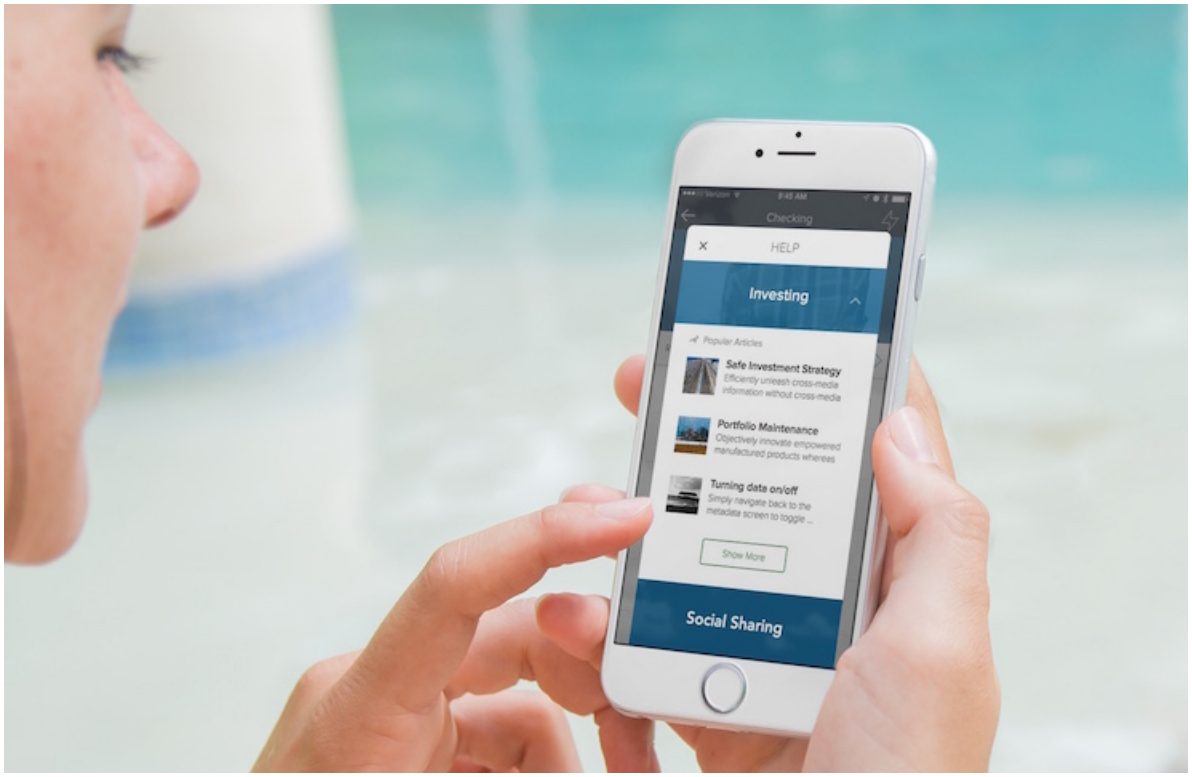
Service SDK Developer's Guide for Android

Version 0.4.0



@salesforcedocs

Last updated: December 21, 2016

CONTENTS

SERVICE SDK DEVELOPER'S GUIDE FOR ANDROID

The Service SDK for mobile devices makes it easy to give customers access to powerful Service Cloud features right from within your native mobile app. You can make these Service Cloud elements feel organic to your app and have things up and running quickly. This developer's guide helps you get started with the Service SDK.

This documentation describes the following component versions:

Component	Version Number
Knowledge	0.4.0 (Beta)
SOS	2.1.0

Note: This release contains a beta version of the Knowledge SDK, which means it has high-quality features with known limitations.

Release Notes

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

Knowledge Version 0.4.0 / SOS Version 2.1.0

New Features:

- SOS: Added server logging for Quality of Service (QOS) and error case scenarios.

Knowledge Version 0.4.0 / SOS Version 2.0.0

New Features:

- Knowledge: You can customize fonts in the Knowledge UI. To learn more, Knowledge: Customize Fonts.
- Knowledge: Image caching supported in Knowledge. To learn more, see Cache Images for Offline Access.
- Knowledge: Knowledge supports listening for analytics data. To learn more, see Analytics.
- SOS: You can programmatically stop and start screen sharing. To learn more, see Control an SOS Session.
- SOS: SOS now supports elegant reconnects when there are network issues.

API Changes:

- Knowledge: Repository moved from `http://salesforcesos.com/android/maven/knowledge/sdk/release` to `https://salesforcesos.com/android/maven/release`.
- Knowledge: Added `OfflineResourceConfig` and `OfflineResourceCache` API for offline caching.
- Knowledge: Added `cacheImages` option to `ArticleDetailsRequest.Builder`.
- Knowledge: Added `offlineResourceConfig` option to `KnowledgeConfiguration.Builder`.
- Knowledge: `ArticleListRequestBuilder#dataCategory` now takes two strings: a group name and a category name, instead of a `DataCategorySummary`.
- Knowledge: Category group name argument removed from `KnowledgeConfiguration#create`.
- Knowledge: Removed `KnowledgeClient#getDataCategoryGroupName` method.
- Knowledge: Removed `KnowledgeConfiguration#getDataCategoryGroupName` method.
- Knowledge: Removed `KnowledgeUIClient#getCommunityUrl`, `KnowledgeUIClient#getDataCategoryGroupName`, `KnowledgeUIClient#getRootDataCategory`, and added `KnowledgeUIClient#getConfiguration` methods.
- Knowledge: `KnowledgeUIConfiguration#create` now requires a `dataCategoryGroupName`.
- SOS: Repository moved from `http://salesforcesos.com/android/maven/sos/sdk/release` to `https://salesforcesos.com/android/maven/release`.
- SOS: Artifact Group ID changed from `com.salesforce.android` to `com.salesforce.service`.
- SOS: All color branding resource names have been changed from a having a `sos_` prefix to having a `salesforce_` prefix.
- SOS: Added `connectingUi` option to `SosConfiguration` that allows disabling the default SOS UI until the agent has connected to the session. This configuration option allows consuming applications to show their own connecting experience instead of the default UI.
- SOS: Removed the public `SosDialogPresenter` interface, `SosDefaultDialogPresenter` class, and the `dialogs` method from the `SessionBuilder` class. Consuming applications should no longer need to consume this API, as the new UI uses a non-blocking mechanism for presenting prompts instead of dialog boxes.
- SOS: Added `setScreenSharingEnabled` method to the `Sos` static API. Passing `false` to this method will prevent all screen captures, and will also prevent all masked fields from rendering. All normal SOS operations are unaffected, and this still allows two-way video to transmit video normally. This flag will persist between sessions.
- SOS: Removed the `haloUi` session configuration option.

- SOS: Removed the following unused dimension resources:
  - `sos_agent_radial_container_button_padding`
  - `sos_agent_radial_container_padding`
  - `sos_agent_radial_container_button_diameter`
  - `sos_agent_radial_container_border`
  - `sos_agent_radial_container_diameter`
  - `sos_agent_radial_container_recording_icon_left`
  - `sos_agent_radial_container_recording_icon_right`
  - `sos_agent_radial_container_recording_icon_top`
  - `sos_agent_radial_container_recording_icon_bottom`
  - `sos_agent_radial_container_recording_label_padding_left`
  - `sos_agent_radial_container_recording_label_padding_right`
  - `sos_agent_radial_container_recording_label_padding_top`
  - `sos_agent_radial_container_recording_label_padding_bottom`
  - `sos_agent_radial_container_recording_label_margin_left`
  - `sos_agent_radial_container_recording_label_margin_right`
  - `sos_agent_radial_container_recording_label_margin_top`
  - `sos_agent_radial_container_recording_label_margin_bottom`
  - `sos_agent_radial_container_name_margin_left`
  - `sos_agent_radial_container_name_margin_right`
  - `sos_agent_radial_container_name_margin_top`
  - `sos_agent_radial_container_name_margin_bottom`
  - `sos_agent_radial_container_name_padding_left`
  - `sos_agent_radial_container_name_padding_right`
  - `sos_agent_radial_container_name_padding_top`
  - `sos_agent_radial_container_name_padding_bottom`
- SOS: Removed the following unused string resources:
  - `sos_start_camera_share`
  - `sos_start_screen_share`
- SOS: Renamed the following dimension resources:
  - `sos_agent_radial_container_diameter` -> `sos_agent_container_diameter`
- SOS: Removed unused `SosToast` values:
  - `SCREEN_SHARE_FROM_CAMERA`
  - `CAMERA_SHARE`
- SOS: Removed deprecated `userIdentifier` constructor for `SosOptions`. Use custom data constructor instead.
- SOS: Removed deprecated method `Sos.setAgentVideoEnabled(boolean)`. Use `setSessionPaused(boolean)` instead.
- SOS: Removed the Radial UI.

Fixes:

- Knowledge: No longer crashes with an `OutOfMemoryException` when trying to batch download many objects at once.
- Knowledge: The scrolling position is now retained on navigation and rotation.
- Knowledge: Links from article details now open in an external browser.
- Knowledge: The home modal now has a constrained width in landscape orientation.
- Knowledge: The `onClose` listener is properly called when the user presses the back button.
- SOS: Changed `AvailabilityFragment` to use default values for org configuration information.
- SOS: Removed background color from `AgentVideoSurface` for Nougat compatibility.
- SOS: Fixed crash when user selects end call in the UI while initializing.

Known Issues:

- Knowledge: Scrollable elements can be drawn below the bottom of the screen when returning to a previously-viewed fragment. This can result in items getting clipped or not being visible. **Workaround:** Rotate the device or go back one level and reopen the page.
- SOS: Screen sharing during Android 7's multi-window mode may result in black areas around the window and may cause the view of the agent to move off the user's screen. This is the result of incorrect window size reporting from the Android OS. There are two open issues logged against the Android Open Source Project (issue #216099, #209972). **Workaround:** If you're worried about a poor multi-window experience with SOS, consider limiting or disabling use of multi-window in your application.

Knowledge Version 0.3.1 / SOS Version 1.5.1

Fixes:

- Knowledge: Fixed a crash that occurred when navigating back from an article list while simultaneously scrolling through it.
- SOS: Increased polling time for Agent Availability from 5 seconds to 30 seconds.

Knowledge Version 0.3.0 / SOS Version 1.5.0

New Features:

- SOS documentation now incorporated into this developer's guide.
- Knowledge search view now allows for infinite scroll behavior of query results.
- We've introduced a custom loading progress indicator for Knowledge.
- We now require all SSL connections to use TLS v1.2.

API Changes:

- Minimum supported Android API version is now 16 (JELLY_BEAN).
- The instantiation flow for both Knowledge Core and Knowledge UI have changed. To learn more, see Quick Setup: Knowledge and Article Fetching with the Knowledge Core API.
- Knowledge `Async` class now has an `inProgress` method.
- Knowledge `isComplete` method in `Async` class no longer returns `true` when the operation ends in an error.
- Added an `OnCloseListener` interface and a companion `KnowledgeUIClient.setOnCloseListener()` method. The `OnCloseListener.onClose()` callback notifies you when the user explicitly closes the Knowledge UI or dismisses the minimized view. To learn more, see Quick Setup: Knowledge.

Known Issues:

- Knowledge does not currently support Community URLs that result in an HTTP redirect.

Service Cloud Setup

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

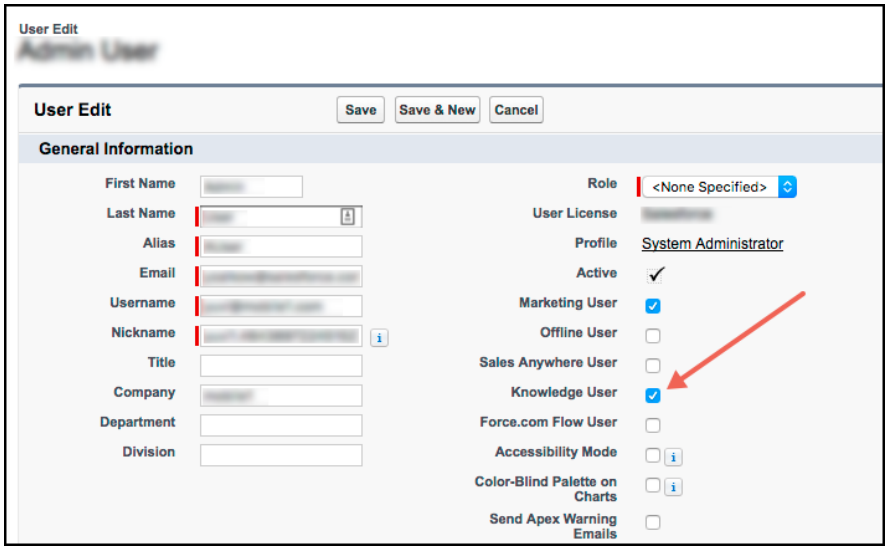
Cloud Setup for Knowledge

Set up your knowledge base and community to use Knowledge in your app.

1. **Salesforce Knowledge** must be enabled and set up in your org.

   Knowledge needs to be enabled and you'll need Knowledge licenses. To learn more, see Enable Salesforce Knowledge in Salesforce Help.

   Once you have Knowledge enabled, you need to turn on **Knowledge User** in the user settings for whoever administers the knowledge base.



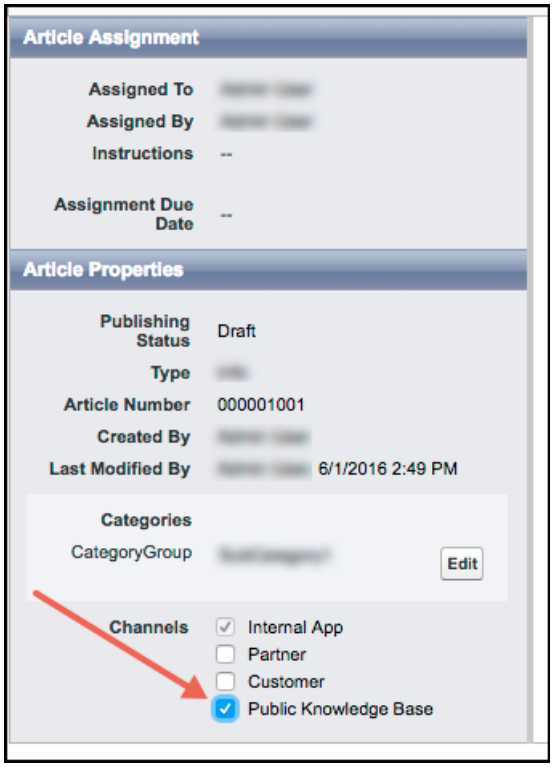
   If you don't see the **Knowledge User** checkbox, verify that you've enabled Knowledge in your org and that your org has Knowledge licenses. To learn more, see Enable Salesforce Knowledge in Salesforce Help.

2. When setting up Knowledge, make sure you define Article Types, create Knowledge articles, and associate articles with data categories within a category group.

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

   Your app developer needs the name of the **Data Category Group** and a root **Data Category** to use the knowledge feature in the SDK.

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



3. 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.

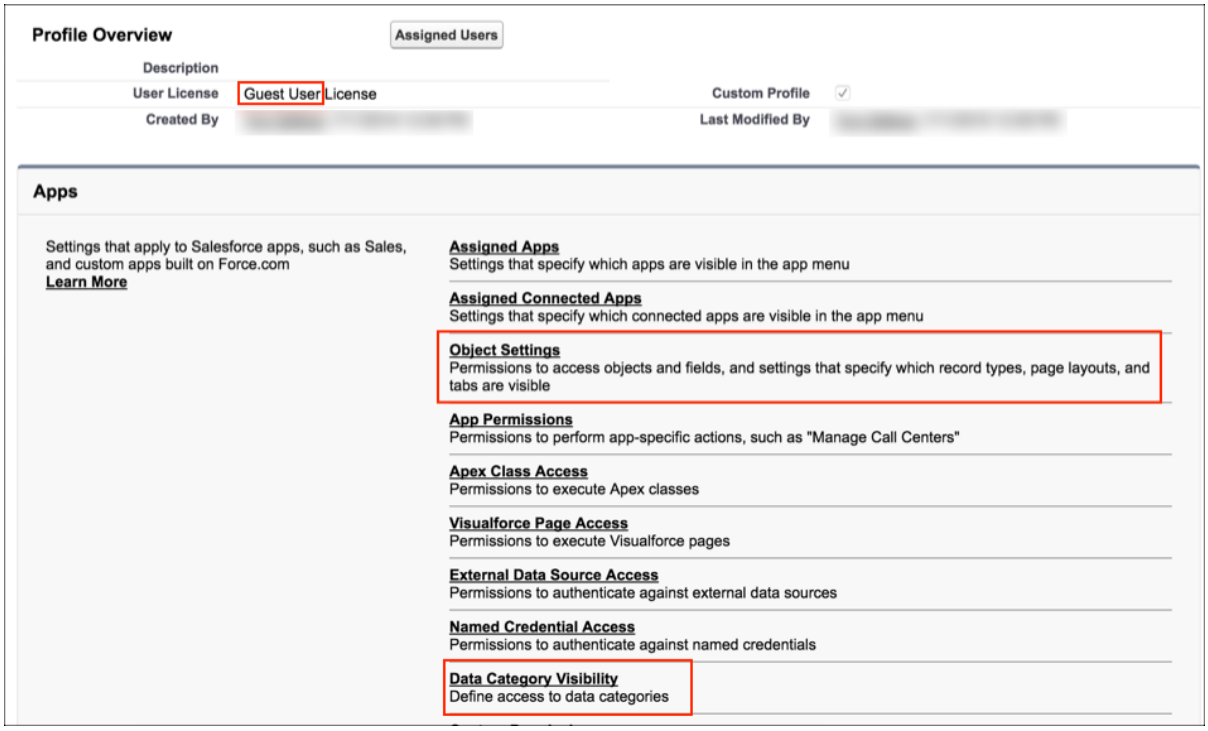
   You can either create a site with Community Builder or you can build a Force.com site. Getting Started With Communities in Salesforce Help provides detailed information about getting a community up and running. If you don't know which type of site is suitable for your needs, check out Choosing Between Community Builder and Force.com Sites.

