Service Cloud Snap-ins for iOS

Version 206.0
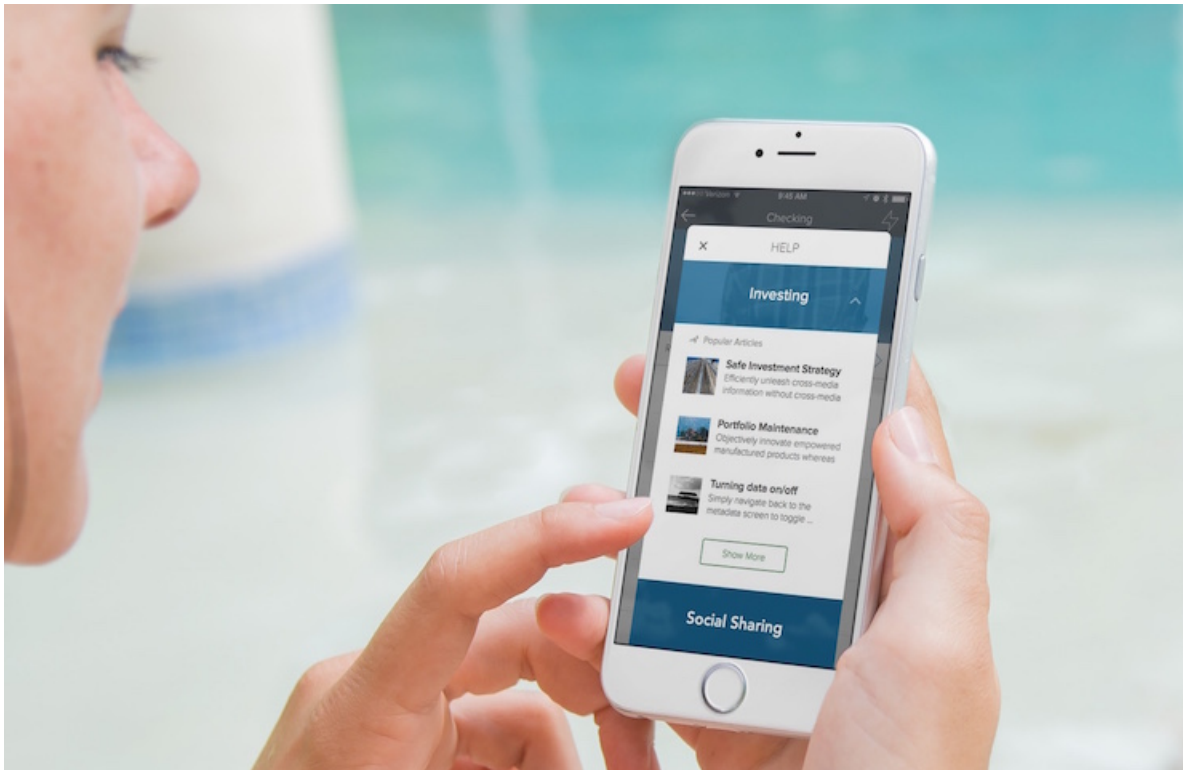
CONTENTS

SERVICE CLOUD SNAP-INS FOR IOS

Service Cloud Snap-ins 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.

February 2017 Release (206.0.0)

This documentation describes the following component versions, which are packaged together in this version of the SDK:

Component	Version Number
Knowledge	2.1.0
Case Management	1.3.0
Live Agent Chat	1.2.0
SOS	3.3.0
ServiceCore (common component used by all features)	1.3.0

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

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

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

Quick Start Tutorials
Get going quickly with these short introductory tutorials.

Using Knowledge
Adding the Knowledge experience to your app.

Using Case Management
Adding the Case Management experience to your app.

Using Live Agent Chat
Adding the Live Agent Chat experience to your app.

Using SOS
Adding the SOS experience to your app.

SDK Customizations
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 guidance on remote notifications and instructions for localizing strings in all supported languages.

Troubleshooting
Get some guidance when you run into issues.

Reference Documentation

Reference documentation for Service Cloud Snap-ins for iOS.

Additional Resources

If you're looking for other resources, check out this list of links to related documentation.

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

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

Cloud Setup for Knowledge

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

Cloud Setup for Case Management

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

Console Setup for Live Agent Chat

To use Live Agent Chat in your app, set up Live Agent for your console.

Console Setup for SOS

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

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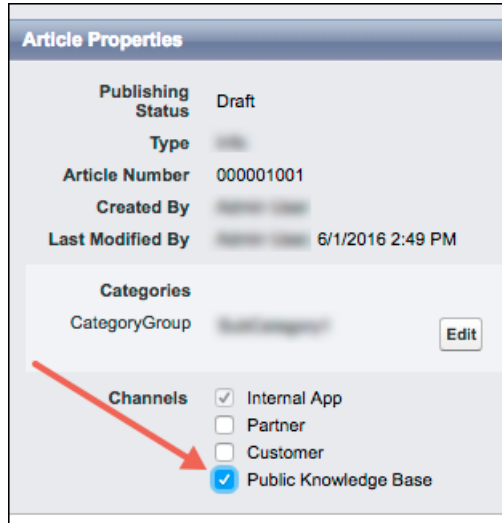
4. Create a community. Your Salesforce org must have an available Community or Force.com site. Your app developer needs the Community URL for the site to use the Knowledge or Case Management feature in the SDK.

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. You'll also need to ensure that **Guest Access to the Support API** is turned on.

> 📝 **Note:** Your Knowledge categories and articles do not appear if the GUEST USER profile isn't set up properly.

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

How to Get the Required Values 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.
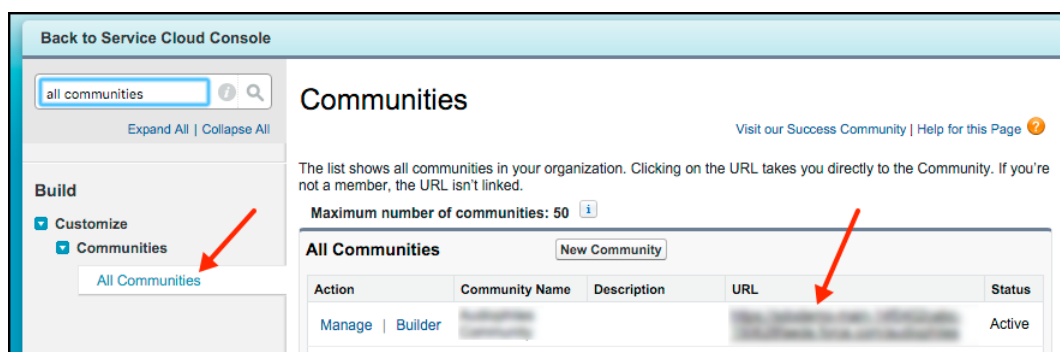
Data Category Group

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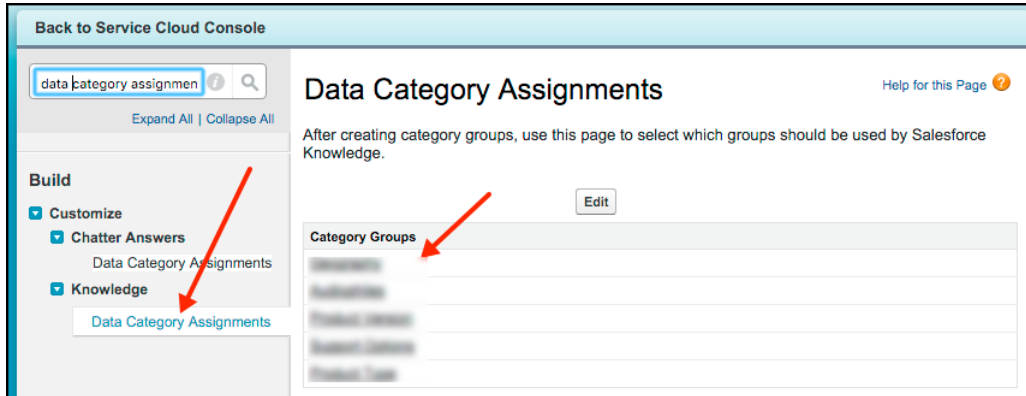
Data Category

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.
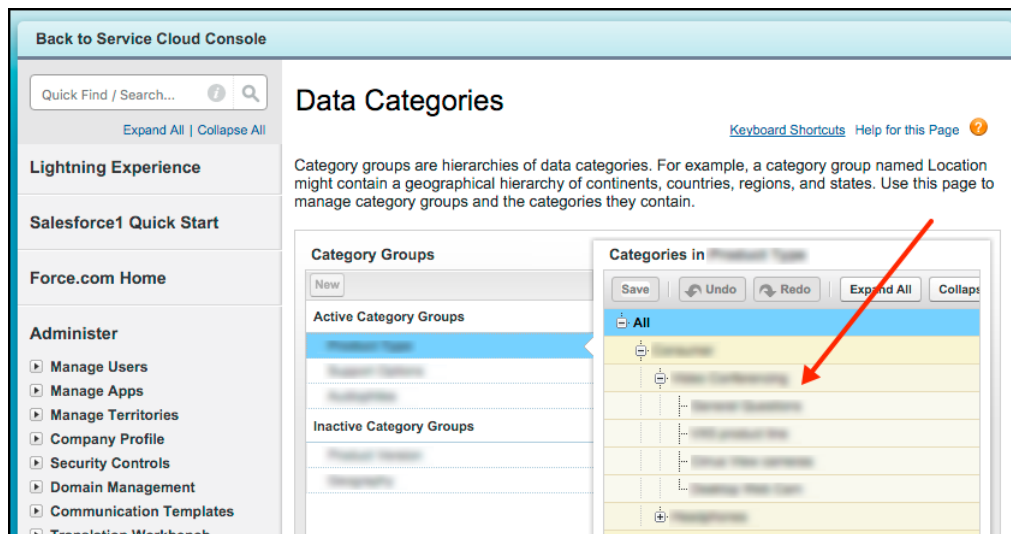
Guest User Access for Your Community

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.

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 Force.com 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. Click **Manage**.
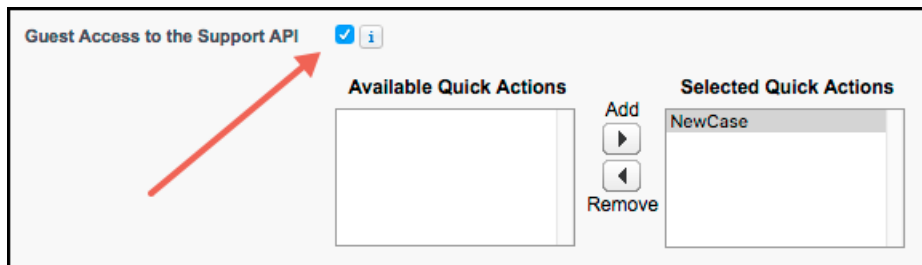
    d. From the sidebar, click **Administration** > **Pages** > **Go to Force.com** to get to the **Site Detail** page.

2. (Force.com sites only) If you are editing the settings for a **Force.com 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.
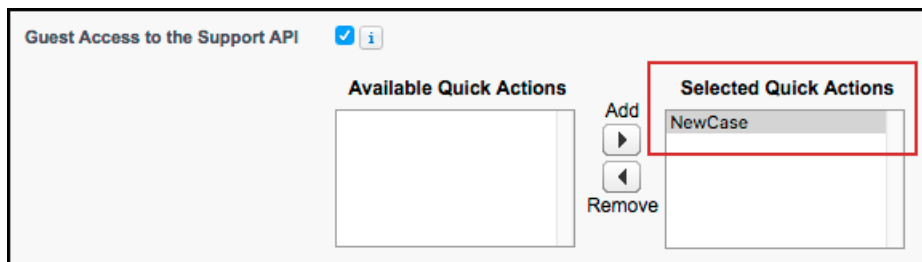
    c. Click **Save**.

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

5. (For Knowledge feature only) Select **Object Settings** and find the Article Types that you want to be visible to users of your app.

   a. Select **Edit** to edit the settings for each Article Type.

   b. Make sure that **Read** is checked for the **Object Permissions** of the Article Type.

   c. Make sure that **Read Access** is checked for the **Field Names** that you want visible.

6. (For Knowledge feature only) Select **Data Category Visibility** and grant visibility to the categories that you want users to see.

Cloud Setup for Case Management

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

1. Create a community. Your Salesforce org must have an available Community or Force.com 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. You'll also need to 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.
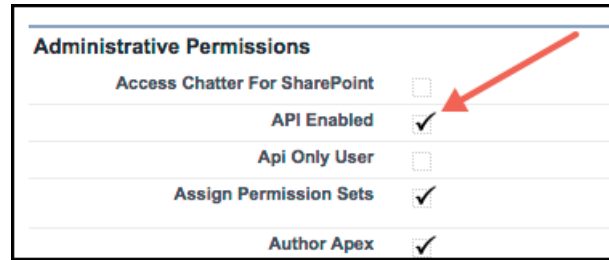
b. You'll need a list view for your cases in Service Cloud. To learn more about creating views, see Create a List View in Salesforce Help. Supply the **Case List Unique Name** for this view to your app developer.

> 📝 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.

How to Get the Required Values 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.
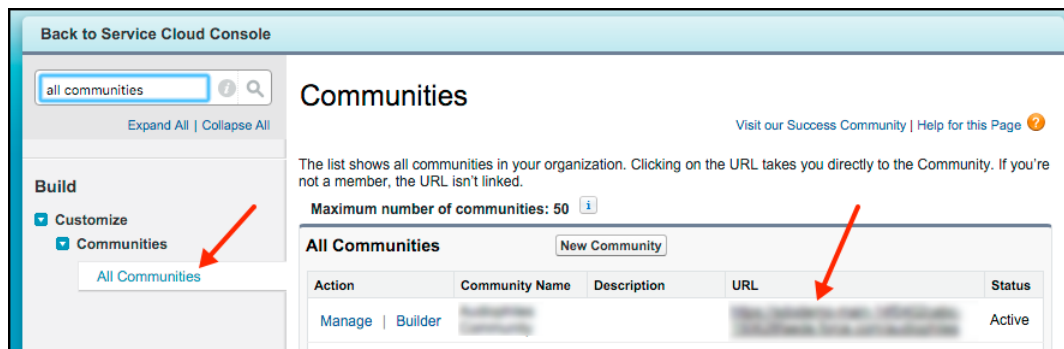
Global Action

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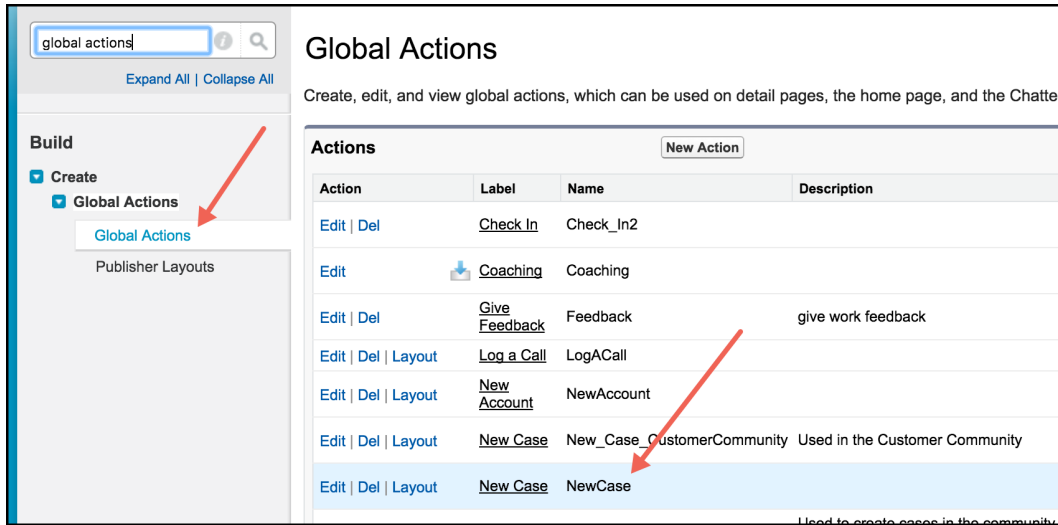
Case List Unique Name

To get this 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.
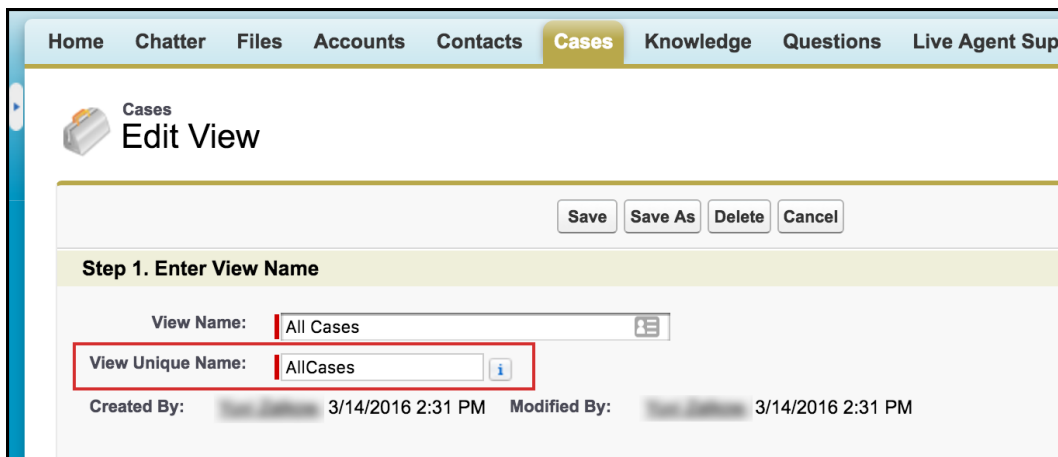
Console Setup for Live Agent Chat

To use Live Agent Chat in your app, set up Live Agent for your console.

📝 Note: If you intend to provide real-time support using *both* Live Agent 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.

1. Create a Live Agent Chat implementation in Service Cloud, as described in Live Agent for Administrators (PDF).

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

   Omni-Channel allows your agents to handle all real-time routing (for example, Live Agent Chat, SOS, email, case management) with the same widget. You can still use Live Agent Chat without setting up Omni-Channel if you don't care for this feature.

> **Note:** Once you've set up Live Agent in the console, **supply your app developer with four values**: the Live Agent pod endpoint, the organization ID, the deployment ID, and the button ID. If you do not have this information handy, you can get it from your org's setup.

How to Get the Required Values from Your Org

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

pod

The hostname for the Live Agent pod that your organization has been assigned. To get this value, from Setup, search for **Live Agent Settings** and copy the hostname from the **Live Agent API Endpoint**. Be sure not to include the protocol or the path — just use the hostname. For example: "d.la12345.salesforceliveagent.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 Live Agent deployment. To get this value, from Setup, select **Live Agent** > **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.

buttonId

The unique button ID for your chat configuration. You can get the button ID by creating a Live Agent chat button (see Create Chat Buttons in the Live Agent help documentation), and instead of using the supplied JavaScript, just copy the `id` attribute. To get this value after creating a button, from Setup, search for **Chat Buttons** and select **Chat Buttons & Invitations**. Copy the `id` for the button from the JavaScript snippet.

Console Setup for SOS

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

📝 Note: If you intend to provide real-time support using *both* Live Agent 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.

You can either use the quick setup or configure your org manually. The quick setup (Quick Setup: SOS Console) is best to get started the first time; manual setup (Manual Setup: SOS Console) is appropriate if you want to customize your org for production. Once you've set up your console, see Additional SOS Console Setup Options for more configuration options.

How to Get the Required Values from Your Org

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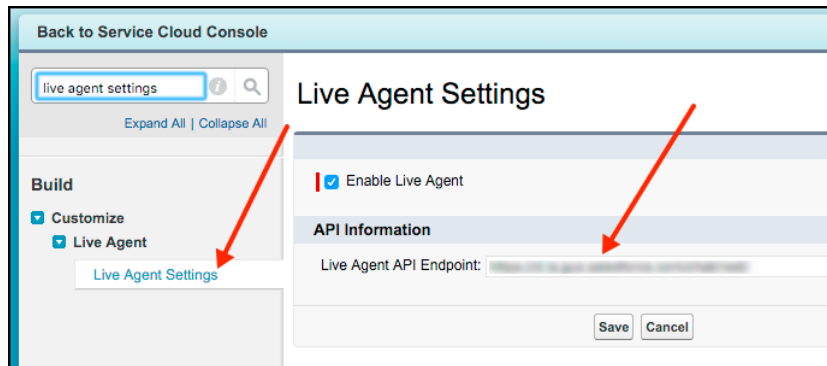
pod

The hostname for the Live Agent pod that your organization has been assigned. To get this value, from Setup, search for **Live Agent Settings** and copy the hostname from the **Live Agent API Endpoint**. Be sure not to include the protocol or the path — just use the hostname. For example: "d.la12345.salesforceliveagent.com".
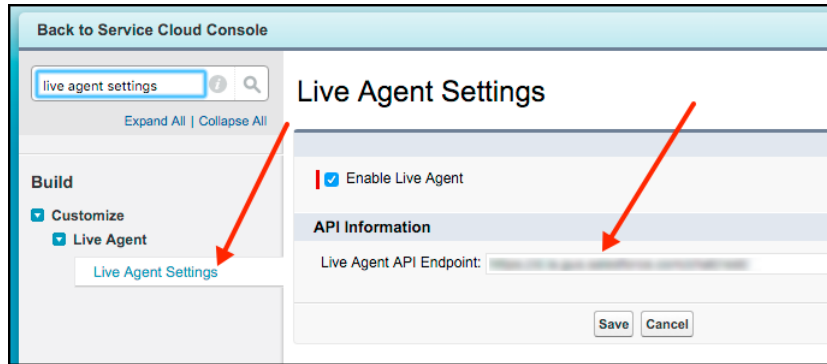
orgId

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**.

Quick Setup: SOS Console

Quick setup is great when you want to try SOS for the first time and you haven't already enabled Omni-Channel or SOS in your org.

Manual Setup: SOS Console

Perform a manual setup when you want to fine-tune your Service Cloud for a production environment.

