    // the existing case publisher layout
    NSMutableSet *deflectionSearchFieldsInLayout = [NSMutableSet new];
    for (NSString *field in deflectionSearch) {
        if ([availableFields containsObject:field]) {
            [deflectionSearchFieldsInLayout addObject:field];
        }
    }

    // Return the list of fields
    return deflectionSearchFieldsInLayout;
}
```

Note: Before you can use case deflection, be sure to configure your environment with a category group and root data category (using the `SCSServiceConfiguration` object). For more information, see Quick Setup: Knowledge in the Service SDK.

Customize the Case Publisher Result View

By default, when a user submits a new case from the Case Publisher screen, a standard success view appears. If you want to provide your users with more specific guidance after a case is created, one solution is to customize the view's message text and default image. If you'd like more control over what is displayed, you can present your own view by implementing the `viewForResult` method in the `SCSCasePublisherViewControllerDelegate` class.

The default Case Publisher success view contains this content:



To customize the standard message text (`ServiceCloud.CasePublisher.SuccessMessage`) and the image (`CaseSubmitSuccess`), see Customize and Localize Strings with the Service SDK and Customize Images with the Service SDK for instructions.

To create your own view:

1. Implement the `SCSCasePublisherViewControllerDelegate` class.

   Implement the `casePublisher(viewFor:withCaseId:error:)` method where you create your custom view and return it to the SDK. If you don't implement this method (or if you return `nil`), the SDK presents the default view.

   In Swift:

   ```
   func casePublisher(_ publisher: SCSCasePublisherViewController,
                      viewFor result: SCSCasePublisherResult,
                      withCaseId caseId: String?,
                      error: Error?)
                      -> UIView? {

     let myResultView = // TO DO: Create my custom view
     return myResultView
   }
   ```

   In Objective-C:

   ```
   - (UIView *)casePublisher:(SCSCasePublisherViewController *)publisher
               viewForResult:(SCSCasePublisherResult)result
                  withCaseId:(nullable NSString *)caseId
                       error:(nullable NSError *)error {

     UIView* myResultView = // TO DO: Create my custom view
     return myResultView;
   }
   ```

   Note: The SDK only calls the `casePublisher(viewFor:withCaseId:error:)` method when a case submission is successful.

2. Register your delegate class with the `SCSCasePublisherViewController`.

   The process for registering your delegate class is different depending on whether you're manually instantiating a Case Publisher view controller or whether you're using the default view controller.

   a. If you're manually instantiating a Case Publisher view controller, register your delegate with the `delegate` property of this class. For more information, see Customize the Presentation and View Controllers for Case Management.

   b. If you're not manually instantiating the Case Publisher view controller, you can get the default view controller using the Service Cloud delegate.

      Implement `SCServiceCloudDelegate` and supply that implementation to `ServiceCloud`.

      In Swift:

      ```
      ServiceCloud.shared().delegate = self
      ```

      In Objective-C:

      ```
      [SCServiceCloud sharedInstance].delegate = self;
      ```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud(willDisplay controller:animated:)` method to find the `SCSCasePublisherViewController` class and register your view controller delegate.

In Swift:

```
func serviceCloud(_ serviceCloud: ServiceCloud,
                  willDisplay controller: UIViewController,
                  animated: Bool) {

  if controller is SCSCasePublisherViewController {

    let publisherController = controller as! SCSCasePublisherViewController

    publisherController.delegate = // TO DO: Specify your case publisher delegate
  }
}
```

In Objective-C:

```
- (void)serviceCloud:(SCServiceCloud *)serviceCloud
       willDisplayViewController:(UIViewController *)controller
                        animated:(BOOL)animated {

  if ([controller isKindOfClass:[SCSCasePublisherViewController class]]) {

    SCSCasePublisherViewController *casePublisherController =
      (SCSCasePublisherViewController *)controller;

    casePublisherController.delegate = // TO DO: Specify your case publisher delegate

  }
}
```

After you supply your view to the Service SDK, the SDK presents it when a case is submitted.

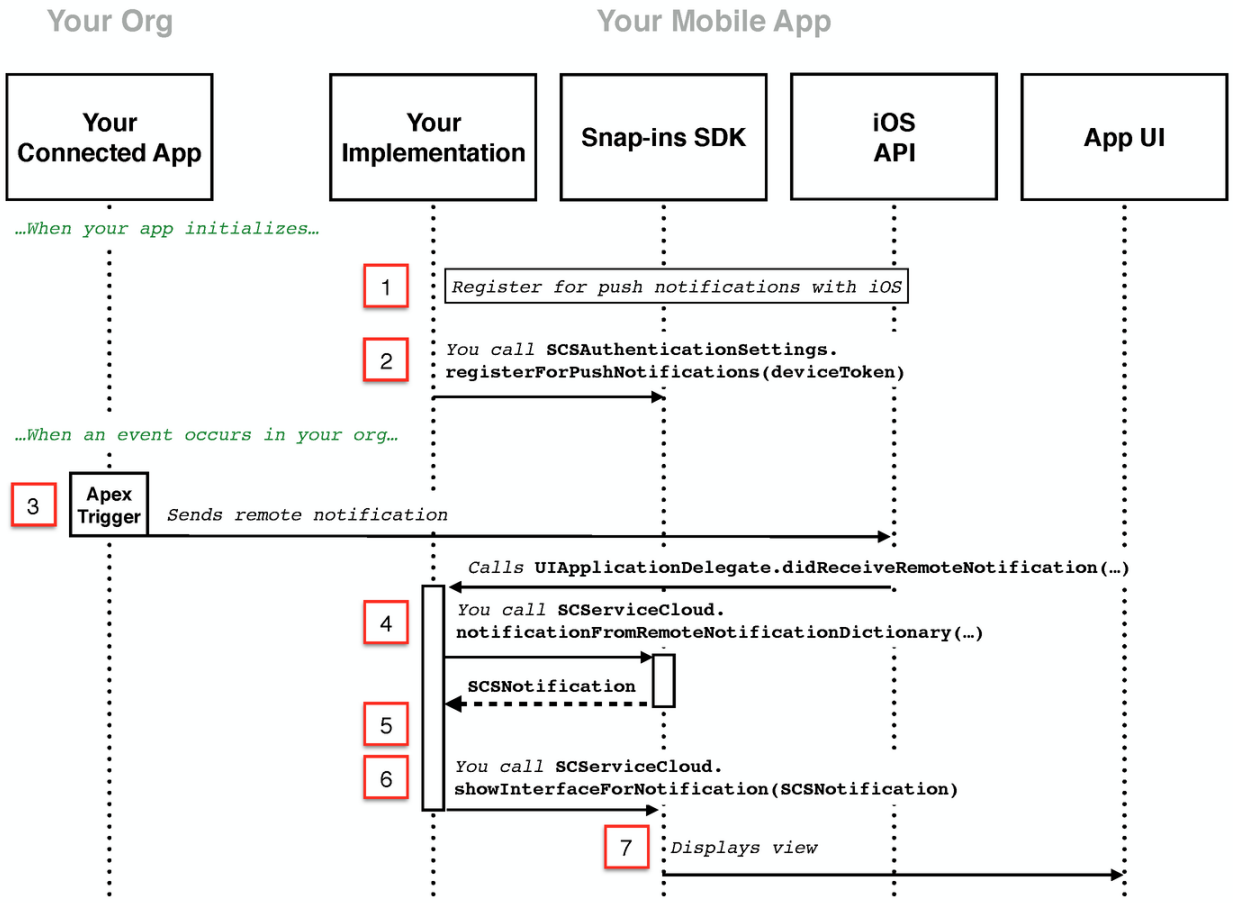
Push Notifications for Case Activity

You can send push notifications from your org when activity associated with a user's case occurs. After you've set up notifications in your org, handle the notification from your app.

Before you begin, set up your app for authenticated Case Management, as described in Case Management as an Authenticated User.

When building your app, you may have remote notifications arrive from your Salesforce org. You can ask the SDK whether it can handle a notification, and if it can, you can tell the SDK to show the appropriate view.

The following sequence diagram illustrates a scenario where an Apex trigger sends a notification to your app and your app displays the associated view.



1. Set up the push notification mechanism for your iOS app. To learn more, see Notifications in the Apple documentation. Make sure that you register for push notifications and that the user is prompted to allow notifications. When iOS calls you back with `UIApplicationDelegate.didRegisterForRemoteNotificationsWithDeviceToken`, hold onto the `deviceToken` object.
2. After authentication, call the `registerForPushNotifications` method on your `SCSAuthenticationSettings` instance using the `deviceToken` object from the previous step. To learn more about authentication, see Authentication with the Service SDK for iOS.
3. Create an Apex trigger in your org to detect activity.

   For general information about Apex triggers, see Using Apex Triggers to Send Push Notifications in the Salesforce Mobile Push Notifications Implementation Guide.

   You can use the following sample Apex code as a starting point. This Chatter Feed Item trigger sends a notification when an agent creates a text post on a case.

```
// THIS APEX TRIGGER IS PROVIDED AS AN EXAMPLE. BE SURE TO REVIEW
// YOUR CODE BEFORE PUTTING ANYTHING INTO PRODUCTION.

trigger newCaseFeedItemNotification on FeedItem (after insert) {

  for(FeedItem feedItem : Trigger.new) {
```

```
try {

  Schema.SObjectType objectType = feedItem.parentId.getSObjectType();

  if(feedItem.body == null ) {
    // Don't push if we have no body.
    break;
  }

  // Ensure Case type
  if (objectType == Case.sObjectType) {

    Case cs = [SELECT contactId, ownerId, caseNumber, subject
               FROM Case
               WHERE id = :feedItem.parentId];
    Set<String> users = new Set<String>();

    // Determine who created or inserted this feed item
    String commentedById = feedItem.CreatedById;
    if (commentedById == null) {
      commentedById = feedItem.InsertedById;
      if (commentedById == null) {
        commentedById = feedItem.LastEditById;
      }
    }

    // If the FeedItem was not created by the owner, send to the owner
    if (cs.ownerId != null && !cs.ownerId.equals(commentedById)) {

      // Ensure the user has access to the feed item before pushing
      List<UserRecordAccess> accessList = [SELECT HasReadAccess, RecordId
        FROM UserRecordAccess
        WHERE UserId = :cs.ownerId
        AND RecordId = :feedItem.Id LIMIT 1];
      if (accessList != null && !accessList.isEmpty()
                             && accessList[0].HasReadAccess) {
        users.add(cs.ownerId);
      }
    }

   // If the FeedItem was not created by the contact on the case send to the contact

    if (cs.contactId != null && !cs.contactId.equals(commentedById)) {

      // Ensure the user has access to the feed item before pushing
      List<UserRecordAccess> accessList = [SELECT HasReadAccess, RecordId
        FROM UserRecordAccess
        WHERE UserId = :cs.contactId
        AND RecordId = :feedItem.Id LIMIT 1];
      if (accessList != null && !accessList.isEmpty()
                             && accessList[0].HasReadAccess) {
        users.add(cs.contactId);
      }
    }
```

```
        // Assemble the necessary payload parameters for the mobile app.
        // Params are:
        // (<alert text>,<alert sound>,<badge count>,<free-form data>)
        // This example doesn't use badge count but does make use of free-form
        // data to pass the caseId in the notification.
        // The number of notifications that haven't been acted
        // upon by the intended recipient is best calculated
        // at the time of the push. This timing helps
        // ensure accuracy across multiple target devices.

        // If subject is not set, use '(No Subject)'
        String subject = cs.subject;
        if (subject == null) {
          subject = '(No Subject)';
        }

        String alertText = 'New comment added to case: ' + subject;

        // Add the caseId so we can handle the push notification within the app
        Map<String, Object> freeFormData = new Map<String, Object>();
        freeFormData.put('caseid', cs.id);

        // Add any other free form data here...

        // Create the payload
        Messaging.PushNotification msg = new Messaging.PushNotification();

        // Format for apple devices
        Map<String, Object> payload =
          Messaging.PushNotificationPayload.apple(alertText, '', null, freeFormData);

        // Add payload to the notification
        msg.setPayload(payload);

        // Needs to match your connected app name
        msg.send('YourConnectedAppName', users);
      }
    }
    catch (Exception e) {
      // Catch everything to ensure push failures do not prevent posts from succeeding.

      // TO DO: Add logging here to record errors or display an error message.
    }

  } // end of for loop
}
```

When this notification is sent to your app, you can have the SDK handle this notification for you. The SDK can display the Case Details or Case List screen for an authenticated user with the relevant case information showing.

Note: If you want the SDK to handle the notification for you, your free-form data **must** contain the case ID, as shown in the code snippet. Without this information, the SDK cannot interpret the contents of the notification.

4. When you receive a remote notification (from your app delegate's `didReceiveRemoteNotification` method), pass notification information to `notification(fromRemoteNotificationDictionary:)` to determine whether the SDK can handle the notification.

   In Swift:

   ```
   func application(_ application: UIApplication,
     didReceiveRemoteNotification userInfo: [AnyHashable : Any],
     fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult)
     -> Void) {

     let notification =
       ServiceCloud.shared().notification(fromRemoteNotificationDictionary: userInfo)

     // TO DO: Handle notification here
   }
   ```

   In Objective-C:

   ```
   - (void)application:(UIApplication *)application
           didReceiveRemoteNotification:(NSDictionary *)userInfo
           fetchCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler {

       SCSNotification *notification =
         [[SCServiceCloud sharedInstance]
         notificationFromRemoteNotificationDictionary:userInfo];