   When setting up the site, be sure that your site has "Guest Access to the Support API" enabled.

4. To show Knowledge articles from your app, enable guest user access for the **Article Types**, **Categories**, and **Fields** associated with your knowledge articles.

   Note: Your Knowledge categories and articles will not appear if you do not have the GUEST USER profile set up properly.

   To view the guest user settings, go to your **Site Detail** page and select **Public Access Settings**. From this view, you can enable the required elements for the guest user profile. You can view the article type and article layout fields from the **Object Settings** page. You can view the categories from the **Data Category Visibility** page.



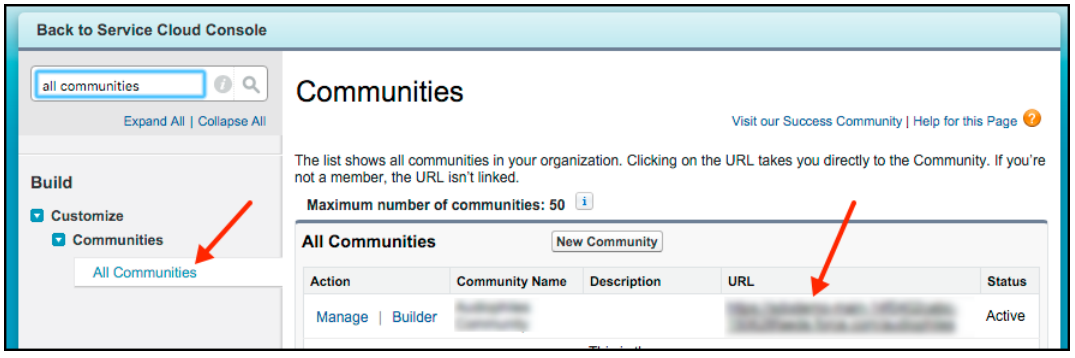
For more thorough instructions, see Guest User Access for Your Community.

Note: Once you've set up your knowledge base and your community, supply your app developer with three values: the community URL, the data category group, and the root data category. 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'S SETUP:

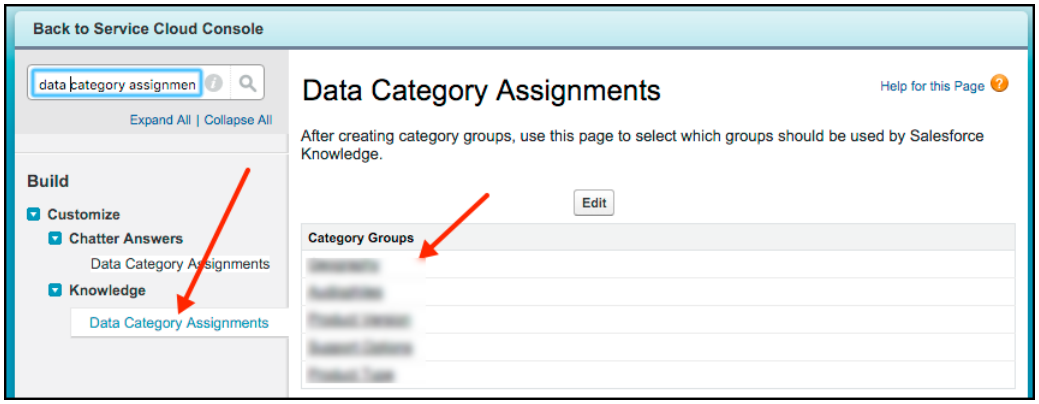
Community URL

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



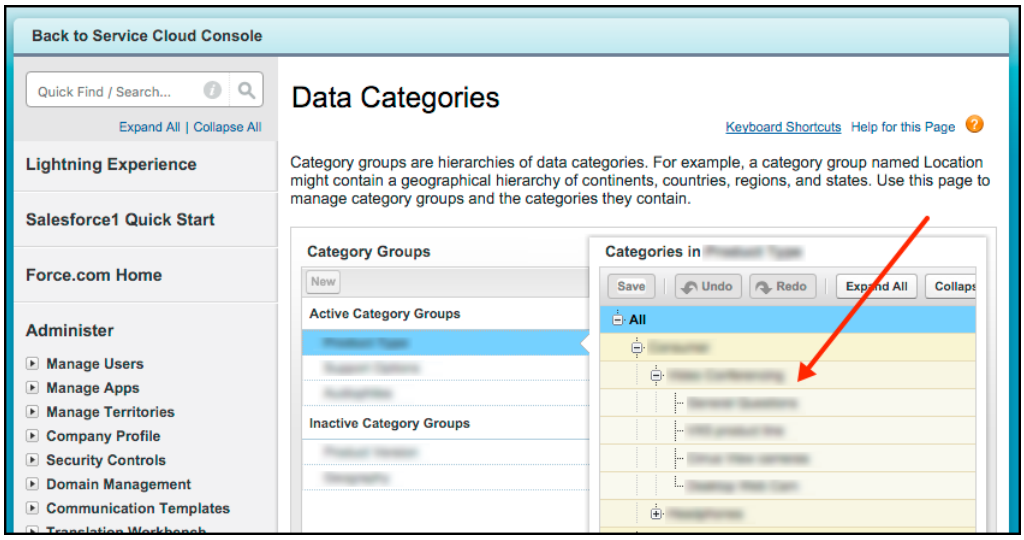
Data Category Group

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

To show Knowledge articles from your app, enable guest user access for the **Article Types**, **Categories**, and **Fields** associated with your knowledge articles.

Guest User Access 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.

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 **Public Access Settings**. This action displays the guest user profile in your org.
4. Select **Data Category Visibility** and grant visibility to the categories that you want users to see.
5. 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.

Console Setup for SOS

Set up Omni-Channel and SOS to use SOS in your app.

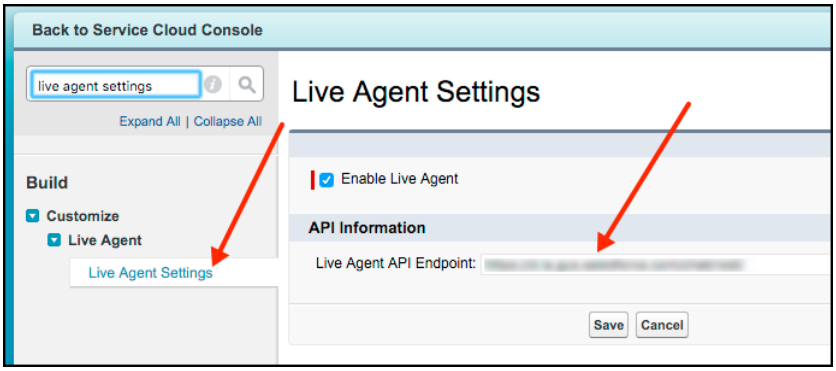
You can either use the quick setup or configure your org manually. The quick setup (Quick Setup: SOS Console) is best to get started the first time; manual setup (Manual Setup: SOS Console) is appropriate if you want to customize your org for production.

Note: Once 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. 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'S SETUP:

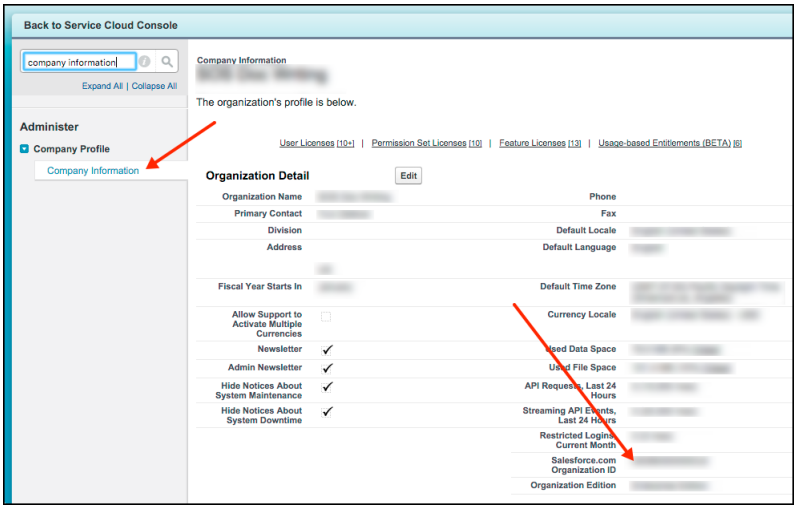
pod

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**.



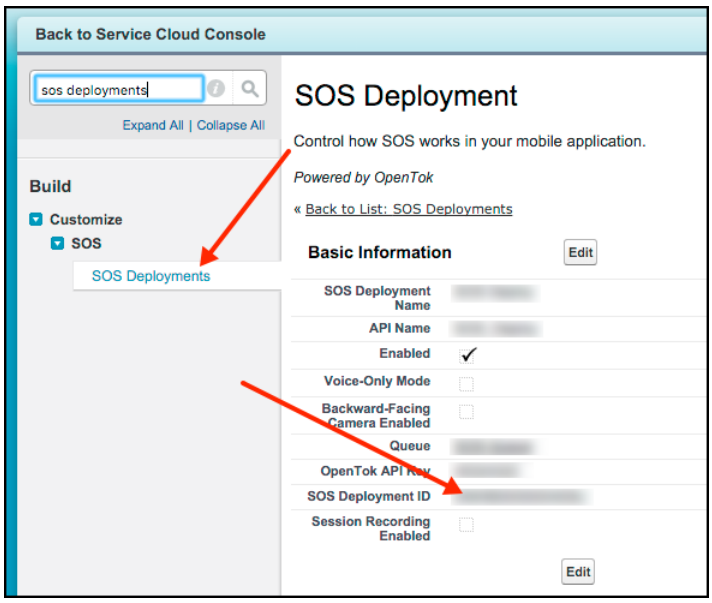
orgId

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 Setup Options: SOS Console

Fine-tune your SOS configuration with additional customizations.

Quick Setup: SOS Console

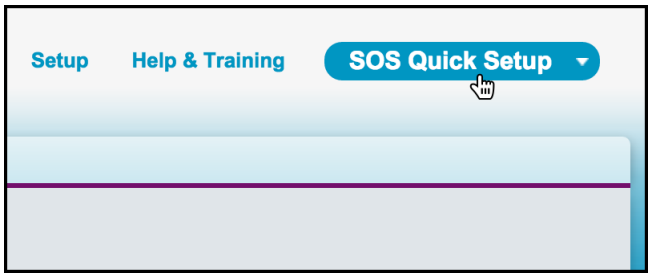
Quick setup is great when you want to try SOS for the first time and you haven't already enabled Omni-Channel or SOS in your org.

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

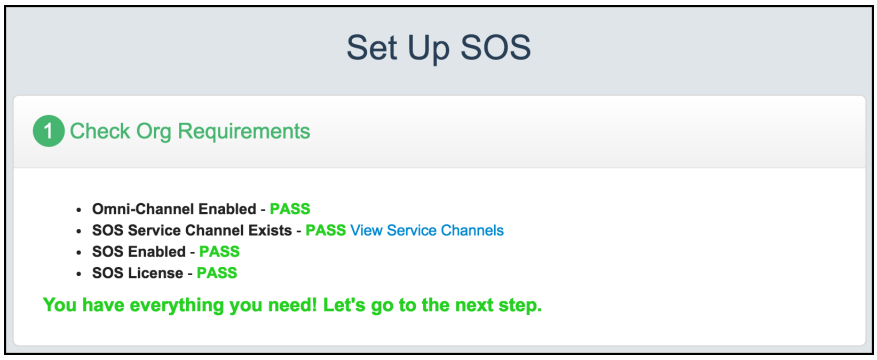
To simplify the configuration for SOS Service Cloud, we have included an easy-to-use SOS quick setup wizard.

1. Log in to your org and select the **SOS Quick Setup** app.



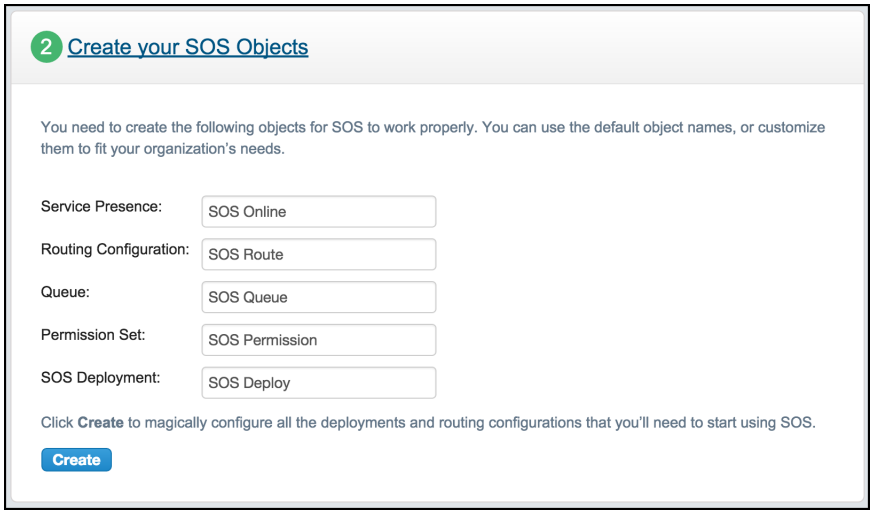
2. Select **Check Org Requirements**.

   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

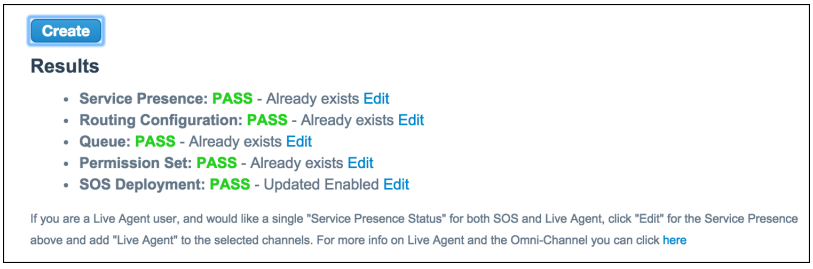
Contains the app permissions to enable the SOS license and to provide access to the Service Presence status. This object must be used to enable the license on a user. All members of the permission set must be assigned an SOS license. It can also be used to enable the Service Presence status.

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.

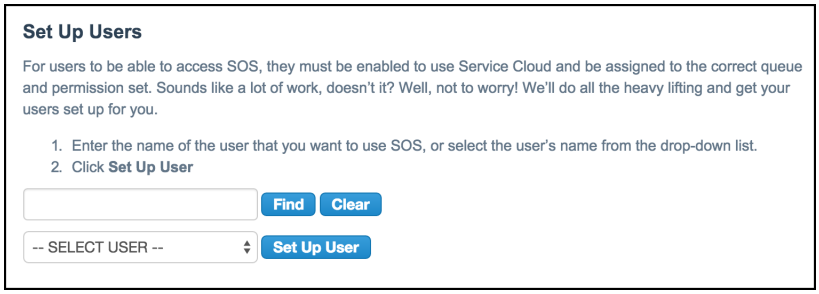
4. Click **Create**.

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



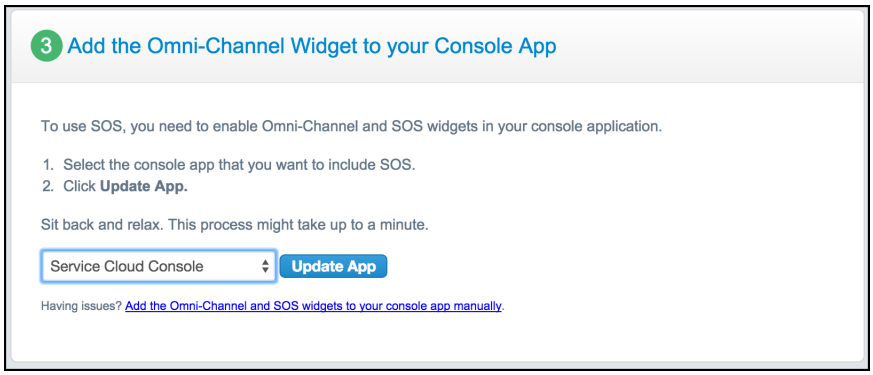
5. Add agents to the **Set Up Users** section.

   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**.

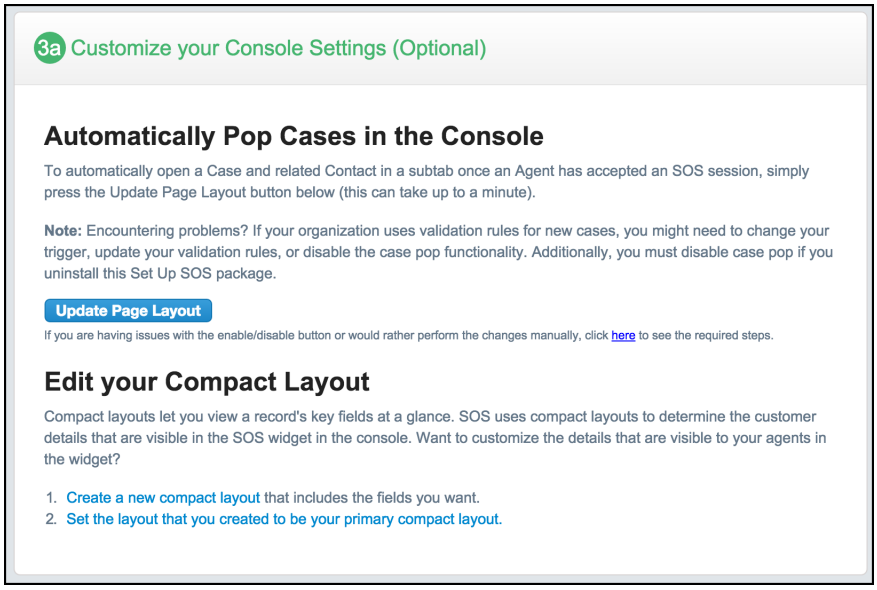


   This step adds the following to your Console App:

   - 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.