Additional SOS Console Setup Options
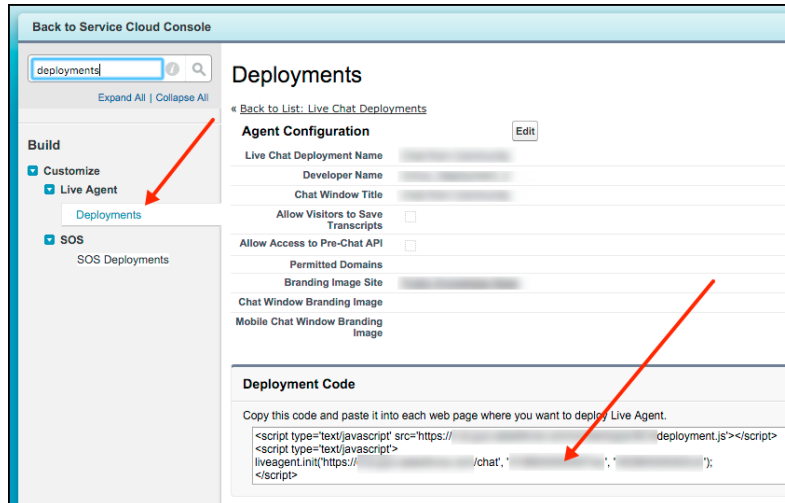
Fine-tune your SOS configuration with additional customizations.

Quick Setup: SOS Console

Quick setup is great when you want to try SOS for the first time and you haven't already enabled Omni-Channel or SOS in your org.

Before running through this quick start, be sure that the SOS Quick Start Package is installed into your Service Cloud instance.

- Install Quick Start Package in your **sandbox** org
- Install Quick Start Package in your **production** org

To simplify the configuration for SOS Service Cloud, we have included an easy-to-use SOS quick setup wizard.

1. Log in to your org and select the **SOS Quick Setup** app.
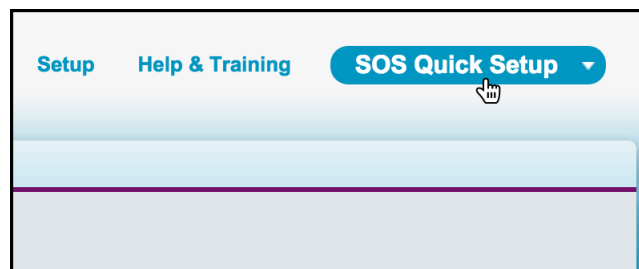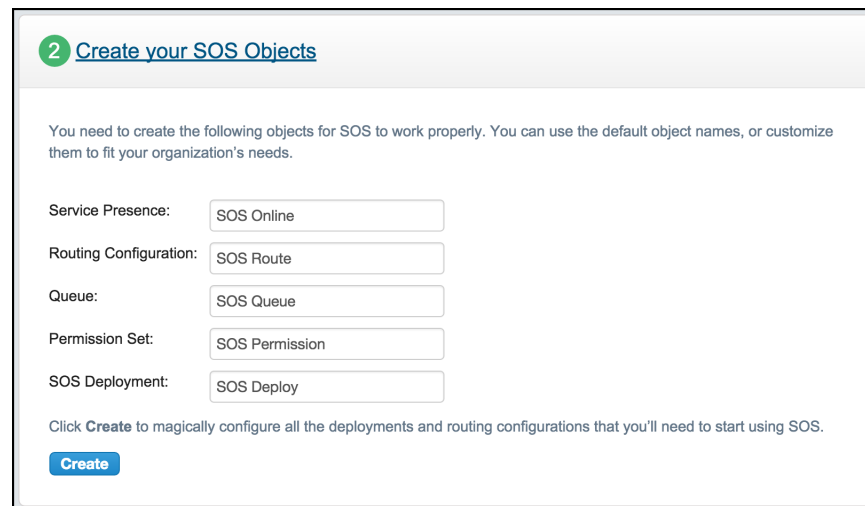
2. Select **Check Org Requirements**.

This step performs basic checks to ensure that your org is set up correctly. Ensure that every line item is marked with **PASS**. You can edit any settings that have not passed.
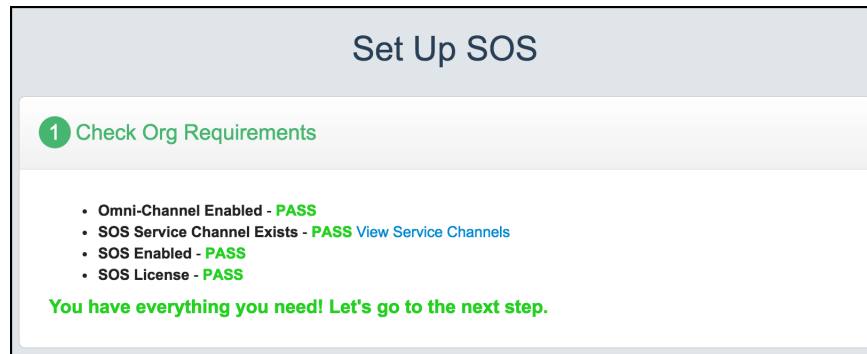
3. Select **Create Your SOS Objects**.

    This step allows you to provide custom names for the new SOS objects. You can leave the default values.

Service Presence
>   Determines what appears in the Omni-Channel widget. You can edit this object and add more Service Channels to the Presence status.

Routing Configuration
>   Allows you to set agent work capacity and priority.

Queue
>   Connects the users to a Routing Configuration. Also allows you to state which objects (in this case, the SOS session) can be owned by this queue.

Permission Set
>   Contains the app permissions to enable the SOS license and to provide access to the Service Presence status. This object must be used to enable the license on a user. All members of the permission set must be assigned an SOS license. It can also be used to enable the Service Presence status.

SOS Deployment
>   Links your customer-facing application to the SOS Queue. Once created, you can configure the deployment to enable session recording by providing Amazon AWS credentials.

4. Click **Create**.

Ensure that every line item is marked with **PASS**. You can edit any settings that have not passed.

5. Add agents to the **Set Up Users** section.

This section allows you to assign the SOS license to an agent and add the agent to the permission set and queue.

6. Select **Add the Omni-Channel Widget to Your Console App**.

Choose the **Service Cloud Console** and click **Update App**.

This step adds the following to your Console App:

- Omni-Channel Widget
- SOS Console component
- Whitelisted domains necessary for SOS
- Report dashboard to the Navigation tab

7. (Optional) Select **Custom Your Console Settings**.

You can set your org to automatically create a case for each SOS session and open it in a subtab.

The case is created by a trigger included with the package. Before the new SOS session is inserted, the trigger creates a case and adds a reference to it to the SOS session object. If you wish to modify the trigger, it can be found in **Setup**, by searching **SOS Sessions** and going to **Triggers**. The name of the trigger is **SOSCreateCase**.

The case is popped in a subtab by a page that is hidden in the SOS session page layout. This hidden page and an altered SOS session page layout are also included with the package.

8. Select **Info for Your SOS App**.

   This step provides you with the three pieces of information required to start an SOS session from the SDK: Organization ID, SOS Deployment ID, and Live Agent API Endpoint. Save this information for later.

Once you've completed these steps, you are ready to start using the SOS feature in the Service SDK.

Manual Setup: SOS Console

Perform a manual setup when you want to fine-tune your Service Cloud for a 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 Additional SOS Console Setup Options.

Additional SOS Console Setup Options

Fine-tune your SOS configuration with additional customizations.

Assign SOS Permissions

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

Automatic 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

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

Record SOS Sessions

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

SOS Reference ID

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

Multiple Queues

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

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 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. 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".

> Note: The **Label** field can contain spaces but the **Name** field cannot.

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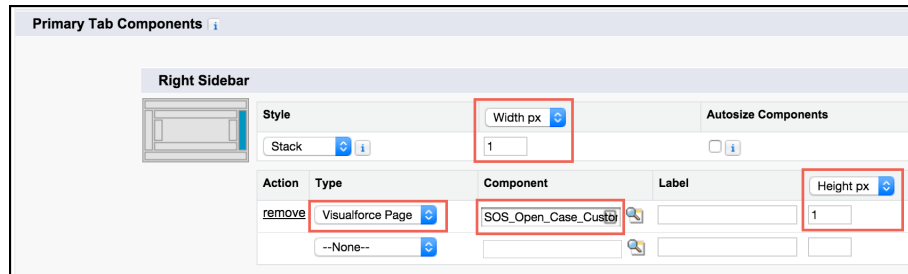
  - Set **Style** to **Stack**.
  - Set **Width px** to **1**.
  - Set **Height px** to **1**. (Change **Height %** to **Height px** if necessary.)
  - Set **Type** to **Visualforce Page**
  - Set **Component** to the page created previously (for example, **SOS_Open_Case_Custom**).

  f. Click **Save**.

Listen for SOS Console Events

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

Event Listener API

To detect state changes from the Salesforce console, you can use JavaScript in your Visualforce page. Specifically, use the `addEventListener` method, which is documented in the Salesforce Console Integration Toolkit Developer Guide. The method syntax is:

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

eventType (String)

  For SOS session state events, this value is `SFORCE_SOS:STATE_CHANGED`.

eventListener (Function)

  This function is called when the registered event is emitted. You receive one JSON `message` object within the payload passed to this function. The payload is described in the Event Listener Payload section.

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.
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.

State	How a Session Can End Without an Error	How a Session Can End with an 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
	  }
	}```

Scenario	Sample Payload

Session ended by agent while customer app is in the background	<pre>{
	  currentState: 'ENDED',
	  previousState: 'PAUSED',
	  reason: {
	    name: 'ENDED_BY_CONSOLE',
	    error: null
	  }
	}</pre>

Session ended by agent after app crash	<pre>{
	  currentState: 'ENDED',
	  previousState: 'WAITING',
	  reason: {
	    name: 'ENDED_BY_CONSOLE',
	    error: null
	  }
	}</pre>

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',</pre>

Error Scenario	Sample Payload

	`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.'`
	`    }`
	`  }`
	`}`
	`}`

Session dies after OpenTok drops connection because of a timeout	`{`
	`  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'`
	`      }`
	`    }`
	`  }`
	`}`

Record SOS Sessions

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

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 Queues

Implement multiple 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.

SDK Setup

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

Requirements

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

SDK Development Requirements

To develop using this SDK, you must have:

- iOS SDK version 9 or newer
- Xcode version 8 or newer

App Requirements

Any app that uses this SDK requires:

- iOS version 9 or newer

SOS Agent Requirements

The agents responding to SOS calls need to 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

Browser requirements:

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

Operating System:

- OSX 10.5 or newer
- Windows 7 or newer

Install the SDK

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. (Live Agent Chat only) If you're using Live Agent 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 Live Agent 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.

6. Turn off **Enable Bitcode** for your project unless you've received a bitcode-enabled SDK.

    a. Open your Project Target Settings.

    b. Select the **Build Settings** tab.

    c. Search for **Enable BitCode** in the provided search bar.

    d. Set the value to **NO**.
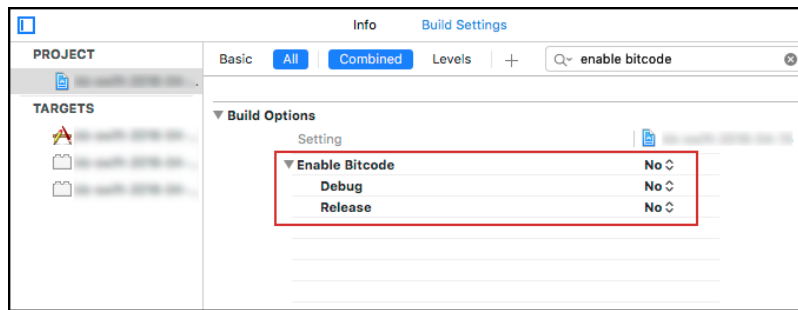
You're now ready to get started using the SDK!

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. Add the Service Cloud SDK dependency.

    a. Change directories into the root directory of your application project and create or edit the file named `Podfile`

    b. If you want to install the 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 Service SDK (with all components)
target '<your app target>' do
    pod 'ServiceSDK'
end
```

    c. 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 with the `pod` command.

Feature	Pod name
Knowledge	`ServiceSDK/Knowledge`
Case Management	`ServiceSDK/Cases`
Live Agent 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'll 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 3.3.0 of SOS, specify 206.0.0 because version 206.0.0 of the Service SDK has version 3.3.0 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', '206.0.0'
end
```

4. Run the CocoaPods installer.

```
pod install
```

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

5. 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

Add the Frameworks Manually

Add the SDK frameworks by manually embedding the appropriate frameworks.

1. Download the SDK frameworks from the Service SDK landing page.

2. Embed the relevant Service SDK frameworks into your project.

   The following frameworks are available to you:

Framework	Description	Required?
SalesforceKit	This framework gives you access to the Salesforce Mobile SDK. See the Mobile SDK Developer's Guide for more info.	Yes
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 Live Agent Chat features of the SDK.	Only if using Live Agent 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

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. You can resolve this problem by using the script provided in the

`SalesforceKit` 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 `SalesforceKit` framework in your project directory.

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

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

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

   ```
   "$PODS_ROOT/ServiceSDK/Frameworks/SalesforceKit.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.

Run Script for Manual Framework Installation
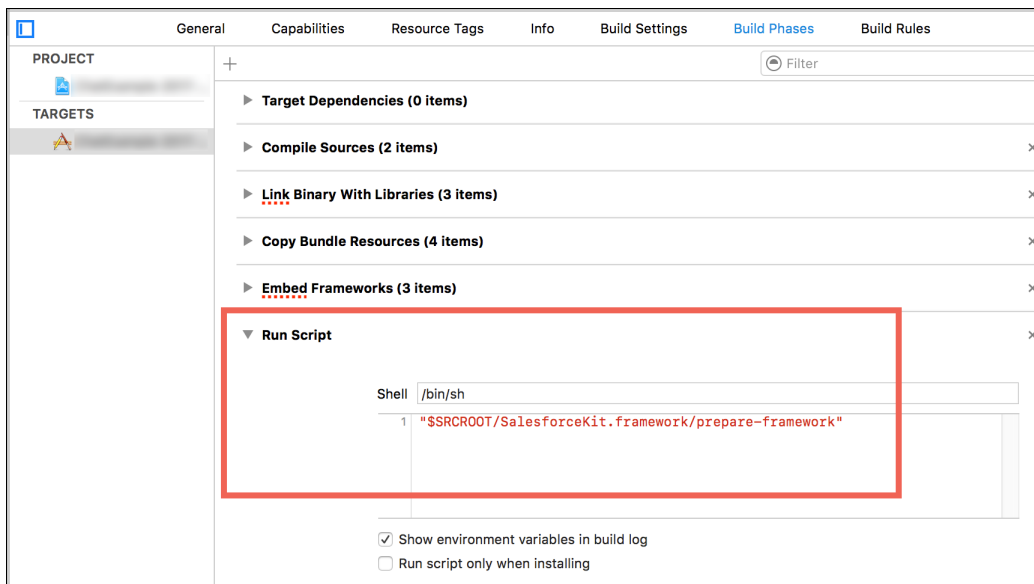
Run Script for CocoaPods Installation
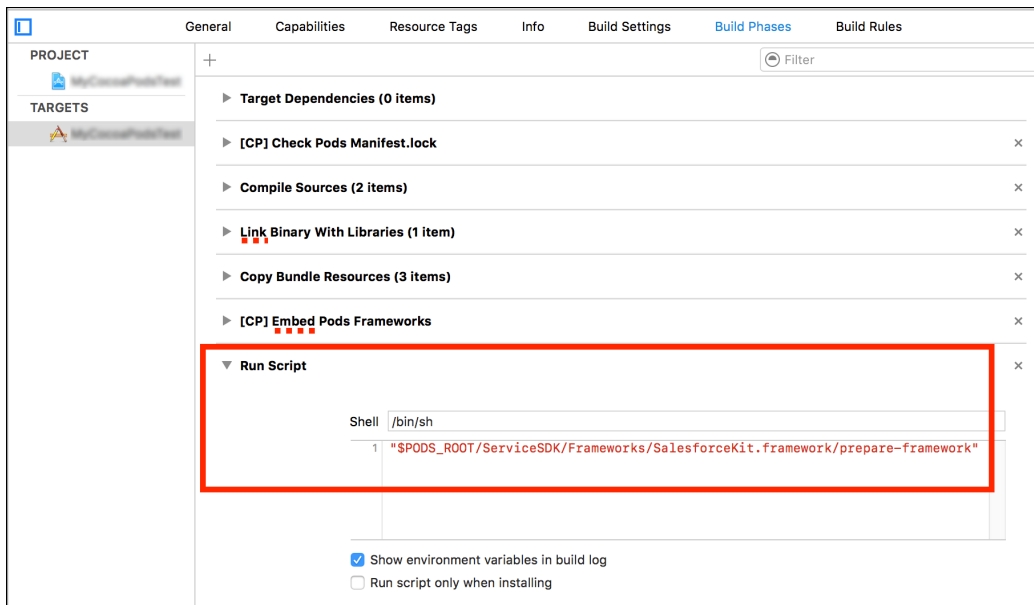
Quick Start Tutorials

Get going quickly with these short introductory tutorials.

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 Live Agent Chat

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

Get Started with SOS

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

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`.
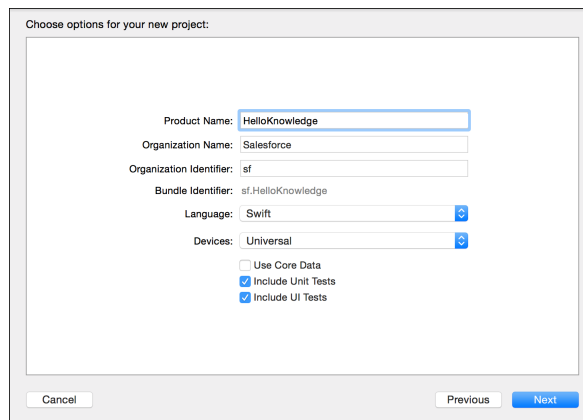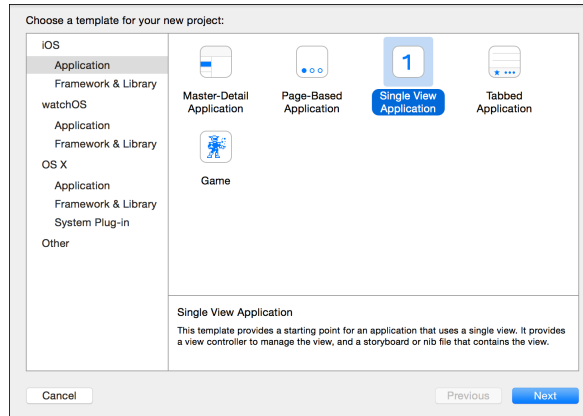
2. Install the SDK as described in Install the SDK.

3. From your app delegate implementation, import the SDK.

    In Objective-C:

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

    In Swift:

    ```
    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 `SCSServiceCloud` shared instance using the
    `initWithCommunity:dataCategoryGroup:rootDataCategory:` method

    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;
```

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
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 Objective-C:

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

   In Swift:

   ```
   import ServiceCore
   import ServiceKnowledge
   ```

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

   In Objective-C:

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

   In Swift:

   ```
   SCServiceCloud.sharedInstance().knowledge.setInterfaceVisible(true,
                                                  animated: true,
                                                  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 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`.

Choose a template for your new project:

iOS				
Application				
Framework & Library	Master-Detail Application	Page-Based Application	Single View Application	Tabbed Application
watchOS				
Application				
Framework & Library	Game			
OS X				
Application				
Framework & Library				
System Plug-in				
Other				

Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Cancel   Previous   Next

Choose options for your new project:

Product Name:	HelloCases
Organization Name:	Salesforce
Organization Identifier:	sf
Bundle Identifier:	sf.HelloCases
Language:	Swift
Devices:	Universal
	☐ Use Core Data
	☑ Include Unit Tests
	☑ Include UI Tests

Cancel   Previous   Next

2. Install the SDK as described in Install the SDK.

3. From your app delegate implementation, import the SDK.

In Objective-C:

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

In Swift:

```
import ServiceCore
import ServiceCases
```

4. Point the SDK to your org from the `applicationDidFinishLaunchingWithOptions` method in your `UIApplicationDelegate` class.

To connect your application to your organization, create an `SCSServiceConfiguration` object containing the community URL. Pass this object to the `SCServiceCloud` shared instance using the `initWithCommunity:` method.

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;
```
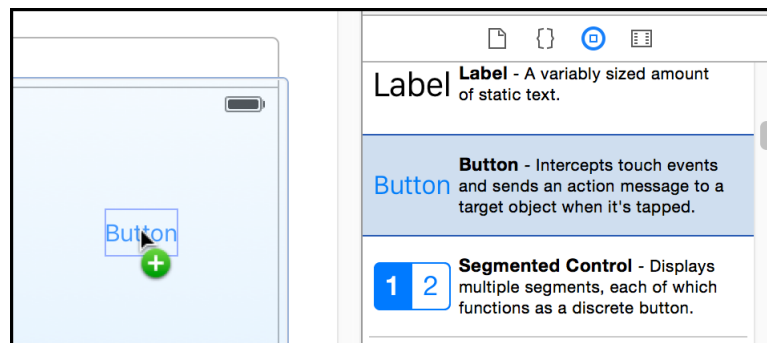
41

In Swift:

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

// Pass configuration to shared instance
SCServiceCloud.sharedInstance().serviceConfiguration = config
```
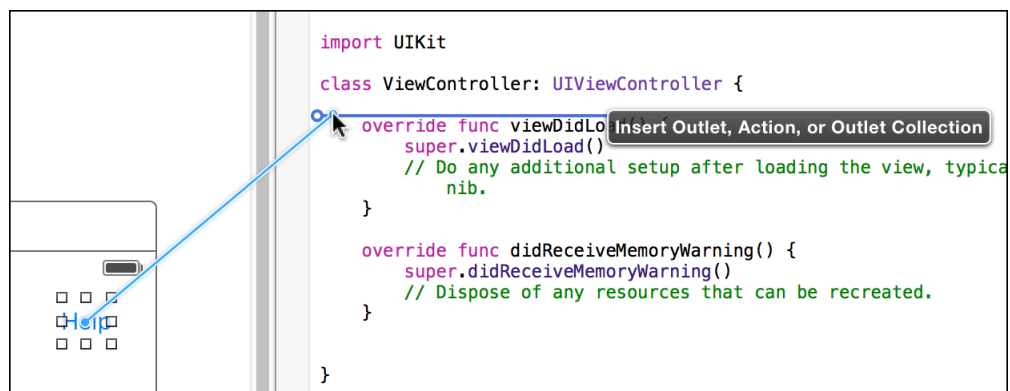
You can get the community URL from your Salesforce org. From Setup, search for **All Communities**, and copy the URL for the desired community. For more help, see Cloud Setup for Case Management.

> Note: If you plan to access Knowledge in addition to Case Management, use the `initWithCommunity:dataCategoryGroup:rootDataCategory:` method instead. This method sets up data categories in addition to setting the community URL. See Quick Setup: Knowledge 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 Objective-C:

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

In Swift:

```
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 Objective-C:

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

In Swift:

```
import ServiceCore
import ServiceCases
```

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

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().cases.setInterfaceVisible(true,
                                        animated: true,
                                        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.