       // TO DO: Handle notification here
   }
   ```

   This method returns `nil` if the SDK can't handle the notification; it returns an `SCSNotification` object if it *can* handle the notification.

5. (Optional) If you want to handle the notification by yourself (like to display your own view), you can inspect the notification type to determine what feature area the notification is associated with. This property returns a `SCSNotificationType` enum.

   ```
   notification.notificationType
   ```

   When dealing with case activity notifications, this value is `SCSNotificationTypeCase`.

6. If you want the SDK to handle the notification for you, call `showInterface(for:)` and pass in the `SCSNotification` object.

   In Swift:

   ```
   ServiceCloud.shared().showInterface(for: notification!)
   ```

   In Objective-C:

   ```
   [[SCServiceCloud sharedInstance] showInterfaceForNotification:notification];
   ```

   Note: If you choose to manually launch the Case Management view controllers, you can't take advantage of the notification-handling mechanism provided by the SDK (using `showInterface(for:)`). See Customize the Presentation and View Controllers for Case Management.

7. The SDK displays or updates the relevant view for the notification.

When dealing with case activity notifications, this command displays the Case List or Case Details screen (depending on the notification), showing the relevant case information. If the appropriate screen is already showing, it refreshes.

Automated Email Responses

Create automated email responses when a case is submitted from your app.

If you'd like to create an automated response when a user of your app submits a case, you can set up a **Case Auto-Response Rule**. To learn more, see the documentation on Salesforce Help: Set Up Auto-Response Rules.

Note: When creating an auto-response rule for guest users, be sure to add the **Web Email** field to the action layout that you've specified with the `caseCreateActionName` property. This field is used for the email response.

Once you've created the rule, a user who submits a case from your app receives an automated response.

Using SOS with the Service SDK

Add the SOS experience to your mobile app.

SOS in the Service SDK for iOS

Learn about the SOS experience using the SDK.

Quick Setup: SOS in the Service SDK

To start an SOS session from your iOS app, use `SOSSessionManager`.

Configure an SOS Session

Before starting an SOS session, you can optionally configure the session using the `SOSOptions` object. These configuration settings allow you to enable or disable cameras, determine what screen a session starts on, specify whether network tests are performed, and control other features.

Two-Way Video

In addition to screen sharing, the SOS SDK lets your customer share their device's live camera feed with an agent. The customer's front-facing camera allows for a video conversation with an agent. The back-facing camera provides a great way for a customer to show something to an agent, rather than have to explain it.

SOS Events and Errors

Implement `SOSDelegate` to be notified about state changes made before, during, and after an SOS call. This delegate also allows you to listen for error conditions so you can present alerts to the user when applicable.

Quality-of-Service Events

Check your audio and video quality-of-service (QoS) to detect packet loss and other streaming issues between the OpenTok media router and your org.

Check SOS Agent Availability

Before starting a session, you can check the availability of your SOS agents and then provide your users with more accurate expectations.

Enable and Disable Screen Sharing

There are some scenarios where you may want to programmatically turn off screen sharing in mid-session. You can enable and disable screen sharing using the `screenSharing` property.

Field Masking

If an application contains sensitive information that an agent shouldn't see during an SOS session, you can hide this information from the agent.

Custom Data

Use custom data to identify customers, send error messages, issue descriptions, or identify the page the SOS session was initiated from.

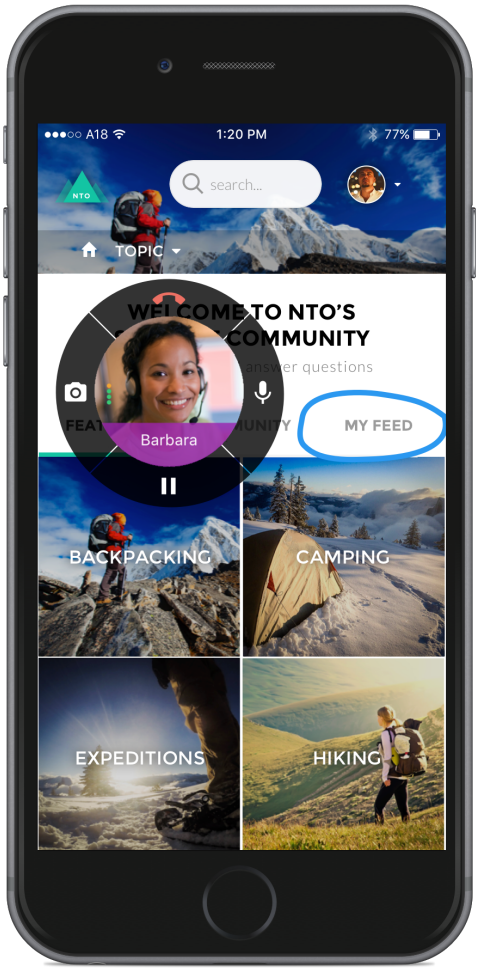
Replace the SOS UI

If you'd like to customize the SOS UI, you can create your own UI by subclassing the `UIViewController` class associated with that phase of the SOS session.

SOS in the Service SDK for iOS

Learn about the SOS experience using the SDK.

SOS lets you easily add real-time video and screen sharing support to your native iOS app. Once you've set up Service Cloud for SOS, it takes just a few calls to the SDK to have your app ready to handle agent calls and to support screen sharing. With screen sharing, agents can even make annotations directly on the customer's screen.



And with just a few more configuration changes, you can provide two-way video support from your app. This functionality can include front-facing camera support, back-facing camera support, or both.

There are several other ways you can set up your SOS environment, including masking sensitive fields and passing custom data back to your org.

You can also customize the look and feel of the interface so that it fits naturally within your app. These customizations include the ability to fine-tune the colors, the fonts, the images, and the strings used throughout the interface.

Check out SOS Events and Errors for information about how to handle event changes. In particular, you'll want to listen for error conditions and present alerts to the user when applicable.

Let's get started.

Quick Setup: SOS in the Service SDK

To start an SOS session from your iOS app, use `SOSSessionManager`.

Before running through these steps, be sure you've already:

- Set up Service Cloud to work with SOS. To learn more, see Console Setup for SOS.
- Installed the SDK. To learn more, see Install the Service SDK for iOS.

Once you've reviewed these prerequisites, you're ready to begin.

1. Import the SDK. Wherever you intend to use the SOS SDK, be sure to import the Service Common framework and the SOS framework.

   In Swift:

   ```
   import ServiceCore
   import ServiceSOS
   ```

In Objective-C:

```
@import ServiceCore;
@import ServiceSOS;
```

2. Create an `SOSOptions` object with information about your LiveAgent pod, your Salesforce org ID, and the deployment ID.

   In Swift:

   ```
   let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                            orgId: "YOUR-ORG-ID",
                            deploymentId: "YOUR-DEPLOYMENT-ID")
   ```

   In Objective-C:

   ```
   SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                       orgId:@"YOUR-ORG-ID"
                                                deploymentId:@"YOUR-DEPLOYMENT-ID"];
   ```

   Note: You can get the required parameters for this method from your Salesforce org. If your Salesforce admin hasn't already set up SOS in Service Cloud or you need more guidance, see Console Setup for SOS.

3. (Optional) Specify additional configuration settings before starting a session.

   Before starting an SOS session, you can optionally configure the session using the `SOSOptions` object. These configuration settings allow you to enable or disable cameras, determine what screen a session starts on, specify whether network tests are performed, and control other features. To learn more, see Configure an SOS Session.

4. (Optional) Customize the appearance with the configuration object.

   You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor`, `setFontDescriptor`, and `setImage`. You can also configure the strings used throughout the interface. See SDK Customizations with the Service SDK for iOS.

5. To start an SOS session, call `startSession` on the `SOSSessionManager` shared instance.

   You can start a session when the view controller loads, or from a UI action.

   In Swift:

   ```
   ServiceCloud.shared().sos.startSession(with: options)
   ```

   In Objective-C:

   ```
   [[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
   ```

   You can provide an optional completion block to execute code when the session has been fully connected to all services. During a successful session initialization, the SDK calls the completion block at the point that the session is active and the user is waiting for an agent to join. If there is a failure, the SDK calls the completion block with the associated error.

   Note: If your app crashes when it is in the process of connecting to an SOS session, check that you've enabled the device privacy permissions for the camera and the microphone. An app will crash if these permissions are not set in Xcode. See Install the Service SDK for iOS.

   For instructions on launching the interface from a web view, see Launch the Service SDK from a Web View in iOS.

6. Listen for events and handle error conditions.

   You can listen for state changes that occur during a session life cycle by implementing `SOSDelegate` methods. Register this delegate using the `add(delegate: SOSDelegate!)` method on your SOS instance (from `ServiceCloud`). In particular,

we suggest that you implement the `sos(didStopWith:error:)` method to handle session termination. See SOS Events and Errors.

Note: The SDK doesn't show an alert when a session fails to start, or when a session ends. It's your responsibility to listen to events and display an error when appropriate.

7. (Optional) If you want to programmatically stop a session, call the `stopSession` method on the `SOSSessionManager` shared instance.

   In Swift:

   ```
   ServiceCloud.shared().sos.stopSession()
   ```

   In Objective-C:

   ```
   [[SCServiceCloud sharedInstance].sos stopSession];
   ```

   Alternatively, you can call the `stopSession(completion:)` method with a completion block.

For additional details on customizing the SOS experience in your app, see the other topics covered in Using SOS with the Service SDK. If you run into network issues while connecting with an agent, see SOS Network Troubleshooting Guide.

Example: To use this example code, create a Single View Application and Install the Service SDK for iOS.

Use the storyboard to add a button to the view. Add a `Touch Up Inside` action in your `UIViewController` implementation with the name `startSOS`. In the view controller code:

- Implement the `SOSDelegate` protocol so that you can be notified when there are errors or state changes.
- Specify `self` as an SOS delegate.
- Start an SOS session in the button action.
- Implement the `sos(didStopWith:error:)` method and show a dialog when appropriate.

In Swift:

```
import UIKit
import ServiceCore
import ServiceSOS

class ViewController: UIViewController, SOSDelegate {

  override func viewDidLoad() {
    super.viewDidLoad()

    // Add our SOS delegate
    ServiceCloud.shared().sos.add(self)
  }

  @IBAction func startSOS(sender: AnyObject) {

    // Create options object
    let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                             orgId: "YOUR-ORG-ID",
                             deploymentId: "YOUR-DEPLOYMENT-ID")

    // Start the session
    ServiceCloud.shared().sos.startSession(with: options)
  }
```

```
// Delegate method for session stop event.
// You can also check for fatal errors with this delegate method.
func sos(_ sos: SOSSessionManager!, didStopWith reason: SOSStopReason,
         error: Error!) {

  var title = ""
  var description = ""

  // If there's an error...
  if (error != nil) {

    switch (error as NSError).code {

      // No agents available
      case SOSErrorCode.SOSNoAgentsAvailableError.rawValue:
        title = "Session Failed"
       description = "It looks like there are no agents available. Try again later."


      // Insufficient network error
      case SOSErrorCode.SOSInsufficientNetworkError.rawValue:
        title = "Session Failed"
        description = "Insufficient network. Check network quality and try again."

      // TO DO: Use SOSErrorCode to check for ALL other error conditions
      //        in order to give a more clear explanation of the error.
      default:
        title = "Session Error"
        description = "Unknown session error."
      }

  // Else if session stopped without an error condition...
  } else {

    switch reason {

      // Handle the agent disconnect scenario
      case .agentDisconnected:
        title = "Session Ended"
        description = "The agent has ended the session."

      // TO DO: Use SOSStopReason to check for
      //        other reasons for session ending...
      default:
        break
    }
  }

  // Display dialog if we have something to say...
  if (title != "") {

    let alert = UIAlertController(title: title,
                                  message: description,
```

```
                                    preferredStyle: .alert)
      let okAction = UIAlertAction(title: "OK",
                                   style: .default,
                                   handler: nil)
      alert.addAction(okAction)
      self.present(alert, animated: true, completion: nil)
    }
  }
}
```

In Objective-C:

```
#import <UIKit/UIKit.h>
@import ServiceCore;
@import ServiceSOS;

@interface ViewController : UIViewController <SOSDelegate>

@end

@implementation ViewController

- (void)viewDidLoad {
  [super viewDidLoad];

  // Add our SOS delegate
  [[SCServiceCloud sharedInstance].sos addDelegate:self];
}

- (IBAction)startSOS:(id)sender {

  // Create options object
  SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                      orgId:@"YOUR-ORG-ID"
                                               deploymentId:@"YOUR-DEPLOYMENT-ID"];

  // Start the session
  [[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
}

// Delegate method for session stop event.
// You can also check for fatal errors with this delegate method.
- (void)sos:(SOSSessionManager *)sos didStopWithReason:(SOSStopReason)reason
                                                 error:(NSError *)error {
  NSString *title = nil;
  NSString *description = nil;

  // If there's an error...
  if (error != nil) {

    switch (error.code) {

      // No agents available
      case SOSNoAgentsAvailableError: {
        title = @"Session Failed";
```

```
      description = @"It looks like there are no agents available. Try again later.";

      break;
    }

    // Network test failure
    case SOSNetworkTestError: {
      title = @"Session Failed";
      description = @"Insufficient network. Check network quality and try again.";
      break;
    }

    // TO DO: Use SOSErrorCode to check for ALL other error conditions
    //        in order to give a more clear explanation of the error.
    default: {
      title = @"Session Error";
      description = @"Unknown session error.";
      break;
    }
  }

// Else if session stopped without an error condition...
} else {

  switch (reason) {

      // Handle the agent disconnect scenario
      case SOSStopReasonAgentDisconnected: {
          title = @"Session Ended";
          description = @"The agent has ended the session.";
          break;
      }

      // TO DO: Use SOSStopReason to check for
      //        other reasons for session ending...
      default: {
          break;
      }
  }
}

// Display dialog if we have something to say...
if (title != nil) {
    UIAlertController *alert = [UIAlertController
                                alertControllerWithTitle:title
                                message:description
                                preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction* okAction = [UIAlertAction
                               actionWithTitle:@"OK"
                               style:UIAlertActionStyleDefault
                               handler:^(UIAlertAction * action)
                               {
                                   NSLog(@"OK action");
```

```
                                    }];

        [alert addAction:okAction];
        [self presentViewController:alert animated:YES completion:nil];
    }
}

@end
```

Configure an SOS Session

Before starting an SOS session, you can optionally configure the session using the `SOSOptions` object. These configuration settings allow you to enable or disable cameras, determine what screen a session starts on, specify whether network tests are performed, and control other features.

When you start an SOS session using `SOSSessionManager`, you pass an `SOSOptions` object as one of the arguments. This object contains all the configuration settings necessary for SOS to start a session. To create an `SOSOptions` object, you specify information about your org and deployment (as described in Quick Setup: SOS in the Service SDK), and that is all that is required. However, there are many other options you can set using `SOSOptions`.

Note: Be sure not to start an SOS session until you've fully configured the `SOSOptions` object.

The following features are available for configuration:

Feature	Description	Type & Default Value
`customFieldData`	Dictionary that can be used to send custom data to your Salesforce org. See Custom Data.	`NSMutableDictionary` — Default: `nil`
`featureAgentVideoStreamEnabled`	Whether the agent video stream is enabled for the session.	`BOOL` — Default: `YES/true`
`featureClientBackCameraEnabled`	Whether the back-facing camera is enabled for the session. See Two-Way Video.	`BOOL` — Default: `NO/false`
`featureClientFrontCameraEnabled`	Whether the front-facing (selfie) camera is enabled for the session. See Two-Way Video.	`BOOL` — Default: `NO/false`
`featureClientScreenSharingEnabled`	Whether screen sharing is enabled for the session.	`BOOL` — Default: `YES/true`
`featureNetworkTestEnabled`	Whether the network test is enabled before and during a session.	`BOOL` — Default: `YES/true`
`initialAgentStreamPosition`	The initial center position of the UI containing the agent video and SOS control buttons.	`CGPoint`
`initialAgentVideoStreamActive`	Whether the agent video stream is active when starting a session.	`BOOL` — Default: `YES/true`

Feature	Description	Type & Default Value
`initialCameraType`	The initial view (screen sharing, front-facing camera, or back-facing camera) when starting a session.	`SOSCameraType` enumerated type — Default: `screenSharing`
`remoteLoggingEnabled`	Determines whether session logs are sent for collection. Logs sent remotely do not collect personal information. Unique IDs are created for tying logs to sessions and those IDs cannot be correlated back to specific users.	`BOOL` — Default: `YES/true`
`sessionRetryTime`	The length of time (in seconds) before SOS prompts the user to retry or cancel.	`NSTimeInterval` — Default: 30
`setViewControllerClass`	Lets you override the SOS UI. See Replace the SOS UI.	`SOSUIPhase` enumerated type.

As you can see from the table, by default, an SOS session starts in screen sharing mode, with both cameras disable. If you don't want default values, manually change the options before starting a session.

Example: The following example starts a session with the back-facing camera showing, the front-facing camera enabled, and screen sharing disabled.

In Swift:

```
let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                         orgId: "YOUR-ORG-ID",
                         deploymentId: "YOUR-DEPLOYMENT-ID")

options!.featureClientBackCameraEnabled = true
options!.featureClientFrontCameraEnabled = true
options!.featureClientScreenSharingEnabled = false
options!.initialCameraType = .backFacing

ServiceCloud.shared().sos.startSession(with: options)
```

In Objective-C:

```
SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                    orgId:@"YOUR-ORG-ID"
                                             deploymentId:@"YOUR-DEPLOYMENT-ID"];

[options setFeatureClientBackCameraEnabled: YES];
[options setFeatureClientFrontCameraEnabled: YES];
[options setFeatureClientScreenSharingEnabled: NO];
[options setInitialCameraType: SOSCameraTypeBackFacing];

[[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
```

Two-Way Video

In addition to screen sharing, the SOS SDK lets your customer share their device's live camera feed with an agent. The customer's front-facing camera allows for a video conversation with an agent. The back-facing camera provides a great way for a customer to show something to an agent, rather than have to explain it.

By default, after a connection is established, the camera shows the agent in the full-screen view and the customer's camera in the picture-in-picture view. If a device has both a front-facing and back-facing camera, the customer can swap cameras by double-tapping the screen during the two-way video session. The customer can also tap the picture-in-picture view to swap the full-screen view with the picture-in-picture view.

You can programmatically configure which cameras are available and which camera the session starts with.

Configure Two-Way Video

Two-way video for SOS is disabled by default. You can enable it by accessing one or both cameras in the user's device with the `SOSOptions` object used when starting the SOS session. You can also configure the initial camera view when the session starts.

Configure Two-Way Video

Two-way video for SOS is disabled by default. You can enable it by accessing one or both cameras in the user's device with the `SOSOptions` object used when starting the SOS session. You can also configure the initial camera view when the session starts.

Note: Always test two-way video on an actual device, rather than using the simulator. The Xcode simulator doesn't have access to a camera, so it doesn't provide you with an accurate experience.

1. Create an `SOSOptions` object using `SOSOptions(liveAgentPod:orgId:deploymentId:)`, as described in Quick Setup: SOS in the Service SDK.
2. Initialize access to one or both of the cameras on the user's device by setting `featureClientFrontCameraEnabled` and `featureClientBackCameraEnabled`.

   In Swift:

   ```
   options?.featureClientFrontCameraEnabled = true
   options?.featureClientBackCameraEnabled = true
   ```

   In Objective-C:

   ```
   [options setFeatureClientFrontCameraEnabled: YES];
   [options setFeatureClientBackCameraEnabled: YES];
   ```

   To learn more, see Configure an SOS Session.

3. Determine what is displayed when a session starts.

   By default, a session starts in screen sharing mode. Alternatively, you can start a session using one of the cameras. For example, the following command starts a session using the back-facing camera.

   In Swift:

   ```
   options?.initialCameraType = .backFacing
   ```

   In Objective-C:

   ```
   [options setInitialCameraType: SOSCameraTypeBackFacing];
   ```

   If you'd rather start a session with the front-facing camera, use this command instead.

   In Swift:

   ```
   options?.initialCameraType = .frontFacing
   ```

   In Objective-C:

   ```
   [options setInitialCameraType: SOSCameraTypeFrontFacing];
   ```

   To learn more, see Configure an SOS Session.

4. Determine whether you want to allow the screen sharing feature.

   Even if you start a session with the camera, the screen sharing function is still enabled by default. This functionality may be appropriate for your use case. However, you can disable screen sharing altogether with the following call.

   In Swift:

   ```
   options?.featureClientScreenSharingEnabled = false
   ```

In Objective-C:

```
[options setFeatureClientScreenSharingEnabled: NO];
```

To learn more, see Configure an SOS Session.

5. Start an SOS session.

   In Swift:

   ```
   ServiceCloud.shared().sos.startSession(with: options)
   ```

   In Objective-C:

   ```
   [[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
   ```

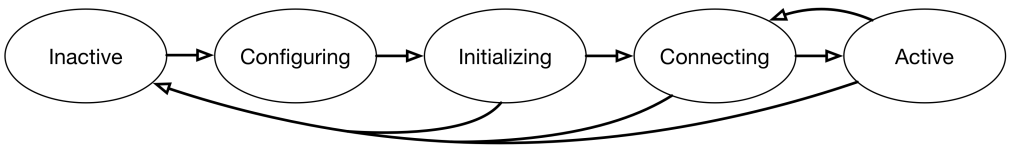
   To learn more, see Quick Setup: SOS in the Service SDK.

SOS Events and Errors

Implement `SOSDelegate` to be notified about state changes made before, during, and after an SOS call. This delegate also allows you to listen for error conditions so you can present alerts to the user when applicable.

Note: This topic covers how to handle state changes from your mobile app. For information about how to handle state changes from the Salesforce console, see Listen for SOS Console Events.

The `SOSSessionManager` singleton (`sos`) maintains all information related to SOS sessions. One of its properties (`state`) is an `SOSSessionState` object, which maintains the current state of SOS. This state object can be in one of five different states:



Inactive
No active session; no outgoing/incoming SOS traffic.

Configuring
Performing a pre-initialization configuration step, such as network testing.

Initializing
Preparing to connect.

Connecting
Attempting a connection to an agent.

Active
Connected with agent; session is fully established.

Throughout a session, your application might want information related to the session state. You can monitor state changes by implementing `SOSDelegate`. Use the `add(delegate: SOSDelegate!)` method on `SOSSessionManager` to register your delegate.

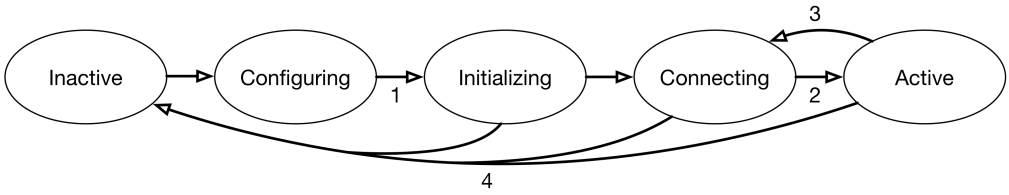
Listening to State Changes

If you want to know every time an SOS session state changes, the `sos(stateDidChange:current:previous:)` method is called for every change.

A more specific set of delegate methods are available to track specific state changes:

1. `sosDidStart` — invoked when a session is possible and the user has agreed to start a session.
2. `sosDidConnect` — invoked when the SOS session has connected.
3. `sosWillReconnect` — invoked if network connectivity issues arise, and the session will attempt to reconnect.
4. `sos(didStopWith:error:)` — invoked when the session has ended.

These state changes are labeled with an associated number in the diagram below.



Handling Session Termination and Error Conditions

The SDK doesn't present UI alerts for session termination or error conditions so you'll need to listen for these events and decide what to show your users. There are two `SOSDelegate` methods for this purpose:

1. To track session termination, use the `sos(didStopWith:error:)` method . Inspect the reason (`SOSStopReason`) to determine why the session stopped. Typically, the session stops due to a normal event (for example, `SOSStopReasonUserDisconnected` or `SOSStopReasonAgentDisconnected`). If the reason is `SOSStopReasonSessionError`, check the `error` parameter for more detail and compare the error code to `SOSErrorCode` values. For instance, when there are no agents available to take a call, the error is `SOSNoAgentsAvailableError`.
2. You can track all SOS errors with the `sos(didError:)` method. Compare the error code to `SOSErrorCode` to determine what kind of error occurred.

See the sample code below for a basic implementation of `sos(didStopWith:error:)` and `sos(didError:)`.

Example: **Basic `SOSDelegate` Example**

This sample code does the following:

- Implements the `SOSDelegate` protocol.
- Implements the `sos(stateDidChange:current:previous:)` method.
- Implements the `sosDidConnect` method.
- Implements the `sos(didStopWith:error:)` method and includes some error handling logic.
- Implements the `sos(didError:)` method.

In Swift:

```
import UIKit
import ServiceCore
import ServiceSOS

class MySOSDelegateImplementation: NSObject, SOSDelegate {
```

```
  // TO DO: Register this delegate using
SCServiceCloud.sharedInstance().sos.addDelegate(self)

  // Delegate method for state change.
  func sos(_ sos: SOSSessionManager!, stateDidChange current: SOSSessionState,
           previous: SOSSessionState) {

    NSLog("SOS state changed...")

    if (current == .connecting) {
      NSLog("SOS now connecting...")
    }
  }

  // Delegate method for connect state.
  func sosDidConnect(_ sos: SOSSessionManager!) {
    NSLog("SOS session connected...")
  }

  // Delegate method for session stop event.
  // You can also check for fatal errors with this delegate method.
  func sos(_ sos: SOSSessionManager!, didStopWith reason: SOSStopReason,
           error: Error!) {

    var title = ""
    var description = ""

    // If there's an error...
    if (error != nil) {

      switch (error as NSError).code {

        // No agents available
        case SOSErrorCode.SOSNoAgentsAvailableError.rawValue:
          title = "Session Failed"
         description = "It looks like there are no agents available. Try again later."


        // Insufficient network error
        case SOSErrorCode.SOSInsufficientNetworkError.rawValue:
          title = "Session Failed"
          description = "Insufficient network. Check network quality and try again."

        // TO DO: Use SOSErrorCode to check for ALL other error conditions
        //        in order to give a more clear explanation of the error.
        default:
          title = "Session Error"
          description = "Unknown session error."
        }

    // Else if session stopped without an error condition...
    } else {
```

```
      switch reason {

        // Handle the agent disconnect scenario
        case .agentDisconnected:
          title = "Session Ended"
          description = "The agent has ended the session."

        // TO DO: Use SOSStopReason to check for
        //        other reasons for session ending...
        default:
          break
      }
    }

    // Do we have an error to report?
    if (title != "") {
      // TO DO: Display an alert using title & description
      NSLog("\nSOS ALERT Title: %@\nDescription: %@",title, description)
    }
  }

  // Delegate method for error conditions.
  func sos(_ sos: SOSSessionManager!, didError error: Error!) {
    NSLog("SOS error (%d): '%@'", (error as NSError).code, error.localizedDescription)
  }
}
```

In Objective-C:

```
#import <UIKit/UIKit.h>
@import ServiceCore;
@import ServiceSOS;

@interface MySOSDelegateImplementation : NSObject <SOSDelegate>

@end

@implementation MyDelegateImplementation

// TO DO: Register this delegate using [[SCServiceCloud sharedInstance].sos
addDelegate:self]

// Delegate method for state change.
- (void)sos:(SOSSessionManager *)sos stateDidChange:(SOSSessionState)current
                                          previous:(SOSSessionState)previous {
  NSLog(@"SOS state changed...");
  if (current == SOSSessionStateConnecting) {
    NSLog(@"SOS now connecting...");
  }
}

// Delegate method for connect state.
- (void)sosDidConnect:(SOSSessionManager *)sos {
  NSLog(@"SOS session connected...");
}
```

```
// Delegate method for session stop event.
// You can also check for fatal errors with this delegate method.
- (void)sos:(SOSSessionManager *)sos didStopWithReason:(SOSStopReason)reason
                                                 error:(NSError *)error {

  NSString *title = nil;
  NSString *description = nil;

  // If there's an error...
  if (error != nil) {

    switch (error.code) {

      // No agents available
      case SOSNoAgentsAvailableError: {
        title = @"Session Failed";
       description = @"It looks like there are no agents available. Try again later.";

        break;
      }

      // Network test failure
      case SOSNetworkTestError: {
        title = @"Session Failed";
        description = @"Insufficient network. Check network quality and try again.";
        break;
      }

      // TO DO: Use SOSErrorCode to check for ALL other error conditions
      //        in order to give a more clear explanation of the error.
      default: {
        title = @"Session Error";
        description = @"Unknown session error.";
        break;
      }
    }

  // Else if session stopped without an error condition...
  } else {

    switch (reason) {

        // Handle the agent disconnect scenario
        case SOSStopReasonAgentDisconnected: {
            title = @"Session Ended";
            description = @"The agent has ended the session.";
            break;
        }

        // TO DO: Use SOSStopReason to check for
        //        other reasons for session ending...
        default: {
            break;
```

```
        }
    }
  }

  // Do we have an error to report?
  if (title != nil) {
    // TO DO: Display an alert using title & description
    NSLog(@"\nSOS ALERT Title: %@\nDescription: %@",title, description);
  }
}

- (void)sos:(SOSSessionManager *)sos didError:(NSError *)error {
  NSLog(@"SOS error (%d): '%@'", error.code, error.localizedDescription);
}

@end
```

Quality-of-Service Events

Check your audio and video quality-of-service (QoS) to detect packet loss and other streaming issues between the OpenTok media router and your org.

Note: This SDK allows you to track streaming issues on the other side of the conversation (from the agent to the media router). To track QoS issues on this side (from the app to the media router), see SOS Quality-of-Service Console Events.

1. Implement `SOSNetworkReporterDelegate`.

   In Swift:

```
func audioNetworkStatsDidUpdate(withSessionId sessionId: String,
                                bytesReceived: NSNumber,
                                packetsReceived: NSNumber,
                                packetsLost: NSNumber,
                                timeStamp: NSNumber) {

  // Handle audio network stats updates
}

func videoNetworkStatsDidUpdate(withSessionId sessionId: String,
                                bytesReceived: NSNumber,
                                packetsReceived: NSNumber,
                                packetsLost: NSNumber,
                                videoDimensions: CGSize,
                                timeStamp: NSNumber) {

  // Handle video network stats updates
}
```

   In Objective-C:

```
- (void)audioNetworkStatsDidUpdateWithSessionId:(NSString *)sessionId
                                  bytesReceived:(NSNumber *)bytesReceived
                                packetsReceived:(NSNumber *)packetsReceived
                                    packetsLost:(NSNumber *)packetsLost
```

```
                                        timeStamp:(NSNumber *)timeStamp {

  // Handle audio network stats updates
}

- (void)videoNetworkStatsDidUpdateWithSessionId:(NSString *)sessionId
                                  bytesReceived:(NSNumber *)bytesReceived
                                packetsReceived:(NSNumber *)packetsReceived
                                    packetsLost:(NSNumber *)packetsLost
                                videoDimensions:(CGSize *)videoDimensions
                                      timeStamp:(NSNumber *)timeStamp {

  // Handle video network stats updates
}
```

The `audioNetworkStatsDidUpdate` method specifies how many bytes were received, how many packets were received, and the number of packets lost for a 30-second span of audio.

The `videoNetworkStatsDidUpdate` method specifies how many bytes were received, how many packets were received, the number of packets lost, and the video dimensions for a 30-second span of video.

2. Subscribe to network events from your SOS instance.

   In Swift:

   ```
   ServiceCloud.shared().sos.networkReporter.add(myNetworkDelegate)
   ```

   In Objective-C:

   ```
   [SCServiceCloud.sharedInstance.sos.networkReporter addDelegate:myNetworkDelegate];
   ```

3. When done, unsubscribe to network events from your SOS instance.

   In Swift:

   ```
   ServiceCloud.shared().sos.networkReporter.remove(myNetworkDelegate)
   ```

   In Objective-C:

   ```
   [SCServiceCloud.sharedInstance.sos.networkReporter removeDelegate:myNetworkDelegate];
   ```

Check SOS Agent Availability

Before starting a session, you can check the availability of your SOS agents and then provide your users with more accurate expectations.

To use agent availability, implement the `SOSAgentAvailabilityDelegate` methods and start polling your org.

Note: When subscribing to the agent availability delegate, it can take several seconds before any of your delegate methods are called. We don't suggest that you block a user's ability to start a session during this period.

1. Implement the `SOSAgentAvailabilityDelegate` methods in your `UIViewController`.

   In Swift:

   ```
   func agentAvailability(_ agentAvailability: Any!,
                          didChange availabilityStatus: SOSAgentAvailabilityStatusType) {
     // TO DO: Handle event...
   }
   ```