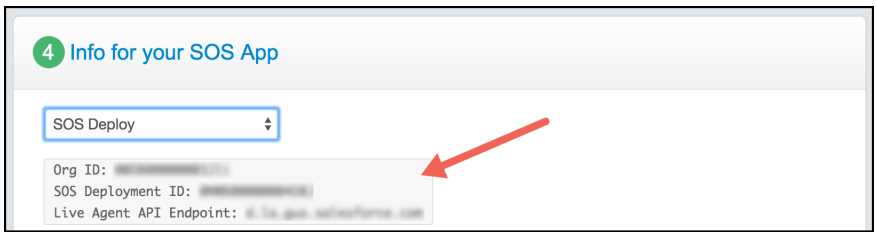


The case is created by a trigger included with the package. Before the new SOS session is inserted, the trigger creates a case and adds a reference to it to the SOS session object. If you wish to modify the trigger, it can be found in **Setup**, by searching **SOS Sessions** and going to **Triggers**.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

Additional Setup Options: SOS Console

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.

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.

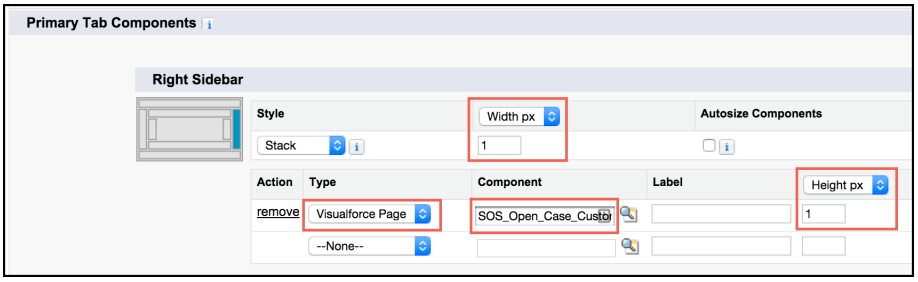
a. From **Setup**, search for **SOS Sessions**.

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:



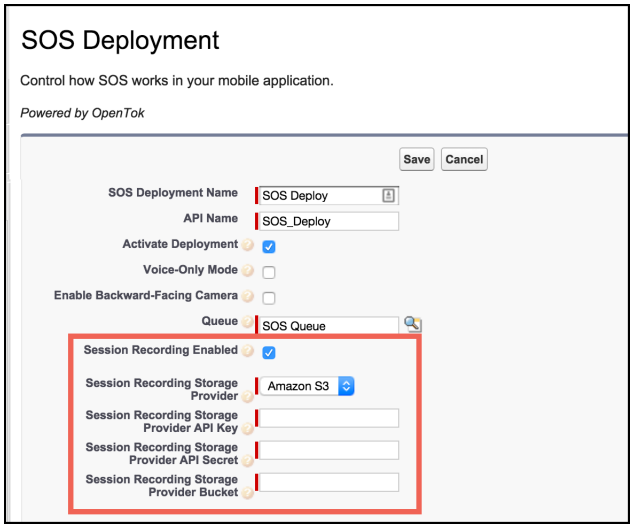
- Set **Style** to **Stack**.
- Set **Width px** to **1**.
- Set **Height px** to **1**. (Change **Height %** to **Height px** if necessary.)
- Set **Type** to **Visualforce Page**
- Set **Component** to the page created previously (for example, **SOS_Open_Case_Custom**).

f. Click **Save**.

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 Requirements

This SDK requires Android API level 16 (4.1, JELLY_BEAN) or above.

Knowledge Beta API Requirements

If you're using the Knowledge feature on an Android device, you must be in the **Snap-ins Beta**. This program enables Service SDK functionality in your org. Contact your Salesforce account team for more information.

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 Android SDK

Install the Service SDK for Android using Gradle.

Before running through these steps, be sure you've checked the Requirements.

Note: This release contains a beta version of the Knowledge SDK, which means it has high-quality features with known limitations.

To get started with the Service SDK for Android:

1. Install the SDK using Gradle.

   The Service SDK is hosted in a maven repository.

   To install **Knowledge** functionality, add the following maven repository and compile-time dependency to your `build.gradle` file.

```
repositories {
  maven {
    url 'https://salesforcesos.com/android/maven/release'
  }
}

dependencies {
  compile 'com.salesforce.service:knowledge-ui:0.4.0'
}
```

   To install **SOS** functionality, add the following maven repositories and compile-time dependency to your `build.gradle` file.

```
repositories {
    maven {
        url 'https://salesforcesos.com/android/maven/release'
    }
    maven {
        url 'http://tokbox.bintray.com/maven/'
    }
}

dependencies {
  compile 'com.salesforce.service:sos:2.1.0'
}
```

   To install **all** the Service SDK features, add the following maven repositories and compile-time dependencies to your `build.gradle` file.

```
repositories {
    maven {
        url 'https://salesforcesos.com/android/maven/release'
    }
    maven {
        url 'http://tokbox.bintray.com/maven/'
    }
}

dependencies {
  compile 'com.salesforce.service:knowledge-ui:0.4.0'
  compile 'com.salesforce.service:sos:2.1.0'
}
```

2. Declare permissions.

To install **Knowledge** functionality, you must declare the following permissions in your `AndroidManifest.xml`.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

To install **SOS** functionality, you must declare the following permissions in your `AndroidManifest.xml`.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

If you wish to use **two-way video** with SOS, you must also declare the camera permission.

```
<uses-permission android:name="android.permission.CAMERA"/>
```

You can now start using the Service SDK for Android.

Analytics

You can listen to user-driven events from the Service SDK using the `ServiceAnalytics` system. This system works only for Knowledge UI activities.

Implement `ServiceAnalyticsListener` to start receiving events.

```
ServiceAnalytics.addListener(new ServiceAnalyticsListener() {
  @Override public void onServiceAnalyticsEvent(String behaviorId,
                                                Map<String, Object> eventData) {
    // TO DO: Do something with analytics data
  }
});
```

When you receive an event, inspect the `behaviorId` to see the behavior that caused the event (for example, `KNOWLEDGE_UI_USER_LAUNCH`). Inspect the `eventData` map for contextual data related to the event (for example, `KnowledgeUIAnalytics.DATA_CATEGORY_GROUP_NAME`). For a list of behaviors and the key constants for parsing the `eventData` map, see the `KnowledgeUIAnalytics` class in the Reference Documentation.

Using Knowledge

Adding the Knowledge experience to your app.

Knowledge Overview

Learn about the Knowledge experience using the SDK.

Quick Setup: Knowledge

It's easy to get started with Knowledge for your Android app. To set up Knowledge, create a configuration object that points to your knowledge base and then create a Knowledge UI client.

Cache Images for Offline Access

By default, articles are cached for offline access, but the associated images are not. However, you can use the `OfflineResourceConfig` class to cache image content.

Provide Images for Categories and Articles

By implementing the `KnowledgeImageProvider`, you can provide images for use with your Knowledge categories and articles. Category images appear behind the category name when viewing a category list and they appear on the category description view. Article images appear in article lists and at the top of the article detail view.

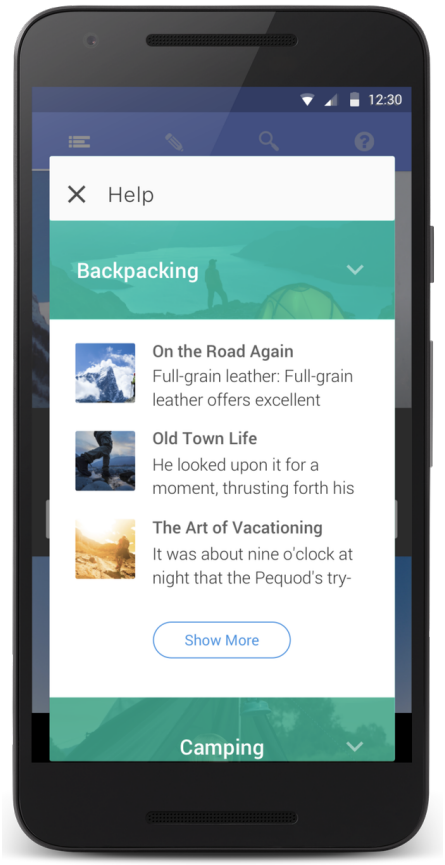
Article Fetching with the Knowledge Core API

If you want more control over the interface, you can work with the Knowledge Core API. This API lets you query your community for categories and articles and handle them as objects. You can then display the contents of these objects in your own custom UI.

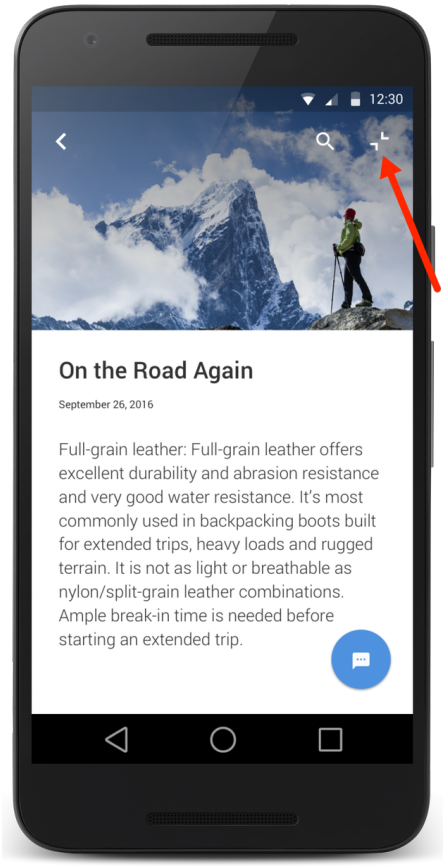
Knowledge Overview

Learn about the Knowledge experience using the SDK.

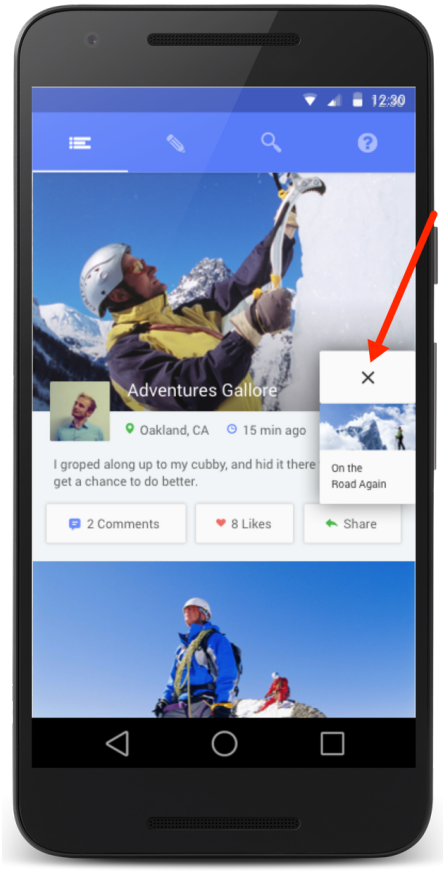
The Knowledge feature in the SDK gives you access to your org's knowledge base directly from within your app. Once you point your app to your community URL with the right category group and root data category, you can display your knowledge base to your users.



From the knowledge home, a user can navigate through articles that are organized by category. Articles are also searchable from within the app. When a user views an article, they can minimize it using the minimize button at the top right of the article so that they can continue to navigate your app.



Once minimized, the user can drag the article thumbnail 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.

Once you've configured Knowledge to work within your app, customize the color branding and the strings of the interface by using our UI Customization guidelines.

There are other ways to customize the Knowledge experience in your app. If you want to design your own UI, see our Knowledge Core API to access knowledge content directly. You also have finer-grained control over how images are cached and displayed, using Cache Images for Offline Access and Provide Images for Categories and Articles.

Let's get started.

Quick Setup: Knowledge

It's easy to get started with Knowledge for your Android app. To set up Knowledge, create a configuration object that points to your knowledge base and then create a Knowledge UI client.

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 Android SDK.

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

1. Import the classes you'll be using for Knowledge.

```
// Add these imports to your code...
import com.salesforce.android.knowledge.core.KnowledgeConfiguration;
import com.salesforce.android.service.common.utilities.control.Async;
```

```
import com.salesforce.android.knowledge.ui.KnowledgeUI;
import com.salesforce.android.knowledge.ui.KnowledgeUIConfiguration;
import com.salesforce.android.knowledge.ui.KnowledgeUIClient;
```

2. Specify the community URL, the data category group, and the root data category that you want to use as the starting point for the Knowledge user experience.

```
// Specify the community url, category group, and root category
String communityUrl = "https://your-community-url";
String categoryGroup = "your-category-group";
String rootCategory = "your-root-category";
```

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

3. Create a `KnowledgeConfiguration` object using the community URL and the data category group.

```
// Create a core configuration instance
KnowledgeConfiguration coreConfiguration =
  KnowledgeConfiguration.create(communityUrl);
```

   Note: By default, knowledge articles are cached locally but images are not cached. For guidance on setting up a configuration that caches images, see Cache Images for Offline Access.

4. Create a `KnowledgeUIConfiguration` instance using the configuration object and the root category to use as the starting place for the knowledge base.

```
// Create a UI configuration instance from a core instance
KnowledgeUIConfiguration uiConfiguration =
  KnowledgeUIConfiguration.create(coreConfiguration, categoryGroup, rootCategory);
```

5. (Optional) Provide images for categories and articles.

   If you'd like to provide images for Knowledge categories and Knowledge articles, implement the `KnowledgeImageProvider` interface and pass it to the `KnowledgeUIConfiguration` object.

```
// Specify image provider
uiConfiguration.setImageProvider(new MyImageProvider());
```

   Category images appear behind the category name when viewing a category list and they appear on the category description view. Article images appear in article lists and at the top of the article detail view.

   To learn more about `KnowledgeImageProvider`, see Provide Images for Categories and Articles.

6. (Optional) Perform any customizations to the interface.

   You can customize the colors, the strings, and other aspects of the interface. You can also localize the strings into other languages. To learn more about customizations, see UI Customizations.

7. Create a `KnowledgeUIClient` instance, show the UI, and maintain a reference to this instance.

   Note: If you'd like to design your own UI and access Knowledge objects directly, see Article Fetching with the Knowledge Core API.

   To start the Knowledge UI, call the static `KnowledgeUI.configure` method, which creates a `KnowledgeUI` instance. From this instance, create a client (asynchronously) with the `createClient` method. This asynchronous call gives you a `KnowledgeUIClient` instance. You can start the UI using the `launch` method. Be sure to maintain a reference to this instance

for as long as you want to use the Knowledge UI. You can remove your reference to this instance using `OnCloseListener` as shown in the code snippet.

```
// Create a client asynchronously (passing 'myContext' for the Context)
KnowledgeUI.configure(uiConfiguration).createClient(myContext)
  .onResult(new Async.ResultHandler<KnowledgeUIClient>() {

    @Override public void handleResult (Async<?> operation,
      KnowledgeUIClient uiClient) {

        // Maintain a reference to the UI client within your Application lifecycle
        mKnowledgeUIClient = uiClient;

        // Handle the close action
        uiClient.setOnCloseListener(new KnowledgeUIClient.OnCloseListener() {

          @Override public void onClose () {

            // Clear reference to the Knowledge UI client
            mKnowledgeUIClient = null;
          }

        });

        // Launch the UI (passing 'myActivity' for the activity)
        uiClient.launch(myActivity);
    }
});
```

Make sure that you only call the `launch` method one time. If you call it multiple times, multiple UIs will appear.

Example:

```
// Add these imports to your code...
import com.salesforce.android.knowledge.core.KnowledgeConfiguration;
import com.salesforce.android.service.common.utilities.control.Async;
import com.salesforce.android.knowledge.ui.KnowledgeUI;
import com.salesforce.android.knowledge.ui.KnowledgeUIConfiguration;
import com.salesforce.android.knowledge.ui.KnowledgeUIClient;

public class MyActivity extends Activity {

  // You should normally maintain a reference to the UI
  // client within the Application lifecycle...
  // Putting it in the Activity just for this example.
  KnowledgeUIClient mKnowledgeUIClient = null;

  // Specify the community url, category group, and root category
  String communityUrl = "https://your-community-url";
  String categoryGroup = "your-category-group";
  String rootCategory = "your-root-category";
```

```
  // Create a core configuration instance
  KnowledgeConfiguration coreConfiguration =
    KnowledgeConfiguration.create(communityUrl);

  // Create a UI configuration instance from core instance
  KnowledgeUIConfiguration uiConfiguration =
    KnowledgeUIConfiguration.create(coreConfiguration, categoryGroup, rootCategory);

  // Call this method when you want to show the Knowledge UI
  private void startKnowledge() {

    // Create a client asynchronously
    KnowledgeUI.configure(uiConfiguration).createClient(MyActivity.this)
      .onResult(new Async.ResultHandler<KnowledgeUIClient>() {

        @Override public void handleResult (Async<?> operation,
          KnowledgeUIClient uiClient) {

            // Store reference to the Knowledge UI client
            mKnowledgeUIClient = uiClient;

            // Handle the close action
            uiClient.setOnCloseListener(new KnowledgeUIClient.OnCloseListener() {

              @Override public void onClose () {

                // Clear reference to the Knowledge UI client
                mKnowledgeUIClient = null;
              }
            });

            // Launch the UI
            uiClient.launch(MyActivity.this);
        }
    });
  }

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
}
```

Cache Images for Offline Access

By default, articles are cached for offline access, but the associated images are not. However, you can use the `OfflineResourceConfig` class to cache image content.