Get Started with Live Agent Chat

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

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

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

1. Create an Xcode project. For this example, let's make a Single View Application. Name it `HelloLiveAgentChat`.
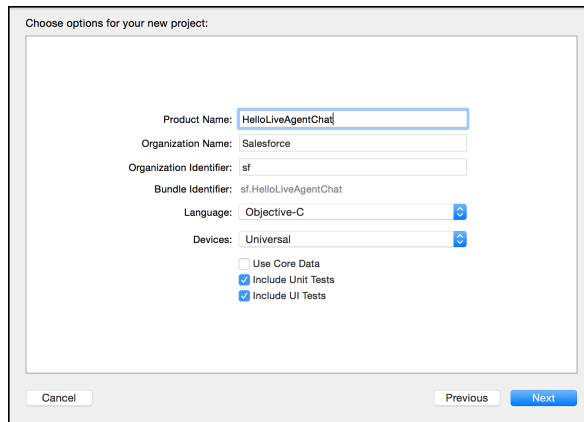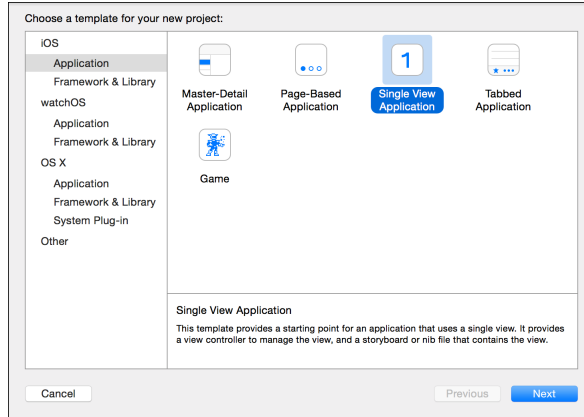
2. Install the SDK as described in Install the SDK.

3. Go to your storyboard and place a button somewhere on the view. Name it `Chat`.
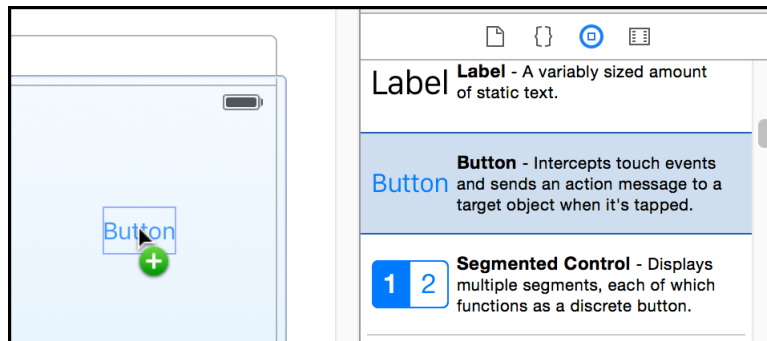
4. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `launchChat`.
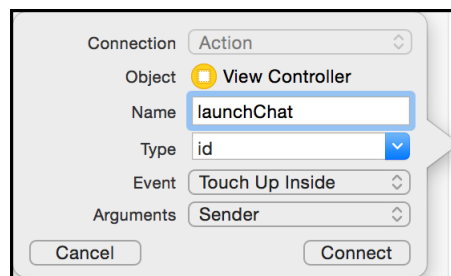
5. Import the SDK. Wherever you intend to use the Live Agent Chat SDK, be sure to import the Service Common framework and the Live Agent Chat framework.

In Objective-C:

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

In Swift:

```
import ServiceCore
import ServiceChat
```

6. Launch a Live Agent Chat session from within the `launchChat` method.

From the button action implementation, launch Live Agent Chat using the `startSessionWithOptions` method.

In Objective-C:

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

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

  // Start the session.

  [[SCServiceCloud sharedInstance].chat startSessionWithConfiguration:config];
}
```

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.

  SCServiceCloud.sharedInstance().chat.startSessionWithConfiguration(config)
}
```

Fill in the placeholder text for the Live Agent pod, the org ID, the deployment ID, and the button ID.

pod

The hostname for the Live Agent pod that your organization has been assigned. To get this value, from Setup, search for **Live Agent Settings** and copy the hostname from the **Live Agent API Endpoint**. Be sure not to include the protocol or the path — just use the hostname. For example: "d.la12345.salesforceliveagent.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 Live Agent deployment. To get this value, from Setup, select **Live Agent** > **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.

buttonId

The unique button ID for your chat configuration. You can get the button ID by creating a Live Agent chat button (see Create Chat Buttons in the Live Agent help documentation), and instead of using the supplied JavaScript, just copy the `id` attribute. To get this value after creating a button, from Setup, search for **Chat Buttons** and select **Chat Buttons & Invitations**. Copy the `id` for the button from the JavaScript snippet.

7. Launch **Service Cloud Console**. From the **Omni-Channel** widget, ensure that a Live Agent agent is online.
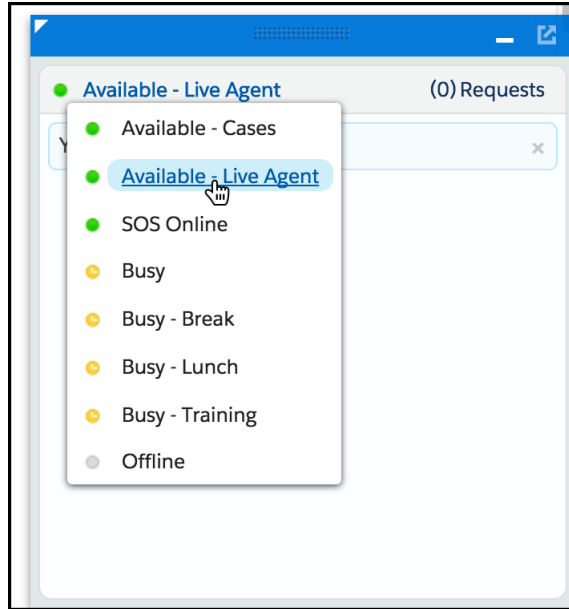
Now you can build and run the app. When you tap the **Chat** button, the app requests a Live Agent chat session, which an agent can accept from the **Service Cloud Console**. From the console, an agent can real-time chat with a customer.
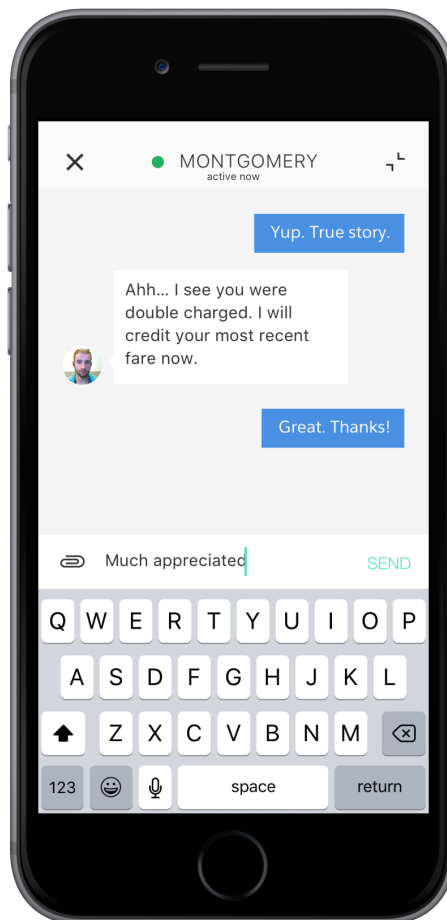
Get Started with SOS

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`.
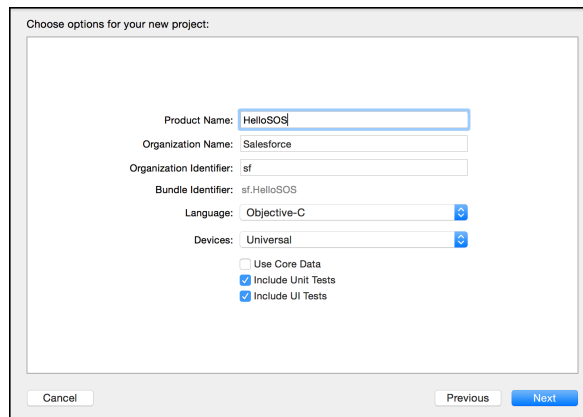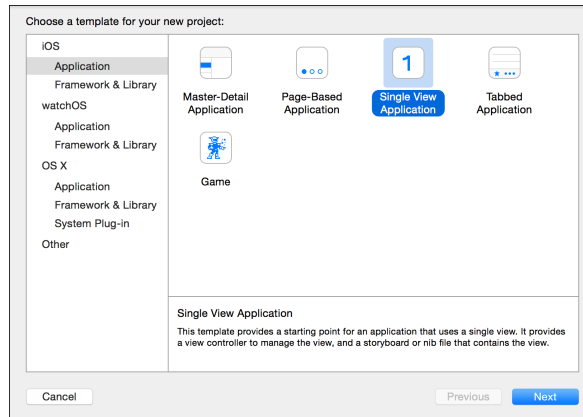
2. Install the SDK as described in Install the SDK.
3. Go to your storyboard and place a button somewhere on the view. Name it `SOS`.
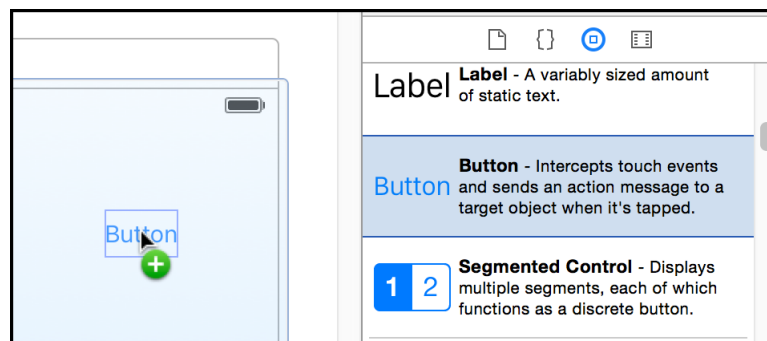
4. Add a `Touch Up Inside` action to your `UIViewController` implementation. Name it `launchSOS`.
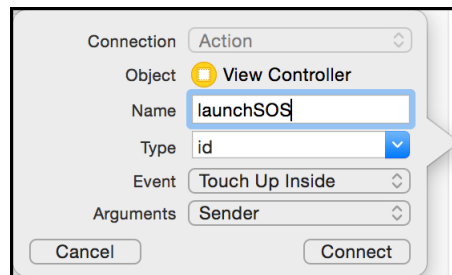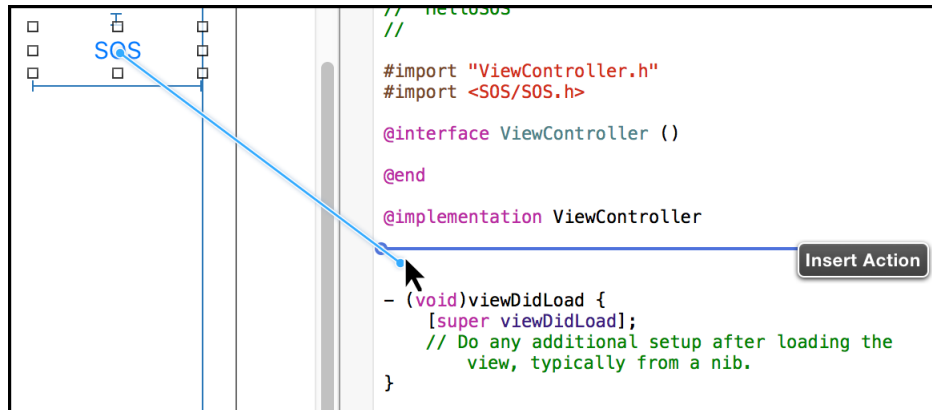
5. Import the SDK. Wherever you intend to use the SOS SDK, be sure to import the Service Common framework and the SOS framework.

In Objective-C:

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

In Swift:

```
import ServiceCore
import ServiceSOS
```

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

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

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];
}
```

In Swift:

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

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

  SCServiceCloud.sharedInstance().sos.startSessionWithOptions(options)
}
```

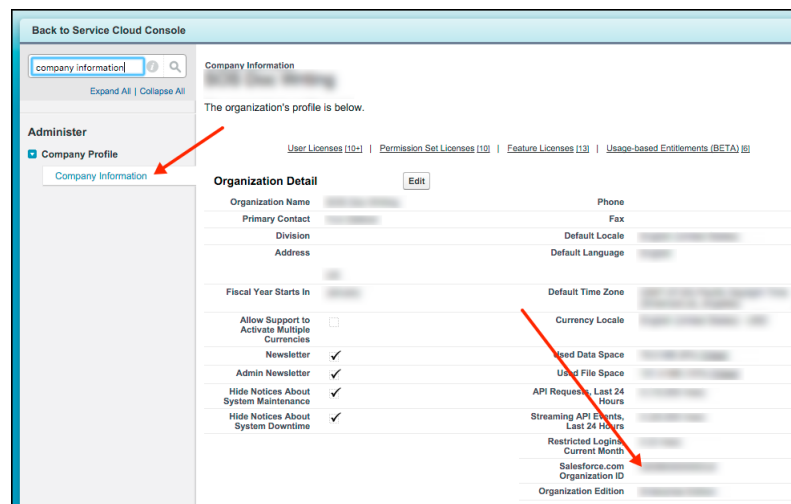Fill in the placeholder text for the Live Agent pod, the org ID, and the deployment ID.

pod

The hostname for the Live Agent pod that your organization has been assigned. To get this value, from Setup, search for **Live Agent Settings** and copy the hostname from the **Live Agent API Endpoint**. Be sure not to include the protocol or the path — just use the hostname. For example: "d.la12345.salesforceliveagent.com".
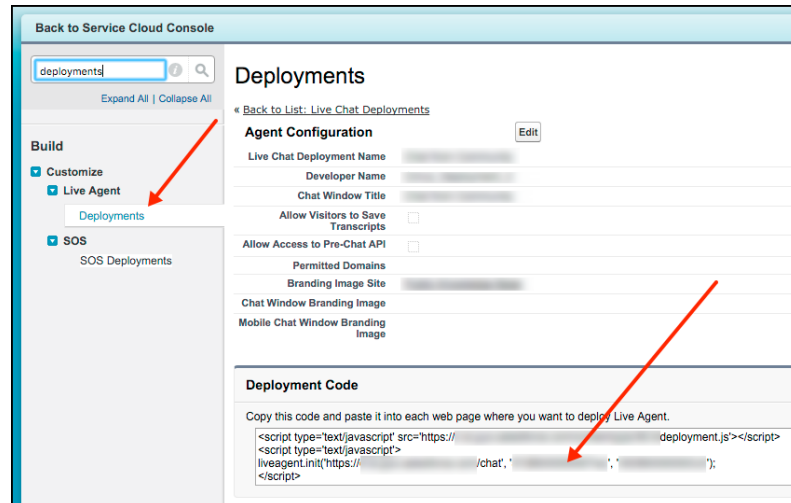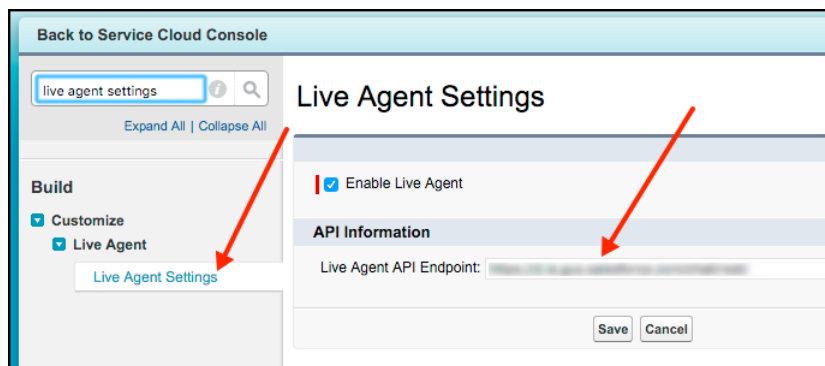

orgId

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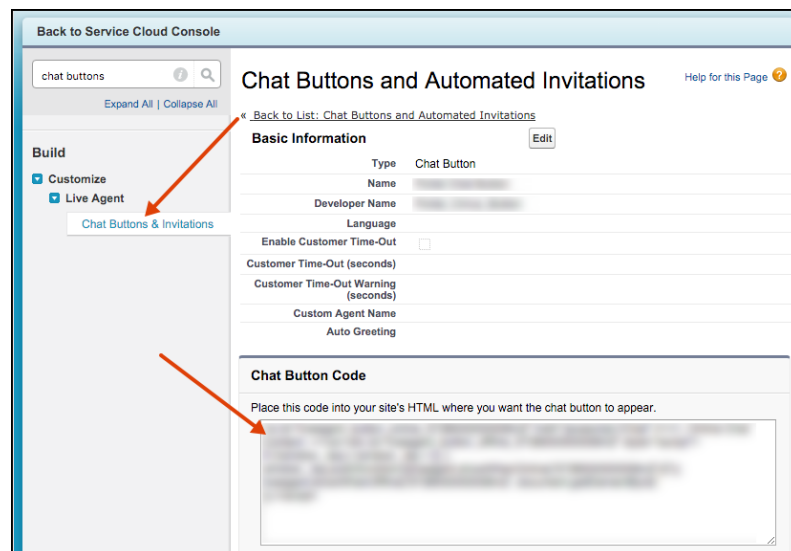

deploymentId

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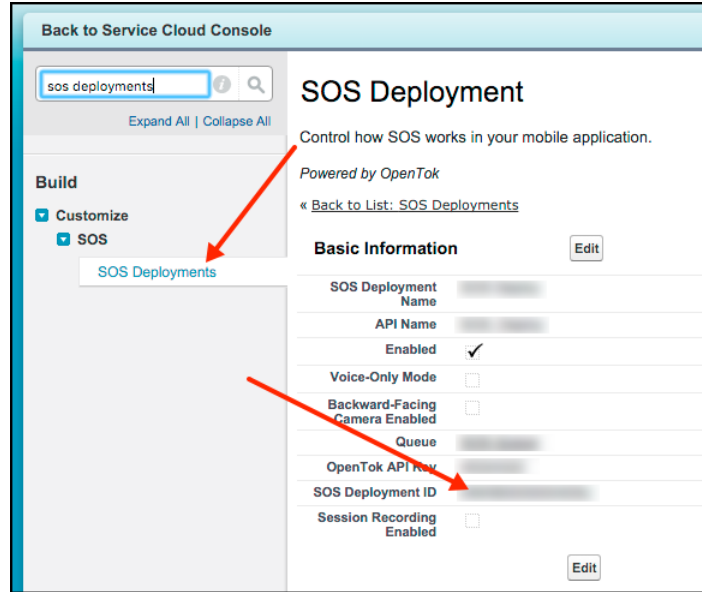
7. Launch **Service Cloud Console**. From the **Omni-Channel** widget, ensure that an SOS agent is online.
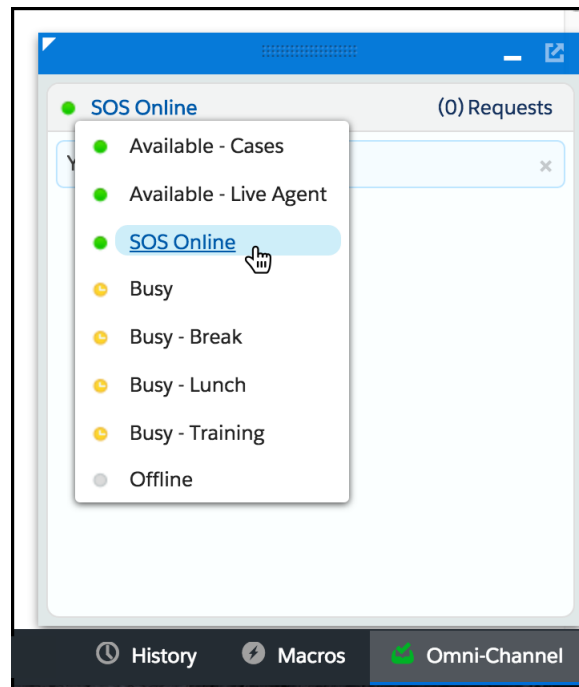
Now you can build and run the app. When you tap the **SOS** button, the app requests an SOS session, which an agent can accept from the **Service Cloud Console**. From the console, you can chat with the customer, annotate things on their screen, and perform a two-way video session (if enabled).
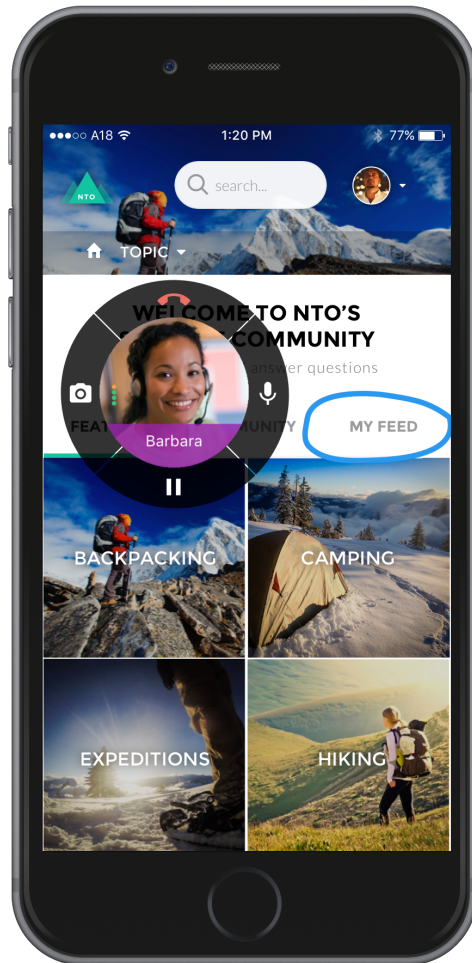
Using Knowledge

Adding the Knowledge experience to your app.