```
func agentAvailability(_ agentAvailability: Any!,
                       didError error: Error!) {
  // TO DO: Handle error...
}
```

In Objective-C:

```
- (void)agentAvailability:(__weak id)agentAvailability
        didChange:(SOSAgentAvailabilityStatusType)availabilityStatus {

  // TO DO: Handle event...
}

- (void)agentAvailability:(__weak id)agentAvailability
        didError:(NSError *)error {

  // TO DO: Handle error...
}
```

Refer to the `SOSAgentAvailabilityStatusType` enumerated type for list of potential status messages.

2. Add your `UIViewController` as a delegate to the`SOSAgentAvailability` object and call `startPolling(withOrganizationId:deploymentId:liveAgentPod:)`.

   In Swift:

   ```
   let agentAvailability = ServiceCloud.shared().sos.agentAvailability!
   agentAvailability.add(self)
   agentAvailability.startPolling(withOrganizationId: "YOUR-ORG-ID",
                                  deploymentId: "YOUR-DEPLOY-ID",
                                  liveAgentPod: "YOUR-LA-POD")
   ```

   In Objective-C:

   ```
   SOSAgentAvailability *agentAvailability =
     [SCServiceCloud sharedInstance].sos.agentAvailability;
   [agentAvailability addDelegate:self];
   [agentAvailability startPollingWithOrganizationId:@"YOUR-ORG-ID"
                                       deploymentId:@"YOUR-DEPLOY-ID"
                                       liveAgentPod:@"YOUR-LA-POD"];
   ```

   This method takes the same values you specified when starting an SOS session. For more info, see Quick Setup: SOS in the Service SDK.

Example: This example accesses the Agent Availability feature and handles the appropriate events. In the code, `_sosBtn` is a `UIButton` with text that turns green when an agent is available, red when no agents are available, and gray when the status is unknown.

In Swift:

```
override func viewDidLoad() {
  super.viewDidLoad()

  let agentAvailability = ServiceCloud.shared().sos.agentAvailability
  agentAvailability?.add(self)
  agentAvailability?.startPolling(withOrganizationId: "YOUR-ORG-ID",
```

```
                                    deploymentId: "YOUR-DEPLOY-ID",
                                    liveAgentPod: "YOUR-LA-POD")
}

// Delegate methods
func agentAvailability(_ agentAvailability: Any!,
  didChange availabilityStatus: SOSAgentAvailabilityStatusType) {

  var color: UIColor?

  switch availabilityStatus {
    case .available:
      color = UIColor.green
      sosBtn.isEnabled = true
    case .unavailable:
      color = UIColor.red
      sosBtn.isEnabled = false
    case .unknown:
      color = UIColor.gray
      sosBtn.isEnabled = false
  }

  sosBtn.setTitleColor(color!, for: .normal)
}

func agentAvailability(_ agentAvailability: Any!, didError error: Error!) {

  // TO DO: Handle error
}
```

In Objective-C:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    SOSAgentAvailability *agentAvailability =
        [SCServiceCloud sharedInstance].sos.agentAvailability;
    [agentAvailability addDelegate:self];
    [agentAvailability startPollingWithOrganizationId:@"YOUR-ORG-ID"
                                         deploymentId:@"YOUR-DEPLOY-ID"
                                         liveAgentPod:@"YOUR-LA-POD"];
}

// Delegate methods

- (void)agentAvailability:(__weak id)agentAvailability
        didChange:(SOSAgentAvailabilityStatusType)availabilityStatus {

    UIColor *color;

    switch (availabilityStatus) {
        case SOSAgentAvailabilityStatusAvailable: {
            color = [UIColor greenColor];
            [_sosBtn setEnabled:YES];
            break;
        }
```

```
        case SOSAgentAvailabilityStatusUnavailable: {
            color = [UIColor redColor];
            [_sosBtn setEnabled:NO];
            break;
        }
        case SOSAgentAvailabilityStatusUnknown:
        default: {
            color = [UIColor grayColor];
            [_sosBtn setEnabled:NO];
            break;
        }
    }
    [_sosBtn setTitleColor:color forState:UIControlStateNormal];
}

- (void)agentAvailability:(__weak id)agentAvailability
        didError:(NSError *)error {

    // TO DO: Handle error
}
```

Enable and Disable Screen Sharing

There are some scenarios where you may want to programmatically turn off screen sharing in mid-session. You can enable and disable screen sharing using the `screenSharing` property.

You can control screen sharing using the `screenSharing` object on the `SOSSessionManager` shared instance. The `screenSharing.enabled` property enables or disables the screen sharing functionality.

The following code disables screen sharing.

In Swift:

```
ServiceCloud.shared().sos.screenSharing.enabled = false
```

In Objective-C:

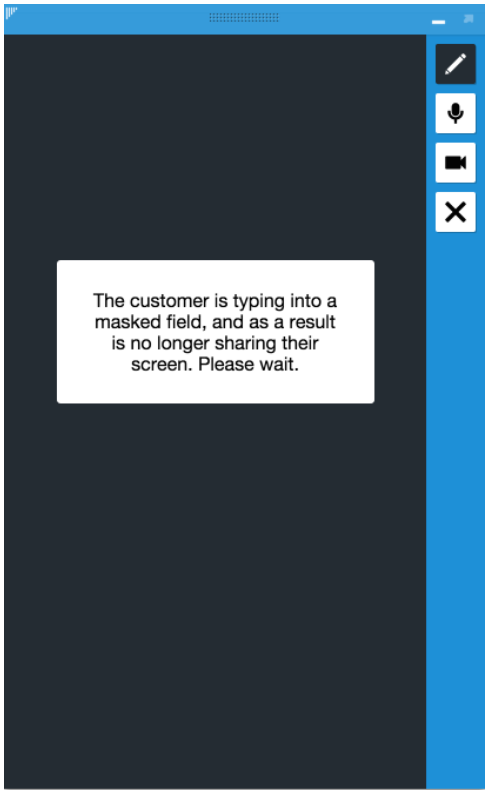
```
[SCServiceCloud sharedInstance].sos.screenSharing.enabled = NO;
```

Field Masking

If an application contains sensitive information that an agent shouldn't see during an SOS session, you can hide this information from the agent.

When a customer enters information into a masked field, screen sharing is disabled and the agent is notified that sharing is unavailable until the user has finished. Field masking is an integral feature to help bring your application to PII, PCI, and HIPAA compliance.

Agent's view when a field is masked:



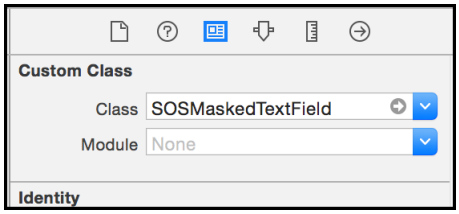
To use field masking, replace the `UITextField` containing sensitive information with a `SOSMaskedTextField`.

When there isn't an SOS session running, a masked text field appears the same as a standard `UITextField`. However, when an SOS session is running, an `SOSMaskedTextField` appears different from a standard `UITextField` field. When the user interacts with the `SOSMaskedTextField`, screen sharing stops while the contents of that field is visible to the user. When the user finishes editing the masked field, screen-sharing resumes.

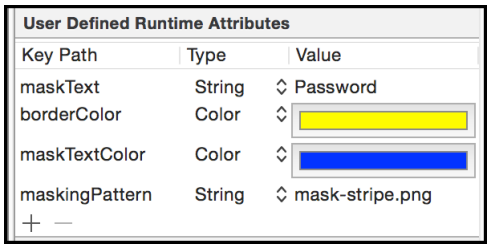
Create Masked Field Using Storyboard

To create a masked field using the storyboard, specify the `SOSMaskedTextField` custom class and set the user-defined runtime attributes.

1. Create a standard `UITextField` with the storyboard.
2. From the **Identity Inspector**, specify `SOSMaskedTextField` for the **Custom Class**.



3. From the **Identity Inspector**, specify the text field style settings in the **User Defined Runtime Attributes** section.



The follow key path attributes are available:

maskPattern (String)
Name of the image file used to fill the background of the masked field when an SOS session is active

borderColor (Color)
`UIColor` of the border around the masked field

maskText (String)
Text to appear in the masked field

maskPattern (Color)
`UIColor` of the text in the masked field

Create Masked Field Programmatically

To create a masked field manually, instantiate and style a `SOSMaskedTextField` instance.

1. Define an `SOSMaskedTextField` instead of a `UITextField`.

   In Swift:

   ```
   var maskCodeExample: SOSMaskedTextField!
   ```

   In Objective-C:

   ```
   @property (strong, nonatomic) IBOutlet SOSMaskedTextField *maskCodeExample;
   ```

2. Instantiate and style the `SOSMaskedTextField`.

   In Swift:

   ```
   // Set the size of the masked field and the look when masking is active
   maskCodeExample = SOSMaskedTextField(
     frame: CGRect.init(x: 20, y: 300, width: 200, height: 300),
   ```

```
  maskPattern: "mask-stripBY.png",
  borderColor: UIColor.yellow,
  text: "Password",
  textColor: UIColor.blue)

// Make the field look like a UITextField created by the interface builder
maskCodeExample.borderStyle = .roundedRect
maskCodeExample.autocorrectionType = .no
maskCodeExample.keyboardType = .default
maskCodeExample.returnKeyType = .done
maskCodeExample.clearButtonMode = .whileEditing
maskCodeExample.contentVerticalAlignment = .center

maskCodeExample.delegate = self

self.view.addSubview(maskCodeExample)
```

In Objective-C:

```
// Set the size of the masked field and the look when masking is active
maskCodeExample = [[SOSMaskedTextField alloc]
                       initWithFrame:CGRectMake(20, 300, 200, 30)
                         maskPattern:@"mask-stripeBY.png"
                         borderColor:[UIColor yellowColor]
                                text:@"Password"
                           textColor:[UIColor blueColor]];

// Make the field look like a UITextField created by the interface builder
maskCodeExample.borderStyle = UITextBorderStyleRoundedRect;
maskCodeExample.font = [UIFont systemFontOfSize:15];
maskCodeExample.autocorrectionType = UITextAutocorrectionTypeNo;
maskCodeExample.keyboardType = UIKeyboardTypeDefault;
maskCodeExample.returnKeyType = UIReturnKeyDone;
maskCodeExample.clearButtonMode = UITextFieldViewModeWhileEditing;
maskCodeExample.contentVerticalAlignment =
                UIControlContentVerticalAlignmentCenter;

[maskCodeExample setDelegate:self];

[self.view addSubview:maskCodeExample];
```

Custom Data

Use custom data to identify customers, send error messages, issue descriptions, or identify the page the SOS session was initiated from.

When an agent receives an SOS call, it can be helpful to have information about the caller before starting the session. Use the custom data feature to identify customers, send error messages, identify the currently viewed page, or send other information. Custom data populates custom fields on the SOS Session object that is created within your Salesforce org for each SOS session initiated by a user.

Before using custom data, create the corresponding fields within the SOS Session object of your Salesforce org. To learn more, see Create Custom Fields.

To use this feature, set the `customFieldData` property on the `SOSOptions` object that you use to start an SOS session. The keys in this `NSMutableDictionary` should reference the API Name for fields defined in your SOS Session object and the values should reflect the desired values for those fields.

Note: If the field associated with the custom data that you specify in `customFieldData` is not set up correctly in your Salesforce org, the SOS session will fail with an error.

Example: This example shows how to pass email information from your app to Service Cloud.

Before trying this example, be sure to define an "Email" custom field on the SOS Session object in your Salesforce org:

Note: By default, your org contains no custom field data. To learn about custom fields, see Create Custom Fields.

Once you have created a custom field, you can build a dictionary object and add it to the `SOSOptions` object that you create when starting a session.

In Swift:

```
let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                         orgId: "YOUR-ORG-ID",
                         deploymentId: "YOUR-DEPLOYMENT-ID")

// Here we are passing the customer's email address as a String. Note the use
// of the field's API Name as the key in the map. We are only populating a single
// field here, but we may put an arbitrary number of entries into the map to
// populate multiple different fields.
options!.customFieldData = ["Email__c": "laurenboyle@example.com"]
```

In Objective-C:

```
SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                    orgId:@"YOUR-ORG-ID"
                                             deploymentId:@"YOUR-DEPLOYMENT-ID"];

// Here we are passing the customer's email address as a String. Note the use
// of the field's API Name as the key in the map. We are only populating a single
// field here, but we may put an arbitrary number of entries into the map to
// populate multiple different fields.
NSMutableDictionary *myCustomData =
    [NSMutableDictionary dictionaryWithObjectsAndKeys:
    @"laurenboyle@example.com", @"Email__c", nil];

[options setCustomFieldData:myCustomData];
```

When the user creates an SOS session, the `Email` field is prepopulated with the value specified in your custom data field.

Replace the SOS UI

If you'd like to customize the SOS UI, you can create your own UI by subclassing the `UIViewController` class associated with that phase of the SOS session.

To replace the SOS UI, you'll need to register your view controllers for whichever phases you want to replace. The following phases are supported:

Onboarding (`SOSUIPhaseOnboarding`)
The onboarding process before a session starts.

Connecting (`SOSUIPhaseConnecting`)
The connecting process before a session starts.

Screen Sharing (`SOSUIPhaseScreenSharing`)
An active screen sharing session.

To register a view controller, use the `SOSOptions` object and call the `setViewControllerClass` method before attempting to start a session. This method takes the class type for your view controller, and an `SOSUIPhase` enumerated type. When creating your view controller, make sure it subclasses the view controller for that phase of the session. Each view controller must also implement a protocol associated with that session phase.

Table 11: SOS Phases

Phase Name	View Controller to Subclass	Protocol to Implement
`onboarding`	`SOSOnboardingBaseViewController`	`SOSOnboardingViewController`
`connecting`	`SOSConnectingBaseViewController`	`SOSConnectingViewController`
`screenSharing`	`SOSScreenSharingBaseViewController` (which subclasses `SOSSessionBaseViewController`)	`SOSSessionViewController`, `SOSUIAgentStreamReceivable`, `SOSUILineDrawingReceivable`

For example, the following code overrides the onboarding UI.

In Swift:

```
let options = SOSOptions(liveAgentPod: "YOUR-POD-NAME",
                         orgId: "YOUR-ORG-ID",
                         deploymentId: "YOUR-DEPLOYMENT-ID")!

// Register a custom onboarding view controller
options.setViewControllerClass(SOSOnboardingViewController.self, for: .onboarding)

// Perform other SOS configuration here
// ...

ServiceCloud.shared().sos.startSession(with: options)
```

In Objective-C:

```
SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                    orgId:@"YOUR-ORG-ID"
                                             deploymentId:@"YOUR-DEPLOYMENT-ID"];

// Register a custom onboarding view controller
[options setViewControllerClass:[SOSMyOnboardingViewController class]
for:SOSUIPhaseOnboarding];

// Perform other SOS configuration here
// ...
```

```
[[SCServiceCloud sharedInstance].sos startSessionWithOptions:options];
```

To learn more about configuring an SOS session, see Configure an SOS Session.