You can store and retrieve cached images by creating an `OfflineResourceConfig` instance and passing it to the `KnowledgeConfiguration` instance during the setup process. See Quick Setup: Knowledge for setup instructions using the default UI and Article Fetching with the Knowledge Core API for direct access to knowledge objects.

The `OfflineResourceConfig` class allows you to specify the maximum cache size and the number of concurrent requests using the builder design pattern. For example:

```
import com.salesforce.android.knowledge.core.offline.OfflineResourceConfig;

...

OfflineResourceConfig offlineConfig = OfflineResourceConfig.fromContext(this)
  .maxSize(60 * 1024 * 1024)   // 60 MB cache
  .concurrentRequests(8)       // 8 concurrent requests
  .build();
```

Once you've instantiated an `OfflineResourceConfig`, you can pass that instance to `KnowledgeConfiguration` using the `offlineResourceConfig` method during construction. For example:

```
KnowledgeConfiguration.builder(communityUrl).offlineResourceConfig(offlineConfig).build();
```

From here, you can proceed with the instructions for setting up the default Knowledge UI (Quick Setup: Knowledge) or for direct access to the Knowledge objects (Article Fetching with the Knowledge Core API).

When using `OfflineResourceConfig`, images are automatically cached locally. When the cache reaches its max size, oldest content is removed.

Provide Images for Categories and Articles

By implementing the `KnowledgeImageProvider`, you can provide images for use with your Knowledge categories and articles. Category images appear behind the category name when viewing a category list and they appear on the category description view. Article images appear in article lists and at the top of the article detail view.

The `KnowledgeImageProvider` interface has one method for article images and one method for category images:

```
public interface KnowledgeImageProvider {
  Drawable getImageForArticle (Context context, ArticleSummary article);
  Drawable getImageForDataCategory (Context context, DataCategorySummary dataCategory);
}
```

The SDK calls these methods whenever it attempts to load an image for a given data category or article. `getImageForArticle` passes you an `ArticleSummary` object and `getImageForDataCategory` passes you a `DataCategorySummary` object.

You'll need a way to correlate articles and categories with images. Data categories can be uniquely identified by using `DataCategorySummary.getName()` to get the category name (for example, "Asia"); articles can be uniquely identified by using `ArticleSummary.getArticleNumber()` to get the article ID (for example, "000005256").

Once you've implemented the `KnowledgeImageProvider` interface, pass an instance of your image provider class to the Knowledge UI configuration object.

```
KnowledgeUIConfiguration uiConfiguration =
  KnowledgeUIConfiguration.create(coreConfiguration, categoryGroup, rootCategory)
    .setImageProvider(new MyImageProvider());
```

Refer to Quick Setup: Knowledge for more information about starting the Knowledge interface.

Example: Since the `KnowledgeImageProvider` interface does not specify how the images are sourced, this example illustrates how to load them from disk as project assets. For performance reasons, this example uses Android's `LruCache` API for efficient loading of assets for future requests. This method helps avoid high UI latency and excess memory consumption.

```
/**
 * KnowledgeImageProvider implementation that loads images used by Knowledge UI.
 * This implementation assumes that images are stored as PNG files in the assets
 * directory of the project. Data Category images are stored in the "categories"
 * subfolder with the category name as the file name. Article images are stored
 * in the "articles" subfolder with the article number as the file name.
 */
public class ImageProvider implements KnowledgeImageProvider {

  private final LruCache<String, BitmapDrawable> mCategoryImageCache;
  private final LruCache<String, BitmapDrawable> mArticleImageCache;

  ImageProvider () {
    // The maximum amount of memory available to the application in kilobytes.
    final int maxMemory = ((int) Runtime.getRuntime().maxMemory() / 1024);

    // Use 1/8th the maximum available memory to cache category images. Article
    // images can use less because we don't display as many of them at once.
    final int categoryCacheSize = maxMemory / 8;
    final int articleCacheSize = maxMemory / 16;

    // Create an LruCache that is bounded by the maximum available amount of memory.
    mCategoryImageCache = makeCache(categoryCacheSize);
    mArticleImageCache = makeCache(articleCacheSize);
  }

  //--------------------------------------------------------------------------
  // KnowledgeImageProvider API
  //--------------------------------------------------------------------------

  @Override public Drawable getImageForArticle (Context context,
                                                ArticleSummary article) {
    String name = "articles/" + article.getArticleNumber() + ".png";
    return getDrawable(context, name, mArticleImageCache);
  }

  @Override
  public Drawable getImageForDataCategory (Context context,
                                           DataCategorySummary dataCategory) {
    String name = "categories/" + dataCategory.getName().toLowerCase() + ".png";
    return getDrawable(context, name, mCategoryImageCache);
  }
```

```
  //--------------------------------------------------------------------------------
  // Helpers
  //--------------------------------------------------------------------------------

  private static LruCache<String, BitmapDrawable> makeCache (int size) {
    return new LruCache<String, BitmapDrawable>(size) {
      @Override protected int sizeOf (String key, BitmapDrawable drawable) {
        return drawable.getBitmap().getByteCount() / 1024;
      }
    };
  }

  private static BitmapDrawable getDrawable (Context context, String name,
                                             LruCache<String, BitmapDrawable> cache)
{
    // Try to fetch from the in-memory cache.
    BitmapDrawable drawable = cache.get(name);

    // If it's not there then load it from disk.
    if (drawable == null) {
      drawable = loadBitmapFromDisk(context, name);
      if (drawable != null) {
        cache.put(name, drawable);
      }
    }

    return drawable;
  }

  private static BitmapDrawable loadBitmapFromDisk (Context context, String name) {
    AssetManager assets = context.getAssets();

    InputStream stream = null;
    try {
      stream = assets.open(name);

      // Lower the sample size so the images take less memory.
      BitmapFactory.Options options = new BitmapFactory.Options();
      options.inSampleSize = 2;

      Bitmap bitmap = BitmapFactory.decodeStream(stream, null, options);
      return new BitmapDrawable(context.getResources(), bitmap);
    } catch (IOException e) {
      return null;
    } finally {
      if (stream != null) {
        try {
          stream.close();
        } catch (IOException e) {
          // Log the stream closing error.
        }
      }
    }
```

```
    }
}
```

Article Fetching with the Knowledge Core API

If you want more control over the interface, you can work with the Knowledge Core API. This API lets you query your community for categories and articles and handle them as objects. You can then display the contents of these objects in your own custom UI.

To access Knowledge Core objects, first instantiate a `KnowledgeCore` client by calling the `createClient(...)` method.

```
// Create a core configuration instance
String communityUrl = "https://your.community.url.force.com";
KnowledgeConfiguration config = KnowledgeConfiguration.create(communityUrl);

// Create a Knowledge Core object
KnowledgeCore.configure(config).createClient(getContext())
  .onResult(new Async.ResultHandler<KnowledgeClient>() {
    @Override public void handleResult (Async<?> operation, @NonNull KnowledgeClient client)
 {

      // TO DO: Handle the client instance.
    }
  }
);
```

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

Once you've launched the Knowledge Core client, you can query for categories and articles.

Submitting Queries

You can query for data by calling the overloaded `KnowledgeClient.submit(...)` method with a request object that contains the criteria for your specific request. The submit method asynchronously returns an object representing the data your requested.

Table 1: Submitting a Request

Request Object	Response Type	Description
DataCategoryGroupListRequest	DataCategoryGroupList	Represents a list of data category groups, which are groups that contain a hierarchy of data categories within them.
DataCategoriesRequest	DataCategoryList	A hierarchical list of data categories.
ArticleListRequest	ArticleList	A list of articles with their summaries.
ArticleDetailRequest	ArticleDetails	Complete contents of an article.

You create a request by using the builder method on a particular request object.

To request a list of the high-level **data category groups** within your community:

```
DataCategoryGroupListRequest request = DataCategoryGroupListRequest.builder().build();
```

To request a single data category group by name ("Travel" in this case):

```
DataCategoryGroupRequest.builder("Travel").build();
```

To request a list of **data categories** within a data category group or within a parent data category, build a `DataCategoriesRequest` object with the name of the data category group *and* the name of a data category:

```
String categoryGroup = "Travel";
String rootCategory = "Canada";
DataCategoriesRequest request =
  DataCategoriesRequest.builder(categoryGroup, rootCategory).build();
```

You can also use the term "All" to grab all subcategories:

```
DataCategoriesRequest request = DataCategoriesRequest.builder(categoryGroup, "All").build();
```

When querying for **articles**, you have a few ways to optionally refine your query:

```
DataCategorySummary
  canadaDataCategorySummary = // Obtained from DataCategoryList.getDataCategories()
ArticleListRequest request = ArticleListRequest.builder()
    .dataCategory(categoryGroup, canadaDataCategorySummary.getName())
    .pageNumber(1)
    .pageSize(10)
    .searchTerm("Hockey")
    .build();
```

Specifically, you can use the following query parameters:

pageNumber
The page number of the results. This value is 1-based. Default value is 1.

pageSize
The number of articles per page. Default value is 3.

searchTerm
The search term used to refine the search. By default, no search term is used.

sortBy
What field you want to use for sorting. Possible values are: `SORT_BY_LAST_PUBLISHED_DATE`, `SORT_BY_TITLE`, and `SORT_BY_VIEW_SCORE`. Default value is `SORT_BY_LAST_PUBLISHED_DATE`.

sortOrder
Whether you want to sort results in descending (`SORT_DESC`) or ascending (`SORT_ASC`) order. Default is `SORT_DESC`.

The results are sorted by last modified date, in descending order.

Once you have an article list, you can use the article summary to request the **contents of an article**:

```
ArticleSummary
  hockeyInCanada = // Obtained from ArticleList.getArticles()
ArticleDetailRequest request = ArticleDetailRequest.builder(hockeyInCanada).build();
```

To submit a request, call the overloaded `KnowledgeClient.submit(...)` method and asynchronously handle the query response. The following example illustrates how to submit a request once you've built a request object:

```
knowledgeClient.submit(request).onResult(new Async.ResultHandler<RESPONSE_TYPE_GOES_HERE>()
 {

  @Override
```

```
  public void handleResult (Async<?> operation, @NonNull RESPONSE_TYPE_GOES_HERE result)
{
    // TO DO: Inspect the response object
  }

}).onComplete(new Async.CompletionHandler() {
  // TO DO: Handle completion
});
```

Caching Query Results

Every request object has a `cacheResults` builder method, which you can use to **cache the results** of the query. By default, this value is set to `true`. Although the article text is cached, images are not automatically cached. To **cache the images** for offline access, you'll need to use `OfflineResourceConfig`. To learn more, see Cache Images for Offline Access.

Handling Queries

Handle the result of a request using the `Async<T>` interface and its associated handler interfaces: `ResultHandler`, `CompletionHandler`, and `ErrorHandler`. Alternatively you can implement the broad `Handler` interface which itself inherits from all three interfaces. For convenience, the `Async<T>` methods that accept handlers as parameters return the `Async<T>` instance so you can chain the syntax.

Keep in mind that a `ResultHandler` instance can be called multiple times before the `Async<T>` operation is complete, and some of the results can be duplicates. All Knowledge objects have unique IDs so you can check for duplicates.

```
public interface Async<T> {
  Async<T> onResult(ResultHandler<? super T> handler);
  Async<T> onComplete(CompletionHandler handler);
  Async<T> onError(ErrorHandler handler);
  Async<T> removeResultHandler (ResultHandler<? super T> handler);
  Async<T> removeCompletionHandler (CompletionHandler handler);
  Async<T> removeErrorHandler (ErrorHandler handler);
  void cancel();
  boolean inProgress();
  boolean isComplete();
  boolean isCancelled();
  boolean hasFailed();

  interface ResultHandler<T> {
    void handleResult (Async<?> operation, @NonNull T result);
  }

  interface CompletionHandler {
    void handleComplete (Async<?> operation);
  }

  interface ErrorHandler {
    void handleError (Async<?> operation, @NonNull Throwable throwable);
  }

  interface Handler<T> extends ResultHandler<T>, CompletionHandler, ErrorHandler {
```

```
  }
}
```

Examples

Fetching **data category groups**:

```
DataCategoryGroupListRequest dataCategoryGroupRequest =
  DataCategoryGroupListRequest.builder().build();
knowledgeClient.
  submit(dataCategoryGroupRequest).onResult(new Async.ResultHandler<DataCategoryGroupList>()
 {

  @Override
  public void handleResult (Async<?> operation, @NonNull DataCategoryGroupList result) {
    // The result is a DataCategoryGroupList objects, which contains DataCategoryGroup
objects
  }
}).onComplete(new Async.CompletionHandler() {

  @Override public void handleComplete (Async<?> operation) {
    // Called when the operation is complete
  }
}).onError(new Async.ErrorHandler() {

  @Override public void handleError (Async<?> operation, @NonNull Throwable throwable) {
    // Called if an error has been encountered
  }
});
```

Fetching **data categories** within the "Canada" category of "Travel":

```
String categoryGroup = "Travel";
String rootCategory = "Canada";
DataCategoriesRequest dataCategoriesRequest =
  DataCategoriesRequest.builder(categoryGroup, rootCategory).build();
knowledgeClient.
  submit(dataCategoriesRequest).onResult(new Async.ResultHandler<DataCategoryList>() {

  @Override public void handleResult (Async<?> operation, @NonNull DataCategoryList result)
 {
    // The result is a DataCategoryList object, which contains a list of DataCategorySummary
 objects
  }
}).onComplete(new Async.CompletionHandler() {

  @Override public void handleComplete (Async<?> operation) {
    // Called when the operation is complete
  }
}).onError(new Async.ErrorHandler() {

  @Override public void handleError (Async<?> operation, @NonNull Throwable throwable) {
    // Called if an error has been encountered
```

```
  }
});
```

Fetching **articles** within a data category:

```
String categoryGroup = "Travel";
String categoryName = "Canada";
ArticleListRequest articleListRequest = ArticleListRequest.builder()
    .dataCategory(categoryGroup, categoryName)
    .pageNumber(1)
    .pageSize(10)
    .queryMethod(ArticleListRequest.QUERY_BELOW)
    .searchTerm("Hockey")
    .build();
knowledgeClient.submit(articleListRequest).onResult(new Async.ResultHandler<ArticleList>()
 {

  @Override public void handleResult (Async<?> operation, @NonNull ArticleList result) {
    // The result is an ArticleList object which contains a list of ArticleSummary objects

  }
}).onComplete(new Async.CompletionHandler() {

  @Override public void handleComplete (Async<?> operation) {
    // Called when the operation is complete
  }
}).onError(new Async.ErrorHandler() {

  @Override public void handleError (Async<?> operation, @NonNull Throwable throwable) {
    // Called if an error has been encountered
  }
});
```

Fetching the **contents of an article**:

```
ArticleSummary articleSummary = // Obtained from ArticleList.getArticles()
ArticleDetailRequest articleDetailRequest =
ArticleDetailRequest.builder(articleSummary).build();
knowledgeClient.submit(articleDetailRequest).onResult(new
Async.ResultHandler<ArticleDetails>() {

  @Override public void handleResult (Async<?> operation, @NonNull ArticleDetails result)
 {
    // The result is an ArticleDetails object that contains the contents of the article
  }
}).onComplete(new Async.CompletionHandler() {

  @Override public void handleComplete (Async<?> operation) {
    // Called when the operation is complete
  }
}).onError(new Async.ErrorHandler() {

  @Override public void handleError (Async<?> operation, @NonNull Throwable throwable) {
    // Called if an error has been encountered
```

```
    }
});
```

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

To start an SOS session, call the `SOS` class from an activity. Once the session is started, SOS automatically handles activity transitions for you.

Configure an SOS Session

Before starting an SOS session, you can optionally configure the session using an `SosConfiguration` object. This object allows you to enable or disable cameras, determine what screen a session starts on, and control other features.

Control an SOS Session

During an SOS session, you can control the session using various static methods on the `Sos` class. These methods can pause the session, disable screen sharing, stop the session, change what is being shown on the screen, and perform other functions.

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.

Check Agent Availability

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

Listen to Events

The SOS SDK provides listeners that allow you to listen to and respond to various events emitted during an SOS session. These listeners can be used to keep your application informed about the state of the SOS session or to implement custom behavior based on the SOS session progress.

Detect the Keyboard

Because the Android OS manages the keyboard, its representation on the screen is inaccessible to an application. This situation presents a challenge when sending an image of the device to an agent during an SOS session. For this reason, you should explicitly pass along information about the keyboard.

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.

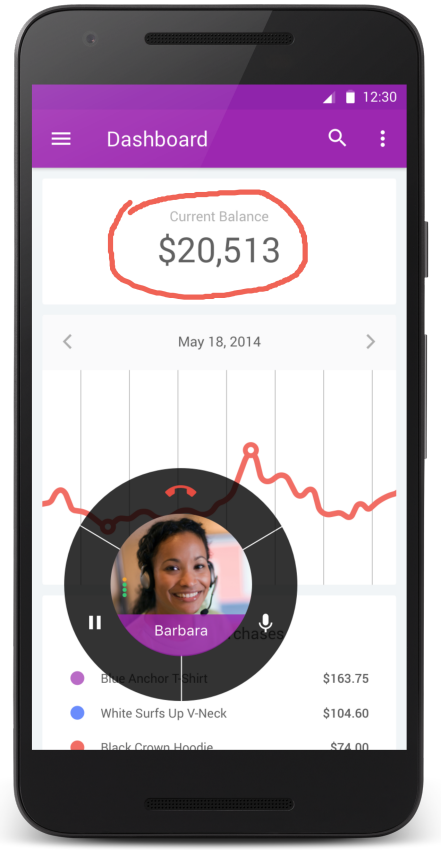
Custom Data

Use custom data to identify customers, send error messages, issue descriptions, or identify the page the SOS session was initiated from.