Knowledge Overview

Learn about the Knowledge experience using the SDK.

Quick Setup: Knowledge

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

Knowledge as an Authenticated User

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

Customize the Presentation for Knowledge

The simplest way to show and hide the Knowledge interface is by calling the `setInterfaceVisible` method on the `SCServiceCloud.knowledge` property. Alternatively, you can present the interface using a custom presentation. You can even manually display a Knowledge article yourself.

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.

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 `SCSServiceCloudDelegate`.

Knowledge Overview

Learn about the Knowledge experience using the SDK.

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.
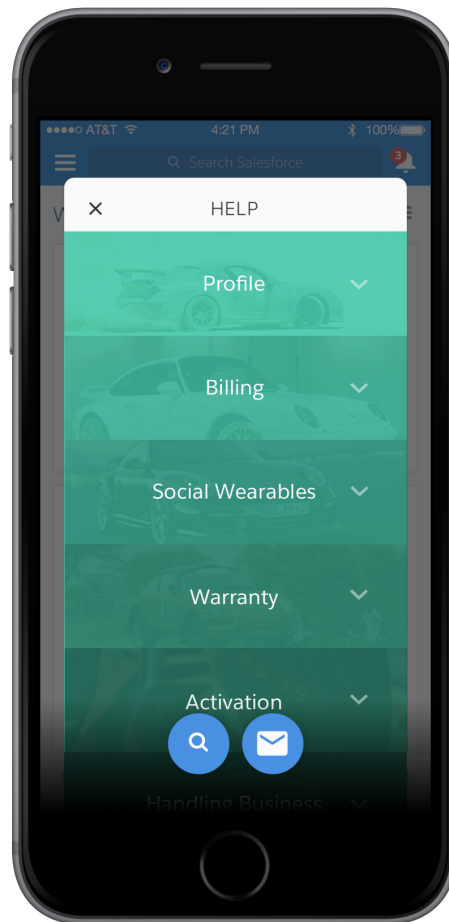
By default, knowledge appears as a floating dialog on top of your app's existing content, though you can customize the presentation if you'd like. From the knowledge home, a user can navigate through articles that are organized by category. Articles are also searchable from within the app. By default, 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.
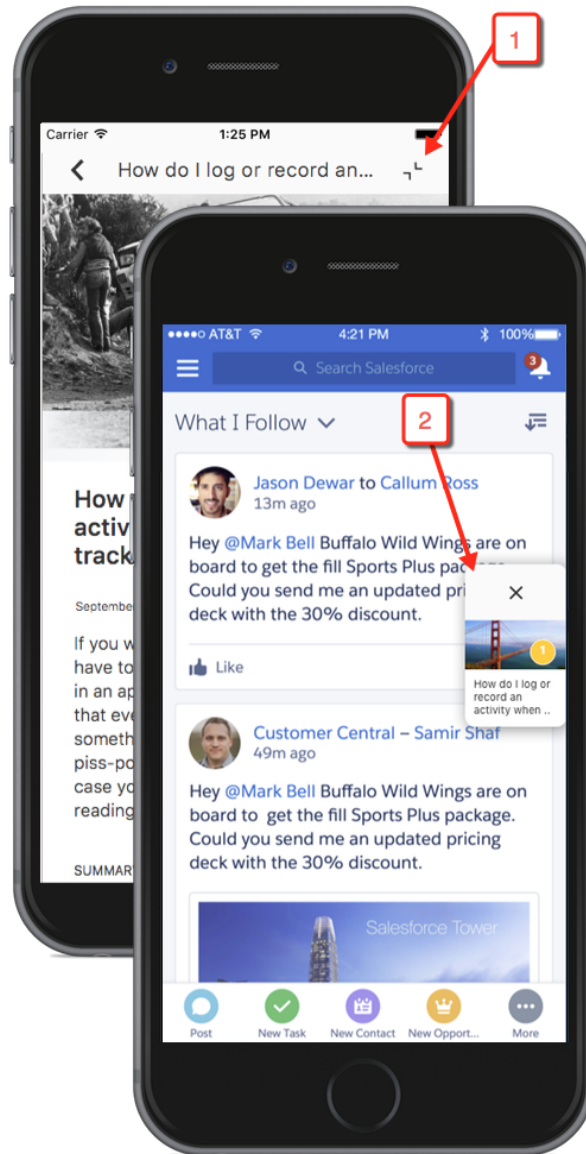
You can also customize the look and feel of the interface so that it fits naturally within your app. These customizations include the ability to fine-tune the colors, the fonts, the images, and the strings used throughout the interface.

Let's get started.

Quick Setup: Knowledge

To set up Knowledge, point the shared instance to your community, 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 Knowledge. To learn more, see Cloud Setup for Knowledge.
- Installed the SDK. To learn more, see Install the SDK.

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

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 Objective-C:

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

    In Swift:

    ```
    import ServiceCore
    import ServiceKnowledge
    ```

2. Point the SDK to your org from the `applicationDidFinishLaunchingWithOptions` method in your `UIApplicationDelegate` class.

    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 `SCSServiceCloud` shared instance using the `initWithCommunity:dataCategoryGroup:rootDataCategory:` method

    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;
    ```

    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
    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 with the configuration object.

You can configure the colors, fonts, and images to your interface with an `SCAppearanceConfiguration` instance. It contains the methods `setColor:forName:`, `setFontDescriptor:fontFileName:forWeight:`, and `setImage:compatibleWithTraitCollection:forName:`. Here's an example of how you can change the colors.

In Objective-C:

```
// Create appearance configuration object with some appearance changes
SCAppearanceConfiguration *appearance = [SCAppearanceConfiguration new];
[appearance setColor:[UIColor greenColor]
            forName:SCSAppearancePrimaryBrandColor];
[appearance setColor:[UIColor redColor]
            forName:SCSAppearanceBrandContrastColor];
[appearance setColor:[UIColor blackColor]
            forName:SCSAppearanceContrastPrimaryColor];

// Pass appearance configuration to shared instance
[SCServiceCloud sharedInstance].appearanceConfiguration = appearance;
```

In Swift:

```
// Create appearance configuration object with some appearance changes
let appearance = SCAppearanceConfiguration()
appearance.setColor(UIColor.green, forName: SCSAppearancePrimaryBrandColor)
appearance.setColor(UIColor.red, forName: SCSAppearanceBrandContrastColor)
appearance.setColor(UIColor.black, forName: SCSAppearanceContrastPrimaryColor)

// Pass appearance configuration to shared instance
SCServiceCloud.sharedInstance().appearanceConfiguration = appearance
```

You can also configure the strings used throughout the interface and which action buttons are visible when viewing your knowledge base.

See SDK Customizations for more info.

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 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:animated:completion:`.

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

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().knowledge.setInterfaceVisible(true,
                                            animated: true,
                                            completion: nil)
```

By default, the interface appears as a floating dialog. Alternatively, you can present the interface using a custom presentation. See Customize the Presentation for Knowledge for more info.

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

If you run into issues accessing your community, check out Unable to Access My Community.

👁 **Example:** To use this example code, create a Single View Application and Install the SDK.

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

In Objective-C:

```
#import "AppDelegate.h"
@import ServiceCore;
@import ServiceKnowledge;

@implementation AppDelegate


- (BOOL)application:(UIApplication *)application
            didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

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

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

  // Customize the appearance with the configuration object
  SCAppearanceConfiguration *appearanceConfiguration =
      [SCAppearanceConfiguration new];
  [appearanceConfiguration setColor:[UIColor greenColor]
                          forName:SCSAppearancePrimaryBrandColor];
  [appearanceConfiguration setColor:[UIColor redColor]
                          forName:SCSAppearanceBrandContrastColor];
  [appearanceConfiguration setColor:[UIColor blackColor]
                          forName:SCSAppearanceContrastPrimaryColor];
  [SCServiceCloud sharedInstance].appearanceConfiguration =
      appearanceConfiguration;

  return YES;
}
@end
```

In Swift:

```
import UIKit
import ServiceCore
```

```swift
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
    SCServiceCloud.sharedInstance().serviceConfiguration = config

    // Create appearance configuration object with some appearance changes
    let appearance = SCAppearanceConfiguration()
    appearance.setColor(UIColor.green, forName: SCSAppearancePrimaryBrandColor)
    appearance.setColor(UIColor.red, forName: SCSAppearanceBrandContrastColor)
    appearance.setColor(UIColor.black, forName: SCSAppearanceContrastPrimaryColor)

    // Pass appearance configuration to shared instance
    SCServiceCloud.sharedInstance().appearanceConfiguration = appearance

    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.

In Objective-C:

```objc
#import "ViewController.h"
@import ServiceCore;
@import ServiceKnowledge;

@implementation ViewController

- (IBAction)showHelp:(id)sender {

  // Show the Knowledge interface
  [[SCServiceCloud sharedInstance].knowledge setInterfaceVisible:YES
                                                        animated:YES
                                                      completion:nil];
}
@end
```

In Swift:

```swift
import UIKit
import ServiceCore
import ServiceKnowledge

class ViewController: UIViewController {

  @IBAction func showHelp(_ sender: AnyObject) {

    SCServiceCloud.sharedInstance().knowledge.setInterfaceVisible(true,
        animated: true,
        completion: nil)
  }
}
```

Knowledge as an Authenticated User

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

These instructions set up your knowledge base as an authenticated user. 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 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. Review the steps in Quick Setup: Knowledge.

The basic steps for setting up and displaying the interface still apply for authenticated users.

2. If you're using a Salesforce community and authenticating with the Salesforce Mobile SDK, configure the login endpoint as described in the Salesforce Mobile SDK documentation: Configure the Login Endpoint.

The documentation describes how to use the `SFDCOAuthLoginHost` property in your `info.plist` file to create a custom login URI.

3. Write the code to authenticate a user with your org.

You can authenticate in several ways.

- **Create normal user accounts on your Salesforce org.** If you have (or want to create) Salesforce user accounts for each user of your app, see Digging Deeper into OAuth 2.0 on Force.com, which describes how to set up a connected app on Salesforce. For developers already using the Salesforce Mobile SDK, the Mobile SDK Developer's Guide contains instructions for authenticating with that SDK.

- **Use your own user credentials to create community portal users on Salesforce.** You can authenticate using existing credentials on your server. One approach is to create an API endpoint on your server that is accessible to your app. For each user, your server can formulate unique information representing that user. Use this information in a JSON Web Token (JWT) payload. Sign and submit this information to the OAuth endpoint on your Salesforce org. Using Apex code configured within Salesforce, you can use the JWT information to identify a previously created community user or implicitly create a new user. That user's OAuth keys can be created and supplied back to your server.

  - Documentation about using JWT for Salesforce authorization: OAuth 2.0 JWT Bearer Token Flow.

- Documentation about creating a community portal user with an Apex trigger: Apex Developer's Guide: Force.com Sites Examples.

- More documentation about creating a community portal user: Authenticating Users on Force.com Sites.

The Service SDK does not contain any additional method to authenticate.

4. When authenticated, hold onto an `SFUserAccount` object with your credentials.

Whichever method you use to authenticate, you need an `SFUserAccount` object to pass to the Service SDK. This class is part of the `SalesforceKit` framework from the Salesforce Mobile SDK. The account object must have correct information in its `credentials` property. If you're using the Mobile SDK to authenticate, the `SFOAuthCoordinator` instance you're given in your delegate handler contains a credentials property that you can use.

5. Implement `SCServiceCloudDelegate` and pass account information from within the `serviceCloud:shouldAuthenticateService:completion:` method. The SDK calls this method whenever it plans to display a view that can support authenticated users.

The `SCServiceCloud` shared instance has a delegate you can implement.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().delegate = self
```

If you want an authenticated connection, implement the `serviceCloud:shouldAuthenticateService:completion:` method from this delegate and return `YES`/`true`. This method is a good place to pass in the user account with the supplied completion block, which runs asynchronously.

In Objective-C:

```
- (BOOL)serviceCloud:(SCServiceCloud *)serviceCloud
        shouldAuthenticateService:(NSString *)service
                      completion:(void (^)(SFUserAccount * _Nullable))completion {

  SFUserAccount* user = // TO DO: Get user account

  // After acquiring user information, call
  // this completion block to set the new user:
  completion(user);

  return YES;
}
```

In Swift:

```
func serviceCloud(serviceCloud: SCServiceCloud,
     shouldAuthenticateService service: String,
                     completion: (SFUserAccount?) -> Void) -> Bool {

  let user: SFUserAccount = // TO DO: Get user account

  // After acquiring user information, call
  // this completion block to set the new user:
  completion(user)
```

```
  return true
}
```

6. For error handling, implement `serviceCloud:serviceAuthenticationFailedWithError:` from `SCServiceCloudDelegate`.

In Objective-C:

```
- (void)serviceCloud:(SCServiceCloud*)serviceCloud
        serviceAuthenticationFailedWithError:(NSError*)error
        forService:(NSString*)service {

  // TO DO: Inspect error and handle appropriately.
}
```

In Swift:

```
func serviceCloud(serviceCloud: SCServiceCloud,
    serviceAuthenticationFailedWithError error: Error,
    forService service: String) {

  // TO DO: Inspect error and handle appropriately.
}
```

The `service` param is `SCServiceTypeKnowledge` for Knowledge and `SCServiceTypeCases` for Case Management.

The `NSError` object contains information about the error. The `code` property on this object contains the error code. The following errors are the most common error codes you can encounter:

Error Code	Description
`SCServiceUserSessionExpiredOrInvalidError` (401)	Occurs when the session has expired or the ID is invalid. Use the refresh token to acquire another access token and then update the `account` property, as shown in the next step.
`SCServiceUserRequestRefusedError` (403)	Occurs when the user does not have sufficient access for the request. Verify that the logged in user has sufficient credentials.
`SCServiceUserResourceNotFoundError` (404)	Occurs when the requested resource was not found. Check the URI and other parameters for errors.

See Status Codes and Error Responses for a full set of possible error codes.

7. When account information changes, update the `account` property.

Account information can change if the access token expires, the user logs out, another user logs in, and for various other reasons. You can update the `account` property on the `SCServiceCloud` shared instance. The Service SDK updates the interface to correspond with the updated account information. Preferably, instead of setting the property, use the `setAccount:completion:` method, which allows you to specify a completion block for error handling.

In Objective-C:

```
SFUserAccount *user = // Get user account
[[SCServiceCloud sharedInstance]
                setAccount:user completion:^(NSError *error) {
```

```
  // TO DO: Handle error
}];
```

In Swift:

```
let user = // Get user account

SCServiceCloud.sharedInstance()
             .setAccount(user, completion: { (error: Error?) in

  // TO DO: Handle error
})
```

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

Customize the Presentation for Knowledge

The simplest way to show and hide the Knowledge interface is by calling the `setInterfaceVisible` method on the `SCServiceCloud.knowledge` property. Alternatively, you can present the interface using a custom presentation. You can even manually display a Knowledge article yourself.

Activating Interface Using the Default Presentation

Use the `setInterfaceVisible:animated:completion:` 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 Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().knowledge.setInterfaceVisible(true,
                                          animated: true,
                                          completion: nil)
```

Activating Interface Using a Custom Transitioning Delegate

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

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

    In Objective-C:

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

    In Swift:

    ```
    SCServiceCloud.sharedInstance().delegate = mySCServiceCloudDelegate
    ```

2. Implement the `serviceCloud:transitioningDelegateForViewController:` method in your delegate and
return a custom `UIViewControllerTransitioningDelegate` from this method.

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;
}
```

In Swift:

```
func serviceCloud(_ serviceCloud: SCServiceCloud,
                   transitioningDelegateFor controller: UIViewController)
                   -> UIViewControllerTransitioningDelegate? {

  // TO DO: Put your logic here and then return your transitioning delegate...

  return myTransitioningDelegate
}
```

Showing Article Content Manually

You can manually display an article by instantiating an `SCSArticleViewController`, specifying the article using the `article`
property, and then presenting the view yourself. This method is useful if you want to display specific articles within your app. 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.

Article caching is valuable for 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.

There are several classes associated with article caching:

SCSKnowledgeManager
  Manager class for interacting with Knowledge at the data level, including article caching functionality.

SCSCategoryGroup
  Class that represents a category group.

SCSCategory
  Class that represents a category or subcategory.

SCSMutableArticleQuery

> Class used to build your article query. The immutable parent class is `SCSArticleQuery`.

SCSArticle

> Class that contains information about an article.

When using the caching feature, there are several areas to consider:

1. Fetching Categories

2. Querying for Articles

3. Downloading Content

4. Displaying Content

We've included an example after discussing these topics.

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 `fetchAllCategoriesWithCompletionHandler:`. 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 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

  }];
}
```

In Swift:

```
let knowledgeManager = SCSKnowledgeManager.default()
if (!knowledgeManager.hasFetchedCategories()) {

  knowledgeManager.fetchAllCategories(completionHandler: {
    (categoryGroups: [SCSCategoryGroup]?, error: Error?) in

      // 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 an preexisting instance of `SCSKnowledgeManager`. After you authenticate a user, call the static `defaultManager` method on `SCSKnowledgeManager` 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 `fetchArticlesWithQuery:completion:`. To fetch local (already cached) articles for a given query, call `articlesMatchingQuery:completion:`. These methods take an `SCSMutableArticleQuery` 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 an `SCSMutableArticleQuery` instance, you can determine the type of query using the following properties:

Table 4: Query Types

Query Type	Property Name / Type	Notes
Articles matching article ID	`articleId: NSString`	The 18-character article ID. This property cannot be used with `searchTerm`, `queryMethod`, `sortOrder`, or `sortByField`.
Articles matching search term	`searchTerm: NSString`	This property cannot be used with `articleId`, `sortOrder`, or `sortByField`.
Articles within a set of categories	`categories`: array of `SCSCategory` 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 5: 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. For example: title, published date, view score.

You can filter the query results with the following properties:

Table 6: Query Filters

Query Filter	Property Name / Type / Default	Description
Number of articles to fetch	`pageSize: NSUInteger = 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

Query Filter	Property Name / Type / Default	Description
		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 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

}];
```

In Swift:

```
let query = SCSMutableArticleQuery()
query.searchTerm = "login issues"
query.pageSize = 5

knowledgeManager.fetchArticles(with: query, completion: {
  (articles: [SCSArticle], error: Error?) in

    // TO DO: Download articles using downloadContentWithOptions

})
```

Downloading Content

Once you've fetched the articles, you'll need to download them before displaying. Use the downloadContentWithOptions:completion: method in the SCSArticle 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 Objective-C:

```
SCSArticle *article = // once you have an article
```

```
[article downloadContentWithOptions:
  (SCSArticleDownloadOptionRefetchArticleContent|SCSArticleDownloadOptionImages)
  completion:^(NSError * _Nullable error) {

  // TO DO: Handle completion

}]
```

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

})
```

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 two ways to present knowledge content:

1. Use the default presentation. If you use the `setInterfaceVisible:animated:completion:`, the SDK automatically displays the content, and it uses cached content before trying to get content online.

2. You can manually display an article by instantiating an `SCSArticleViewController`, specifying the article using the `article` property, and then presenting the view yourself. This method is useful if you want to display specific articles within your app. When using this method, you can use the `SCSArticleViewControllerDelegate` class to listen for events.

    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
```

```objc
  }
  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];
      }
    }];
  }
}];
```

In Swift:

```swift
// Get Knowledge Manager instance
let knowledgeManager = SCSKnowledgeManager.default()

// Create query for a specific article
let query = SCSMutableArticleQuery()
query.articleId = "TO_DO:QUERY_ID"

// Fetch article
knowledgeManager.fetchArticles(with: query, completion: {
    (articles: [SCSArticle], error: Error?) in

  if (error != nil) {
    // TO DO: Handle error
  }
  else if (articles.count == 0) {
    // TO DO: Handle no results
  }
  else {
    let article:SCSArticle = 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)
      }
    })
  }
})
```

👁 Example: This code shows an example of how to download the top three articles in each category.