You can use the following code samples to get started.

SOS Onboarding Sample Code

Below you'll find some boilerplate sample code for the onboarding experience. This class must subclass `SOSOnboardingBaseViewController`.

In Swift:

```
override func willHandleConnectionPrompt() -> Bool {
  return true
}

override func connectionPromptRequested() {
  // TO DO: Show the onboarding view
}

// Call `handleStartSession` to start a session
// Call `handleCancel` to cancel a session

// See `SOSOnboardingBaseViewController`, `SOSOnboardingViewController`
// for additional functionality
```

In Objective-C:

```
- (BOOL)willHandleConnectionPrompt {
  return YES;
}

- (void)connectionPromptRequested {
  // TO DO: Show the onboarding view
}
```

SOS Connecting Sample Code

Below you'll find some boilerplate sample code for the connecting experience. This class must subclass `SOSConnectingBaseViewController`.

In Swift:

```
override func initializingNotification() {
  // TO DO: Show initializing view
}

override func waitingForAgentNotification() {
  // TO DO: Show waiting for agent view
}

override func agentJoinedNotification() {
  // TO DO: Show agent joined notification
```

```
}

// Call `handleEndSession` to cancel a session

// See `SOSConnectingBaseViewController`, `SOSConnectingViewController`
// for additional functionality
```

In Objective-C:

```
- (void)initializingNotification {
  // TO DO: Show initializing view
}

- (void)waitingForAgentNotification {
  // TO DO: Show waiting for agent view
}

- (void)agentJoinedNotification:(NSString *)name {
  // TO DO: Show agent joined notification
}
```

Screen Sharing Sample Code

Below you'll find some boilerplate sample code for the screen sharing experience. This class must subclass `SOSScreenSharingBaseViewController`.

In Swift:

```
override func willHandleAgentStream() -> Bool {
  // This determines whether you wish to display an agent stream in your view.
  // If you return NO you will not receive a view containing the agent video feed.
  return true
}

override func willHandleAudioLevel() -> Bool {
  // When this returns YES, you will receive updates about the audio level you can use
  // to implement an audio meter.
  return false
}

override func willHandleLineDrawing() -> Bool {
  // This determines whether you want to handle line drawing during the session.
  return true
}

override func willHandleRemoteMovement() -> Bool {
  // When this returns YES, you will receive screen space coordinates which represent
  // the center of where the agent has moved the view. You can use this to update
  // the position of your containing view.
  return false
}

override func didReceiveLineDraw(_ drawView: UIView) {
  // TO DO: Handle line draw
}
```

```
override func didReceiveAgentStreamView(_ agentStreamView: UIView) {
  // TO DO: Handle stream view
}

// See `SOSSessionBaseViewController` for how to handle pause, mute, end session events

// See `SOSScreenSharingBaseViewController`, `SOSUIAgentStreamReceivable`,
//     `SOSUILineDrawingReceivable` for additional functionality
```

In Objective-C:

```
- (BOOL)willHandleAgentStream {
  // This determines whether you wish to display an agent stream in your view.
  // If you return NO you will not receive a view containing the agent video feed.
  return YES;
}

- (BOOL)willHandleAudioLevel {
  // When this returns YES, you will receive updates about the audio level you can use
  // to implement an audio meter.
  return NO;
}

- (BOOL)willHandleLineDrawing {
  // This determines whether you want to handle line drawing during the session.
  return YES;
}

- (BOOL)willHandleRemoteMovement {
  // When this returns YES, you will receive screen space coordinates which represent
  // the center of where the agent has moved the view. You can use this to update
  // the position of your containing view.
  return NO;
}

- (void)didReceiveLineDrawView:(UIView * _Nonnull __weak)drawView {
  // TO DO: Handle line draw
}

- (void)didReceiveAgentStreamView:(UIView * _Nonnull __weak)agentStreamView {
  // TO DO: Handle stream view
}
```

SDK Customizations with the Service SDK for iOS

Once you've played around with some of the SDK features, use this section to learn how to customize the Service SDK user interface so that it fits the look and feel of your app. This section also contains instructions for localizing strings in all supported languages.

Many UI customizations are handled with the `SCAppearanceConfiguration` object. You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor`, `setFontDescriptor`, and `setImage`. To use this object, create an `SCAppearanceConfiguration` instance, specify

values for each token you want to change, and store the instance in the `appearanceConfiguration` property of the `ServiceCloud sharedInstance`.

There are other ways to customize the interface. When using Service Cloud features, various action buttons are available to the user. You can control the visibility of these buttons and even create new action buttons. You can also customize the strings used in the UI for any of the supported languages. String customization is performed using a standard localization mechanism provided to Apple developers.

Customize Colors with the Service SDK

Customize the colors by defining the branding token colors used throughout the interface.

To customize colors, create an `SCAppearanceConfiguration` instance, specify values for each token you want to change, and store the instance in the `appearanceConfiguration` property of the `ServiceCloud sharedInstance`.

In Swift:

```
// Create appearance configuration instance
let appearance = SCAppearanceConfiguration()

// Customize color tokens
appearance.setColor(COLOR_VALUE, forName: TOKEN_NAME)

// Add other customizations here...

// Save configuration instance
ServiceCloud.shared().appearanceConfiguration = appearance
```

In Objective-C:

```
// Create appearance configuration instance
SCAppearanceConfiguration *appearance = [SCAppearanceConfiguration new];

// Customize color tokens
[appearance setColor:COLOR_VALUE forName:TOKEN_NAME];

// Add other customizations here...

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = appearance;
```

The following branding tokens are available for customization:

Token Name / Swift Value / Objective-C Value	Default Value	Description / Sample Uses

Navigation Bar Background
`navbarBackground`
`SCSAppearanceColorTokenNavbarBackground`
	#FAFAFA	Background color for the navigation bar.

Navigation Bar Inverted
`navbarInverted`
`SCSAppearanceColorTokenNavbarInverted`
	#010101	Navigation bar text and icon color.

Brand Primary
`brandPrimary`
`SCSAppearanceColorTokenBrandPrimary`
	#007F7F	Knowledge: First data category, the Show More button, the footer stripe, the selected article.
		SOS: Used for various icons.

Brand Secondary
`brandSecondary`
`SCSAppearanceColorTokenBrandSecondary`
	#2872CC	Used throughout the UI for button colors.
		Chat: Agent text bubbles.
		SOS: Background color for action items.

Primary Brand Inverted
`brandPrimaryInverted`
`SCSAppearanceColorTokenBrandPrimaryInverted`
	#FBFBFB	Knowledge: Text on data category headers and the chevron on the Knowledge home page.

Secondary Brand Inverted
`brandSecondaryInverted`
`SCSAppearanceColorTokenBrandSecondaryInverted`
	#FCFCFC	Text on areas where a brand color is used for the background.

Token Name / Swift Value / Objective-C Value	Default Value	Description / Sample Uses

Contrast Primary
`contrastPrimary`
`SCSAppearanceColorTokenContrastPrimary`
	#000000	Primary body text color.
		SOS: Background color for buttons on the UI.

Contrast Secondary
`contrastSecondary`
`SCSAppearanceColorTokenContrastSecondary`
	#6D6D6D	Knowledge: Subcategory headers.

Contrast Tertiary
`contrastTertiary`
`SCSAppearanceColorTokenContrastTertiary`
	#BABABA	Knowledge: Search background color.
		SOS: Dots in UI.

Contrast Quaternary
`contrastQuaternary`
`SCSAppearanceColorTokenContrastQuaternary`
	#F1F1F1	Knowledge: Icon image background color.
		Case Management: Link color.
		Chat: Background color.

Contrast Inverted
`contrastInverted`
`SCSAppearanceColorTokenContrastInverted`
	#FFFFFF	Page background, navigation bar, table cell background.
		SOS: Color of the icons.

Feedback Primary
`feedbackPrimary`
`SCSAppearanceColorTokenFeedbackPrimary`
	#E74C3C	Text color for error messages.
		SOS: Mute indicator. Disconnect icon.

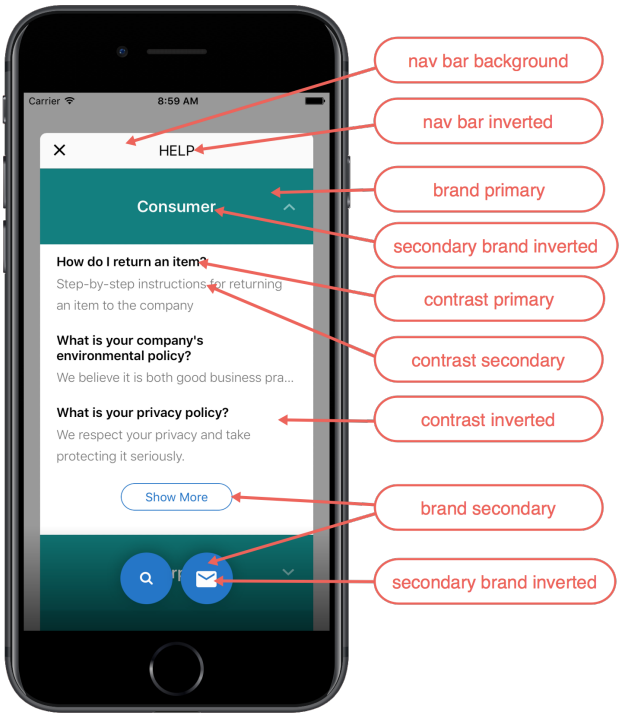
Feedback Secondary
`feedbackSecondary`
`SCSAppearanceColorTokenFeedbackSecondary`
	#2ECC71	SOS: Connection quality indicators. Background color for the Resume button when the two-way camera is active.

Feedback Tertiary
`feedbackTertiary`
`SCSAppearanceColorTokenFeedbackTertiary`
	#F5A623	SOS: Connection quality indicators.

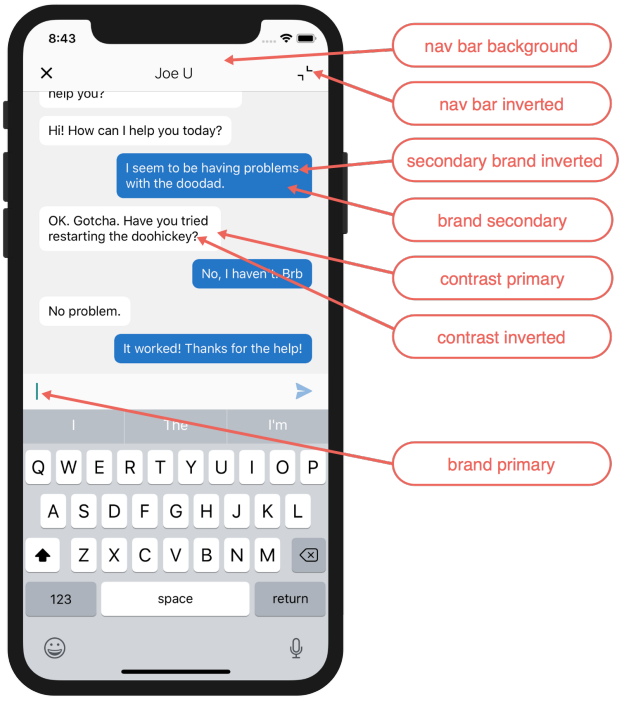
Overlay
`overlay`
`SCSAppearanceColorTokenOverlay`
	Contrast Primary
	(at 40% alpha)
		Knowledge: Background for the Knowledge home screen.

The screenshots below illustrate how the branding tokens affect the UI.

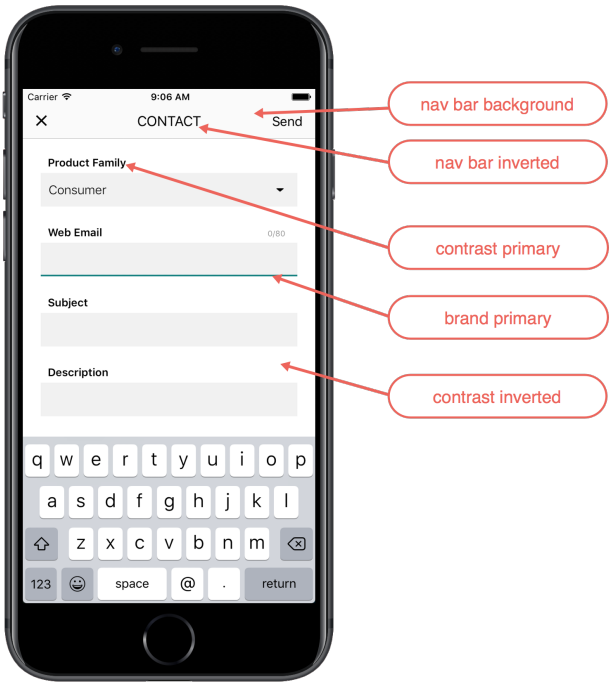
Knowledge UI Branding:



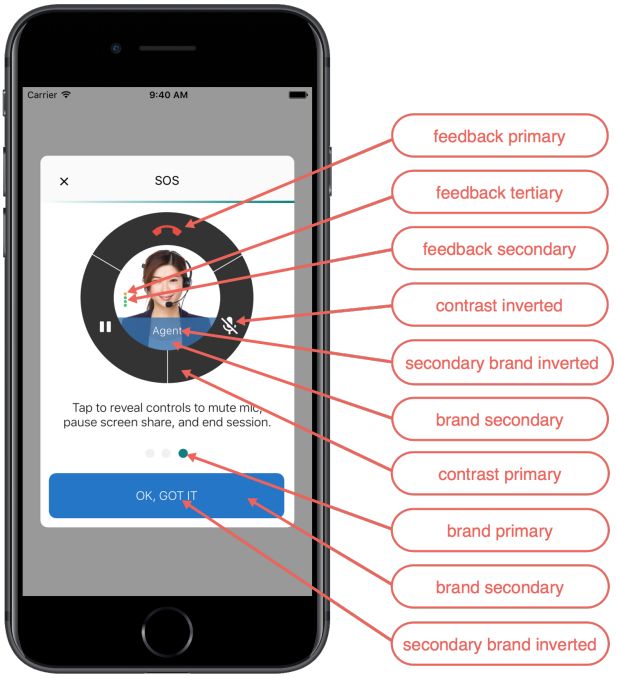
Chat UI Branding:



Case Management UI Branding:



SOS UI Branding:



Example: The following code sample changes three of the branding tokens.