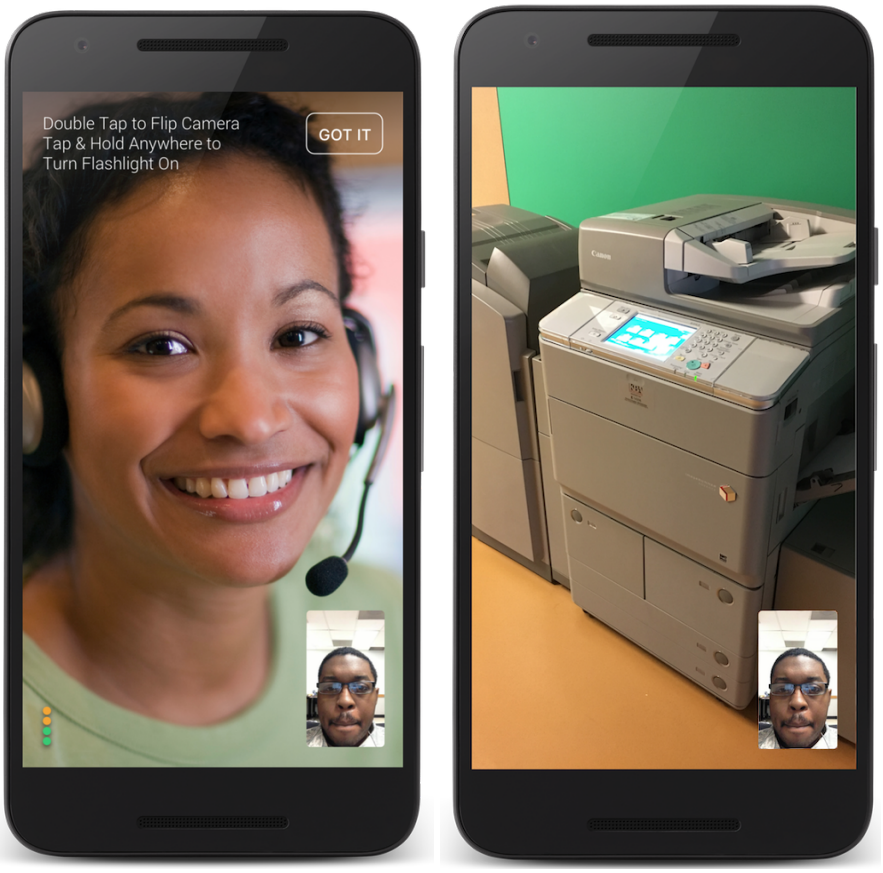
SOS Overview

Learn about the SOS experience using the SDK.

SOS lets you easily add real-time video and screen sharing support to your native Android 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. And if you want to handle state changes and other events, check out Listen to Events.

Once you've configured SOS to work within your app, customize the look and feel of the interface so that it fits naturally within your app by using our UI Customization guidelines.

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.

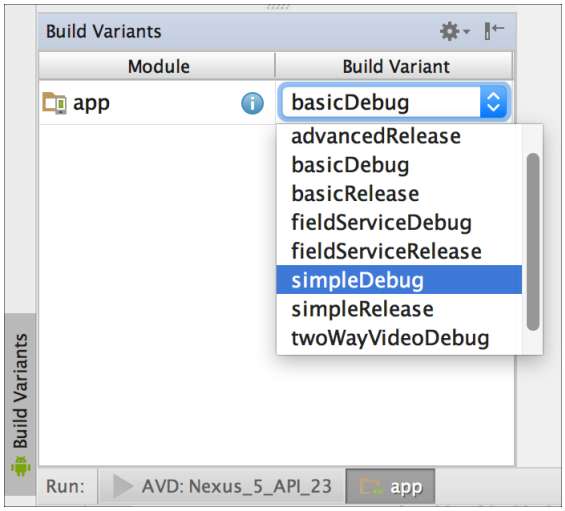
1. Clone the git repository where the example project is located: https://github.com/goinstant/android-sdk-guides.

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

2. Open the project in Android Studio.
3. Open the `SOSSettings.properties` file located in the `assets` folder, 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.

4. From the **Build Variants** toolbar in Android Studio, select the build variant.



This project contains several build variants to illustrate different ways of taking advantage of the SOS SDK.

simple
: Demonstrates connecting to an SOS session with minimal integration. This variant uses the default SOS experience without any customizations.

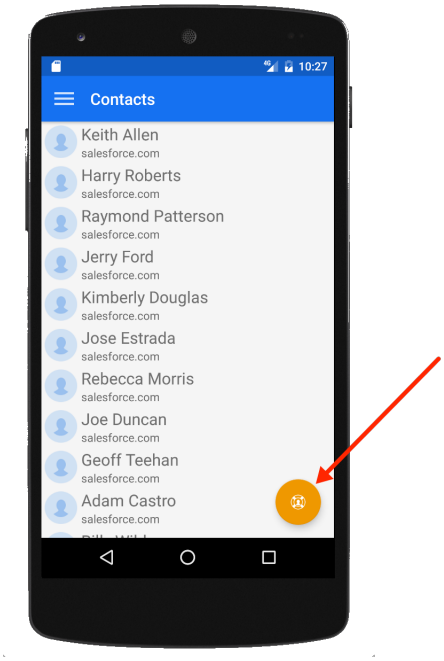
basic
: Demonstrates a basic configuration of the SOS experience using resource files to override the SOS library resources. This variant illustrates an easy mechanism for changing the default content without affecting behavior.

advanced
: Overrides portions of the SOS logic with custom code. This variant illustrates how to implement more customizations and demonstrates branding colors by overriding XML resources. It has a higher integration cost and requires an increased level of responsibility for the application developer.

twoWayVideo
: Enables the two-way video feature in SOS. With this variant, the user is presented with a camera toggle button to switch between video and screen sharing.

fieldService
: Launches the SOS session in field services mode, in which only the front and back cameras are available. Screen sharing is disabled.

When you run the app, a Contacts UI appears with a floating action button in the bottom right of the display. Tap this button to start an SOS session.



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

Quick Setup: SOS

To start an SOS session, call the `SOS` class from an activity. Once the session is started, SOS automatically handles activity transitions for you.

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 Android SDK.

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

1. Create an `Activity` subclass.

   Create a class that subclasses the `Activity` class.

2. Import the classes you'll be using for SOS.

```
// Add these imports to your code...
import com.salesforce.android.sos.api.SosOptions;
import com.salesforce.android.sos.api.Sos;
```

3. From within a method of the `Activity` object, create an `SosOptions` instance with information about your LiveAgent pod, your Salesforce org ID, and the deployment ID.

   For example:

```
// Specify your pod identifier, your org ID, and your deployment ID
String pod = "your-pod.com";
String orgId = "your-orgId";
String deploymentId = "your-deploymentId";
```

```
// Create an SosOptions object
SosOptions options = new SosOptions(pod, orgId, deploymentId);
```

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.

4. (Optional) Configure the SOS session before starting.

   Before starting an SOS session, you can optionally configure the session using an `SosConfiguration` object. This object allows you to enable or disable cameras, determine what screen a session starts on, and control other features. For more information about configuring a session, see Configure an SOS Session.

5. (Optional) Perform any customizations to the interface.

   You can customize the colors, the strings, and other aspects of the interface. You can also localize the strings into other languages. To learn more about customizations, see UI Customizations.

6. Call `start` on a new SOS session using the `SosOptions` object created earlier.

```
// Start an SOS session
Sos.session(options).start(this);
```

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:

```
// Add these imports to your code...
import com.salesforce.android.sos.api.SosOptions;
import com.salesforce.android.sos.api.Sos;

public class MyActivity extends Activity {

  // Call this method when you want to start an SOS session
  private void startSos() {

    // Specify your pod identifier, your org ID, and your deployment ID
    String pod = "your-pod.com";
    String orgId = "your-orgId";
    String deploymentId = "your-deploymentId";

    // Create an SosOptions object
    SosOptions options = new SosOptions(pod, orgId, deploymentId);

    // Start an SOS session
    Sos.session(options).start(this);
  }

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
}
```

Configure an SOS Session

Before starting an SOS session, you can optionally configure the session using an `SosConfiguration` object. This object allows you to enable or disable cameras, determine what screen a session starts on, and control other features.

`SosConfiguration` objects are constructed using `SosConfiguration.Builder`. For instance, the following code snippet enables a two-way video session:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .build();
```

Once you've built a configuration object, add it to the session that you're starting:

```
Sos.session(options)
  .configuration(config)
  .start(this);
```

You can use this configuration object for many different types of configuration settings.

Session Startup

By default there are several UIs that are presented to the user during the startup flow of an SOS session. This flow includes the device permissions UI, the onboarding UI, and the network test. Each of the sections of the startup flow can be disabled using the configuration builder.

Note: Disabling the permissions UI means that the application itself must acquire the necessary permissions before starting an SOS session.

To disable the permissions UI:

```
SosConfiguration config = SosConfiguration.builder()
  .permissionUi(false)
  .build();
```

To disable the onboarding UI:

```
SosConfiguration config = SosConfiguration.builder()
  .onboardingUi(false)
  .build();
```

To change the layout of the onboarding UI:

```
SosConfiguration config = SosConfiguration.builder()
  .onboardingCardLayouts(R.layout.onboardingCardOne, R.layout.onboardingCardTwo)
  .build();
```

To disable network tests:

```
SosConfiguration config = SosConfiguration.builder()
  .networkTestEnabled(false)
  .build();
```

Screen Sharing and Camera Configuration

By default, an SOS session begins by sharing the user's device screen with an agent. But you can also share the output from the device's cameras with an agent by enabling two-way video.

To enable two-way video:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .build();
```

To enter into "field services" mode, where screen sharing is disabled and SOS exclusively uses two-way video, use the following configuration:

```
SosConfiguration config = SosConfiguration.builder()
  .fieldServices(true)
  .build();
```

There are many other ways you can configure a two-way video session. For more info about two-way video, see Two-Way Video.

Fonts

Although most UI customization is done by overriding XML resources, changing the font that is used to display messages is done by specifying the font asset with `typefaceFromAsset`.

```
SosConfiguration config = SosConfiguration.builder()
  .typefaceFromAsset("fonts/myTypeface.ttf")
  .build();
```

Connecting UI

By default, connection messages appear on the SOS interface when a session connects. You can override this behavior and present your own UI during the connection process. To present your own UI for the connection process, disable the default UI at configuration time with the `connectingUi` method.

```
SosConfiguration config = SosConfiguration.builder()
  .connectingUi(false)
  .build();
```

Now you can handle state changes with your own UI. To learn more, see SOS: Customize the Connecting UI.

Sounds

By default, the user's default system notification sound is played when an SOS session is fully connected and the agent appears on the screen. It is possible to customize the sound by providing a URI, providing a resource ID for an embedded sound file, or by disabling sounds altogether.

To use an embedded sound file:

```
SosConfiguration config = SosConfiguration.builder()
  .agentJoinSound(R.raw.custom_agent_join_sound)
  .build();
```

To use a URI:

```
SosConfiguration config = SosConfiguration.builder()
  .agentJoinSound(Uri.parse("content://media/external/audio/media/710"))
  .build();
```

If you use a URI, we recommend using local or content provider paths only; a network resource may negatively impact the user experience.

To disable sound:

```
SosConfiguration config = SosConfiguration.builder()
  .playSounds(false)
  .build();
```

Debugging

When you are debugging and testing your application, it is often convenient to turn off some features. For instance, developing locally can cause an audio feedback loop between a device and your computer. If you are not working on audio features, you can use `audio` to disable all audio in your test sessions:

```
SosConfiguration config = SosConfiguration.builder()
  .audio(false) // No audio will ever be available in the session
  .build();
```

Another level of audio and video disabling is available using `agentPublish`. When set to `false` there is no creation, configuration, or use of the `OpenTok` agent. This is useful when using automated testing suites that do not have audio or video capabilities.

```
SosConfiguration config = SosConfiguration.builder()
  .agentPublish(false) // OpenTok agent is never used
  .build();
```

Control an SOS Session

During an SOS session, you can control the session using various static methods on the `Sos` class. These methods can pause the session, disable screen sharing, stop the session, change what is being shown on the screen, and perform other functions.

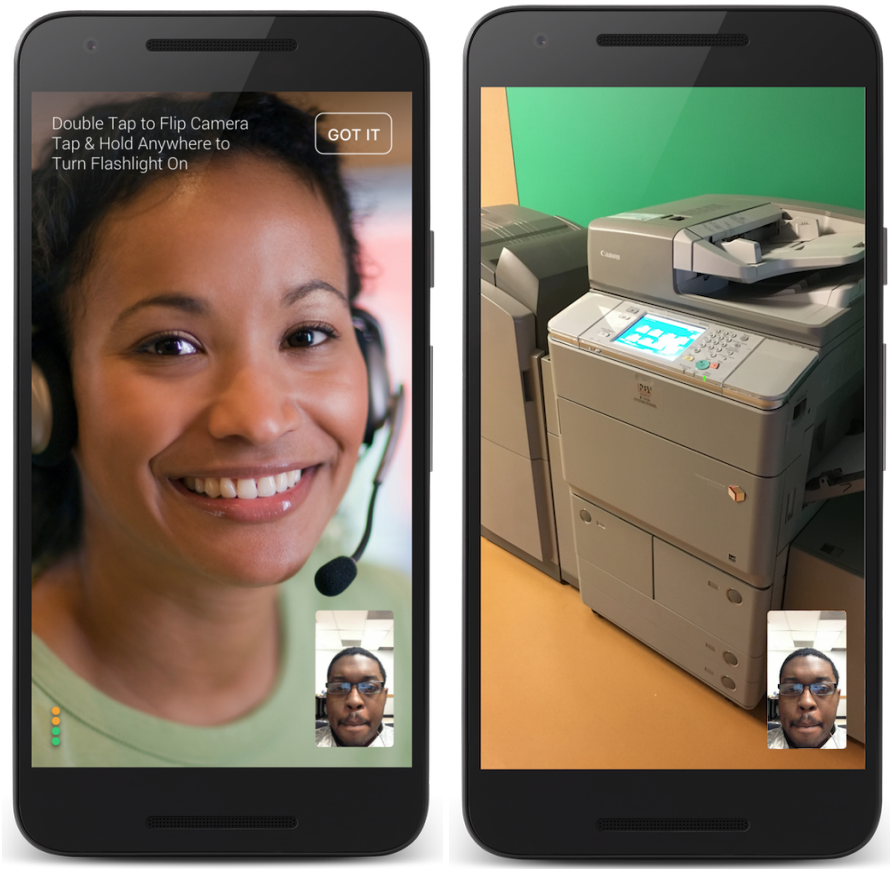
Method Name	Description
`void setSessionPaused (boolean paused)`	Pauses or unpauses the session. When paused, no audio or video is sent to the agent.
`void endSession (boolean showPrompt)`	Ends the current SOS session. The `showPrompt` argument allows you to control whether a prompt is displayed to the user before the session ends.
`boolean isSessionActive ()`	Returns whether a session is active.
`void setScreenSharingEnabled (boolean enabled)`	Enables or disables the screen sharing functionality.
`void setAudioEnabled (boolean enabled)`	Enables or disables the audio.
`void setTwoWayVideoEnabled (boolean enabled)`	Enables or disables the two-way video feature without prompting the user. You can only turn on two-way video if the session has been configured for two-way video and the device has a camera. See Two-Way Video to learn more.
`void setShareType (SosShareType shareType)`	Changes the share type to the specified value. Possible values are: `BackFacingCamera`, `FrontFacingCamera`, `ScreenSharing`. You can only turn on a camera if the session

Method Name	Description
	has been configured for two-way video and the device has a camera. See Two-Way Video to learn more.

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 enable the camera features by creating an `SosConfiguration` object with the desired settings and adding this configuration object to the `Sos.SessionBuilder`. For details about using this mechanism, see Configure an SOS Session.

The following video-related configuration settings are available:

Table 2: Two-Way Video Configuration Settings

Setting	Default Value	Description
`twoWayVideo`	`false`	Whether two-way video is enabled. This setting is required if you want to use either of the cameras.
`frontFacingCamera`	`true`	Whether the front-facing camera is enabled. When two-way video is enabled, the front-facing camera is enabled by default, if available.
`backFacingCamera`	`true`	Whether the back-facing camera is enabled. When two-way video is enabled, the back-facing camera is enabled by default, if available.
`defaultShareType`	`SosShareType.ScreenSharing`	Default sharing mode when a session starts. Can be set to `SosShareType.ScreenSharing`, `SosShareType.FrontFacingCamera`, or `SosShareType.BackFacingCamera`. If you don't specify a default share type, when you open the camera, it will try to use the front-facing camera first. If you specify `FrontFacingCamera` or `BackFacingCamera`, be sure to enable `twoWayVideo` as well.
`fieldServices`	`false`	Whether the session is in field services mode. In field services mode, screen sharing is disabled and two-way video is enabled. The app launches in full-screen video mode and you can't go into screen sharing mode. By default, the front-facing camera is used (if available) when the session starts.

When you turn on the camera using SOS, the user sees the Camera UI. The SDK never sends camera frames to the agent without a clear indication for the user on the device.

The configuration process for some of the more common use cases are shown below.

Example: **Basic Two-Way Video**

To turn on basic two-way video:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .build();
```

In this scenario, two-way video is accessible to the user (by tapping the camera icon on the SOS UI), but the session still starts in screen sharing mode.

Example: **Start with Front-Facing Camera**

To start an SOS session with the front-facing camera:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .defaultShareType(SosShareType.FrontFacingCamera)
  .build();
```

In this scenario, the session starts with the front-facing camera, but the user can return to screen sharing if desired.

Example: **Field Services Mode**

To start an SOS session in field services mode:

```
SosConfiguration config = SosConfiguration.builder()
  .fieldServices(true)
  .build();
```

In this scenario, the session starts in full-screen video mode (using the front-facing camera first, if available) and screen sharing mode is disabled.