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();
}
```

In Swift:

```
let knowledgeManager = SCSKnowledgeManager.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 = SCSMutableArticleQuery()

      // ... 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: [SCSArticle], 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: [SCSCategoryGroup]?, error: Error?) in

    // TO DO: Check for error

    // And then download the articles
    articleDownloadBlock();
  })
}
else {
  // Download the articles
  articleDownloadBlock();
}
```

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 `SCSServiceCloudDelegate`.

To disable the Case Management button, follow the instructions in Customize Action Buttons. Use the `SCSActionCasePublisher` named constant value when determining which button to disable.

Using Case Management

Adding the Case Management experience to your app.

Case Management Overview

Learn about the Case Publisher and Case Management experience using the SDK.

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:
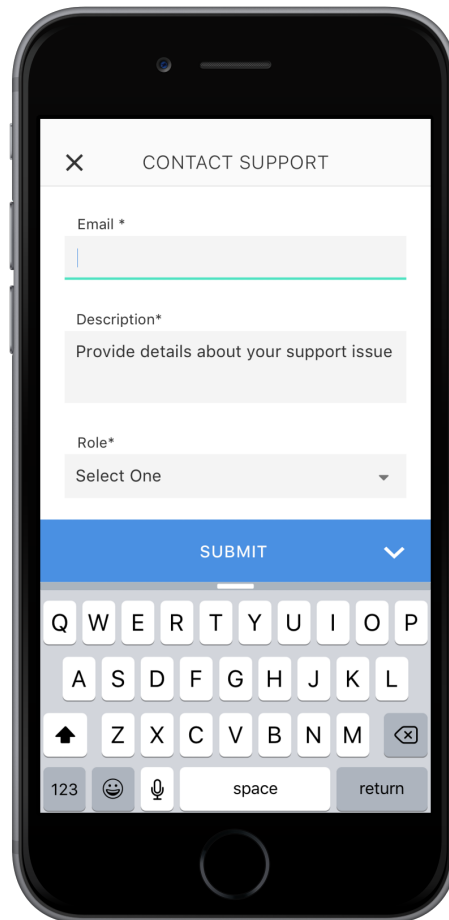
If you authenticate the user, they can also view and manage their list of cases. In the default 'guest user' flow, launching the interface causes the **Case Publisher** screen to appear. From this screen, a user can create a case as an anonymous guest user. In the default 'authenticated user' flow, launching 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).
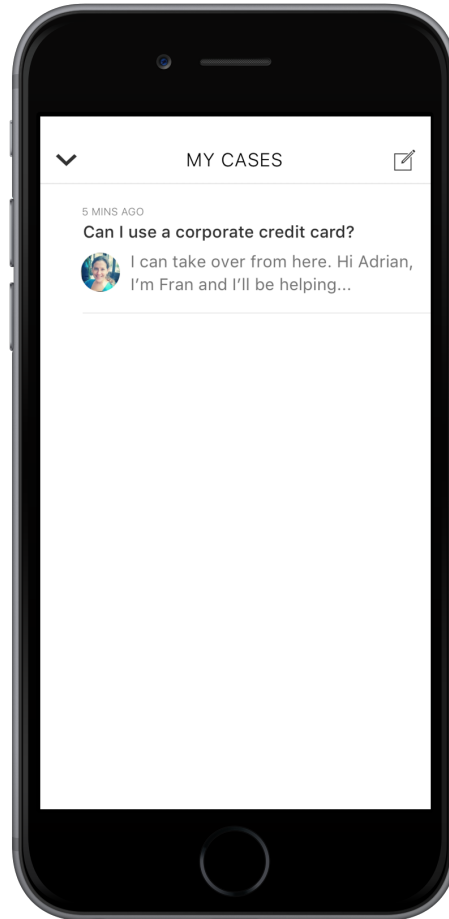
If you'd prefer, you can manually control the Case Management view controllers.

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.

Let's get started.

Quick Setup: Case Publisher as a Guest User

To set up case publisher, 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 SDK.

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

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 Objective-C:

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

   In Swift:

   ```
   import ServiceCore
   import ServiceCases
   ```

2. Point the SDK to your org from the `applicationDidFinishLaunchingWithOptions` method in your `UIApplicationDelegate` class.

   To connect your application to your organization, create an `SCSServiceConfiguration` object containing the community URL. Pass this object to the `SCServiceCloud` shared instance using the `initWithCommunity:` method.

   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;
   ```

   In Swift:

   ```
   // Create configuration object with your community URL
   let config = SCSServiceConfiguration(
               community: URL(string: "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 the `initWithCommunity:dataCategoryGroup:rootDataCategory:` method instead. This method sets up data categories in addition to setting the community URL. See Quick Setup: Knowledge 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 Objective-C:

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

   In Swift:

   ```
   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:forName:`, `setFontDescriptor:fontFileName:forWeight:`, and `setImage:compatibleWithTraitCollection:forName:`. You can also configure the strings used throughout the interface. See SDK Customizations for more info.

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:animated:completion:`.

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 Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().cases.setInterfaceVisible(true,
                                        animated: true,
                                        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 for Case Management for more info.

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

> **Example:** To use this example code, create a Single View Application and Install the SDK.
>
> Set up the Case Management interface within the `AppDelegate` implementation.
>
> In Objective-C:

```
#import "AppDelegate.h"
@import ServiceCore;
@import ServiceCases;
```

```objc
@implementation AppDelegate


- (BOOL)application:(UIApplication *)application
                didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

  // Point the SDK to your community
  SCSServiceConfiguration *config = [[SCSServiceConfiguration alloc]
    initWithCommunity:[NSURL URLWithString:@"https://mycommunity.example.com"]];
  [SCServiceCloud sharedInstance].serviceConfiguration = config;

  // Assign global action to use for case layout
  [SCServiceCloud sharedInstance].cases.caseCreateActionName = @"NewCase";

  return YES;
}

@end
```

In Swift:

```swift
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")!)
    SCServiceCloud.sharedInstance().serviceConfiguration = config

    // Assign global action to use for case layout
    SCServiceCloud.sharedInstance().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.

In Objective-C:

```objc
#import "ViewController.h"
@import ServiceCore;
@import ServiceCases;

@implementation ViewController
```

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

  // Show the Case Management interface
  [[SCServiceCloud sharedInstance].cases setInterfaceVisible:YES
                                                    animated:YES
                                                  completion:nil];
}

@end
```

In Swift:

```
import UIKit
import ServiceCore
import ServiceKnowledge

class ViewController: UIViewController {

  @IBAction func showHelp(_ sender: AnyObject) {

    SCServiceCloud.sharedInstance().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 both create and manage cases from your app.

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. 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.

2. If you're using a Salesforce community and authenticating with the Salesforce Mobile SDK, configure the login endpoint as described in the Salesforce Mobile SDK documentation: Configure the Login Endpoint.

   The documentation describes how to use the `SFDCOAuthLoginHost` property in your `info.plist` file to create a custom login URI.

3. Write the code to authenticate a user with your org.

   You can authenticate in several ways.

   • **Create normal user accounts on your Salesforce org.** If you have (or want to create) Salesforce user accounts for each user of your app, see Digging Deeper into OAuth 2.0 on Force.com, which describes how to set up a connected app on Salesforce. For developers already using the Salesforce Mobile SDK, the Mobile SDK Developer's Guide contains instructions for authenticating with that SDK.

- • **Use your own user credentials to create community portal users on Salesforce.** You can authenticate using existing credentials on your server. One approach is to create an API endpoint on your server that is accessible to your app. For each user, your server can formulate unique information representing that user. Use this information in a JSON Web Token (JWT) payload. Sign and submit this information to the OAuth endpoint on your Salesforce org. Using Apex code configured within Salesforce, you can use the JWT information to identify a previously created community user or implicitly create a new user. That user's OAuth keys can be created and supplied back to your server.

  - – Documentation about using JWT for Salesforce authorization: OAuth 2.0 JWT Bearer Token Flow.
  - – Documentation about creating a community portal user with an Apex trigger: Apex Developer's Guide: Force.com Sites Examples.
  - – More documentation about creating a community portal user: Authenticating Users on Force.com Sites.

  The Service SDK does not contain any additional method to authenticate.

4. When authenticated, hold onto an `SFUserAccount` object with your credentials.

   Whichever method you use to authenticate, you need an `SFUserAccount` object to pass to the Service SDK. This class is part of the `SalesforceKit` framework from the Salesforce Mobile SDK. The account object must have correct information in its `credentials` property. If you're using the Mobile SDK to authenticate, the `SFOAuthCoordinator` instance you're given in your delegate handler contains a credentials property that you can use.

5. Implement `SCServiceCloudDelegate` and pass account information from within the `serviceCloud:shouldAuthenticateService:completion:` method. The SDK calls this method whenever it plans to display a view that can support authenticated users.

   The `SCServiceCloud` shared instance has a delegate you can implement.

   In Objective-C:

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

   In Swift:

   ```
   SCServiceCloud.sharedInstance().delegate = self
   ```

   If you want an authenticated connection, implement the `serviceCloud:shouldAuthenticateService:completion:` method from this delegate and return `YES`/`true`. This method is a good place to pass in the user account with the supplied completion block, which runs asynchronously.

   In Objective-C:

   ```
   - (BOOL)serviceCloud:(SCServiceCloud *)serviceCloud
           shouldAuthenticateService:(NSString *)service
                        completion:(void (^)(SFUserAccount * _Nullable))completion {

     SFUserAccount* user = // TO DO: Get user account

     // After acquiring user information, call
     // this completion block to set the new user:
     completion(user);

     return YES;
   }
   ```

In Swift:

```
func serviceCloud(serviceCloud: SCServiceCloud,
     shouldAuthenticateService service: String,
                          completion: (SFUserAccount?) -> Void) -> Bool {

  let user: SFUserAccount = // TO DO: Get user account

  // After acquiring user information, call
  // this completion block to set the new user:
  completion(user)

  return true
}
```

6. For error handling, implement `serviceCloud:serviceAuthenticationFailedWithError:` from
`SCServiceCloudDelegate`.

In Objective-C:

```
- (void)serviceCloud:(SCServiceCloud*)serviceCloud
       serviceAuthenticationFailedWithError:(NSError*)error
       forService:(NSString*)service {

  // TO DO: Inspect error and handle appropriately.
}
```

In Swift:

```
func serviceCloud(serviceCloud: SCServiceCloud,
     serviceAuthenticationFailedWithError error: Error,
     forService service: String) {

  // TO DO: Inspect error and handle appropriately.
}
```

The `service` param is `SCServiceTypeKnowledge` for Knowledge and `SCServiceTypeCases` for Case Management.

The `NSError` object contains information about the error. The `code` property on this object contains the error code. The following errors are the most common error codes you can encounter:

Error Code	Description
`SCServiceUserSessionExpiredOrInvalidError` (401)	Occurs when the session has expired or the ID is invalid. Use the refresh token to acquire another access token and then update the `account` property, as shown in the next step.
`SCServiceUserRequestRefusedError` (403)	Occurs when the user does not have sufficient access for the request. Verify that the logged in user has sufficient credentials.
`SCServiceUserResourceNotFoundError` (404)	Occurs when the requested resource was not found. Check the URI and other parameters for errors.

See Status Codes and Error Responses for a full set of possible error codes.

7. 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 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 Objective-C:

   ```
   [SCServiceCloud sharedInstance].cases.caseListName = @"AllOpenCases";
   ```

   In Swift:

   ```
   SCServiceCloud.sharedInstance().cases.caseListName = "AllOpenCases"
   ```

8. (Optional) Incorporate notifications whenever there is a new text post on an existing case.

   Using Salesforce's push notification implementation guide, you can send 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.

9. When account information changes, update the `account` property.

   Account information can change if the access token expires, the user logs out, another user logs in, and for various other reasons. You can update the `account` property on the `SCServiceCloud` shared instance. The Service SDK updates the interface to correspond with the updated account information. Preferably, instead of setting the property, use the `setAccount:completion:` method, which allows you to specify a completion block for error handling.

   In Objective-C:

   ```
   SFUserAccount *user = // Get user account
   [[SCServiceCloud sharedInstance]
                   setAccount:user completion:^(NSError *error) {

     // TO DO: Handle error
   }];
   ```

   In Swift:

   ```
   let user = // Get user account

   SCServiceCloud.sharedInstance()
                 .setAccount(user, completion: { (error: Error?) in

     // TO DO: Handle error
   })
   ```

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 for Case Management

The simplest way to show and hide the Case Management interface is by calling the `setInterfaceVisible` method on the `SCServiceCloud.cases` property. 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:animated:completion:` method to show the Case Management interface using the default presentation.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().cases.setInterfaceVisible(true,
                                        animated: true,
                                        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 `SCServiceCloud`.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().delegate = self
```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud:willDisplayViewController:animated:` method to find out if it's a view controller you're interested in, and if so, assign the view controller delegate.

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;
   }
}
```

In Swift:

```
func serviceCloud(_ serviceCloud: SCServiceCloud,
                  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
  }

}
```

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. Just implement a
`UIViewControllerTransitioningDelegate`.

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

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().delegate = mySCServiceCloudDelegate
```

2. Implement the serviceCloud:transitioningDelegateForViewController: method in your delegate and
return a custom UIViewControllerTransitioningDelegate from this method.

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;
}
```

In Swift:

```
func serviceCloud(_ serviceCloud: SCServiceCloud,
                    transitioningDelegateFor controller: UIViewController)
                    -> UIViewControllerTransitioningDelegate? {

  // TO DO: Put your logic here and then return your transitioning delegate...

  return myTransitioningDelegate
}
```

Showing the View Controllers Manually

Instead of having the SDK manage the case management flow, 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).

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!

For a use case that involves custom view controllers, see Send Custom Data Using Hidden Fields.

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 `showInterfaceForNotification:`). See Handle Remote Notifications.

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 Objective-C:

```objectivec
- (NSSet *)casePublisher:(SCSCasePublisherViewController *)publisher
         fieldsToHideFromCaseFields:(NSArray *)availableFields {

  NSSet *hideFieldsSet = [NSSet setWithObjects: @"MyHiddenField",
                                                @"MyOtherHiddenField", nil];
  return hideFieldsSet;
}
```

In Swift:

```swift
func casePublisher(_ publisher: SCSCasePublisherViewController,
                   fieldsToHideFromCaseFields availableFields: [String])
                   -> Set<String> {

  let hideFieldSet: Set = ["MyHiddenField", "MyOtherHiddenField"]
  return hideFieldSet
}
```

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 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;
}
```

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
}
```

> 📝 **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 Objective-C:

```
- (NSSet *)caseDetail:(SCSCaseDetailViewController *)caseDetailController
         fieldsToHideFromCaseFields:(NSArray *)availableFields {

  NSSet *hideFieldsSet = [NSSet setWithObjects:@"MyHiddenField",
                                               @"MyOtherHiddenField", nil];
  return hideFieldsSet;
}
```

In Swift:

```
func caseDetail(_ caseDetailController: SCSCaseDetailViewController,
                fieldsToHideFromCaseFields availableFields: [String])
                -> Set<String> {
  let hideFieldSet: Set = ["MyHiddenField", "MyOtherHiddenField"]
  return hideFieldSet
}
```

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 `SCServiceCloud`.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().delegate = self
```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud:willDisplayViewController:animated:` method to find out if it's a view controller you're interested in, and if so, assign the view controller 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 = self;

  } else if ([controller isKindOfClass:[SCSCaseDetailViewController class]]) {
    SCSCaseDetailViewController *caseDetailController =
      (SCSCaseDetailViewController *)controller;
    caseDetailController.delegate = self;
  }
}
```

In Swift:

```
func serviceCloud(_ serviceCloud: SCServiceCloud,
                  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
  }
}
```

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 `SCSCaseListViewController` implementation.

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];
}
```

In Swift:

```
func caseList(_ caseList: SCSCaseListViewController,
              selectedCaseWithId caseId: String) {

  let controller = SCSCaseDetailViewController(caseId: caseId)
  controller.delegate = self
  caseList.navigationController!.pushViewController(controller, animated: true)
}
```

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.
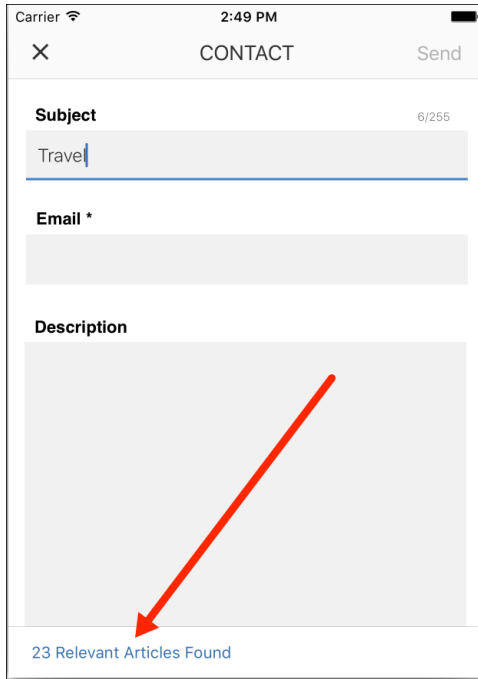
By default, this feature is enabled and it searches the Subject and Description fields for relevant articles using those search terms. You can change this behavior using methods in the `SCSCasePublisherViewControllerDelegate` class. This is the delegate class for `SCSCasePublisherViewController`. You 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:animated:completion:` method), refer to the **Activating Interface Using the Default Presentation** section of Customize the Presentation for Case Management.

2. If you are displaying the Case Publisher by manually presenting the view controller, refer to the **Showing the View Controllers Manually** section of Customize the Presentation for Case Management.

You can turn case deflection on and off with the `shouldEnableCaseDeflectionForPublisher:` method. You can control which fields are used for searching knowledge base articles using the `casePublisher:fieldsForCaseDeflection:` method.

In Objective-C:

```objc
- (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];
```

```objc
    for (NSString *field in deflectionSearch) {
        if ([availableFields containsObject:field]) {
            [deflectionSearchFieldsInLayout addObject:field];
        }
    }

    // Return the list of fields
    return deflectionSearchFieldsInLayout;
}
```

In Swift:

```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)
}
```

📝 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.

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:
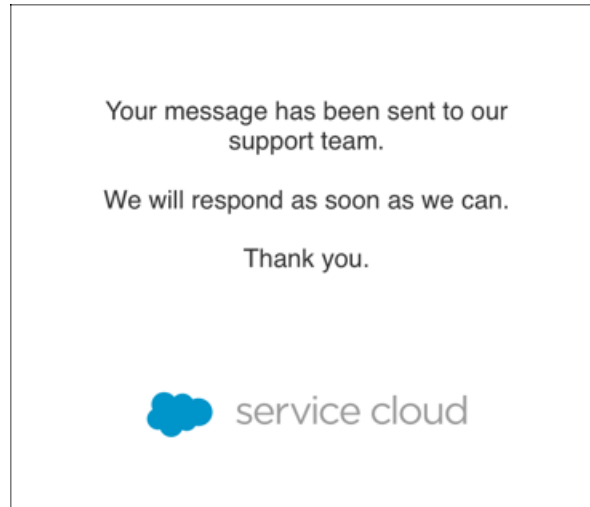
To customize the standard message text (`ServiceCloud.CasePublisher.SuccessMessage`) and the image (`CaseSubmitSuccess`), see Customize and Localize Strings and Customize Images for instructions.

To create your own view:

1. Implement the `SCSCasePublisherViewControllerDelegate` class.

Implement the `casePublisher:viewForResult:withCaseId:error:` method where you create your custom view and return it to the Service SDK. If you don't implement this method (or if you return `nil`), the SDK presents the default view.

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;
}
```

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
}
```

> 📝 **Note:** The SDK only calls the `casePublisher:viewForResult: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 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 `SCServiceCloud`.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().delegate = self
```

In your `SCServiceCloudDelegate` implementation, use the `serviceCloud:willDisplayViewController:animated:` method to find the `SCSCasePublisherViewController` class and register your view controller 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

  }
}
```

In Swift:

```
func serviceCloud(_ serviceCloud: SCServiceCloud,
                  willDisplay controller: UIViewController,
                  animated: Bool) {

  if controller is SCSCasePublisherViewController {

    let publisherController = controller as! SCSCasePublisherViewController

    publisherController.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

Using Salesforce's push notification implementation guide, you can send 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.

The Salesforce Mobile Push Notifications Implementation Guide has instructions on how to set up push notifications in your app. Basically, you create a connected app in your org, create an Apex trigger, and then package and install the components in your org.

To set up a trigger, you can use the following sample Apex code as a starting point. This code triggers a notification when an agent creates a text post.

```apex
// 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) {

    Schema.SObjectType objectType = feedItem.parentId.getSObjectType();

    // 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);

     // Create the payload and add it to the notification
     Messaging.PushNotification msg = new Messaging.PushNotification();
     Map<String, Object> payload =
       Messaging.PushNotificationPayload.apple(alertText, '', null, freeFormData);
     msg.setPayload(payload);

     // Needs to match your connected app name
     msg.send('YourConnectedAppName', users);
   }
 }
}
```

When this notification is sent to your app, you can have the SDK to 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. To make this happen, see Handle Remote Notifications.

📝 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.

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 Live Agent Chat

Adding the Live Agent Chat experience to your app.

Live Agent Chat Overview

Learn about the Live Agent Chat experience using the SDK.

Quick Setup: Live Agent Chat

To add Live Agent Chat to your app, create an `SCSChatConfiguration` object and pass it to the `startSessionWithConfiguration:` method.

Configure a Live Agent Chat Session

Before starting a Live Agent Chat session, you have several ways to optionally configure the session using the `SCSChatConfiguration` object. These configuration settings allow you to specify pre-chat fields and determine whether a session starts minimized or full screen.

Live Agent Chat Events and Errors

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

Send Custom Data to Live Agent

Send custom data from your app to the agent so that your agent has all the needed information.

Check Live Agent Availability

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

Transfer File to Agent

With Live Agent Chat, a user can transfer a file to an agent after the agent requests a file transfer.

Live Agent Chat Overview

Learn about the Live Agent Chat experience using the SDK.

With Live Agent Chat, you can provide real-time chat sessions from within your native app. Once you've set up Live Agent Chat for Service Cloud, it takes just a few calls to the SDK to have your app ready to handle agent chat sessions.
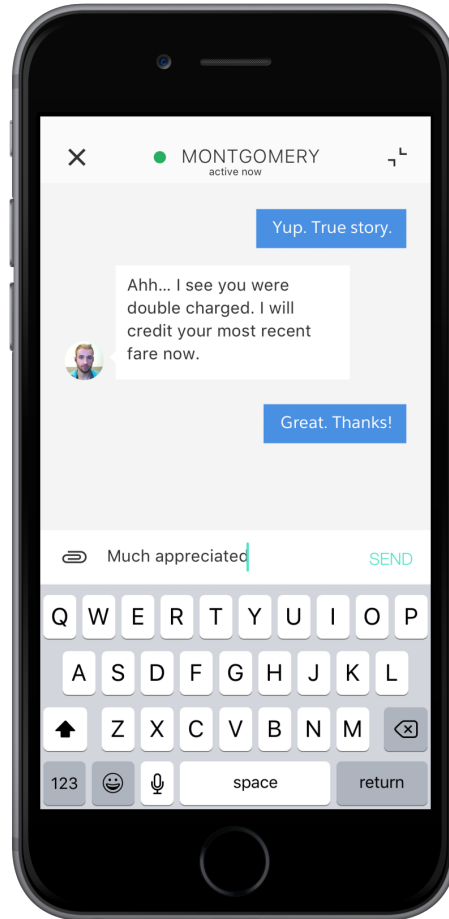
This chat session can be minimized so that the user can continue to navigate from within the app while speaking with an agent.

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: Live Agent Chat

To add Live Agent Chat to your app, create an `SCSChatConfiguration` object and pass it to the `startSessionWithConfiguration:` method.

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

- Set up Service Cloud to work with Live Agent Chat. To learn more, see Console Setup for Live Agent Chat.
- Installed the SDK. To learn more, see Install the SDK.

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

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

    In Objective-C:

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

In Swift:

```
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 Objective-C:

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

In Swift:

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

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

3. (Optional) Determine whether you want Live Agent Chat to appear as a minimized view or as a full screen view.