In Swift:

```
// Create appearance configuration instance
let appearance = SCAppearanceConfiguration()
```

```
// Customize color tokens
appearance.setColor(
       UIColor(red: 80/255, green: 227/255, blue: 194/255, alpha: 1.0),
       forName: .brandPrimary)
appearance.setColor(
       UIColor(red: 74/255, green: 144/255, blue: 226/255, alpha: 1.0),
       forName: .brandSecondary)
appearance.setColor(
       UIColor(red: 252/255, green: 252/255, blue: 252/255, alpha: 1.0),
       forName: .brandSecondaryInverted)

// Save configuration instance
ServiceCloud.shared().appearanceConfiguration = appearance
```

In Objective-C:

```
// Create appearance configuration instance
SCAppearanceConfiguration *appearance = [SCAppearanceConfiguration new];

// Customize color tokens
[appearance setColor:[UIColor colorWithRed: 80/255
                                    green: 227/255
                                     blue: 194/255
                                    alpha: 1.0]
             forName:SCSAppearanceColorTokenBrandPrimary];
[appearance setColor:[UIColor colorWithRed: 74/255
                                    green: 144/255
                                     blue: 226/255
                                    alpha: 1.0]
             forName:SCSAppearanceColorTokenBrandSecondary];
[appearance setColor:[UIColor colorWithRed: 252/255
                                    green: 252/255
                                     blue: 252/255
                                    alpha: 1.0]
             forName:SCSAppearanceColorTokenBrandSecondaryInverted];

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = appearance;
```

Customize and Localize Strings with the Service SDK

You can change the text used throughout the user interface. To customize text, create string resource values in a `Localizable.strings` file in the Localization bundle for the languages you want to update. Create string tokens that match the tokens you intend to override.

Service SDK text is translated into more than 25 different languages. In order for your string customizations to take effect in a given language, provide a translation for that language. For any language you do not override manually in your app, the SDK uses its default values for that language.

Refer to Internationalization at developer.apple.com for more info about localization.

The following list of string tokens are available for customization:

- ServiceChat (Chat) String Resources
- ServiceKnowledge (Knowledge) String Resources

- ServiceCases (Case Management) String Resources
- ServiceSOS (SOS) String Resources
- ServiceCore (Common) String Resources

The following languages are currently supported:

Table 12: Supported Languages

Language Code	Language
ar	Arabic
cs	Czech
da	Danish
de	German
el	Greek
en	English
es	Spanish
fi	Finnish
fr	French
hu	Hungarian
id	Indonesian
it	Italian
iw	Hebrew
ja	Japanese
ko	Korean
nb	Norwegian Bokmål
nl	Dutch
pl	Polish
pt	Portuguese
ro	Romanian
ru	Russian
sv	Swedish
th	Thai
tr	Turkish
uk	Ukranian
vi	Vietnamese

Language Code	Language
zh_TW	Chinese (Taiwan)
zh-Hans	Chinese (Simplified)
zh-Hant	Chinese (Traditional)
zh	Chinese

Customize Fonts with the Service SDK

There are three customizable font settings used throughout the UI: `SCFontWeightLight`, `SCFontWeightRegular`, `SCFontWeightBold`.

You can customize three font settings used throughout the Service SDK interface:

Font Setting	Default Value	Samples Uses in the SDK
`SCFontWeightLight`	Helvetica Neue - Light	Knowledge article cell summary, Case Management field text, Case Management submit success view, content of error messages
`SCFontWeightRegular`	Helvetica Neue	Navigation bar, Chat text, Knowledge data category cell in detail view, Knowledge "show more" article footer, Knowledge "show more" button cell
`SCFontWeightBold`	Helvetica Neue - Semibold	Knowledge category headers, Knowledge article cell title, Case Management field labels, Case Management submit button, title of error messages

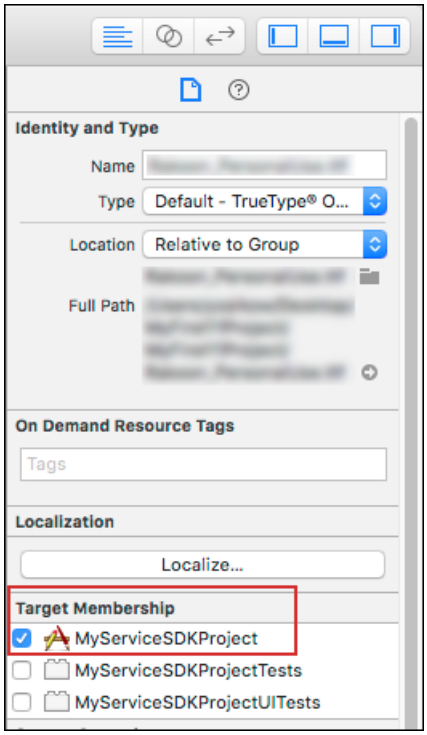
To configure your app to use different fonts:

1. Add new fonts to your Xcode project.

   Any new fonts must be added as a resource to your Xcode project. When adding, be sure to select **Copy items if needed**.

2. Add new fonts to your project target.

   For each new font, add it to your project target under **Target Membership**.



3. Add the font to your app's `Info.plist`.

   You'll need to add all new fonts into a string array. Each string element of the array must be the name of each font resource file.

   If you're viewing your `Info.plist` as a **Property List**, add an Array named **Fonts provided by application**.

   If you're viewing your `Info.plist` as **Source Code**, add an array named `UIAppFonts`. For example:

```
<key>UIAppFonts</key>
  <array>
    <string>MyCustomFont1.ttf</string>
    <string>MyCustomFont2.ttf</string>
    <string>MyCustomFont3.ttf</string>
</array>
```

4. Customize any of the Service SDK font values using `SCAppearanceConfiguration`.

   To customize the fonts, create an `SCAppearanceConfiguration` instance, set the font descriptor for each font setting you want to change, and store the `SCAppearanceConfiguration` instance in the `appearanceConfiguration` property of the `ServiceCloud` shared instance.

   Swift Example:

```
// Create appearance configuration instance
let config = SCAppearanceConfiguration()

// Customize font
let descriptor = UIFontDescriptor(fontAttributes:
    [UIFontDescriptor.AttributeName.family : "Proxima Nova"])
config.setFontDescriptor(descriptor,
                         fontFileName: "ProximaNova-Light.otf",
                         forWeight: SCFontWeightLight)
```

```
// Add other customizations here...

// Save configuration instance
ServiceCloud.shared().appearanceConfiguration = config
```

Objective-C Example:

```
// Create appearance configuration instance
SCAppearanceConfiguration *config =
    [SCAppearanceConfiguration new];

// Customize font
UIFontDescriptor *descriptor =
    [UIFontDescriptor fontDescriptorWithFontAttributes:@{
     UIFontDescriptorFamilyAttribute: @"Proxima Nova",
       UIFontDescriptorFaceAttribute: @"Light" }];
[config setFontDescriptor:descriptor
                     fontFileName:@"ProximaNova-Light.otf"
                        forWeight:SCFontWeightLight];

// Add other customizations here...

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = config;
```

Be sure to use the exact font descriptor attribute name and font file name for your custom font.

Customize Images with the Service SDK

You can specify custom images used throughout the UI, along with the images used by Knowledge categories and within Knowledge articles.

The method to customize an image is different depending on the type of image you want to customize. This table describes what customization techniques are supported for each image type.

Table 13: Custom Image Types

Image Type	How to Customize
Stock images	Use the `setImage` method on the `SCAppearanceConfiguration` object to **replace a stock image** with your image. Use the enumeration value for the image you intend to replace. See Replacing Stock Images for more guidance.
Data category image (Knowledge) Article image (Knowledge)	Data category images and article images can be customized in either one of two different ways: 1. Put the images in a specific **image folder**. This technique uses the `imageFolderPath` property on the`SCSServiceConfiguration`. For data category images, the image name must match the unique name for that category. For article images, the image name must match

Image Type	How to Customize
	the article number. See Supplying Knowledge Images using an Image Folder for more guidance. 2. Supply images by implementing a **delegate** method. This technique uses the `SCKnowledgeInterfaceDelegate protocol`. See Supplying Knowledge Images with a Delegate for more guidance.

Supported image file formats include: tiff, tif, jpg, jpeg, gif, png, bmp, BMPF, ico, cur.

Replacing Stock Images

For specific images, use the `SCSAppearanceImageToken` enumeration specified by the SDK and add it to the `SCAppearanceConfiguration` object with the `setImage` method.

Table 14: Stock Image Enum Values

Image Description	Enum Value
Common Images	*Images used throughout the SDK*
Close button	`close`
Done button	`done`
Small warning icon used when an error occurs	`error`
Error image used in a view for timeouts, when no agents are available, or for an unknown error	`genericError`
Minimize button (Knowledge and Chat)	`minimizeButton`
No connection	`noConnection`
Send button (Case Publisher and Chat)	`send`
Next field button (Case Publisher and Chat)	`submitButtonNextArrow`
Previous field button (Case Publisher and Chat)	`submitButtonPreviousArrow`
Knowledge Images	*Images used by Knowledge*
Search action button	`actionSearch`
Category header arrow	`categoryHeaderArrow`
Article is empty	`emptyArticle`
Category section has no articles	`emptySection`
No search results	`noSearchResult`
Placeholder image before search completes	`searchPlaceholder`
Icon used when showing the list of subcategories	`subCategoryIcon`

Image Description	Enum Value
Case Management Images	*Images used by Case Management*
Case publisher action button	`actionCasePublisher`
Case publisher success message	`caseSubmitSuccess`
Compose icon on Case List screen for creating a case	`compose`
Empty icon for the Case List screen if no cases are present	`emptyIcon`
Pick case list button	`picklistDropdown`
Chat Images	*Images used by Chat*
Attachment button when the user can attach a file	`attachmentClipIcon`
Avatar used for the agent	`chatAgentAvatar`
Avatar used for Einstein bot	`chatBotAvatar`
Icon used for Einstein bot persistent footer menu	`chatBotFooterMenu`
Icon used in the pre-chat screen	`preChatIcon`
SOS Images	*Images used by SOS*
Agent muted icon	`sosAgentMutedIcon`
Agent placeholder icon (must be 61x55 pixels @1x, 122x110 @2x, 183x165 @3x)	`sosAgentPlaceHolderIcon`
Camera icon	`sosCameraIcon`
Cancel icon	`sosCancel`
Confirm icon	`sosConfirmIcon`
End icon	`sosEndIcon`
Expand icon	`sosExpandIcon`
Flashlight icon	`sosFlashlightIcon`
Info icon	`sosInfoIcon`
Mask stripe	`sosMasking`
Microphone icon	`sosMicrophoneIcon`
User muted icon	`sosMicrophoneMutedIcon`
Pause icon	`sosPauseIcon`
Resume icon	`sosResumeIcon`

In Swift:

```
// Create appearance configuration instance
let config = SCAppearanceConfiguration()

// Specify images
config.setImage(MY_CUSTOM_IMAGE,
                compatibleWithTraitCollection: MY_TRAITS,
                forName: ENUM_VALUE)

// Add other customizations here...

// Save configuration instance
ServiceCloud.shared().appearanceConfiguration = config
```

In Objective-C:

```
// Create appearance configuration instance
SCAppearanceConfiguration *config = [SCAppearanceConfiguration new];

// Specify images
[config setImage:MY_CUSTOM_IMAGE compatibleWithTraitCollection: MY_TRAITS
         forName: ENUM_VALUE];

// Add other customizations here...

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = config;
```

Supplying Knowledge Images using an Image Folder

For knowledge articles images and data category images, specify the location of the image using the `imageFolderPath` property on the`SCSServiceConfiguration` object. We suggest you do this at the time that you configure the object to connect to your org. To learn more about connecting to your org, see Quick Setup: Knowledge in the Service SDK.

For data category images, the image name must match the unique name for that category. For article images, the image name must match the article number.

Here is some sample code to get you started.

In Swift:

```
// Create configuration object with init params
let config = SCSServiceConfiguration(
               community: URL(string: "https://mycommunity.example.com")!,
               dataCategoryGroup: "Regions",
               rootDataCategory: "All")

// Specify image folder
config.imageFolderPath = "my/path/"

// Pass configuration to shared instance
ServiceCloud.shared().serviceConfiguration = config
```

In Objective-C:

```
// Create configuration object with init params
SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
  initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]
  dataCategoryGroup:@"Regions"
   rootDataCategory:@"All"];

// Specify image folder
config.imageFolderPath = @"my/path/";

// Pass configuration to shared instance
[SCServiceCloud sharedInstance].serviceConfiguration = config;
```

Supplying Knowledge Images with a Delegate

For knowledge articles images and data category images, you can instead supply images using a delegate.

1. Provide the `ServiceCloud` instance with an implementation of `SCKnowledgeInterfaceDelegate`.

   In Swift:

   ```
   ServiceCloud.shared().knowledge.delegate =
        mySCKnowledgeInterfaceDelegateImplementation
   ```

   In Objective-C:

   ```
   [SCServiceCloud sharedInstance].knowledge.delegate =
       mySCKnowledgeInterfaceDelegateImplementation;
   ```

2. Implement the two delegate methods.

   In Swift:

   ```
   func knowledgeInterface(_ interface: SCKnowledgeInterface,
          imageForArticle articleId: String,
          compatibleWith traitCollection: UITraitCollection?)
          -> UIImage? {

     // Your code that returns an image given an article id

     return myImage
   }

   func knowledgeInterface(_ interface: SCKnowledgeInterface,
          imageForDataCategory categoryName: String,
          compatibleWith traitCollection: UITraitCollection?)
          -> UIImage? {

     // Your code that returns an image given a data category

     return myImage
   }
   ```

In Objective-C:

```
- (UIImage*)knowledgeInterface:(SCKnowledgeInterface *)interface
                 imageForArticle:(NSString *)articleId
 compatibleWithTraitCollection:(UITraitCollection *)traitCollection {

  // Your code that returns an image given an article id

  return myImage;
}

- (UIImage*)knowledgeInterface:(SCKnowledgeInterface *)interface
           imageForDataCategory:(NSString *)categoryName
 compatibleWithTraitCollection:(UITraitCollection *)traitCollection {

  // Your code that returns an image given a data category

  return myImage;
}
```

Customize Action Buttons with the Service SDK

You can customize the action buttons used throughout the UI. You can override the look and the behavior of existing buttons, and you can create buttons associated with new actions. Use the `actions` property on `ServiceCloud` to get access to the action button API.

The SDK provides action buttons that automatically appear when they apply to the context of what you're viewing. For instance, a Search button and a Case Publisher button appear when browsing Knowledge articles. You can further customize the behavior of these built-in action buttons, and you can create your own action buttons.

Several classes are associated with the action button API.

`SCSActionManager`
: This class is your entry point into the action button API. Access this object using the `actions` property on the `ServiceCloud` singleton.

`SCSActionManagerDelegate`
: This delegate contains optional methods that you can implement to customize the behavior of action buttons.

`SCSActionItem`
: If you want to create your own button, create a `View`-derived class that implements `SCSActionItem`.

`SCSActionItemContainer`
: This class represents the container that holds all the `SCSActionItem` buttons. To access this container object, use the `actionItemContainer` property from `SCSActionManager`.

The Action Manager (`SCSActionManager`)

This class is your entry point into the action button API. Access this object using the `actions` property on the `ServiceCloud` singleton.

In Swift:

```
ServiceCloud.shared().actions
```

In Objective-C:

```
[SCServiceCloud sharedInstance].actions
```

Some methods of this class give you access to the container (`SCSActionItemContainer`) that holds all the action buttons.

Table 15: Action Manager: Container Methods and Properties

Method	Description
`setContainerVisible`	Makes the action button container appear or disappear.
`isContainerVisible`	Tells you whether the action button container is visible.
`actionItemContainer`	Gives you access to the action button container object. See *The Action Container* section.