Example: **Disable a Camera**

To enable two-way video but disable the back-facing camera:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .backFacingCamera(false)
  .build();
```

To enable two-way video but disable the front-facing camera:

```
SosConfiguration config = SosConfiguration.builder()
  .twoWayVideo(true)
  .frontFacingCamera(false)
  .build();
```

Check Agent Availability

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

You can subscribe as a listener to be notified whenever the agent availability state changes.

1. Implement the `SosAvailability.Listener` interface.

   ```
   public class MyActivity extends Activity implements SosAvailability.Listener {
   ```

2. Register your object as a listener.

   ```
   SosAvailability.addListener(this);
   ```

3. Upon registration, check the current status of agent availability to get initial status information.

   ```
   SosAvailability.Status status = SosAvailability.getStatus();
   ```

4. Upon registration, start polling for state changes.

```
if (!SosAvailability.isPolling()) {
  SosAvailability.startPolling(this, orgId, deploymentId, liveAgentPod);
}
```

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.

5. Implement the `onSosAvailabilityChange` method to handle state changes.

```
@Override
public void onSosAvailabilityChange (SosAvailabilty.Status status) {
  // React to status updates, either UNKNOWN, AVAILABLE, or UNAVAILABLE
}
```

6. When done, unregister listener and stop polling.

```
SosAvailability.removeListener(this);
SosAvailability.stopPolling();
```

Refer to the `SosAvailability` Javadoc for more details.

Example: The following example handles agent availability changes from an `Activity` class.

```
public class MyActivity extends Activity implements SosAvailability.Listener {

  @Override
  protected void onCreate (Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    SosAvailability.addListener(this);

    // check if there is any known status, react to it if so
    updateAvailability(SosAvailability.getStatus());

    if (!SosAvailability.isPolling()) {
      SosAvailability.startPolling(this, orgId, deploymentId, liveAgentPod);
    }
  }

  @Override
  public void onDestroy () {
    super.onDestroy();

    SosAvailability.removeListener(this);

    // Stop polling, if needed.  Don't do this if you would like to continue
    // polling in the background
    SosAvailability.stopPolling();
  }

  //--------------------------------------------------------------------------
  // SosAvailability.Listener implementation
  //--------------------------------------------------------------------------
```

```
  @Override
  public void onSosAvailabilityChange (SosAvailabilty.Status status) {
    // React to status updates, either UNKNOWN, AVAILABLE, or UNAVAILABLE
    // This value is an enum

    // A primary use-case is to show a button to initiate an SOS session only if
    // there is an agent AVAILABLE
  }

  //--------------------------------------------------------------------------
  // private helpers
  //--------------------------------------------------------------------------

  private void updateAvailablity (SosAvailability.Status status) {
    // React to the given status, e.g. show a button if it is AVAILABLE
  }
}
```

Listen to Events

The SOS SDK provides listeners that allow you to listen to and respond to various events emitted during an SOS session. These listeners can be used to keep your application informed about the state of the SOS session or to implement custom behavior based on the SOS session progress.

The following SOS listeners are available:

SosListener
Covers the high-level status of the session, including session creation, session termination and the state of the session.

SosHoldListener
Includes events for when the hold status of the session has changed.

SosConnectionListener
Allows you to listen for connection quality changes.

SosAVListener
Includes events for when the status of the audio or video playback changes during a session.

SosMaskingListener
Includes events for when masked fields are hidden or exposed during an SOS session.

SosNetworkTestListener
Can be used to listen to the status of the pre-session network test.

SosShareTypeListener
Allows you to monitor share type changes during a session.

Refer to the linked Javadoc for more details about these listeners.

Checking Initial State

A few event types have static methods in order for you to manually check their states. Depending on your implementation, these methods can be used to check the state when you initially start listening for an event.

Sos.getState()
Returns the same state sent to an SosListener.

Sos.getHoldState()
Returns the same state sent to an SosHoldListener.

Using a Listener

Instances of each of the above listener interfaces may be added or removed using the appropriate static APIs in the SOS class. Listeners may be added and removed at any time before a session starts, or during an active session, and will be triggered for all events that occur until they are removed.

Note: Be sure to remove any listeners you've added when your application is finished listening to events. The added listeners are statically referenced, and failure to properly remove them before they go out of scope will result in a memory leak in your application.

We recommend that you do not tie an `SosListener` implementation to the lifecycle of an `Activity` if you expect to receive SOS events for the duration of the session. Since Activities are created and destroyed with regularity, it is preferable to create an instance of `SosListener` that exists at the Application level, or by some other means to ensure it will persist for the duration of a session.

Example: **Listener in Activity Class**

A common way to use the listener interfaces is to implement them in an `Activity` class. The `Activity` may add itself as a listener in one of its pre-visible lifecycle callbacks, and remove itself in a post-visible callback.

```
public class MyActivity extends Activity implements SosListener {

  // ...

  @Override
  public void onCreate (Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // ... Do your normal activity setup here.

    // Add this activity as an SOS event listener. The SosListener methods
    // implemented below will now be called for any new or existing SosSession
    // while this Activity instance exists.
    Sos.addListener(this);
  }

  @Override
  public void onDestroy () {
    super.onDestroy();

    // ... Do your normal activity teardown here.

    // Remove ourselves as an SOS event listener. This call ensures that you
    // will not leak a reference to your Activity.
    Sos.removeListener(this);
  }

  // ...

  @Override
  public void onSessionCreated () {
    // Handle a new session being created here by e.g. changing the UI to
    // reflect that you're in an SOS session.
```

```
  }

  @Override
  public void onSessionEnded (SosEndReason reason) {
    // Handle a session being ended here.
  }

  @Override
  public void onSessionStateChange (SosState state, SosState oldState) {
    // Handle state changes during the session here.
  }
}
```

Example: **Listener in Anonymous Class**

You may also choose to add a listener using an anonymous class for a more short-term use case. It is still important to remember to remove such a listener or it may leak itself and its enclosing scope.

```
public class MyActivity extends Activity {

  public static final String TAG = MyActivity.class.getSimpleName();

  private SosAVListener mListener;

  public void addListener () {
    mListener = new SosAVListener () {
      @Override
      public void onAudioToggled (boolean enabled) {
        LOG.d(TAG, "SOS audio is " + (enabled ? "enabled" : "disabled"));
      }

      @Override
      public void onVideoToggled (boolean enabled) {
        LOG.d(TAG, "SOS video is " + (enabled ? "enabled" : "disabled"));
      }
    };
  }

  public void removeListener () {
    if (mListener != null) {
      Sos.removeAVListener(mListener);
      mListener = null;
    }
  }

  @Override
  public void onDestroy () {
    // Make sure the listener is definitely removed.
    removeListener();
  }
}
```

Detect the Keyboard

Because the Android OS manages the keyboard, its representation on the screen is inaccessible to an application. This situation presents a challenge when sending an image of the device to an agent during an SOS session. For this reason, you should explicitly pass along information about the keyboard.

Sending Keyboard Information

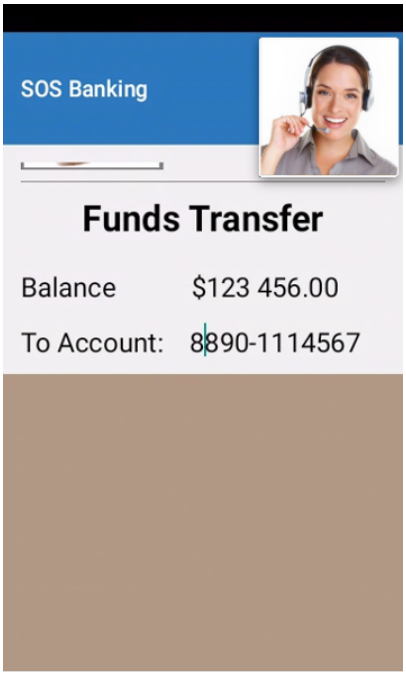
When you determine that keyboard information has changed, use the SDK to send this information to the agent:

`showKeyboard(Rect)`
Informs the agent that the keyboard covers a given rectangle on the screen.

`hideKeyboard()`
Informs the agent that the keyboard is not visible.

The API describes the size and position of the keyboard and not its appearance. Your application cannot know what the keyboard looks like, especially because users are free to install third-party keyboards. When you give the SDK a keyboard position and size the agent sees a shaded rectangle in the specified area. If you wish to show a full screen keyboard, pass in a `null Rect` to the `showKeyboard(Rect)` method.

Keyboard view from agent's perspective:



Techniques to Detect the Keyboard

There is no direct way to detect the keyboard on Android. However, it's possible to infer the existence of the keyboard by using the `InputMethodManager` to determine if input views are active on the screen and if they are able to receive input from the user.

It's also not possible to directly determine the position or dimensions of the keyboard, but sometimes you can infer this information by comparing the Window dimensions of the current `Activity` to the device's default `Display` area. In some situations, especially

in portrait orientation, the `Window` dimensions shrink in height to accommodate the keyboard. If you subtract the shrunken `Window` dimensions from the `Display` dimensions, you can assume the keyboard occupies that area. In landscape orientation, if the `Window` size has not changed, you can assume that the keyboard takes up the entire display.

Detecting the Keyboard Using InputMethodManager

The `InputMethodManager` is provided by Android to facilitate the transfer of input data into an application. It also provides an API for determining if Android is currently accepting input. It doesn't provide any information about the size of the keyboard, but it's possible to infer whether a keyboard is displayed by combining the output from a few of its methods.

Once you have obtained an instance of the `InputMethodManager`, you can check for the presence of input-ready views with the following condition:

```
if (mInputMethodManager != null && mInputMethodManager.isActive() &&
  mInputMethodManager.isAcceptingText()) {
  // Determine keyboard area and call Sos.showKeyboard(...)
} else {
  Sos.hideKeyboard();
}
```

Invoking `InputMethodManager.isActive()` tells you if the current view has any editable views (such as `EditText`) inflated. Invoking `InputMethodManager.isAcceptingText()` tells you if any of those editable views have focus. It's possible for an editable view to have focus while the keyboard is not displayed. For this reason, it's not the only indicator you'll need to determine the on-screen presence of the keyboard.

The `InputMethodManager.isFullscreenMode()` method tells you if the keyboard is occupying the entire screen. In practice, this information isn't correlated with the appearance of a full screen keyboard, but rather that one is about to be displayed, or that the editable view that prompted the display of a full screen keyboard has retained focus. For this reason it's not a reliable indicator of when the full screen keyboard is displayed to the user.

Detecting the Keyboard Using Window Size

Another technique for detecting the keyboard is to obtain the current `Window` size. When a keyboard appears on the screen and doesn't require its own `Window`, as a full screen keyboard does, the current `Window` reduces its height to accommodate it. By subtracting the updated `Window` size from the default `Display` size, you can assume that the keyboard occupies the area in the difference. This technique works particularly well on tablets and in portrait orientation on smaller devices. However, this method does not work when a full screen keyboard is displayed (in landscape orientation on a phone) because the `Window` that contains your `View` does not change its dimensions.

There are other situations in which the `Window` size might change that are unrelated to the keyboard. For this reason, `Window` size is not the only indicator you'll need to determine the presence of the on-screen keyboard. We have found two methods for obtaining `Window` size changes that can help you detect a keyboard:

1. Intercept View Measurement
2. View Tree Observer Listeners

These two techniques only approximate the existence of the keyboard. Used together, they provide a reasonably accurate estimate. Taking into account the characteristics of your application, you can further improve the accuracy.

Window Size Method 1: Intercept View Measurement

In situations when a `Window` is resized to accommodate a keyboard, Android re-measures the views on screen using the Android `View` system. This process causes each layout's `onMeasure(int, int)` method to be called. An effective way to intercept this

event is to find each root layout type you use for your activities and extend them to override the `onMeasure(int, int)` method. You should also change your layout XML files to use your new custom `Layout` classes.

```
package com.salesforce.documentation.layout;

public class CustomLinearLayout extends LinearLayout {

  // { Constructors Here }

  @Override
  protected void onMeasure (int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    Rect rect = new Rect();
    ((Activity) getContext()).getWindow().getDecorView().
      getWindowVisibleDisplayFrame(rect);

    // Continue with Keyboard Detection, SOS API
  }
}
```

```
<com.salesforce.documentation.layout.CustomLinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/main_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Views Here -->

</com.salesforce.documentation.layout.CustomLinearLayout>
```

Window Size Method 2: View Tree Observer Listeners

An alternative method to detect `Window` resizing is to use the `ViewTreeObserver`. Registering a `ViewTreeObserver.OnGlobalLayoutListener` with a `ViewTreeObserver` instance allows you to be notified when a view hierarchy is resized. This event can occur with the presence of an on-screen keyboard.

To obtain a reference to a `ViewTreeObserver`, you must obtain `Activity` instances as they are created. The most flexible way of getting this information is to register an instance of the `ActivityLifecycleCallbacks` with your `Application` context. Keep in mind that listeners you register with `ViewTreeObserver` should be explicitly cleaned up when the `Activity` it belongs to is paused or destroyed.

```
public class ActivityTracker implements Application.ActivityLifecycleCallbacks {

  private ViewTreeObserver mViewTreeObserver;
  private ViewTreeObserver.OnGlobalLayoutListener mGlobalLayoutListener;

  @Override
  public void onActivityResumed (final Activity activity) {
    View view = ((ViewGroup) activity.getWindow().getDecorView().
      findViewById(android.R.id.content)).getChildAt(0);
    mViewTreeObserver = view.getViewTreeObserver();
    mGlobalLayoutListener = new ViewTreeObserver.OnGlobalLayoutListener() {
      @Override
      public void onGlobalLayout () {
        Rect windowRect = new Rect();
        activity.getWindow().getDecorView().getWindowVisibleDisplayFrame(windowRect);
```

```
        // Continue with Keyboard Detection, SOS API
      }
    };

    mViewTreeObserver.addOnGlobalLayoutListener(mGlobalLayoutListener);
  }

  @Override
  public void onActivityPaused (Activity activity) {
    if (mViewTreeObserver == null || !mViewTreeObserver.isAlive() ||
        mGlobalLayoutListener == null) {
      return;
    }

    // Make sure the ViewTreeObserver and listener are cleaned up.
    if (android.os.Build.VERSION.SDK_INT <= 15) {
      mViewTreeObserver.removeGlobalOnLayoutListener(mGlobalLayoutListener);
    } else {
      mViewTreeObserver.removeOnGlobalLayoutListener(mGlobalLayoutListener);
    }
  }

  // { Other Overrides }
}
```

Example: **Keyboard Detector Example**

```
/**
 * KeyboardDetector
 *
 * Attempt to detect the presence of the soft keyboard on screen and prompt SOS
 * to display the keyboard overlay to the Agent.
 *
 * Utilizes the InputMethodManager to detect when a keyboard might be visible and
 * compares the Display size to the current Window size in order to determine what
 * the size of the keyboard could be.
 *
 * The InputMethodManager is able to tell us if there are any active input views
 * and if any of them are accepting input. We can use this to reasonably infer
 * that a soft keyboard could be present on the screen. Since we can only
 * correlate input readiness with the display of a soft keyboard, this method
 * is not 100% accurate.
 *
 * We are able to make an assumption regarding the soft keyboard's on-screen
 * dimensions expecting the Window dimensions to shrink once a keyboard is
 * displayed. By comparing the new, smaller Window dimensions with the
 * device's Display dimensions we can assume that the gap between the bottom of
 * the Window and the bottom of the Display must be the area occupied by a soft
 * keyboard. This method may not work if the Activity is configured to maintain
 * its dimensions, and it cannot account for the dimensions of the full-screen
 * keyboard that is displayed on phones while in landscape orientation.
 */
public class KeyboardDetector {

  private InputMethodManager mInputMethodManager;
```

```
private WindowManager mWindowManager;
private Activity mActivity;

public KeyboardDetector (Activity activity) {
  mActivity = activity;
  mInputMethodManager =
    (InputMethodManager) activity.getSystemService(Context.INPUT_METHOD_SERVICE);
  mWindowManager =
    (WindowManager) activity.getSystemService(Context.WINDOW_SERVICE);
}

/**
 * Attempt to discover if a soft keyboard is displayed on the screen and what its
 * dimensions are. This method will inform SOS of when the keyboard overlay should
 * be hidden from the Agent and when it should be shown (and what its dimensions
 * are).
 */
public void detectKeyboard () {
  if (mInputMethodManager == null ||
      !mInputMethodManager.isActive() ||
      !mInputMethodManager.isAcceptingText()) {
    Sos.hideKeyboard();
    return;
  }

  Rect windowRect = getWindowRect();

  if (mInputMethodManager.isFullscreenMode()) {
    Sos.showKeyboard(windowRect);
  } else {
    Rect displayRect = getDisplayRect();
    Rect keyboardRect = getKeyboardRect(displayRect, windowRect);
    Sos.showKeyboard(keyboardRect);
  }
}

/**
 * Get the dimensions of the Default Display as a Rect instance.
 *
 * @return Rect with the display dimensions
 */
private Rect getDisplayRect () {
  Rect displayRect = new Rect();
  mWindowManager.getDefaultDisplay().getRectSize(displayRect);
  return displayRect;
}

/**
 * Get the dimensions of the Activity's Window as a Rect instance.
 *
 * @return Rect with the Window dimensions
 */
private Rect getWindowRect () {
  Rect rect = new Rect();
```

```
    mActivity.getWindow().getDecorView().getWindowVisibleDisplayFrame(rect);
    return rect;
  }

  /**
   * Get the assumed dimensions of the soft keyboard by assuming that it must
   * occupy the area between the bottom of the Window Rect and the bottom of
   * the Display Rect.
   *
   * @param display Rect instance representing the Default Display area
   * @param window  Rect instance representing the Window area
   * @return Rect with assumed keyboard dimensions
   */
  private Rect getKeyboardRect (final Rect display, final Rect window) {
    return new Rect(display.left, window.bottom, display.right, display.bottom);
  }
}
```

Example: **Integration Example** One of the most reliable ways to invoke the `KeyboardDetector` is by extending the layout class you are using in your app and calling the `detectKeyboard()` method during `View.onMeasure(int, int)`. If you choose to invoke keyboard detection this way, you'll need to do this for each type of layout in your app that contains input views as they could cause a keyboard to appear.

If you choose to implement this on an `Activity` that will not be destroyed during an orientation change, you may want to invoke keyboard detection within an `onConfigurationChange(Configuration)` override as well.