By default, Live Agent Chat launches as a minimized view that does not take up the entire screen. This variant is advantageous when you want your user to be able to interact with the app during a chat session. Alternatively, you may want a chat session to be a full screen view that takes over their display.

See Configure a Live Agent Chat Session for more information.

4. (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.

See Configure a Live Agent Chat Session for more information.

5. (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:forName:`, `setFontDescriptor:fontFileName:forWeight:`, and `setImage:compatibleWithTraitCollection:forName:`. You can also configure the strings used throughout the interface. See SDK Customizations for more info.

6. To start a Live Agent Chat session, call the `startSessionWithConfiguration:` method on `SCSChat`.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().chat.startSession(with: 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 SDK from a Web View.

7. Listen for events and handle error conditions.

You can detect when a session ends by implementing the `chat:didEndWithReason:error:` method on the `SCSChatDelegate` delegate. Register this delegate using the `addDelegate:` method on your `SCSChat` instance. In particular, we suggest that you handle the `SCSChatEndReasonAgent` reason (for when an agent ends a session) as well as the `SCSChatNoAgentsAvailableError` error code (for when there are no agents available). See Live Agent Chat 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.

> 👁 Example: To use this example code, create a Single View Application and Install the SDK.
>
> 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 `SCSChatDelegate` 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 `chat:didEndWithReason:error:` method and show a dialog when appropriate.
>
> In Objective-C:

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

@interface ViewController : UIViewController <SCSChatDelegate>

@end

@implementation ViewController

- (void)viewDidLoad {
  [super viewDidLoad];

  // Add our chat delegate
  [[SCServiceCloud sharedInstance].chat 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
```

```objc
  [[SCServiceCloud sharedInstance].chat startSessionWithConfiguration:config];
}

- (void)chat:(SCSChat *)chat didEndWithReason:(SCSChatEndReason)reason
                                        error:(NSError *)error {

  NSString *description = nil;

  // Here we'll handle the situation where the agent ends the session
  // and when there are no agents available...
  if (reason == SCSChatEndReasonAgent) {
      description = @"The agent has ended the session.";
  } else if (reason == SCSChatEndReasonSessionError &&
             error.code == SCSChatNoAgentsAvailableError) {
      description = @"It looks like there are no agents available. Try again later.";

  }

  if (description != nil) {
    UIAlertController *alert = [UIAlertController
                                alertControllerWithTitle:@"Session Ended"
                                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
```

In Swift:

```swift
import UIKit
import ServiceCore
import ServiceChat

class ViewController : UIViewController, SCSChatDelegate {

  override func viewDidLoad() {
    super.viewDidLoad()

    // Add our chat delegate
    SCServiceCloud.sharedInstance().chat.add(self)
  }
```

```swift
@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
  SCSServiceCloud.sharedInstance().chat.startSession(with: config)

}

func chat(_ chat: SCSChat!, didEndWith reason: SCSChatEndReason,
          error: Error!) {

  var description = ""

  // Here we'll handle the situation where the agent ends the session
  // and when there are no agents available...
  if (reason == .agent) {
      description = "The agent has ended the session."
  } else if (reason == .sessionError &&
    (error as NSError).code == SCSChatErrorCode.noAgentsAvailableError.rawValue) {
      description = "It looks like there are no agents available. Try again later."

  }

  if (description != "") {
      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)
  }
 }
}
```

Configure a Live Agent Chat Session

Before starting a Live Agent Chat session, you have several ways to optionally configure the session using the `SCSChatConfiguration` object. These configuration settings allow you to specify pre-chat fields and determine whether a session starts minimized or full screen.

When you start a Live Agent Chat session, you specify an `SCSChatConfiguration` object as one of the arguments. This object contains all the configuration settings necessary for Live Agent Chat to start a session. To create an `SCSChatConfiguration` object, you specify information about your org and deployment (as described in Quick Setup: Live Agent Chat), and that is all that is required. However, there are other options you can set using `SCSChatConfiguration`.

🗹 **Note:** Be sure not to start a Live Agent Chat session until you've fully configured the `SCSChatConfiguration` object.

The following features are available for configuration:

Method Name	Description	Type : Default Value
presentationStyle	By default, Live Agent Chat launches as a minimized view that does not take up the entire screen. This variant is advantageous when you want your user to be able to interact with the app during a chat session. Alternatively, you may want a chat session to be a full screen view that takes over their display. To configure the presentation style, call the `presentationStyle` method on the `SCSChatConfiguration` object. Specify `SCSChatPresentationStyleNonBlocking` for the minimized view; specify `SCSChatPresentationStyleFullscreen` to initialize in the full-screen chat experience. The user can always switch between these states during a chat session — when in full-screen mode, the user taps the minimize button to minimize the session; when in minimized mode, the user taps the chat thumbnail to go into full-screen mode	`SCSChatPresentationStyle` enum : `SCSChatPresentationStyleNonBlocking`
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. Specify each field as an item in an array of `SCSPrechatObject` objects and pass the array to the `prechatFields` property on the `SCSChatConfiguration` object. There are three types of pre-chat fields: • `SCSPrechatObject` does not require user input. • `SCSPrechatTextInputObject` (a subclass of `SCSPrechatObject`) can take user input. • `SCSPrechatPickerObject` (a subclass of `SCSPrechatObject`) provides the user with a dropdown list of options. You initialize a pre-chat object with a string used for the pre-chat label. When using a `SCSPrechatTextInputObject`, you can control several other characteristics: • `required`— to specify whether the field is required. • `keyboardType`— to use other standard keyboards (such as `UIKeyboardTypeEmailAddress`). • `autocapitalizationType`— to control how text capitalization works. • `autocorrectionType`— to control auto-correction. • `maxLength`— to specify the maximum length of the field.	`SCSPrechatPickerObject` array : `nil`

Method Name	Description	Type : Default Value
	When using a `SCSPrechatPickerObject`, you can access two properties: • `required`— to specify whether the field is required. • `options` — which is an array of `SCSPrechatPickerOption` objects. Each `SCSPrechatPickerOption` object is one item in the dropdown list.	

👁 **Example:** The following example starts a session in full-screen mode with a few pre-chat fields: one required text field for email, one optional text field, one optional dropdown field, and one hidden custom field.

In Objective-C:

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


// Set full screen mode
config.presentationStyle = SCSChatPresentationStyleFullscreen;

// 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];

// Add an optional name field (with no auto-correction and word capitalization)
SCSPrechatTextInputObject* nameField = [[SCSPrechatTextInputObject alloc]
                                        initWithLabel:@"Name"];
nameField.required = NO;
nameField.autocorrectionType = UITextAutocorrectionTypeNo;
nameField.autocapitalizationType = UITextAutocapitalizationTypeWords;
[config.prechatFields addObject:nameField];

// Add a dropdown field for department name
NSMutableArray<SCSPrechatPickerOption *> *pickerOptions = [NSMutableArray new];
[pickerOptions addObject:[[SCSPrechatPickerOption alloc] initWithLabel:@"Sales"
                                                                value:@"sales"]];
[pickerOptions addObject:[[SCSPrechatPickerOption alloc] initWithLabel:@"Billing"
                                                                value:@"billing"]];
SCSPrechatPickerObject* pickerObject = [[SCSPrechatPickerObject alloc]
initWithLabel:@"Department"

options:pickerOptions]];
pickerObject.required = NO;
```

```objectivec
[config.prechatFields addObject:pickerObject];

// Add custom data directly to agent
SCSPrechatObject* customData = [[SCSPrechatObject alloc]
                                  initWithLabel:@"DataForAgent"
                                          value:@"CUSTOM-DATA-HERE"];
[config.prechatFields addObject:customData];

// Start session!
[[SCSServiceCloud sharedInstance].chat startSessionWithConfiguration:config];
```

In Swift:

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

// Set full screen mode
config?.presentationStyle = .fullscreen

// 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)

// Add an optional name field (with no auto-correction and word capitalization)
let nameField = SCSPrechatTextInputObject(label: "Name")
nameField?.isRequired = false
nameField?.autocorrectionType = .no
nameField?.autocapitalizationType = .words
config?.prechatFields.add(nameField)

// Add a dropdown field for department name
let pickerOptions = NSMutableArray()
pickerOptions.add(SCSPrechatPickerOption(label:"Sales", value:"sales"))
pickerOptions.add(SCSPrechatPickerOption(label:"Billing", value:"billing"))
let pickerObject = SCSPrechatPickerObject(label: "Department",
                                          options: pickerOptions as NSArray as!
[SCSPrechatPickerOption])
pickerObject?.isRequired = false
config?.prechatFields.add(pickerObject)

// Add custom data directly to agent
let customData = SCSPrechatObject(label: "DataForAgent", value: "CUSTOM-DATA-HERE")
config?.prechatFields.add(customData)

// Start session!
SCSServiceCloud.sharedInstance().chat.startSession(with: config)
```

Live Agent Chat Events and Errors

Implement `SCSChatDelegate` to be notified about state changes made before, during, and after a Live Agent 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 Live Agent Chat session can be in one of the following states:

Inactive

No active session; no incoming or outgoing traffic.

Loading

Session is loading.

Prechat

The user is currently entering pre-chat details.

Connecting

A connection with Live Agent 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.

Ended

Session has ended and will proceed to the inactive state.

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 `SCSChatDelegate`. Use the `addDelegate:` method on `SCSChat` to register your delegate. Use the `chat:stateDidChange:previous:` 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 `SCSChatDelegate` methods for this purpose:

1. To track session termination, use the `chat:didEndWithReason:error:` method. Inspect the reason (`SCSChatEndReason`) to determine why the session stopped. Typically, the session stops due to a normal event (for example, `SCSChatEndReasonUser`). If the reason is `SCSChatEndReasonSessionError`, check the `error` parameter for more detail and compare the error code to `SCSChatErrorCode` values. For instance, when there are no agents available to take a call, the error is `SCSChatNoAgentsAvailableError`.

2. You can track all Live Agent Chat errors (including the session errors that are passed to `chat:didEndWithReason:error:`) with the `chat:didError:` method. Compare the error code to `SCSChatErrorCode` to determine what kind of error occurred.

👁 Example: **Basic `SCSChatDelegate` Example**

This sample code does the following:

- Implements the `SCSChatDelegate` protocol.

- Implements the `chat:stateDidChange:previous:` method to listen for state changes.
- Implements the `chat:didEndWithReason:error:` method and includes some error handling logic.
- Implements the `chat:didError:` method to listen for errors.

```swift
import UIKit
import ServiceCore
import ServiceChat

class MyChatDelegateImplementation: NSObject, SCSChatDelegate {

  // TO DO: Register this delegate using SCSServiceCloud.sharedInstance().chat.add(self)


  // Delegate method for state change.
  func chat(_ chat: SCSChat!, stateDidChange current: SCSChatSessionState,
            previous: SCSChatSessionState) {

    NSLog("Chat state changed...")

    if (current == .connecting) {
      NSLog("Chat now connecting...")
    }
  }

  // Delegate method for session stop event.
  // You can also check for fatal errors with this delegate method.
  func chat(_ chat: SCSChat!, didEndWith reason: SCSChatEndReason,
            error: Error!) {

    var title = ""
    var description = ""

    // If there's an error...
    if (error != nil) {

      switch (error as NSError).code {

      // No agents available
      case SCSChatErrorCode.noAgentsAvailableError.rawValue:
        title = "Session Failed"
        description = "It looks like there are no agents available. Try again later."


      // Communication error
      case SCSChatErrorCode.communicationError.rawValue:
        title = "Session Failed"
        description = "Communication error. Check network and try again."

      // TO DO: Use SCSChatErrorCode to check for 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 .agent:
          title = "Session Ended"
          description = "The agent has ended the session."

        // TO DO: Use SCSChatEndReason 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("\nChat End Session. Title: %@\nDescription: %@",title, description)
      }
    }

  // Delegate method for error conditions.
  func chat(_ chat: SCSChat!, didError error: Error!) {
    NSLog("Chat error (%d): '%@'", (error as NSError).code, error.localizedDescription)
}
```

Send Custom Data to Live Agent

Send custom data from your app to the agent so that your agent has all the needed information.

In some situations, you want to send app-specific or user-specific information directly to an agent without requiring user input. By automatically sending known information, your agents can help customers more effectively without having to ask for the applicable data.

1. In your app, add a pre-chat object with the custom data.

   This code creates a SCSPrechatObject with a label name of CustomerEmailField and a value of lauren@example.com.

   In Swift:

```
let customData = SCSPrechatObject(label: "CustomerEmailField", value:
"lauren@example.com")
config.prechatFields.add(customData)
```

   In Objective-C:

```
SCSPrechatObject* customData = [[SCSPrechatObject alloc]
                                 initWithLabel:@"CustomerEmailField"
                                      value:@"lauren@example.com"];
[config.prechatFields addObject:customData];
```

For more information, see Configure a Live Agent Chat Session.

2. From your Live Agent page layout in your org, add a Visualforce page to read the custom data.

This JavaScript code is called when a Live Agent session starts (`sforce.console.chat.onChatStarted`). After a few checks, it searches for the custom data label. From there, you can extract the value for that label and pass the information to the agent.

```javascript
document.addEventListener('DOMContentLoaded', function () {

  // Called when a new chat session starts
  sforce.console.chat.onChatStarted(function(result) {
    console.log('onChatStarted',result);

    // Returns the details of the chat associated with the chat key
    sforce.console.chat.getDetailsByChatKey(result.chatKey, function(record) {
      console.log('getDetailsByChatKey' , record);
      if (!record || !record.success) {
        console.log('Can not determine session Id');
        return;
      } else {
        if (record.details.customDetails && record.details.customDetails.length > 0) {
          var customDetail = null;

          for (var i = record.details.customDetails.length - 1; i >= 0; i--) {
            customDetail = record.details.customDetails[i];

            // Is it the record we want?
            if (customDetail.label == "CustomerEmailField") {
              // TO DO: Do something with your custom data here...
              // TO DO: Use this value: customDetail.value
            }
          }
        }
      }
    })
  })
})
```

To learn more about writing scripts in your Visualforce page, see Salesforce Console Integration Toolkit Developer Guide.

Check Live Agent Availability

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

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

In Objective-C:

```objectivec
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
  }

});
```

In Swift:

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

SCServiceCloud.sharedInstance().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
  }

})
```

Transfer File to Agent

With Live Agent Chat, a user can transfer a file to an agent after the agent requests a file transfer.

In addition to text-based conversations, an agent can request that the user transfer a file by clicking the **Attach File** button from the Service Cloud Console.
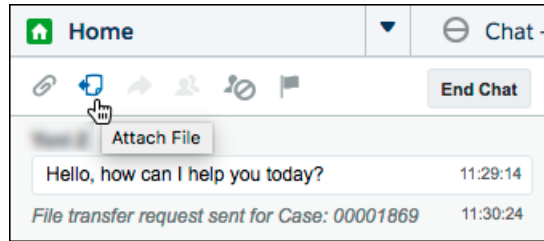
The user sees a **FILE TRANSFER REQUESTED** message within the app, and can then send a file using the paperclip button.
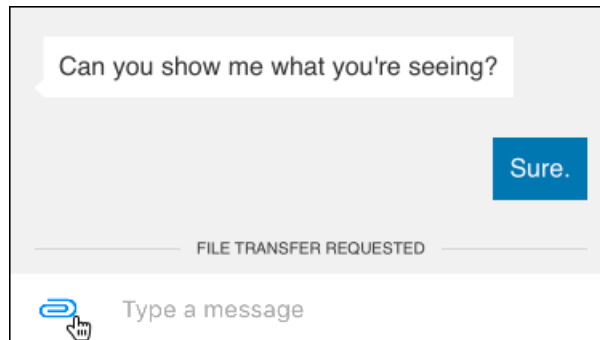
See Transfer Files During a Chat in Salesforce Help for details about setting up this functionality in Service Cloud. No coding is necessary in your app to make this behavior work.

> 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 SDK.

Using SOS

Adding the SOS experience to your app.

SOS Overview
Learn about the SOS experience using the SDK.

SOS Example App
The SOS example app demonstrates many of the SOS features and can help you get up to speed.

Quick Setup: SOS
Use `SOSSessionManager` to start an SOS session.

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.

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 Overview

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.
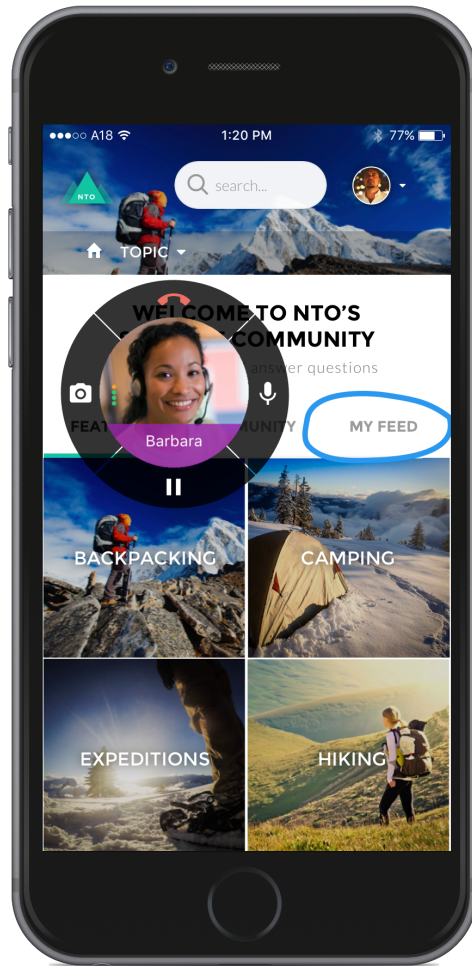
And with just a few more configuration changes, you can provide two-way video support from your app. This functionality can include front-facing camera support, back-facing camera support, or both.

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.

SOS Example App

The SOS example app demonstrates many of the SOS features and can help you get up to speed.

Before using the example app, you must:

- Set up Service Cloud to work with SOS. To learn more, see Console Setup for SOS.
- Install CocoaPods, which is needed for the project's dependencies.

Once you've reviewed these prerequisites, you're ready to download and use the SOS example app.

1. Clone the git repository where the example project is located: https://github.com/goinstant/ios-sdk-guides.

```
git clone https://github.com/goinstant/ios-sdk-guides
```

2. Change directories into the example folder.

```
cd ios-sdk-guides
```

3. Run the CocoaPods installer.

```
pod install
```

This command installs all the dependencies required for this project.

4. Launch the project workspace in Xcode.

```
open SOSExamples.xcworkspace
```

When using CocoaPods, you must be sure to the launch the workspace file (`.xcworkspace`) and NOT the project file (`.xcodeproj`). `SOSExamples.xcodeproj` will not work on its own.

5. Open the `SOSSettings.plist` file and update it with your org's setup values (Salesforce Organization ID, Deployment ID, Live Agent Pod). If the settings are not valid, an error occurs when you start an SOS session.

To learn more about setting up your org, see Console Setup for SOS.

6. Build and run the `SOSExamples` project!

This sample application shows how you can approach integrating SOS into your own app. This app is almost entirely wired together using the storyboard.

If you run into network issues while connecting with an agent, see SOS Network Troubleshooting Guide.

Quick Setup: SOS

Use `SOSSessionManager` to start an SOS session.

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 SDK.

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 Objective-C:

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

In Swift:

```
import ServiceCore
import ServiceSOS
```

2. Create an `SOSOptions` object with information about your LiveAgent pod, your Salesforce org ID, and the deployment ID.

In Objective-C:

```
SOSOptions *options = [SOSOptions optionsWithLiveAgentPod:@"YOUR-POD-NAME"
                                                    orgId:@"YOUR-ORG-ID"
                                             deploymentId:@"YOUR-DEPLOYMENT-ID"];
```

In Swift:

```
let options = SOSOptions(liveAgentPod: "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:forName:`, `setFontDescriptor:fontFileName:forWeight:`, and `setImage:compatibleWithTraitCollection:forName:`. You can also configure the strings used throughout the interface. See SDK Customizations for more info.

5. To start an SOS session, call `startSessionWithOptions:` on the `SOSSessionManager` shared instance.

You can start a session when the view controller loads, or from a UI action.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().sos.startSession(with: 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 SDK.

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

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 `addDelegate:` method on your SOS instance (from `SCServiceCloud`). In particular, we suggest that you implement the `sos:didStopWithReason: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 Objective-C:

```
[[SCServiceCloud sharedInstance].sos stopSession];
```

In Swift:

```
SCServiceCloud.sharedInstance().sos.stopSession()
```

Alternatively, you can call the `stopSessionWithCompletion:` method with a completion block.

For additional details on customizing the SOS experience in your app, see the other topics covered in Using SOS. 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 SDK.

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:didStopWithReason:error:` method and show a dialog when appropriate.

In Objective-C:

```objc
#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;
```

```objc
// 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
```

```objc
                                        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
```

In Swift:

```swift
import UIKit
import ServiceCore
import ServiceSOS

class ViewController: UIViewController, SOSDelegate {

  override func viewDidLoad() {
    super.viewDidLoad()

    // Add our SOS delegate
    SCServiceCloud.sharedInstance().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
    SCServiceCloud.sharedInstance().sos.startSessionWithOptions(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)
      }
    }
}
```

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 specify 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), 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`
`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`

Feature	Description	Type & Default Value
sessionRetryTime	The length of time (in seconds) before SOS prompts the user to retry or cancel.	NSTimeInterval — Default: 30
setViewControllerClass:for:	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 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];
```

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

SCServiceCloud.sharedInstance().sos.startSession(with: 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 the optionsWithLiveAgentPod:orgId:deploymentId: method, as described in Quick Setup: SOS.