Other methods give you access to the buttons (`SCSActionItem`) and their actions.

Table 16: Action Manager: Item Methods and Properties

Method	Description
`setActionItemVisible`	Makes a particular action item appear or disappear. See the list of actions after this table.
`isActionItemWithNameVisible`	Tells you whether a particular action item is visible. See the list of actions after this table.
`performAction`	Performs the specified action. See the list of actions after this table.
`setNeedsUpdateActionItems`	Tells the SDK that the action items should update.

When working with built-in actions, use the following values to represent these actions.

`casePublisher`
Launches the Case Publisher interface from where a user can create a case.

`caseList`
Launches the Case List interface from where a user can see a list of their cases.

`caseInterface`
Performs the default Case Management action. If the user is authenticated, this action performs the `SCSActionCaseList` action. If the user is a guest user, this action performs `SCSActionCasePublisher`.

`articleSearch`
Launches the Knowledge article search interface.

`chatInterface`
Launches the Chat interface.

`sosInterface`
Launches the SOS interface.

In addition to these functions, the `SCSActionManager` gives you access to the action delegate, which allows you to create new actions and customize the behavior of existing actions.

The Action Delegate (`SCSActionManagerDelegate`)

This delegate contains optional methods that you can implement to customize the behavior of action buttons. To override the behavior of the action button mechanism, implement `SCSActionManagerDelegate` and pass this delegate to `SCSActionManager`.

Table 17: Delegate Methods That Affect the Buttons

Method	Description
`actionManager(SCSActionManager, actionsToShowFor controller: UIViewController?, withDefaultActions defaultActions: Set<SCSAction>) -> Set<SCSAction>?`	Asks the delegate for the named actions to show when the specified controller is presented. The default actions are passed to this method. You can add your custom actions to this default `Set` or return the default `Set` as is. If this method is not implemented, the default actions are shown.
`actionManager(SCSActionManager, viewForActionItemWithName name: SCSAction) -> UIView?`	Asks the delegate to supply a custom view (that is, a button) for the action of the specified name. If the return value is `nil`, the system-provided view is used, if any. The view is expected to trigger the action using the `performActionWithName:actionItem:` method.
`actionManager(SCSActionManager, sortIndexForActionItemWithName name: SCSAction) -> Int`	Asks the delegate to indicate the relative sort positioning of the specified action item. By default, the sort order for built-in actions is: SCSActionItemDefaultSortArticleSearch = 10, SCSActionItemDefaultSortCaseInterface = 20, SCSActionItemDefaultSortChatInterface = 30, SCSActionItemDefaultSortSOSInterface = 40.
`actionManager(SCSActionManager, shouldShowContainerFor controller: UIViewController?) -> Bool`	Asks the delegate whether to show the action container when the specified controller becomes visible. This method provides the delegate with the opportunity to conditionally show or hide action items when the state changes.
`actionManager(SCSActionManager, shouldShowActionWithName name: SCSAction) -> Bool`	Asks the delegate whether the specified action button should be present.

Table 18: Delegate Methods That Affect the Actions

Method	Description
`actionManager(SCSActionManager, performActionWithName actionName: SCSAction, actionItem: UIView?) -> Bool`	Asks the delegate to perform a specified action. This method is called either in response to a button tap or from a call to `SCSActionManager.performAction`.

Table 19: Delegate Methods for Notification

Method	Description
`actionManager(SCSActionManager, containerWillChangeVisibility visible: Bool, animated: Bool)`	Tells the delegate when the action container will change its visibility.

Method	Description
`actionManager(SCSActionManager, containerDidChangeVisibility visible: Bool, animated: Bool)`	Tells the delegate after the action container has changed its visibility.

Action Items (`SCSActionItem`)

If you want to create your own button, create a `View`-derived class that implements `SCSActionItem`. To create a button that looks and feels like the default buttons, instantiate an `SCSActionButton` object, which is a `UIButton` that also implements `SCSActionItem`. Use the standard button methods, like `setTitle` and `setImage`, to control what is on the button.

If you are creating buttons for new actions, ensure that these actions are visible with the `actionsToShowFor controller: UIViewController?, withDefaultActions defaultActions: Set<SCSAction>)` delegate handler.

To show your button, pass the button to the SDK from the `viewForActionItemWithName name: SCSAction)` delegate handler.

When your custom button is tapped, perform the appropriate action.

The Action Container (`SCSActionItemContainer`)

This class represents the container that holds all the `SCSActionItem` buttons. To access this container object, use the `actionItemContainer` property from `SCSActionManager`. Do not instantiate your own container. This class gives you a few advanced controls related to how the action buttons are displayed.

Table 20: Action Container Methods

Method	Description
`addActionView`	Adds an action view for the specified action name. This method inserts the view into the view hierarchy.
`removeActionView`	Removes the action view with the specified action name.
`actionView(forName: SCSAction)`	Returns the action view with the specified action name.
`visibleActionNames`	Returns the names for the visible action views.
`shouldAdjustVisibilityWhenContentScrolls`	Indicates whether the item container should hide when the content on the screen scrolls. If not implemented, the default is `false`.

Example: **Swift Example**

This simple example adds a custom button whenever the action button container is visible.

```
/**
 Enum extension to include our new action button.
 */
extension SCSAction {
  // TO DO: Update this custom action name as desired...
  static var myCustomAction: SCSAction {
    return SCSAction(rawValue: "MyCustomActionName")
```

```
  }
}
```

```
/**
 TO DO: Add this code to a class that implements SCSActionManagerDelegate
 */
extension MyClassThatImplementsActionManagerDelegate : SCSActionManagerDelegate {

    /**
     Determines which actions to show for a given controller.
     */
    func actionManager(_ actionManager: SCSActionManager,
                       actionsToShowFor controller: UIViewController?,
                       withDefaultActions defaultActions: Set<SCSAction>) ->
Set<SCSAction>?
    {
        var mySet = defaultActions

        // Add our custom action button.
        // (In this case, we're always adding the action button, but
        // you can inspect `controller` to determine whether you want
        // to add a custom button for a given view controller...)
        mySet.insert(.myCustomAction)

        return mySet
    }

    /**
     Shows the button for a given action.
     */
    func actionManager(_ actionManager: SCSActionManager,
                       viewForActionItemWithName name: SCSAction) -> UIView?
    {
        if name == .myCustomAction {

            // Create our custom action button
            let customActionButton = SCSActionButton()
            customActionButton.setTitle("TO DO", for: .normal)
            customActionButton.addTarget(self,
              action: #selector(myCustomButtonHandler),
              for: .touchUpInside)

            return customActionButton
        }

        return nil
    }

    /**
     Handler for the custom action.
     */
    func myCustomButtonHandler(sender: UIButton!) {
        ServiceCloud.shared().knowledge.setInterfaceVisible(false,
          animated: true,
```

```
            completion: nil)

        // TO DO: Perform custom action here!
    }
}
```

Launch the Service SDK from a Web View in iOS

Although this documentation mostly focuses on launching the UI from within your view controller code, you can just as easily launch the UI from a web view.

These instructions assume that you are already familiar with how to launch the UI for the feature you are using:

- Quick Setup: Knowledge in the Service SDK
- Quick Setup: Case Publisher as a Guest User
- Quick Setup: Chat in the Service SDK
- Quick Setup: SOS in the Service SDK

Once you're familiar with the feature, use these instructions to launch the Service SDK UI from a web view.

1. Create a view controller with a single `UIWebView`.
2. Have your view controller implement the `UIWebViewDelegate` protocol.

   In Swift:

   ```
   class ViewController: UIViewController, UIWebViewDelegate {
   ```

   In Objective-C:

   ```
   @interface ViewController : UIViewController <UIWebViewDelegate>
   ```

3. Set your view controller as the web view delegate and configure your Service SDK feature.

   In Swift:

   ```
   override func viewDidLoad() {
     super.viewDidLoad()

     // Point the web view to your web application.
     let url = URL(string: "https://my.web.app")
     let request = URLRequest(url: url!)
     webView.loadRequest(request)

     // Make sure you set your view controller as a delegate to
     // your webview so you can trap requests.
     webView.delegate = self

     // TO DO: Set up and configure your SDK feature...
     // Use SCSServiceConfiguration or SCSChatConfiguration or SOSOptions
     // to configure your feature and point it to your org.
     // See the "Quick Setup" instructions for the appropriate feature.
   }
   ```

In Objective-C:

```
- (void)viewDidLoad {
  [super viewDidLoad];

  // Point the web view to your web application.
  NSURL *url = [NSURL URLWithString:@"https://my.web.app"];
  NSURLRequest *request = [NSURLRequest requestWithURL:url];
  [webView loadRequest:request];

  // Make sure you set your view controller as a delegate to
  // your webview so you can trap requests.
  [webView setDelegate:self];

  // TO DO: Set up and configure your SDK feature...
  // Use SCSServiceConfiguration or SCSChatConfiguration or SOSOptions
  // to configure your feature and point it to your org.
  // See the "Quick Setup" instructions for the appropriate feature.
}
```

4. Add a delegate handler for parsing requests.

   Implement the `webView(UIWebView, shouldStartLoadWith: ULRRequest)` method to trap requests.

   In Swift:

```
func webView(_ webView: UIWebView,
             shouldStartLoadWith request: URLRequest,
             navigationType: UIWebViewNavigationType) -> Bool {

  let url = request.url

  // For this example, we use "servicesdk" as the URL scheme...
  if (url?.scheme == "servicesdk") {

    // "servicesdk://start", corresponds to the host...
    if (url?.host == "start") {

      // TO DO: Launch the SDK here...
      // Use setInterfaceVisible or startSessionWithOptions
      // or startSessionWithConfiguration, depending on what you're launching.
      // See the "Quick Setup" instructions for the appropriate feature.
    }

    // Returning false here ensures that the browser doesn't
    // try to do anything with this request.
    return false
  }

  return true
}
```

   In Objective-C:

```
- (BOOL)webView:(UIWebView *)webView
       shouldStartLoadWithRequest:(NSURLRequest *)request
```

```
                    navigationType:(UIWebViewNavigationType)navigationType {

  NSURL *url = [request URL];

  // For this example, we use "servicesdk" as the URL scheme...
  if ([[url scheme] isEqualToString:@"servicesdk"]) {

    // "servicesdk://start", corresponds to the host...
    if ([[url host] isEqualToString:@"start"]) {

      // TO DO: Launch the SDK here...
      // Use setInterfaceVisible or startSessionWithOptions
      // or startSessionWithConfiguration, depending on what you're launching.
      // See the "Quick Setup" instructions for the appropriate feature.
    }

    // Returning NO here ensures that the browser doesn't
    // try to do anything with this request.
    return NO;
  }

  return YES;
}
```

5. (Optional) If you want to send additional context to your session (such as an email address), you can add a query parameter to the URL (for example, `servicesdk://start?email=new@example.com`), and then parse the parameter before launching the SDK.

   In Swift:

```
func webView(_ webView: UIWebView,
             shouldStartLoadWith request: URLRequest,
             navigationType: UIWebViewNavigationType) -> Bool {

  let url = request.url

  if (url?.scheme == "servicesdk") {

    // Here's the new code to parse a basic query containing an email.
    if (url?.query != nil) {

      // Very simple example that only handles one parameter.
      // In this example query would look like 'email=new@example.com'.
      let query = url?.query!

      // Split on = and get the second component.
      var email = query?.components(separatedBy: "=")[1]

      // In general, you'll want to make sure that this is decoded properly.
      email = email?.removingPercentEncoding

      // TO DO: Do something with this information.
    }

    if (url?.host == "start") {
```

```
      // TO DO: Launch the SDK here...
      // Use setInterfaceVisible or startSessionWithOptions
      // or startSessionWithConfiguration, depending on what you're launching.
      // See the "Quick Setup" instructions for the appropriate feature.
    }

    // Returning false here ensures that the browser doesn't
    // try to do anything with this request.
    return false
  }

  return true

}
```

In Objective-C:

```
- (BOOL)webView:(UIWebView *)webView
        shouldStartLoadWithRequest:(NSURLRequest *)request
                    navigationType:(UIWebViewNavigationType)navigationType {

  NSURL *url = [request URL];
  if ([[url scheme] isEqualToString:@"servicesdk"]) {

    // Here's the new code to parse a basic query containing an email.
    if ([url query]) {

      // Very simple example that only handles one parameter.
      // In this example query would look like 'email=new@example.com'.
      NSString *query = [url query];

      // Split on = and get the second component.
      NSString *email = [query componentsSeparatedByString:@"="][1];

      // In general, you'll want to make sure that this is decoded properly.
      email = [email
        stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

      // TO DO: Do something with this information.
    }

    if ([[url host] isEqualToString:@"start"]) {

      // TO DO: Launch the SDK here...
      // Use setInterfaceVisible or startSessionWithOptions
      // or startSessionWithConfiguration, depending on what you're launching.
      // See the "Quick Setup" instructions for the appropriate feature.
    }

    // Returning NO here ensures that the browser doesn't
    // try to do anything with this request.
    return NO;
  }
```

```
    return YES;
}
```

6. Add the code to your web page that calls the URL. You can do this using HTML or JavaScript.
   a. Call the URL using an HTML anchor tag.

   ```
   <a href="servicesdk://start">Get help!</a>
   ```

   If you want to include additional information, add it to the `href`.

   ```
   <a href="servicesdk://start?email=new@example.com">Get help!</a>
   ```

   b. Call the URL using a JavaScript function.

   ```
   function myFunc() {
       window.location = "servicesdk://start"
   }
   ```

Troubleshooting the Service SDK

Get some guidance when you run into issues.

Enable Debug Logging for the iOS SDK

To configure the Service SDK logs, set the `level` property on the `ServiceLogger` shared instance.

Use the `shared` singleton on `ServiceLogger` to adjust the `level`.

In Swift:

```
ServiceLogger.shared.level = .debug
```

In Objective-C:

```
[SCServiceLogger sharedLogger].level = SCSLoggerLevelDebug;
```

The log level is specified using the `SCSLoggerLevel` enumerated type. It can be one of these values:

- `SCSLoggerLevelDebug` (`.debug` in Swift)
- `SCSLoggerLevelInfo` (`.info` in Swift)
- `SCSLoggerLevelError` (`.error` in Swift)—Default value
- `SCSLoggerLevelFault` (`.fault` in Swift)
- `SCSLoggerLevelOff` (`.off` in Swift)

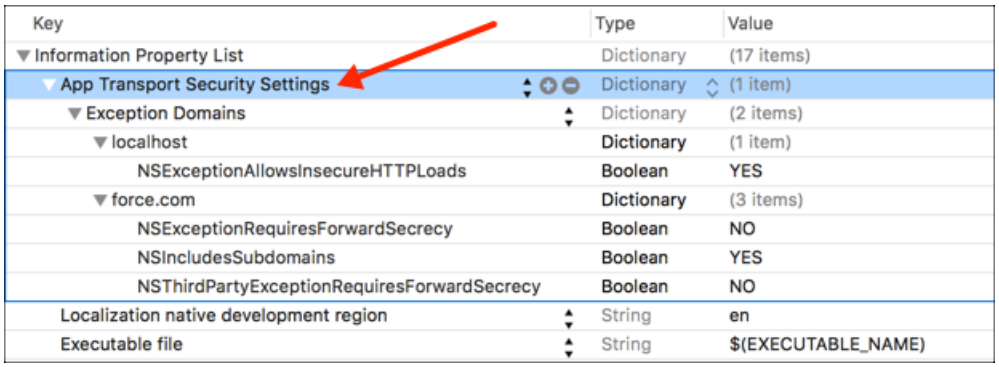
By default, logs go to the console output. You can have logs go to a file using the `filehandle` property.