```
public class KeyboardDetectingDrawerLayout
  extends android.support.v4.widget.DrawerLayout {

  private KeyboardDetector mKeyboardDetector;

  public KeyboardDetectingDrawerLayout (Context context) {
    this(context, null);
  }

  public KeyboardDetectingDrawerLayout (Context context, AttributeSet attrs) {
    this(context, attrs, 0);
  }

  public KeyboardDetectingDrawerLayout (Context context,
    AttributeSet attrs, int defStyle) {

    super(context, attrs, defStyle);
    mKeyboardDetector = new KeyboardDetector((Activity) context);
  }

  @Override
  protected void onMeasure (int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    mKeyboardDetector.detectKeyboard();
  }
}
```

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.

The Android SDK provides five views that mask information sent to an agent during an SOS session. These five views mimic their corresponding native Android views.

- `com.salesforce.android.sos.maskview.AutoCompleteTextView`
- `com.salesforce.android.sos.maskview.EditText`
- `com.salesforce.android.sos.maskview.MultiAutoCompleteTextView`
- `com.salesforce.android.sos.maskview.TextView`
- `com.salesforce.android.sos.maskview.View`

These views are used in the same way as their native Android counterparts, but they automatically obscure their contents when the application screen is being shared with an agent during an SOS session.

Implementing Masked Fields

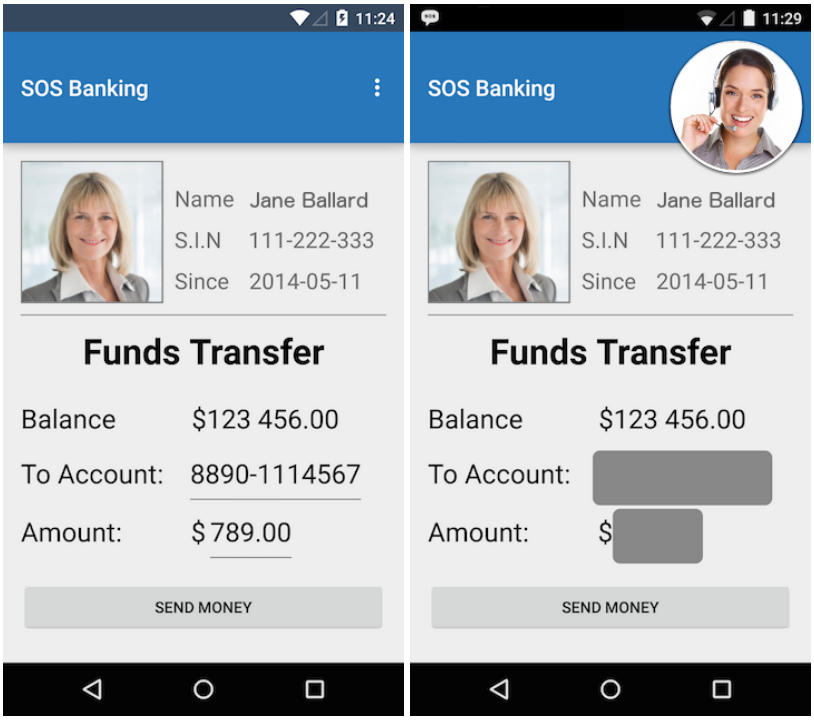
The simplest way to use field masking is to replace a native Android view in your layout file with its masking alternative.

In this example application that allows a user to transfer money to another account, the account number and the amount being transferred are sensitive data and are masked from the agent when in an SOS session. To get this behavior, instead of using a native Android `EditText` view for the account number and transfer amount, use `com.salesforce.android.sos.maskview.EditText`.

```
<com.salesforce.android.sos.maskview.EditText
    android:id="@+id/transfer_account"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_account_value"
    android:textSize="24sp"
    android:gravity="center_horizontal"/>

<com.salesforce.android.sos.maskview.EditText
    android:id="@+id/transfer_amount"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_amount_value"
    android:textSize="24sp"/>
```

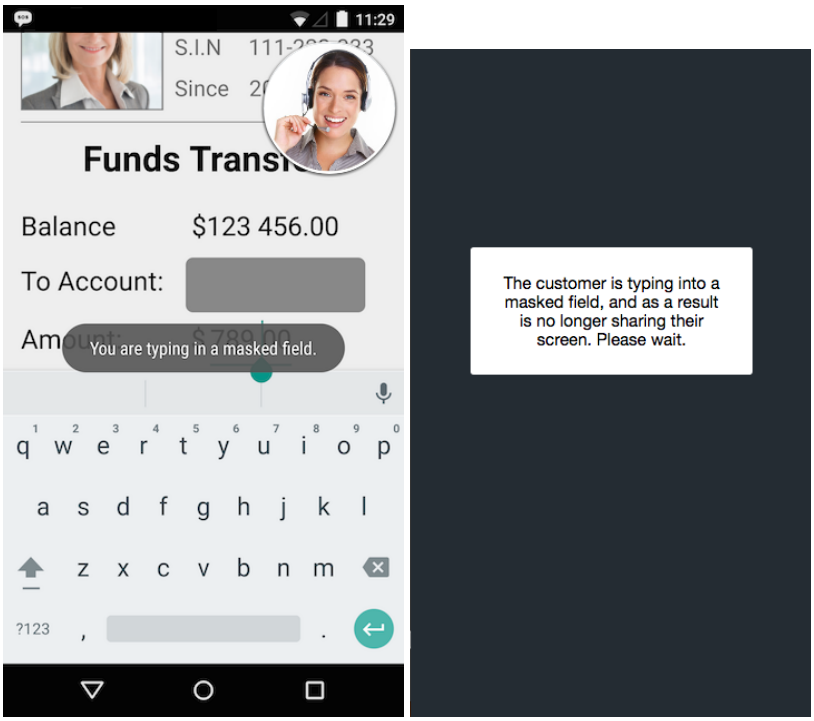
This change is enough to ensure that an agent doesn't see the values in these two views. When the user edits either of the fields, screen sharing halts and the agent sees a notification that sensitive data is being edited.

SOS session with masked field (normal view vs. SOS session view):



When a user begins editing a masked field, the view is exposed and a toast is presented to the user. Editing is then done normally as it would with the corresponding native Android view. The agent on the other end no longer sees the application screen and is presented with a notification.

Behavior when editing masked field (user view vs. agent view):



When the user finishes editing the masked fields and the fields no longer have focus, screen sharing resumes with the fields masked and the agent able to see the application again.

Customizing Masked View Drawable

The default masking behavior is to draw a gray rounded rectangle to cover the masked view. You can customize the appearance of a masked field by specifying a `Drawable` resource to be used to draw the mask over the view. In this example application, we add a reference to a custom `Drawable` to each masked field to match the application's theme. This `Drawable` is used in place of the gray rounded rectangle when drawing.

Define a `Drawable` in `blue_field_mask.xml`:

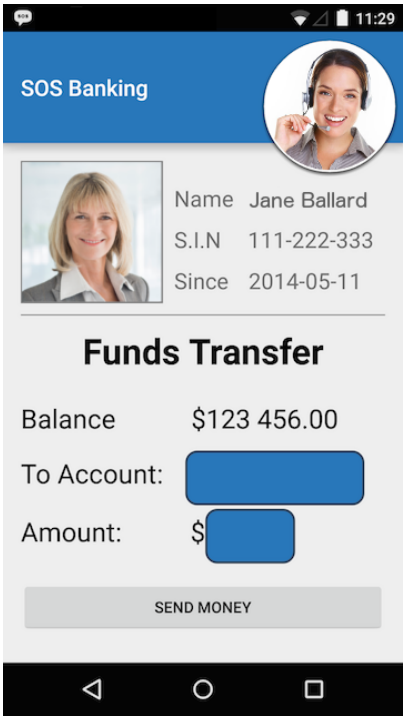
```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">

    <!-- Specify a color for the background -->
    <solid
        android:color="#ff2877ba"/>

    <!-- Specify a darker border -->
    <stroke
        android:width="2dp"
        android:color="#ff36485d"/>

    <!-- Specify rounded corners -->
    <corners
        android:topLeftRadius="10dp"
        android:topRightRadius="10dp"
        android:bottomLeftRadius="10dp"
        android:bottomRightRadius="10dp"/>
</shape>
```

Add the custom `Drawable` to the masked views:

```
<com.salesforce.android.sos.mask.EditText
    app:sos_mask_drawable="@drawable/blue_field_mask"
    android:id="@+id/transfer_account"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_account_value"
    android:textSize="24sp"
    android:gravity="center_horizontal"/>

<com.salesforce.android.sos.mask.EditText
    app:sos_mask_drawable="@drawable/blue_field_mask"
    android:id="@+id/transfer_amount"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_amount_value"
    android:textSize="24sp"/>
```

As a result, the masks are now drawn using the custom `Drawable`. There is no restriction on what makes up the custom `Drawable`. The only caveat is that the `Drawable` stretches to fill the size of the masked view.

Custom Drawable masked field:



You can also change the `Drawable` used by a masked view programmatically by calling `setMask(Drawable)` on the masked view.

```
import com.salesforce.android.sos.mask.EditText;

...

Drawable blueMask = getResources().getDrawable(R.drawable.blue_field_mask);

final EditText transferAccountView = (EditText) rootView.findViewById(R.id.transfer_account);
transferAmountView.setMask(blueMask);

final EditText transferAmountView = (EditText) rootView.findViewById(R.id.transfer_amount);
transferAccountView.setMask(blueMask);
```

This method has the same effect as specifying a `Drawable` in XML, but it allows the mask to be set dynamically during application execution.

Manually Hiding Masked Fields

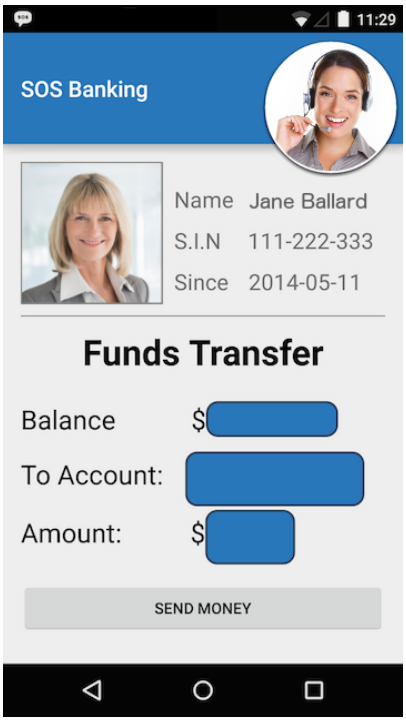
The default behavior for hiding and exposing masked fields is based on when a field has focus. If a masked field has focus, it is exposed (and screen sharing halts), and when it does not have focus, it is hidden. In situations where you want to expose and hide masked views manually, you can call `showMask(boolean)` on the masked view. This method is useful in cases where a masked view does not normally get focus, such as displaying static text in a `TextView`, or when using the more generic `View`.

In this sample application, the account balance is considered sensitive information and is defined as a `com.salesforce.android.sos.maskview.TextView`.

```
<com.salesforce.android.sos.maskview.TextView
    app:sos_mask_drawable="@drawable/blue_field_mask"
    android:id="@+id/account_balance"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/account_balance_value"
    android:textSize="24sp"/>
```

```
import com.salesforce.android.sos.maskview.TextView;

...

final TextView accountBalance = (TextView) rootView.findViewById(R.id.account_balance);

// expose the account balance (do not mask the view)--screen sharing will be paused
accountBalance.showMask(false);

// hide the account balance (mask the view)
accountBalance.showMask(true);
```

Manually masked view:



In all situations where a masked field is exposed, regardless of whether it is exposed manually or automatically, the agent cannot view the application screen. Only when all masked fields are hidden is screen sharing enabled.

Turning Off Focus Masking

If you're planning on manually exposing and hiding a masked view, it can be confusing to have the default behavior based on focus active at the same time. This situation results in a visible masked view regardless of whether it has focus or if it has been manually exposed.

To turn off automatic masking based on focus, add a flag to the masked field definition:

```
<com.salesforce.android.sos.maskview.EditText
    app:sos_use_focus_masking="false"
    app:sos_mask_drawable="@drawable/blue_field_mask"
    android:id="@+id/transfer_account"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_account_value"
    android:textSize="24sp"
    android:gravity="center_horizontal"/>

<com.salesforce.android.sos.maskview.EditText
    app:sos_use_focus_masking="false"
    app:sos_mask_drawable="@drawable/blue_field_mask"
    android:id="@+id/transfer_amount"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FF000000"
    android:text="@string/transfer_amount_value"
    android:textSize="24sp"/>
```

Now, when the user edits one of the masked fields, the mask is not removed. Make this change only if you plan to manage masking manually.

You can also turn off focus masking programmatically by calling `useFocusMasking(boolean)` on the masked view.

```
import com.salesforce.android.sos.maskview.EditText;

...

final EditText transferAccountView = (EditText) rootView.findViewById(R.id.transfer_account);

// turn off focus masking
transferAmountView.useFocusMasking(false);

// turn focus masking back on
transferAmountView.useFocusMasking(true);
```

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, construct a `Map` instance and pass it to the `SosOptions` constructor. The keys in this map should reference the API Name for fields defined in your SOS Session object and the values should reflect the desired values for those fields. The class of the value object should reflect the field type in the SOS Session object.

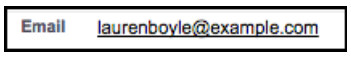
Example: This example shows how to pass email information from your app to Service Cloud. Before trying this example, be sure to define an "Email" custom field on the SOS Session object in your Salesforce org:

To learn more about custom fields, see Create Custom Fields in Salesforce Help.

Once you have created a custom field, construct a `Map` and use the API Name of the field to specify the customer's email address. In this case, the field is defined as an Email data type, so we specify a valid email address as a `String`.

```
Map<String, Object> customData = new HashMap<>();

// 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.
customData.put("Email__c", "laurenboyle@example.com");

// Use the custom data map when initializing the SosOptions instance. From here,
// you may simply create the session as normal and the custom data will be used
// to populate fields in the SOS Session object.
SosOptions sosOpts = new SosOptions(
    customData
    "your.pod.name",
    "your-org-id",
    "your-deploy-id"
);
```

When the user creates an SOS session, the Email field is pre-populated with the value specified in the `SosOptions` custom data map.



UI 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 instructions for localizing strings in all supported languages.

Strings and Localization

You can change the text throughout the user interface. To customize text, create string resource XML files (named `strings.xml`) in your project's `values-[locale]` resource folder for the language(s) you want to update.

Customize Colors

You can customize the look and feel of the interface by specifying the colors used throughout the UI.

Knowledge: Customize Fonts

You can customize the fonts used in the Knowledge UI.

SOS: Customize the Connecting UI

By default, connection messages appear on the SOS interface when a session connects. You can override this behavior and present your own UI during the connection process.

SOS: Agent Annotations

During an SOS session, the agent can annotate a customer's screen to point out something . By default, agent annotations are red lines that are `5dp` wide, but you can customize these values.

SOS: Toast Behavior

The SOS session provides context to the customer through toasts for various events over the lifetime of the session. You can customize these toasts.

Strings and Localization

You can change the text throughout the user interface. To customize text, create string resource XML files (named `strings.xml`) in your project's `values-[locale]` resource folder for the language(s) you want to update.

To see the complete list of string resource values, refer to the string resources document for the feature you want to customize.

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

SDK text is translated into more than 25 different languages. In order for your string customizations to take effect in all languages, provide a translation for each language. To add support for a language, create a resources subdirectory that includes a hyphen and the ISO language code at the end of the directory name. For example, `values-es/` is the directory containing string resources for Spanish. Android loads the appropriate resources according to the locale settings of the device at run time. The system falls back on the strings in the default `values/` directory if the appropriate locale directory isn't found.

The following languages are currently supported:

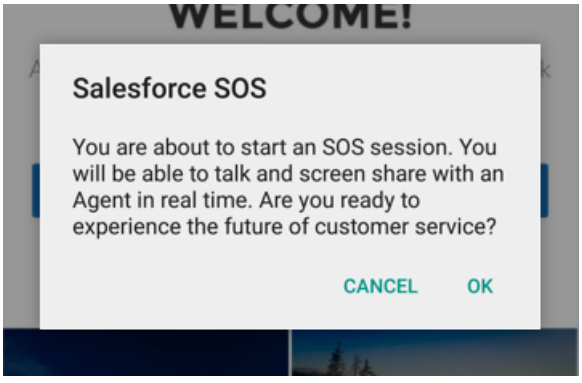
Table 3: Supported Languages

Language Code	Language
values-cs	Czech
values-da	Danish
values-de	German
values-el	Greek
values-en	English
values-es	Spanish
values-fi	Finnish
values-fr	French

Language Code	Language
values-hu	Hungarian
values-in	Indonesian
values-it	Italian
values-ja	Japanese
values-ko	Korean
values-nl	Dutch
values-no	Norwegian
values-pl	Polish
values-pt-rBR	Brazilian Portuguese
values-ro	Romanian
values-ru	Russian
values-sv	Swedish
values-th	Thai
values-tr	Turkish
values-uk	Ukranian
values-vi	Vietnamese
values-zh	Chinese
values-zh-rTW	Traditional Chinese

Check out Supporting Different Languages in the Android Developer documentation for more info about localization.

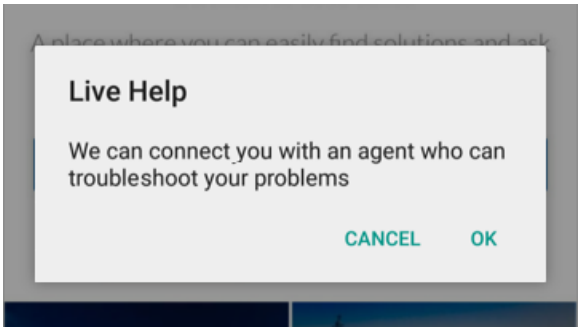
Example: To learn how you can change string values, let's go through an example. The image below shows the default connection prompt dialog text in English:



You can change the title and body of this dialog by changing the `sos_title` and the `sos_connect_prompt` strings in the `strings.xml` file in the `values` folders for your locale (`values-en/` for English):