2. Initialize access to one or both of the cameras on the user's device by setting `featureClientFrontCameraEnabled` and `featureClientBackCameraEnabled`.

In Objective-C:

```
[options setFeatureClientFrontCameraEnabled: YES];
[options setFeatureClientBackCameraEnabled: YES];
```

In Swift:

```
options.featureClientFrontCameraEnabled = true
options.featureClientBackCameraEnabled = true
```

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 Objective-C:

```
[options setInitialCameraType: SOSCameraTypeBackFacing];
```

In Swift:

```
options.initialCameraType = .backFacing
```

If you'd rather start a session with the front-facing camera, use this command instead. In Objective-C:

```
[options setInitialCameraType: SOSCameraTypeFrontFacing];
```

In Swift:

```
options.initialCameraType = .frontFacing
```

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 Objective-C:

```
[options setFeatureClientScreenSharingEnabled: NO];
```

In Swift:

```
options.featureClientScreenSharingEnabled = false
```

To learn more, see Configure an SOS Session.

5. Start an SOS session.

In Objective-C:

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

In Swift:

```
SCServiceCloud.sharedInstance().sos.startSession(with: options)
```

To learn more, see Quick Setup: SOS.

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:
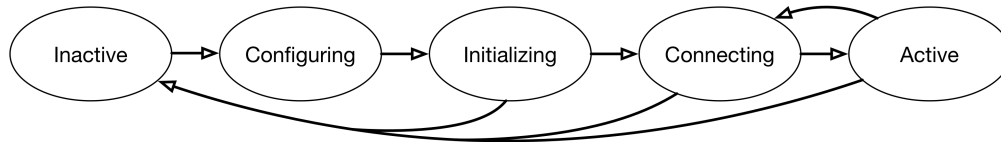
Inactive
> No active session; no outgoing/incoming SOS traffic.

Configuring
> Performing a pre-initialization configuration step, such as network testing.

Initializing
> Preparing to connect.

Connecting
> Attempting a connection to a live 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 `addDelegate:` 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: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:didStopWithReason:error:` — invoked when the session has ended.

These state changes are labeled with an associated number in the diagram below.
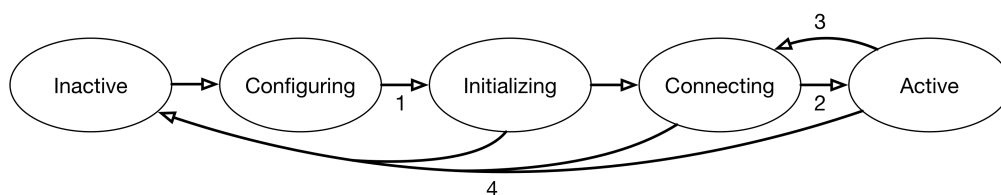
Handling Session Termination and Error Conditions

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:didStopWithReason: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 (including the session errors that are passed to `sos:didStopWithReason:error:`) 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:didStopWithReason:error:` and `sos:didError:`.

> 👁 Example: **Sample Code in SOS Test App**
>
> For a detailed example that implements `SOSDelegate`, refer to the SOS example app: SOS Example App. In particular, look at the `AppDelegate` class: https://github.com/goinstant/ios-sdk-guides/blob/master/SOSExamples/AppDelegate.m.

> 👁 Example: **Basic `SOSDelegate` Example**
>
> This sample code does the following:
>
> - Implements the `SOSDelegate` protocol.
> - Implements the `sos:stateDidChange:previous:` method.
> - Implements the `sosDidConnect:` method.
> - Implements the `sos:didStopWithReason:error:` method and includes some error handling logic.
> - Implements the `sos:didError:` method.
>
> In Objective-C:
>
> ```objc
> #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.
> ```

```objc
- (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
```

```objc
          //        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
```

In Swift:

```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 = ""
```

```swift
    // 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)
  }
}
```

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 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...
   }
   ```

   In Swift:

   ```
   func agentAvailability(_ agentAvailability: Any!,
                           didChange availabilityStatus: SOSAgentAvailabilityStatusType) {
     // TO DO: Handle event...
   }

   func agentAvailability(_ agentAvailability: Any!,
                           didError error: 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 `startPollingWithOrganizationId:deploymentId:liveAgentPod:`.

   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"];
   ```

   In Swift:

   ```
   let agentAvailability = SCServiceCloud.sharedInstance().sos.agentAvailability!
   agentAvailability.add(self)
   agentAvailability.startPolling(withOrganizationId: "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.

👁 **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.

```objc
- (void)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 {

    UIAlertView *alert = nil;
    __weak SOSLaunchViewController *wSelf = self;

    // ensure there is no SOS session currently active
    if (error && [SOSSessionManager sharedInstance].state ==
        SOSSessionStateInactive) {

        alert = [[UIAlertView alloc] initWithTitle:@"Error"
                                           message:[error localizedDescription]
```

131

```
                                                delegate:wSelf
                                        cancelButtonTitle:@"OK"
                                        otherButtonTitles:nil];
                [alert show];
        }
}
```

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 Objective-C:

```
[SCServiceCloud sharedInstance].sos.screenSharing.enabled = NO;
```

In Swift:

```
SCServiceCloud.sharedInstance().sos.screenSharing.enabled = false
```

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:
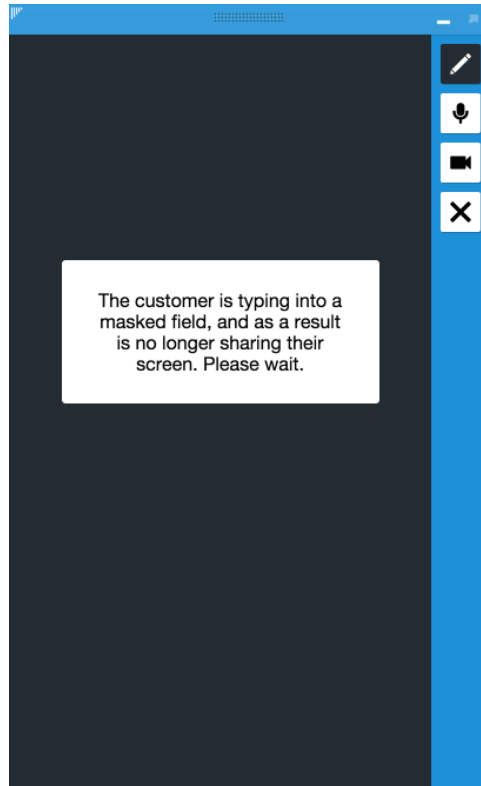
To use field masking, replace the `UITextField` containing sensitive information with a `SOSMaskedTextField`.

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.

Create Masked Field Programmatically

To create a masked field manually, instantiate and style a `SOSMaskedTextField` instance.

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**.
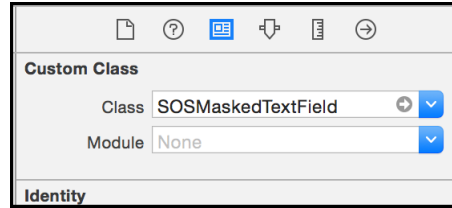
3. From the **Identity Inspector**, specify the text field style settings in the **User Defined Runtime Attributes** section.
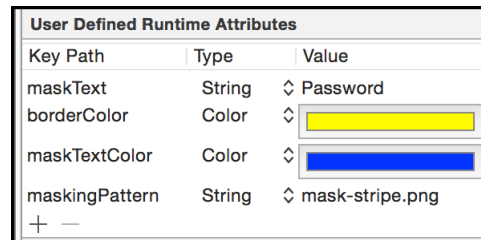
The follow key path attributes are available:

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`.

```
@interface SOSMaskedTestViewController : UIViewController

@property (strong, nonatomic) IBOutlet SOSMaskedTextField *maskCodeExample;

@end
```

2. Instantiate and style the `SOSMaskedTextField`.

```
@interface SOSMaskedTestViewController () <UITextFieldDelegate>
@end

@implementation SOSMaskedTestViewController
```

```
@synthesize maskCodeExample;

- (void)viewDidLoad {
    [super viewDidLoad];

    // 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:

Field Information			
Field Label	Email	Object Name	SOS Session
Field Name	Email	Data Type	Email
API Name	Email__c		

> 📝 **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 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];
```

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"]
```

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:for:` 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 7: SOS Phases

Phase Name	View Controller to Subclass	Protocol to Implement
SOSUIPhaseOnboarding	SOSOnboardingBaseViewController	SOSOnboardingViewController
SOSUIPhaseConnecting	SOSConnectingBaseViewController	SOSConnectingViewController
SOSUIPhaseScreenSharing	SOSScreenSharingBaseViewController (which subclasses SOSSessionBaseViewController)	SOSSessionViewController, SOSUIAgentStreamReceivable, SOSUILineDrawingReceivable

For example, the following code overrides the onboarding UI.

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];
```

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
// ...

SCServiceCloud.sharedInstance().sos.startSession(with: options)
```

To learn more about configuring an SOS session, see Configure an SOS Session.

For a detailed example that has a UI replacement, refer to the SOS example app: SOS Example App. In particular, look at the view controller classes: https://github.com/goinstant/ios-sdk-guides/tree/master/SOSExamples/SOS/ViewControllers. You can turn on the custom UI in the sample app using the `Custom View Controllers` flags in the `SOSSettings.plist` file.

Alternatively, 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.

```
- (BOOL)willHandleConnectionPrompt {
  return YES;
}

- (void)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
```

SOS Connecting Sample Code

Below you'll find some boilerplate sample code for the connecting experience. This class must subclass
SOSConnectingBaseViewController.

```
- (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
}

// Call `handleEndSession:` to cancel a session

// See `SOSConnectingBaseViewController`, `SOSConnectingViewController`
// for additional functionality
```

Screen Sharing Sample Code

Below you'll find some boilerplate sample code for the screen sharing experience. This class must subclass
SOSScreenSharingBaseViewController.

```
- (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
}

// See `SOSSessionBaseViewController` for how to handle pause, mute, end session events

// See `SOSScreenSharingBaseViewController`, `SOSUIAgentStreamReceivable`,
//     `SOSUILineDrawingReceivable` for additional functionality
```

SDK Customizations

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 guidance on remote notifications and 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:forName:`, `setFontDescriptor:fontFileName:forWeight:`, and `setImage:compatibleWithTraitCollection:forName:`. 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 `SCServiceCloud 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. 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.

Notifications are another area for potential customization. With a little bit of code, you can have the SDK automatically handle Service Cloud remote notifications and display the relevant view.

Customize Colors

You can customize the look and feel of the interface by specifying the branding token values in an `SCAppearanceConfiguration`
object.

You customize the colors by defining the branding token colors used throughout the interface. The screenshots below illustrate how
the branding tokens affect the UI.

Knowledge UI Branding:

nav bar background

nav bar inverted

brand primary

brand contrast

contrast primary

contrast secondary

contrast inverted

brand secondary

brand contrast
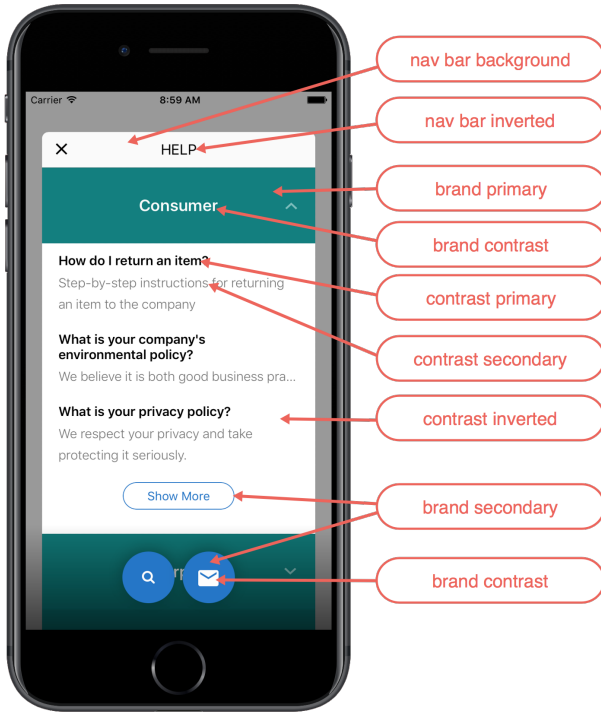
Case Management UI Branding:

nav bar background

nav bar inverted

contrast primary

brand primary

contrast inverted
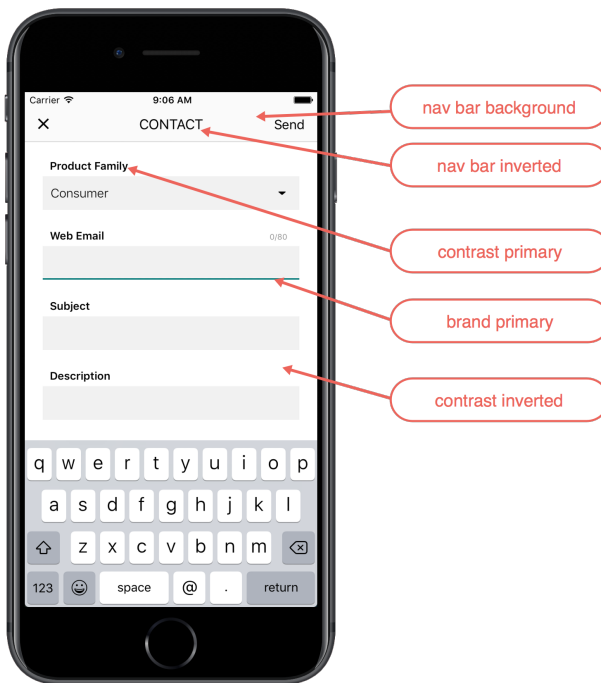
Live Agent Chat UI Branding:
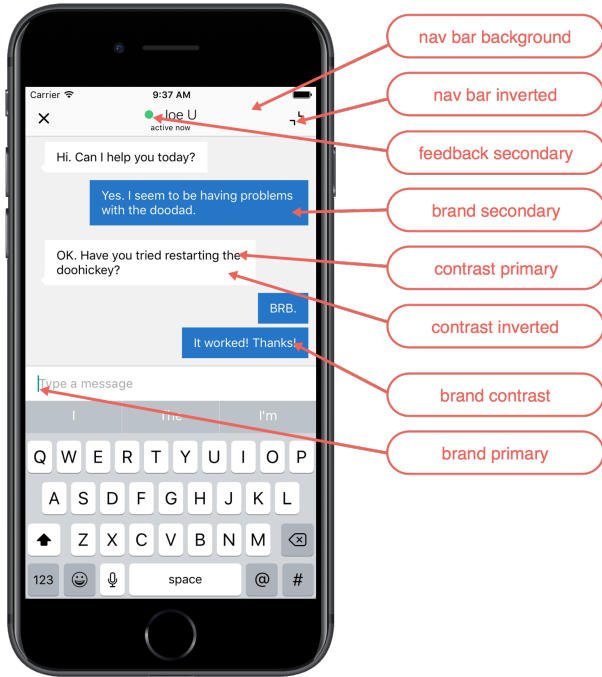
SOS UI Branding:
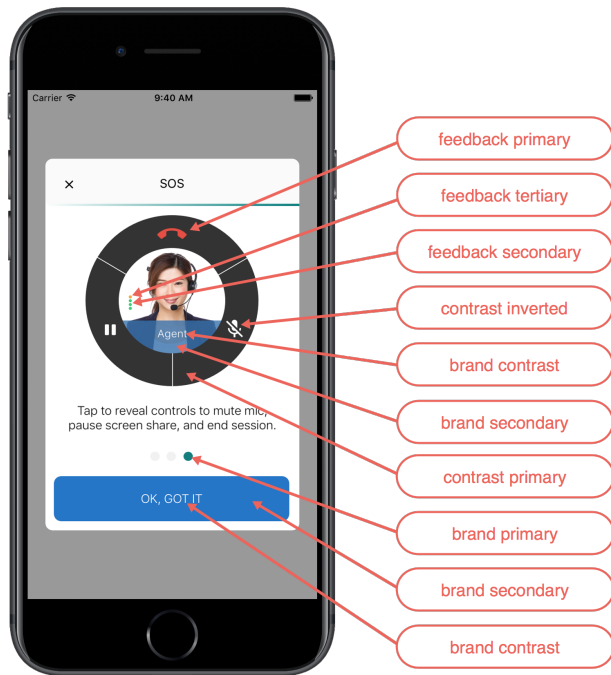
The following branding tokens are available for customization:

Token Name	Default Value	Description / Sample Uses
Navigation Bar Background `SCSAppearanceNavbarBackgroundColor`	#FAFAFA	Background color for the navigation bar.
Navigation Bar Inverted `SCSAppearanceNavbarInvertedColor`	#010101	Navigation bar text and icon color.
Brand Primary `SCSAppearancePrimaryBrandColor`	#007F7F	Knowledge: First data category, the Show More button, the footer stripe, the selected article. SOS: Used for various icons.
Brand Secondary `SCSAppearanceSecondaryBrandColor`	#2872CC	Used throughout the UI for button colors. Live Agent Chat: Agent text bubbles. SOS: Background color for action items.
Brand Contrast `SCSAppearanceBrandContrastColor`	#FCFCFC	Text on areas where a brand color is used for the background. SOS: Color of the icons (when they are selected).
Contrast Primary `SCSAppearanceContrastPrimaryColor`	#000000	Primary body text color. SOS: Background color for buttons on the UI.
Contrast Secondary `SCSAppearanceContrastSecondaryColor`	#767676	Knowledge: Subcategory headers.
Contrast Tertiary `SCSAppearanceContrastTertiaryColor`	#BABABA	
Contrast Quaternary `SCSAppearanceContrastQuaternaryColor`	#F1F1F1	

Token Name	Default Value	Description / Sample Uses
Contrast Inverted SCSAppearanceContrastInvertedColor	#FFFFFF	Page background, navigation bar, table cell background. SOS: Color of the icons (when they aren't selected).
Feedback Primary SCSAppearanceFeedbackPrimaryColor	#E74C3C	Text color for error messages. SOS: Mute indicator. Disconnect icon.
Feedback Secondary SCSAppearanceFeedbackSecondaryColor	#2ECC71	SOS: Connection quality indicators. Background color for the Resume button when the two-way camera is active.
Feedback Tertiary SCSAppearanceFeedbackTertiaryColor	#F5A623	SOS: Connection quality indicators.
Title Text SCSAppearanceTitleTextColor	#FBFBFB	Knowledge: Text on data category headers and the chevron on the Knowledge home page.
Overlay SCSAppearanceOverlayColor	Contrast Primary (at 40% alpha)	Knowledge: Background for the Knowledge home screen.

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 SCServiceCloud sharedInstance.

In Objective-C:

```objc
// Create appearance configuration instance
SCAppearanceConfiguration *config = [SCAppearanceConfiguration new];

// Customize color tokens
[config setColor:COLOR_VALUE forName:TOKEN_NAME];

// Add other customizations here...

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = config;
```

In Swift:

```swift
// Create appearance configuration instance
let config = SCAppearanceConfiguration()
```

```
// Customize color tokens
config.setColor(COLOR_VALUE, forName: TOKEN_NAME)

// Add other customizations here...