Can't Access My Knowledge Base

What to do when you can't get to your knowledge base from within your app.

Run through this checklist to help diagnose the root cause.

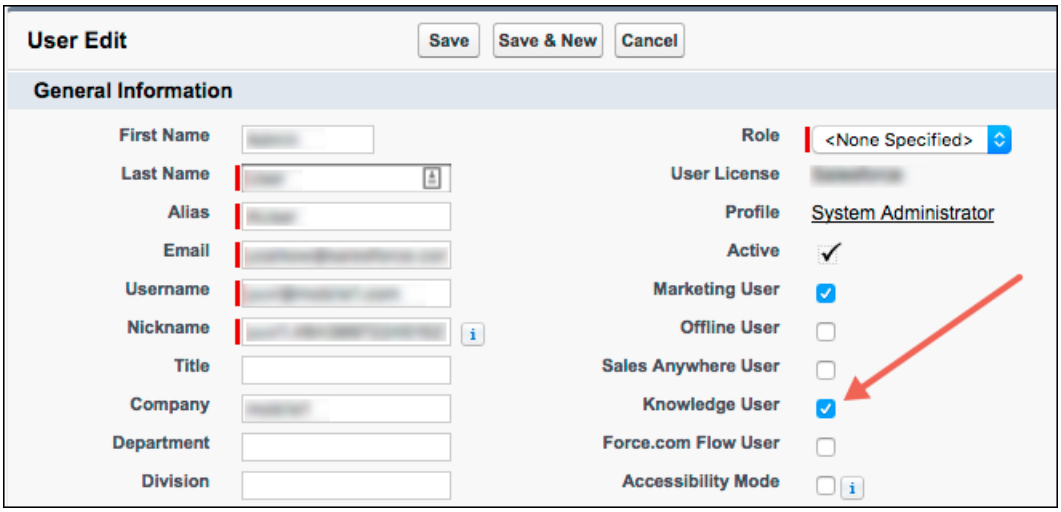
1. Does your `SCSServiceConfiguration` object point to a valid, accessible community URL?
2. Have you set up **App Transport Security** (ATS) exceptions for your community's domain and for `localhost`? See Install the Service SDK for iOS for more info.



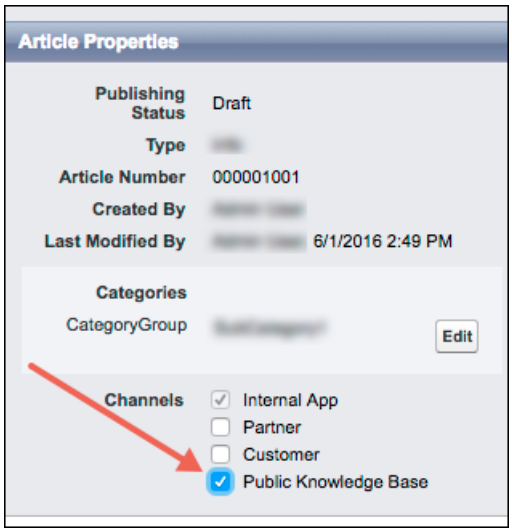
3. Have you set up a Community or Salesforce site? See Cloud Setup for Knowledge for more info.
4. Do you have **Guest Access to the Support API** enabled for your site? See Cloud Setup for Knowledge for more info.

5. (For Knowledge only) Do you have **Knowledge** enabled in your org? Do you have Knowledge licenses? See Cloud Setup for Knowledge for more info.

6. (For Knowledge only) Is the user setting up the knowledge base enabled as a **Knowledge User**? See Cloud Setup for Knowledge for more info.



7. (For Knowledge only) Have you made the article types, the data categories, and the article layout fields visible to guest users? See Guest User Access for Your Community for more info.
8. (For Knowledge only) Have you made your articles accessible to the **Public Knowledge Base** channel? See Cloud Setup for Knowledge for more info.



Can't Connect to Chat

If you can't make a successful connection from your app, even when an agent is standing by, then review how you've set up your Chat implementation.

Run through this checklist to help diagnose the root cause.

1. Verify that the Chat endpoint in your code only specifies the hostname. For instance, if your endpoint is `https://d.la12345.salesforceliveagent.com/chat/rest/`, then use the following value in your code: `d.la12345.salesforceliveagent.com`.
2. Verify that you're using the correct Chat endpoint. See Get Chat Settings from Your Org for more info.
3. Verify that you're using the correct deployment ID and button ID. See Get Chat Settings from Your Org for more info.
4. Verify that you've correctly set up your Chat implementation. See Org Setup for Chat in Lightning Experience with a Guided Flow for more info.

SOS Network Troubleshooting Guide

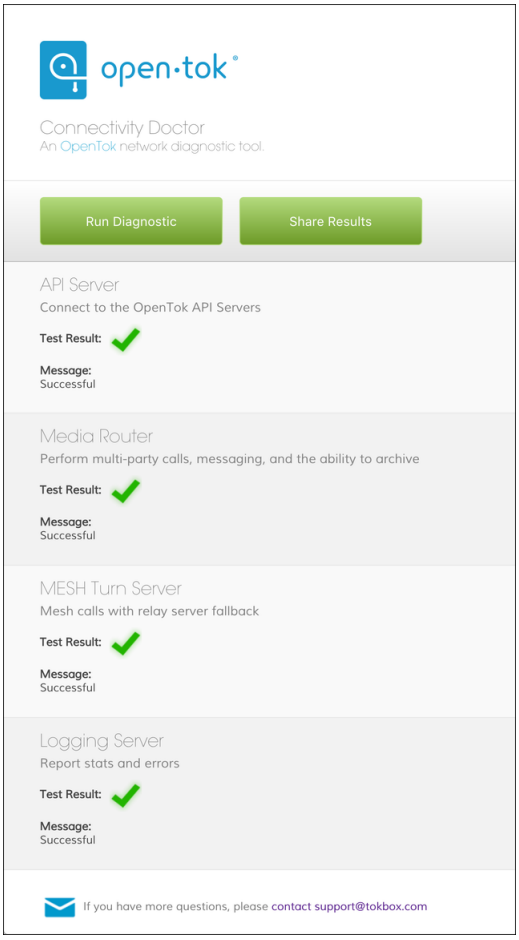
If you can't connect with an SOS agent from your app, you have network connectivity issues, possibly related to your firewall or proxy.

SOS uses the Tokbox OpenTok platform to provide screen sharing and video communication during an SOS session. These guidelines can help you diagnose whether the problem is linked to a networking issue and how to send us diagnostic information if necessary.

Step 1: Run Connectivity Doctor

The Tokbox Connectivity Doctor tests for connectivity issues. You can access this tool via the web, an iOS app, or an Android app. This tool tests network issues in these areas.

1. API server – Session initialization and signaling tests
2. Media router – Whether you can access Tokbox media servers
3. MESH turn server – Relay server fallback mechanism
4. Logging server – Communication of stats and errors to the Tokbox logging server



If all tests pass, go to step 2. If any test fails, you probably need to configure your ports.

API Server or Logging Server Issues

OpenTok clients use HTTP and WSS connections from the client browser to the OpenTok servers on port **TCP/443**. If the only way to access the internet from your network is through a proxy, it must be a transparent proxy. Make sure that TCP/443 is open.

Media Router or Mesh Turn Issues

OpenTok clients can use UDP or TCP connections for media. Salesforce recommends that UDP is enabled to improve the quality of real-time audio and video communications. This connection is bidirectional but always initiated from the client so an external entity can't send malicious traffic in the opposite direction.

- Best experience: We recommend that you open **UDP ports 1025 - 65535**.
- Good experience: Open **UDP port 3478**.
- Minimum experience: Open **TCP port 443**. Some firewall or proxy rules only allow for SSL traffic over port 443. Make sure that non-web traffic can also pass over this port.

Step 2: Test the Tokbox Chat Room

Tokbox has a public-facing chat room site (https://opentokrtc.com/) that you can use for normal chatting. You can also use this site as a test tool.

1. From the chat room site, create a room called “example”. Join that room or click here. You should then see yourself on video. If the video isn't present, make sure that you've given access to the camera and microphone. If nothing happens, go to step 3.

2. Open another browser tab and enter the same URL link as in the previous browser tab (for example, https://opentokrtc.com/room/example). You should see a two-way audio and video-enabled chat. If this process doesn't work, go to step 3.

If you can have a two-way chat, your ports are configured properly and you're done.

Step 3: Gather JSON Metadata

In this step, we'll gather metadata about the failed chat room session. To get this information, open a browser tab and enter https://opentokrtc.com/example.json. If you created a room with a different name, replace the word "example" in the URL accordingly. You'll see JSON content similar to this example.


```
{"apiKey":"45599822","token":"T1==cGyydG5lcl9pZD00NTU5OTgyMiZzaWc9NTcyOWI4NjJhOT
diN2EwYWRmMjZkZjI5MzkxZjkwMjdlNmM0ODNiMTpzZXNzaW9uX2lkPTFfTVg0ME5UVTVPVGd5TW41LU
1UUTNPVEUyTWpVek1USTVObjQ0WkcxSFp6VnNVMnBMZFRKVFp6SmhaRlZ6WVhvM1dVTi1mZyZjcmVhdG
VfdGltZT0xNDc5MTY0MzA4Jm5vbmNlPTAuMzgzODc1MjEwMzAzODEzMiZyb2xlPXB1Ymxpc2hlciZleH
BpcmVfdGltZT0xNDc5MjUwNzA4JmNvbm5lY3Rpb25fZGF0YT0lN0IlMjJ1c2VyTmFtZSUyMiUzQSUyMk
Fub255bW91cyUyMFVzZXI3MiUyMiU3RA==","username":"Anonymous User72","firebaseURL":
"https://ot-archiving.firebaseio.com/sessions//1_MX40NTU5OTgyMn5-MTQ3OTE2MjUzMTI
5Nn43ZG1HZzVsU2pLdTJTZzJhZFVzYXo3WUN-fg","firebaseToken":"eyJ0eXAiOiJKV1QiLCJhbG
ciOiJIUzI1NiJ9.eyJ2IjowLCJkIjp7InVpZCI6IkFub255bW91cyBVc2VyNzIwLjc3NDEwOTM5OTg3N
zQ4ODYiLCJzZXNzaW9uSWQiOiIxX01YNDBOVFU1T1RneU1uNS1NVFEzT1RFMk1qVXpNVEk1Tm40NFpHM
UhaelZzVTJwTGRUSlRaekpoWkZWellYbzNXVU4tZmciLCJyb2xlIjoidXNlciIsIm5hbWUiOiJBbm9ue
W1vdXMgVXNlcjcyIn0sImlhdCI6MTQ3OTE2NDMwN30.ep6l4x_3VGegXVTfwpaOYMGRGOYI944w5Og1h
TyDPfQ","chromeExtId":"undefined","sid":"1_M440NTU5OTgyMn5-MTQ3OTE2MjUzMTI5Nn44Z
G1HZzVsU2pLdTJTZzJhZFVzYXo3WUN-fg"}
```


The output has key/value pairs for the API key ("apiKey"), token ("token"), and session ID ("sid"). Save this information. Tokbox monitors the entire flow for failed and successful sessions (only if you were able to successfully verify the Connectivity Doctor ports requirements first). We'll use this metadata to debug your issue. Go to step 4.

Step 4: Open Support Ticket

Please create a Salesforce support ticket or contact your account team for more help.

Answer the following questions in your support request.

1. TOPOLOGY: What is your network topology? How does data flow through the topology to reach our cloud?
2. ENVIRONMENT: Are you using a virtual environment?
3. PROXY & FIREWALL: What is the type and name of your proxy? What are your firewall restrictions? If you are using a proxy can you execute step 1 and step 2 just through a firewall?
4. PLATFORM & VERSION: What platform are you using? What is the OS version? Which SDK are you using? Which SDK version?
5. DESCRIPTION: Describe the problem you encountered, and the steps you took to try to resolve the problem.
6. LOGS: Can you provide us with any error logs from your side or any other information you deem fit for the problem? Include any relevant network traces or screenshots.
7. CONNECTIVITY DOCTOR: What Connectivity Doctor tests failed? Did you open the required ports and still have issues?
8. JSON METADATA: If applicable, send us the Key/Token/Session information captured in step 3.

My App Crashes

Some tips if your app crashes.

- Chat: If your app crashes when a user attempts to perform a file transfer, check that you've enabled the device privacy permissions for the camera and the photo library. An app will crash if these permissions are not set in Xcode. See Install the Service SDK for iOS.
- SOS: If your app crashes when it is in the process of connecting to an SOS session, check that you've enabled the device privacy permissions for the camera and the microphone. An app will crash if these permissions are not set in Xcode. See Install the Service SDK for iOS.
- For a list of known issues, see the latest Release Notes.

My App Was Rejected

What to do when your app is rejected from the App Store.

- If you receive errors related to unsupported architectures when you upload your app to the App Store, it may be because you didn't strip unneeded architectures from the dynamic libraries used by the Service SDK. See Prepare Your App for Submission for more information.
- If you archive a framework and then export the archive using the Xcode command line tool (`xcodebuild`), you'll get "Invalid Code Signing Entitlements" errors when you try to upload your app to the app store. This is a known issue with Apple's tools. The workaround is to archive and export using Xcode's user interface.

Data Protection and Security in the Service SDK for iOS

The Service SDK does not collect or store personal data from its users. We ensure that data is secure both locally and when in transit.

- **Secure data at rest**. We don't store personal data about the user. We manage keys using iOS Keychain Services. All content fetched from Salesforce servers is stored locally using AES-128 encryption. When the user logs out, we remove all user-specific data from the device.
- **Secure data in transit**. All network communication occurs over SSL using TLS 1.2.

Reference Documentation

Reference documentation for Service SDK for iOS.

To access the reference documentation for the Service SDK for iOS, visit:

- forcedotcom.github.io/ServiceSDK-iOS

This site contains API documentation for the latest version of the SDK.

Reference Index
A list of all classes, protocols, methods, constants, and enums referenced from this developer's guide.

Reference Index

A list of all classes, protocols, methods, constants, and enums referenced from this developer's guide.

Chat Index

Knowledge Index

Case Management Index

SOS Index

Service Common Index

Resource Files

- ServiceChat (Chat) String Resources
- ServiceKnowledge (Knowledge) String Resources
- ServiceCases (Case Management) String Resources
- ServiceSOS (SOS) String Resources
- ServiceCore (Common) String Resources

Additional Resources

If you're looking for other resources, check out this list of links to related documentation.

- Embedded Service SDK for Mobile Apps Resources
  - Service SDK Developer Center
  - Service SDK Trailhead Learning Module
  - iOS: Release Notes, Dev Guide, Reference Docs, Examples
  - Android: Release Notes, Dev Guide, Reference Docs, Examples
- General Resources
  - Service Cloud Developer Center
  - Salesforce Developer Documentation
  - Salesforce Help

INDEX

Q

R

S

T

U

W