```
<!-- other string resources omitted -->

<string name="sos_title">Live Help</string>
<string name="sos_connect_prompt">
  We can connect you with an agent who can troubleshoot your problems</string>
```

Now, whenever you start a session you see the updated dialog text:



Customize Colors

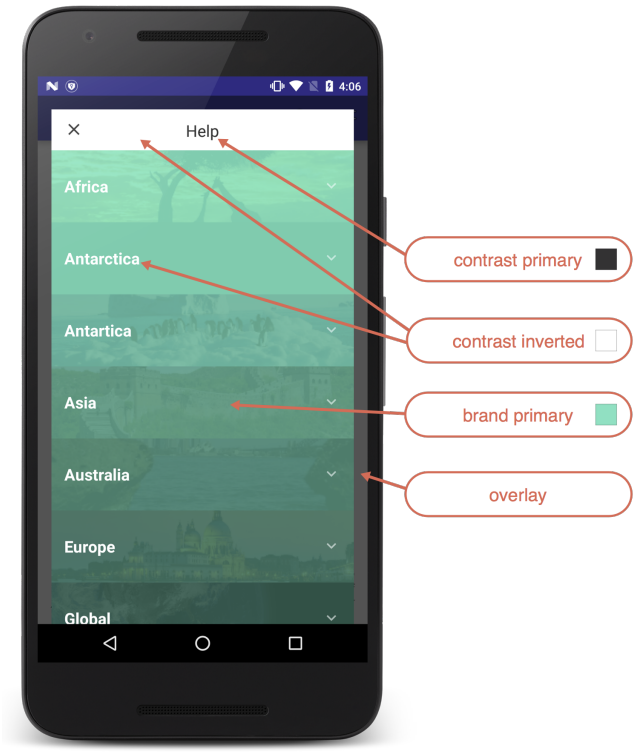
You can customize the look and feel of the interface by specifying the colors used throughout the UI.

You can customize the color scheme used throughout the interface so that the colors look good alongside your app. To customize the colors, create color resource values in your project's `colors.xml` file that correspond to the same resource names specified in this documentation.

For example, the following resource file values customize some of the branding tokens:

```
<resources>
  <color name="salesforce_brand_primary">#50e3c2</color>
  <color name="salesforce_brand_secondary">#4a90e2</color>
  <color name="salesforce_contrast_inverted">#ffffff</color>
  <color name="salesforce_contrast_primary">#333333</color>
  <color name="salesforce_contrast_secondary">#767676</color>
  <color name="salesforce_feedback_primary">#e74c3c</color>
  <color name="salesforce_overlay">#aa000000</color>
</resources>
```

The screenshots below illustrates how the branding tokens affect the UI:



The following branding tokens are available for customization:

Table 4: Service SDK Colors

Token Name	Default Value	Sample Uses
Brand Primary `salesforce_brand_primary`	#50E3C2	In Knowledge, used for first data category, the Show More button, the footer stripe, the selected article. Used by various icons in SOS.
Brand Secondary `salesforce_brand_secondary`	#4A90E2	UI button colors in Knowledge. Background color for action items in SOS
Brand Contrast `salesforce_brand_contrast`	#FCFCFC	Text on areas where a brand color is used for the background. Colors of the icons on the SOS UI (when they are selected).
Contrast Primary `salesforce_contrast_primary`	#333333	Primary body text color throughout the UI. Background color for buttons on the SOS UI.
Contrast Secondary `salesforce_contrast_secondary`	#767676	Used for subcategory headers in Knowledge.
Contrast Inverted `salesforce_contrast_inverted`	#FFFFFF	Used for page background, navigation bar, table cell background. Colors of the icons on the SOS UI (when not selected).
Feedback Primary `salesforce_feedback_primary`	#E74C3C	Text color for error messages. Mute indicator in SOS. Disconnect icon in SOS.
Feedback Secondary `sos_feedback_secondary`	#2ECC71	SOS connection quality indicators; background color for the Resume button when the two-way camera is active.
Feedback Tertiary `sos_feedback_tertiary`	#F5A623	SOS connection quality indicators.

Token Name	Default Value	Sample Uses
Title Text `salesforce_title_text`	#FFFFFF 	In Knowledge, used for text on data category headers and the chevron on the Knowledge home page.
Overlay `salesforce_overlay`	#000000 (at 66% alpha)	Used for background for the Knowledge home screen.

Knowledge: Customize Fonts

You can customize the fonts used in the Knowledge UI.

By default, the Service SDK interface uses the OS default font. If you want to change the font, follow these steps.

1. Add a True Type Font (TTF) file to your project's `assets` directory.
2. Override the `SalesforceFontStyle` style in your project's `styles.xml` resource file and set the `salesforceFont` item to the relative asset path of your TTF file.

   For example:

```
<style name="SalesforceFontStyle">
  <item name="salesforceFont">CustomFont.ttf</item>
</style>
```

The Knowledge UI uses the selected font when you rebuild your app.

SOS: Customize the Connecting UI

By default, connection messages appear on the SOS interface when a session connects. You can override this behavior and present your own UI during the connection process.

To present your own UI for the connection process, disable the default UI at configuration time with the `connectingUi` method.

```
SosConfiguration config = SosConfiguration.builder()
  .connectingUi(false)
  .build();
```

To learn more about using `SosConfiguration`, see Configure an SOS Session.

After you've disabled the default connecting UI, use the SosListener to listen for state changes and display your own UI. To learn more about listeners, see Listen to Events.

Example: The following Fragment class listens to SOS state changes and provides a place for you to add your own UI.

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.salesforce.android.sos.api.Sos;
import com.salesforce.android.sos.api.SosListener;
import com.salesforce.android.sos.api.SosState;
```

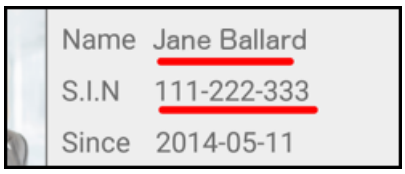
```
import com.salesforce.android.sos.api.SosEndReason;

public class CustomConnectingUI extends Fragment implements SosListener {

  public CustomConnectingUI () {
  }

  public static CustomConnectingUI newInstance () {
    CustomConnectingUI fragment = new CustomConnectingUI();
    Bundle args = new Bundle();
    fragment.setArguments(args);
    return fragment;
  }

  @Override
  public View onCreateView (LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View view = inflater.inflate(R.layout.fragment_custom_connecting_ui, container,
false);

    // Add an SosListener...
    Sos.addListener(this);

    // Set the initial state
    displayState(Sos.getState());

    return view;
  }

  @Override
  public void onDestroyView () {
    super.onDestroyView();

    // Remove listener
    Sos.removeListener(this);
  }

  @Override
  public void onSessionCreated () {
  }

  @Override
  public void onSessionEnded (SosEndReason reason) {
  }

  @Override
  public void onSessionStateChange (SosState state, SosState oldState) {
    displayState(state);
  }

  /**
   * Helper function to display SOS state changes.
   */
```

```
  private void displayState (SosState state) {
    if (state == SosState.Initializing) {
      // TO DO: Display a message that we're creating a session
    }
    if (state == SosState.AgentJoining) {
      // TO DO: Display a message that agent is joining
    }
    if (state == SosState.WaitingForAgent) {
      // TO DO: Display a message that we're waiting for agent to accept
    }
  }
}
```

SOS: Agent Annotations

During an SOS session, the agent can annotate a customer's screen to point out something . By default, agent annotations are red lines that are `5dp` wide, but you can customize these values.

By default, an annotation in SOS looks like this:



The Android SDK offers customization of the line width and color. When you change the line width or color, the value you change it to syncs with the customer and the agent. To change the width or color of the annotation line, modify two XML resources: `sos_drawing_color` and `sos_drawing_width`.

Table 5: Agent Annotation Style

Annotation Name	Type	Default Value
`sos_drawing_color`	Color	#ffff0000
`sos_drawing_width`	Integer	5

For documentation about overriding resources in Android, see Resource Merging in the Android documentation.

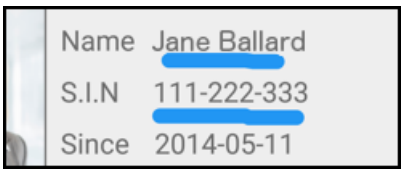
Example: Let's say we want to change the annotation to a thicker blue line. Specify the color value for `sos_drawing_color` (alongside your other resources):

```
<color name="sos_drawing_color">#ff2196f3</color>
```

And specify the integer value for `sos_drawing_width` (alongside your other resources):

```
<integer name="sos_drawing_width">10</integer>
```

Now the application annotation color looks like this:



SOS: Toast Behavior

The SOS session provides context to the customer through toasts for various events over the lifetime of the session. You can customize these toasts.

There are two ways to customize the toast behavior.

Change the Toast Text

You can change any toast text by overriding the string resource that the toast uses. See Strings and Localization for more information about overriding string resources.

Disable Some or All Toasts

If you do not want a particular toast to ever appear during a session, you can disable the toast with `SosConfiguration.Builder.disableToasts(...)`. This method takes an `EnumSet` of `SosToast` elements. Some built-in `enum` sets are already defined in the `SosToast` Javadoc.

For example, this code disables all toasts over the course of the session:

```
SosOptions options = new SosOptions(
  'pod',          // replaced with your real pod identifier
  'orgId',        // replaced with your real orgId
  'deploymentId' // replaced with your real deploymentId
);

SosConfiguration config = SosConfiguration.builder()
    .disableToasts(EnumSet.allOf(SosToast.class))
    .build();

Sos.session(opts)
    .configuration(config)
    .start(this);
```

Troubleshooting

Get some guidance when you run into issues.

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.

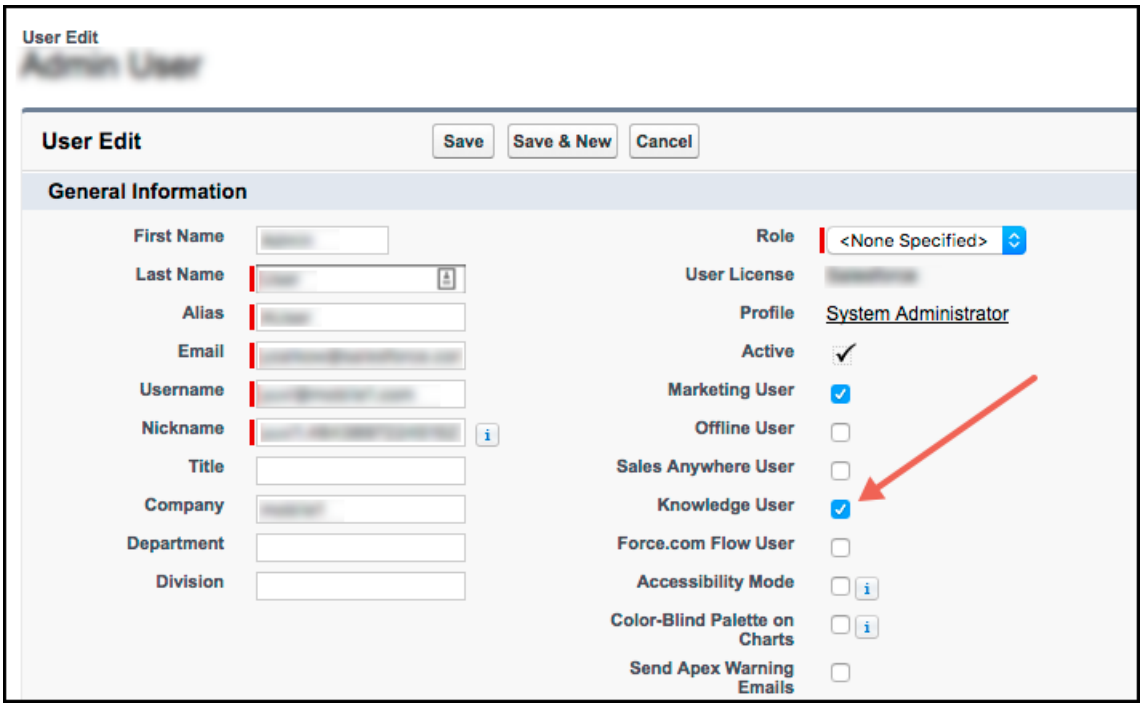
Unable to Access My Community

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

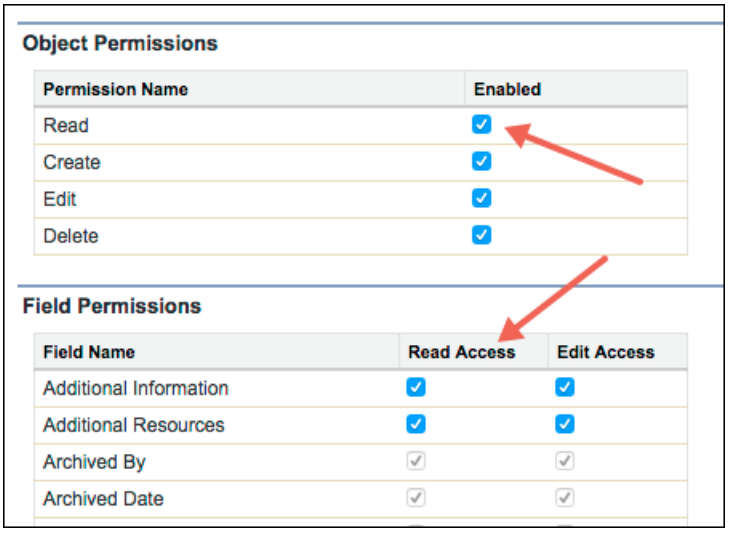
Run through this checklist to help diagnose the root cause.

1. Have you set up a Community or Force.com site? See Cloud Setup for Knowledge for more info.
2. Do you have "Guest Access to the Support API" enabled for your site? See Cloud Setup for Knowledge for more info.

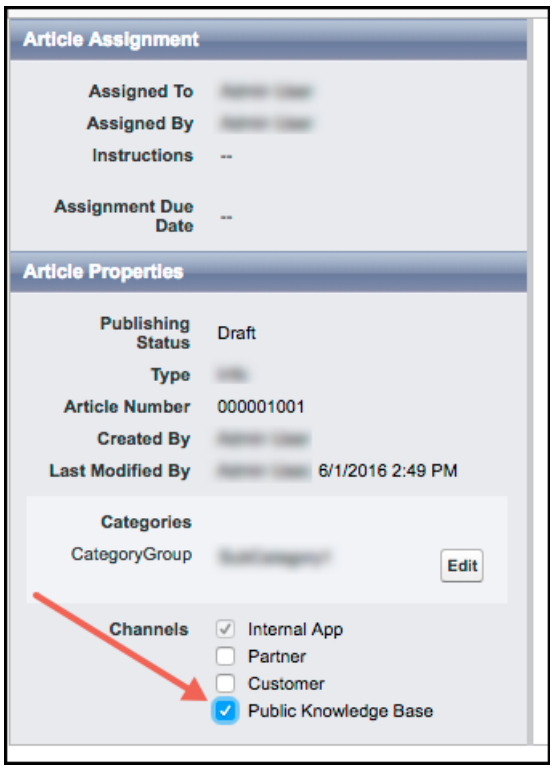
3. (For Knowledge only) Do you have **Knowledge** enabled in your org? Do you have Knowledge licenses? See Cloud Setup for Knowledge for more info.
4. (For Knowledge only) Is the user setting up the knowledge base enabled as a **Knowledge User**? See Cloud Setup for Knowledge for more info.



5. (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.



6. (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

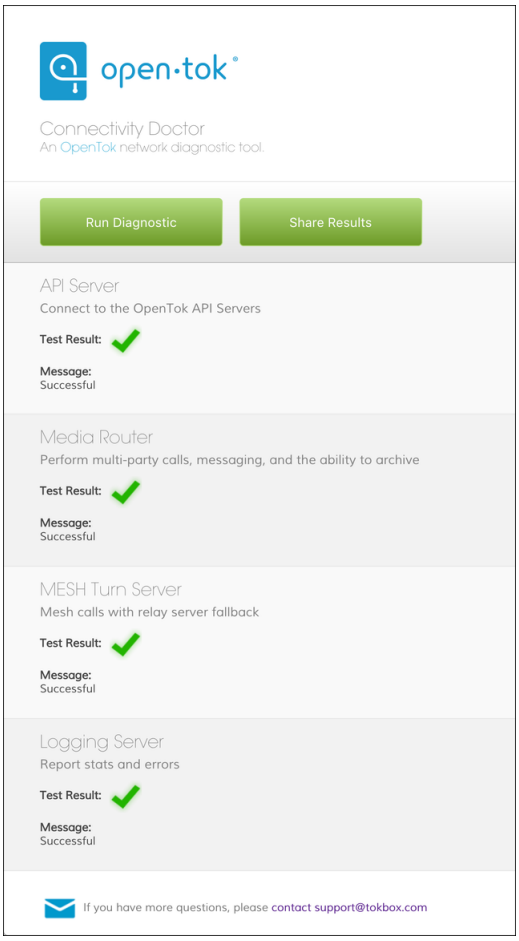
If you can't connect with an SOS agent from your app, you have network connectivity issues, possibly related to your firewall or proxy.

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.

Reference Documentation

Reference documentation for the Service SDK.

To access the reference documentation for the Service SDK for Android, see forcedotcom.github.io/ServiceSDK-Android.

Additional Resources

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

- **Service SDK**: More info about the Service SDK.
  - Service SDK Landing Page
- **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