// Save configuration instance
SCServiceCloud.sharedInstance().appearanceConfiguration = config
```

👁 Example: The following code sample changes three of the branding tokens.

In Objective-C:

```
// Create appearance configuration instance
SCAppearanceConfiguration *config = [SCAppearanceConfiguration new];

// Customize color tokens
[config setColor:[UIColor colorWithRed:80/255 green:227/255 blue:194/255 alpha:1.0]
        forName:SCSAppearancePrimaryBrandColor];
[config setColor:[UIColor colorWithRed:74/255 green:144/255 blue:226/255 alpha:1.0]
        forName:SCSAppearanceSecondaryBrandColor];
[config setColor:[UIColor colorWithRed:252/255 green:252/255 blue:252/255 alpha:1.0]
        forName:SCSAppearanceBrandContrastColor];

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = config;
```

In Swift:

```
// Create appearance configuration instance
let config = SCAppearanceConfiguration()

// Customize color tokens
config.setColor(UIColor(red: 80/255, green: 227/255, blue: 194/255, alpha: 1.0),
        forName: SCSAppearancePrimaryBrandColor)
config.setColor(UIColor(red: 74/255, green: 144/255, blue: 226/255, alpha: 1.0),
        forName: SCSAppearanceSecondaryBrandColor)
config.setColor(UIColor(red: 252/255, green: 252/255, blue: 252/255, alpha: 1.0),
        forName: SCSAppearanceBrandContrastColor)

// Save configuration instance
SCServiceCloud.sharedInstance().appearanceConfiguration = config
```

Customize Fonts

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

Font Setting	Default Value	Samples Uses in the SDK
		submit success view, content of error messages
`SCFontWeightRegular`	Helvetica Neue	Navigation bar, Live Agent 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**.
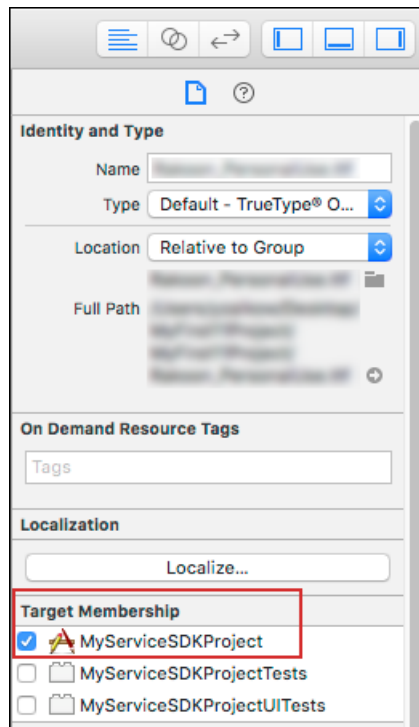
3. Add the font to your app's `Info.plist`.

   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 `SCServiceCloud` shared instance.

   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;
```

   Swift Example:

```
// Create appearance configuration instance
let config = SCAppearanceConfiguration()

// Customize font
let descriptor = UIFontDescriptor(fontAttributes:
    [UIFontDescriptorFamilyAttribute : "Proxima Nova"])
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

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 8: Custom Image Types

Image Type	How to Customize
Stock images	Use the `setImage:compatibleWithTraitCollection:forName:` method on the `SCAppearanceConfiguration` object to **replace a stock image** with your image. Use the named constant 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 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 named constant value specified by the SDK and add it to the `SCAppearanceConfiguration` object with the `setImage:compatibleWithTraitCollection:forName:` method.

Table 9: Stock Image Constant Values

Image Description	Named Constant Value
Common Images	*Images used throughout the SDK*
Close button	`Close`
Done button	`Done`
Error	`Error`

Image Description	Named Constant Value
No connection	NoConnection
Minimize button (Knowledge and Live Agent Chat)	MinimizeButtonImage
Next field button (Case Publisher and Live Agent Chat)	SubmitButtonNext
Previous field button (Case Publisher and Live Agent Chat)	SubmitButtonPrevious
Knowledge Images	*Images used by Knowledge*
Article is empty	EmptyArticle
Category header arrow	CategoryHeaderArrow
Placeholder image before search completes	SearchPlaceholder
Search action button	ActionSearch
Case Management Images	*Images used by Case Management*
Case publisher success message	CaseSubmitSuccess
Case publisher action button	ActionCasePublisher
Pick case list button	PicklistDropdown
Live Agent Chat Images	*Images used by Live Agent Chat*
Attachment button when the user can attach a file	AttachmentClipIcon
Avatar used for the agent	ChatAgentAvatar
Chat icon used in minimized view	ChatIcon
Icon used in the pre-chat screen	PreChatIcon
SOS Images	*Images used by SOS*
Cancel icon	cancel
Confirm icon	confirm
End icon	end
Pause icon	pause
Resume icon	resume
Microphone icon	mic
User muted icon	muted
Agent muted icon	agentMuted
Camera icon	camera
Agent placeholder icon (must be 61x55 pixels @1x, 122x110 @2x, 183x165 @3x)	avatar

Image Description	Named Constant Value
Expand icon	`downCaret`
Flashlight icon	`flashlight`
Info icon	`tip`
Mask stripe	`mask-stripe`

In Objective-C:

```
// Create appearance configuration instance
SCAppearanceConfiguration *config = [SCAppearanceConfiguration new];

// Specify images
[config setImage:MY_CUSTOM_IMAGE compatibleWithTraitCollection: MY_TRAITS
        forName: NAMED_CONSTANT_VALUE];

// Add other customizations here...

// Save configuration instance
[SCServiceCloud sharedInstance].appearanceConfiguration = config;
```

In Swift:

```
// Create appearance configuration instance
let config = SCAppearanceConfiguration()

// Specify images
config.setImage(MY_CUSTOM_IMAGE, compatibleWithTraitCollection: MY_TRAITS,
               forName: NAMED_CONSTANT_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.

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 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;
```

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
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 `SCServiceCloud` instance with an implementation of `SCKnowledgeInterfaceDelegate`.

    In Objective-C:

    ```
    [SCServiceCloud sharedInstance].knowledge.delegate =
        mySCKnowledgeInterfaceDelegateImplementation;
    ```

    In Swift:

    ```
    SCServiceCloud.sharedInstance().knowledge.delegate =
        mySCKnowledgeInterfaceDelegateImplementation
    ```

2. Implement the two delegate methods.

    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;
}
```

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
}
```

Customize Action Buttons

You can determine which action buttons are visible by implementing a protocol method on `SCServiceCloudDelegate`.

📝 **Note:** This customization can be used for Knowledge and Case Management.

When dealing with action buttons, use the named constant value for the button you'd like to show or hide:

SCSActionCasePublisher
    Allows a user to create a case. If the user is authenticated, this will show a list of their existing cases.

SCSActionArticleSearch
    Allows a user to perform a search for a knowledge base article.

To control visibility:

1. Implement the `SCServiceCloudDelegate` protocol and handle the `serviceCloud:shouldShowActionWithName:` method. This method is called to determine whether the specified action button should be present, when applicable.

    In Objective-C:

```
- (BOOL)serviceCloud:(SCServiceCloud *)serviceCloud
        shouldShowActionWithName:(NSString *)name {

  BOOL shouldShowAction = NO;

  if ([name isEqualToString:SCSActionCasePublisher]) {
    // Determine whether we should show the action for case publisher
    // If so, set shouldShowAction to YES; otherwise NO.
    shouldShowAction = YES;
```

```
  } else if ([name isEqualToString:SCSActionArticleSearch]) {
    // Determine whether we should show the action for article searches
    // If so, set shouldShowAction to YES; otherwise NO.
    shouldShowAction = YES;
  }

  return shouldShowAction;
}
```

In Swift:

```swift
func serviceCloud(_ serviceCloud: SCServiceCloud,
     shouldShowActionWithName name: String) -> Bool {

  var shouldShowAction = false

  switch (name) {
    case SCSActionCasePublisher:
      // Determine whether we should show the action for case publisher...
      // If so, set shouldShowAction to true; otherwise false.
      shouldShowAction = true
    case SCSActionArticleSearch:
      // Determine whether we should show the action for article searches...
      // If so, set shouldShowAction to true
      shouldShowAction = true
    default:
      shouldShowAction = false
  }

  return shouldShowAction
}
```

Customize and Localize Strings

You can change the text 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:

- Knowledge String Resources
- Case Management String Resources
- Live Agent Chat String Resources
- SOS String Resources
- ServiceCommon String Resources

The following languages are currently supported:

Table 10: Supported Languages

Language Code	Language
cs	Czech
da	Danish
de	German
el	Greek
en	English
es	Spanish
fi	Finnish
fr	French
hu	Hungarian
id	Indonesian
it	Italian
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
zh_TW	Chinese (Taiwan)
zh-Hans	Chinese (Simplified)
zh-Hant	Chinese (Traditional)
zh	Chinese

Handle Remote Notifications

The SDK can handle remote notifications for events that it understands, such as case feed activity, and display the associated view.

When building your app, you may have remote notifications arrive from many different sources. Some of these sources may be associated with 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 sequence diagram below illustrates a scenario where an Apex trigger from your connected app sends a notification to your mobile app and you instruct the SDK to display the associated view.
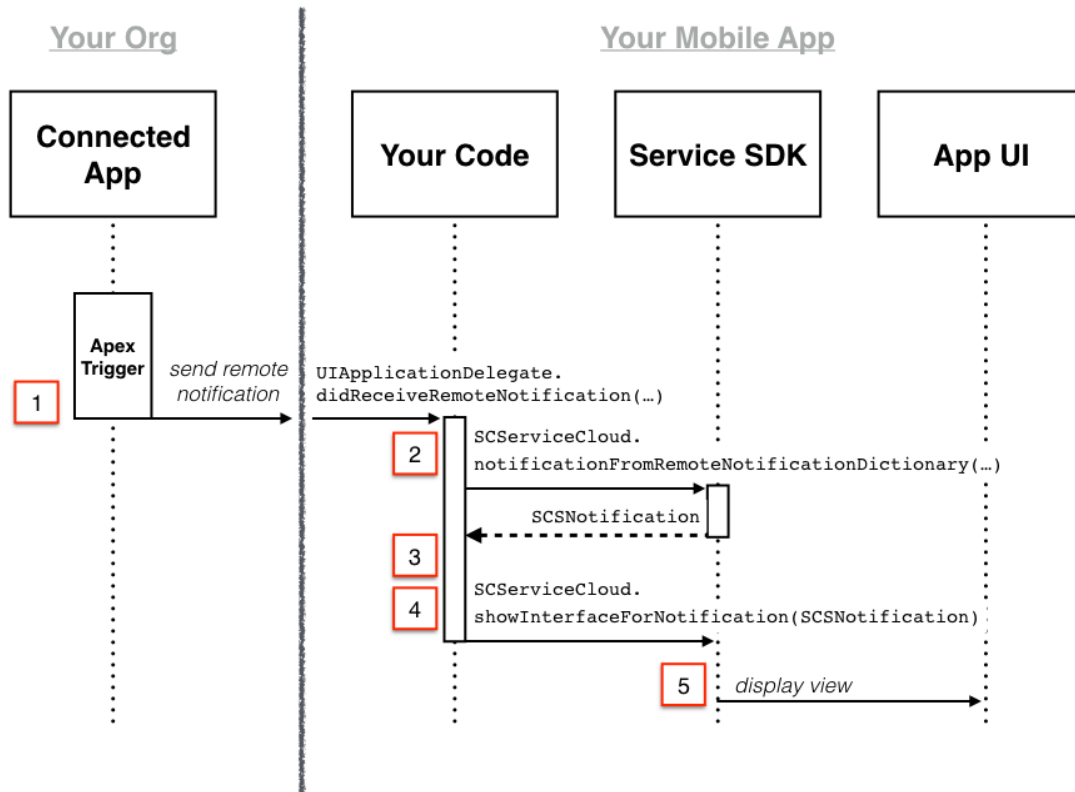
To deal with push notifications from within your app:

1. Send the push notification from your org.

    For help sending push notifications for case feed activity, see Push Notifications for Case Activity.

2. When you receive a remote notification (from your app delegate's `didReceiveRemoteNotification` method), pass notification information to `notificationFromRemoteNotificationDictionary:` to determine whether the SDK can handle the notification.

    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
}
```

In Swift:

```
func application(_ application: UIApplication,
                 didReceiveRemoteNotification userInfo: [AnyHashable : Any],
                 fetchCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult)
                 -> Void) {

  let notification =
    SCServiceCloud.sharedInstance().notification(fromRemoteNotificationDictionary:
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.

3. (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`.

4. If you want the SDK to handle the notification for you, call `showInterfaceForNotification:` and pass in the `SCSNotification` object.

In Objective-C:

```
[[SCServiceCloud sharedInstance] showInterfaceForNotification:notification];
```

In Swift:

```
SCServiceCloud.sharedInstance().showInterface(for: 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 `showInterfaceForNotification:`). See Customize the Presentation for Case Management.

5. The SDK displays 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.

Launch SDK from a Web View

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
- Quick Setup: Case Publisher as a Guest User
- Quick Setup: Live Agent Chat
- Quick Setup: SOS

Once you're familiar with the feature, use this documentation 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 Objective-C:

   ```
   @interface ViewController : UIViewController <UIWebViewDelegate>
   ```

   In Swift:

   ```
   class ViewController: UIViewController, UIWebViewDelegate {
   ```

3. Set your view controller as the web view delegate and configure your Service SDK feature.

   In Objective-C:

   ```objc
   - (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 Service 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 Swift:

   ```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 Service 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:shouldStartLoadWithRequest:navigationType:` method to trap requests.

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;
}
```

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
}
```

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 as you can see in this code sample.

In Objective-C:

```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;
```

159

```
      }

      return YES;
}
```

In Swift:

```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

}
```

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.

```html
<a href="servicesdk://start">Get help!</a>
```

160

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

Get some guidance when you run into issues.

Logging
Learn how to turn on and control the logging framework so that you can debug issues during development.

Unable to Access My Community
What to do when you can't seem to get to your community from within your app.

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.

My App Crashes
Some tips if your app crashes.

My App Was Rejected
What to do when your app is rejected from the App Store.

Logging

Learn how to turn on and control the logging framework so that you can debug issues during development.

If you want to inspect the Service SDK logs, you can turn on logging using the `setLogLevel:forIdentifiersWithPrefix:` method in the `SFLogger` shared instance.

In Objective-C:

```
SFLogger *logger = [SFLogger sharedLogger];
[[SFLogger sharedLogger] setLogLevel:SFLogLevelWarning
            forIdentifiersWithPrefix:@"com.salesforce.ServiceSDK"];
```

In Swift

```
let logger = SFLogger.shared()
logger.setLogLevel(SFLogLevel.warning,
                   forIdentifiersWithPrefix:"com.salesforce.ServiceSDK")
```

Log Level

The log level is specified in the `setLogLevel:forIdentifiersWithPrefix:` method using the `SFLogLevel` enumerated type. It can be one of these values:

- `SFLogLevelVerbose`
- `SFLogLevelDebug`
- `SFLogLevelInfo`
- `SFLogLevelWarning`
- `SFLogLevelError`
- `SFLogLevelOff`

By default, the level is set to `SFLogLevelError` so that only serious issues are logged.

Log Output

By default, logs only go to the Xcode output. You can have logs go to ASL (Apple System Logger) using the `logToASL` property. You can use the `logToFile` property to log messages to a file in your app's directory on the device. This feature supports log rotation every 48 hours, with a maximum of three log files on a device. If you turn off `logToFile`, the files are automatically removed.

In Objective-C:

```
logger.logToFile = YES;
```

In Swift:

```
logger.shouldLogToFile = true
```

You can progammatically access this file with the `logFileContents` property.

Unable to Access My Community

What to do when you can't seem to get to your community 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 SDK for more info.

Key	Type	Value
▼ Information Property List	Dictionary	(17 items)
▼ App Transport Security Settings	Dictionary	(1 item)
▼ Exception Domains	Dictionary	(2 items)
▼ localhost	Dictionary	(1 item)
NSExceptionAllowsInsecureHTTPLoads	Boolean	YES
▼ force.com	Dictionary	(3 items)
NSExceptionRequiresForwardSecrecy	Boolean	NO
NSIncludesSubdomains	Boolean	YES
NSThirdPartyExceptionRequiresForwardSecrecy	Boolean	NO
Localization native development region	String	en
Executable file	String	$(EXECUTABLE_NAME)

3. Have you set up a Community or Force.com 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.
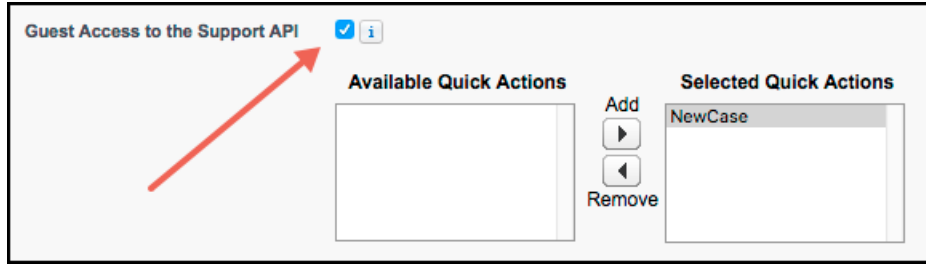
5. (For Knowledge only) Do you have **Knowledge** enabled in your org? Do you have Knowledge licenses? See Cloud Setup for Knowledge for more info.

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.
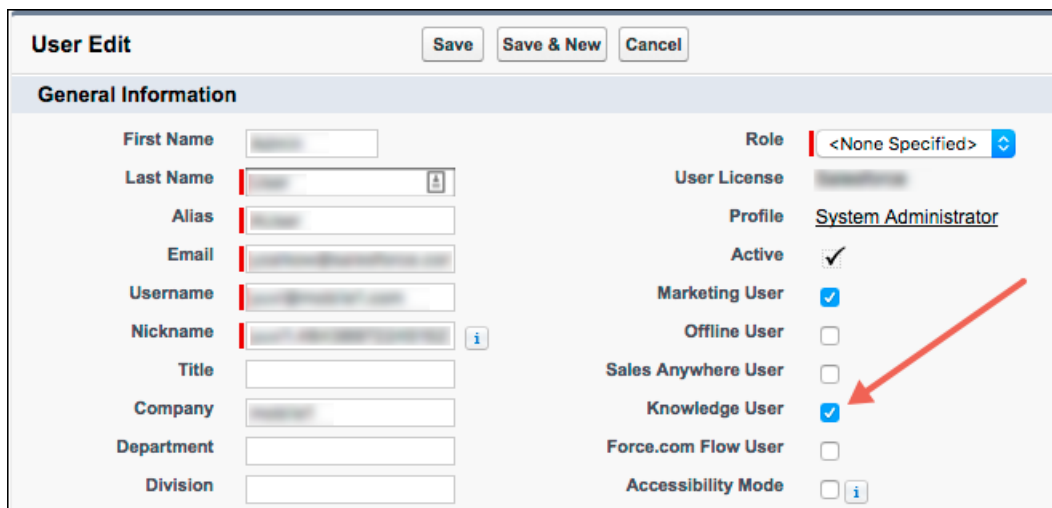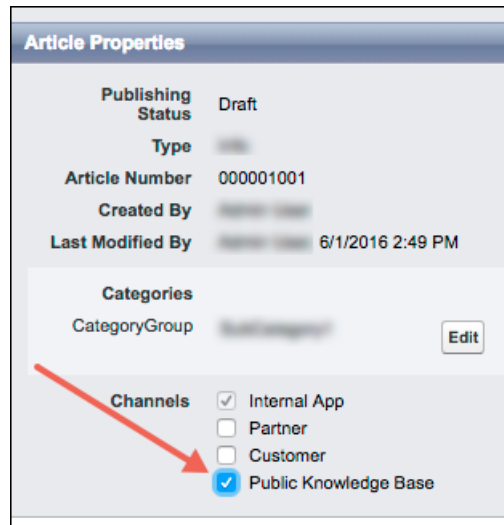
7. (For Knowledge only) Have you made the article types, the data categories, and the article layout fields visible to guest users? See Guest User Access for Your Community for more info.

8. (For Knowledge only) Have you made your articles accessible to the **Public Knowledge Base** channel? See Cloud Setup for Knowledge for more info.

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
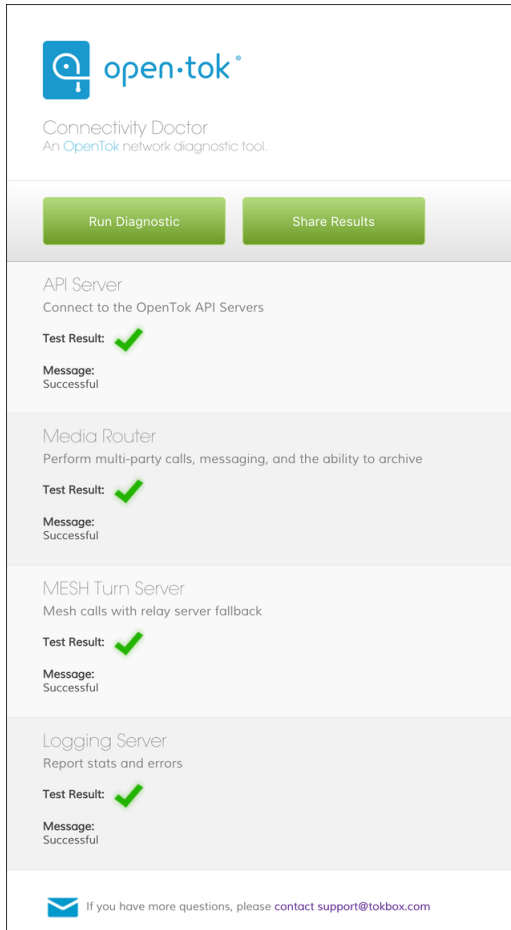
If all tests pass, go to step 2. If any test fails, you probably need to configure your ports.

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.

- Live Agent 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 SDK.

- 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 SDK.

- 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.

Reference Documentation

Reference documentation for Service Cloud Snap-ins for iOS.

To access the reference documentation for Service Cloud Snap-ins 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.

Knowledge Index

- `SCArticleSortByField`
- `SCArticleSortOrder`
- `SCKnowledgeInterface`
  - `setInterfaceVisible:animated:completion:`
- `SCKnowledgeInterfaceDelegate`
- `SCQueryMethod`
- `SCSArticle`

Case Management Index

Live Agent Chat Index

SOS Index

Service Common Index

- SCAppearanceConfigurationDelegate
- SCServiceCloud
    - account
    - appearanceConfiguration
    - cases
    - chat
    - knowledge
    - notificationFromRemoteNotificationDictionary:
    - serviceCloud:shouldShowActionWithName:
    - setAccount:completion:
    - sharedInstance
    - showInterfaceForNotification:
    - sos
- SCServiceCloudDelegate
    - serviceCloud:serviceAuthenticationFailedWithError:
    - serviceCloud:shouldAuthenticateService:completion:
    - serviceCloud:transitioningDelegateForViewController:
    - serviceCloud:willDisplayViewController:animated:
- SCSServiceConfiguration
    - imageFolderPath
    - initWithCommunity:
    - initWithCommunity:dataCategoryGroup:rootDataCategory:
- SCSNotification
- SCSNotificationType

Resource Files

- Knowledge String Resources
- Case Management String Resources
- Live Agent Chat String Resources
- SOS String Resources
- ServiceCommon String Resources

Additional Resources

If you're looking for other resources, check out this list of links to related documentation.

- **Service Cloud Snap-ins for Mobile Apps**: More info about this SDK.
    - Snap-ins Landing Page
    - iOS Reference Documentation

- – iOS Release Notes
- – Android Reference Documentation
- – Android Release Notes

- **Salesforce Mobile SDK**: The SDK that lets you build Salesforce applications for mobile devices.

  - – Mobile SDK Landing Page
  - – Mobile SDK Developer's Guide
  - – Mobile SDK Trailhead

- Salesforce Developer Documentation: Landing page for developer documentation at Salesforce.
- Salesforce Help: Landing page for general documentation at Salesforce.

INDEX