Service Cloud Cookbook (Beta)

Version 35.0, Winter '16

@salesforcedocs

Last updated: December 10, 2015

© Copyright 2000–2015 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

Chapter 1: Service Cloud Cookbook Overview 1

Purpose and Scope 1
Audience 1
Recipe Template 2

Chapter 2: Summary of Recipes 3

Case Recipes 3
- Automatically Update Case Priority During Creation 4
- Synchronize Case Statuses in Case Hierarchies 4
- Validate Case Fields During Creation 7
- Auto-Create Contacts from Web-to-Case or Email-to-Case 8

Salesforce Console Recipes 11
- Create a Marquee Console Footer Component 11
- Create a My Notes Console Footer Component 13
- Create a Log Out Keyboard Shortcut for a Console 15

Organization Sync Recipes 16
- Sync Case Numbers between Organizations 16

Live Agent Recipes 17
- Automatically Send a Custom Message to Customers when They're Transferred to a New Agent 18
- Display Customer Names in the Agent Chat Window 20

Appendix A: Resources 23

INDEX 24

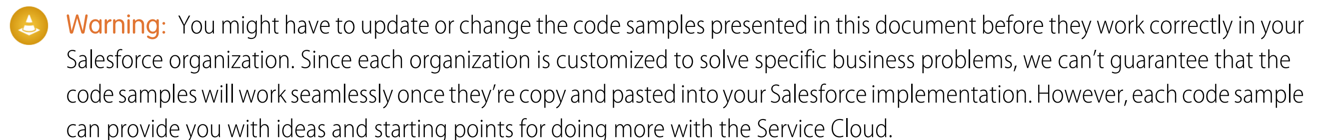
CHAPTER 1 Service Cloud Cookbook Overview

The Service Cloud provides you and your customers with a streamlined service and support experience. Most Service Cloud features are declarative and let you point-and-click your way to a successful implementation. However, with the Force.com platform, you can also code your way to faster response times, happier customers, and more productive support agents.

This document describes code samples that will help you get the most from the Service Cloud. Each description includes an approach to solving a support problem rather than an implementation for a specific feature. In this document, you'll find:

- A number of code samples that address key business problems.
- Recipes for a variety of Service Cloud features.
- New ideas for Service Cloud development.
- Prerequisites and best practices.

Important: This release contains a beta version of the *Service Cloud Cookbook*, which means that it's a high-quality document but has known limitations.



Warning: You might have to update or change the code samples presented in this document before they work correctly in your Salesforce organization. Since each organization is customized to solve specific business problems, we can't guarantee that the code samples will work seamlessly once they're copy and pasted into your Salesforce implementation. However, each code sample can provide you with ideas and starting points for doing more with the Service Cloud.

Purpose and Scope

This document is for developers who need to build or enhance Service Cloud features with the Force.com platform. This document is a catalog of code samples used successfully by many Salesforce developers and consulting architects working with the Service Cloud.

If implemented properly, these code samples enable you to do more with the Service Cloud and demonstrate scenarios that can automate, complement, or reduce your existing support processes. Salesforce's own consulting architects use these code samples as reference points during Service Cloud implementations and are actively engaged in maintaining and improving them.

As with all code samples, they cover many different scenarios, but not all. For example, while this document might touch upon Service Cloud features, such as Live Agent, Open CTI, or custom components for the Salesforce console, please see the developer guides related to those features for more comprehensive code samples. If you feel that your requirements are outside the bounds of what these code samples describe, please speak with your Salesforce representative.

Audience

This document is for developers who are already familiar with the Force.com platform.

It assumes you have knowledge of :

- JavaScript
- Visualforce

- Apex code
- Web services
- API calls and processes

This document also assumes you have a basic familiarity with the Service Cloud and the following features:

- Live agent
- Case feed
- Open CTI
- Call center
- Email-to-case
- Case management
- Salesforce knowledge
- The Salesforce console
- Entitlement management

Some of the recipes in this document might point you towards specific topics to assist you; but if you need more information on the items above, please refer to the Salesforce Help or Salesforce Developer Documentation.

Recipe Template

Each recipe follows a consistent structure. This provides consistency in the information provided in each code sample and also makes it easier to compare recipes.

Name and Business Problem Solved

The recipe's name and a brief description of the problem it solves.

Prerequisites

Any prerequisites to understanding or solving the problem. Prerequisites include features that must be available or implemented before your organization can use the recipe.

Implementation

The tasks and code samples used to implement the solution to the business problem.

CHAPTER 2 Summary of Recipes

You can use the following recipes to extend your existing Service Cloud implementation.

IN THIS SECTION:

Case Recipes

Cases are the foundation of any support team, and they're at the heart of every Service Cloud implementation. Let's make working with cases even more useful for your agents with some simple code samples.

Salesforce Console Recipes

The Salesforce console is your agents' one-stop-shop for assisting customers with support cases. The console is pretty incredible right out of the box, incorporating powerful features like Case Feed, Live Agent, Knowledge, macros, and a host of other goodies all in one place. But guess what—you can make it even more powerful with code!

Organization Sync Recipes

Organization Sync lets you access your Salesforce data at a moment's notice, even during downtime and maintenance periods. With Organization Sync, you can set up a secondary, synced organization where users can work whenever your primary organization is unavailable. Pretty neat, huh? With some simple code, you can streamline your agents' Organization Sync experience even further.

Live Agent Recipes

Live Agent is a chat product that's fully integrated with Salesforce and enables service organizations to connect with customers in real time with Web-based chats. Live Agent is a powerful customer service tool for your agents and supervisors, but we can make it even better with a couple of simple customizations and a bit of code!

Case Recipes

Cases are the foundation of any support team, and they're at the heart of every Service Cloud implementation. Let's make working with cases even more useful for your agents with some simple code samples.

USER PERMISSIONS

To view, edit, create, and delete cases:

- "Read," "Create," "Edit," and "Delete" on cases

IN THIS SECTION:

Automatically Update Case Priority During Creation

Cases with customer feedback don't require immediate attention, but support agents might work on them first unless the case `Priority` is set to Low. You can add an Apex trigger that changes case `Priority` to Low when the case `Reason` equals Feedback.

Synchronize Case Statuses in Case Hierarchies

Case hierarchies seem unreliable when the `Status` of parent and child cases aren't synchronized, meaning that a parent is marked Closed when it has a child marked Open, or all the children are marked Closed and the parent is marked Open. You can add an Apex trigger that automatically synchronizes the `Status` of child and parent cases.

Validate Case Fields During Creation

Every customer case your support team receives should include a `Subject` for easier identification and faster resolution. You can add an Apex trigger that prevents new cases from being created without a `Subject`.

Auto-Create Contacts from Web-to-Case or Email-to-Case

Web-to-Case and Email-to-Case automatically add new cases to existing contacts, but you might want to create new contacts from these cases. You can add an Apex trigger that creates contacts from Web or email cases that don't have a contact.

Automatically Update Case **`Priority`** During Creation

Cases with customer feedback don't require immediate attention, but support agents might work on them first unless the case `Priority` is set to Low. You can add an Apex trigger that changes case `Priority` to Low when the case `Reason` equals Feedback.

Prerequisites

Before you use this recipe, you should have familiarity defining Apex triggers.

See "Define Apex Triggers" in the SalesforceHelp.

Implementation

Here's how you add an Apex trigger to change case `Priority` to Low when the case `Reason` equals Feedback. The field updates whether cases are created manually by support agents or automatically by Email-to-Case or Web-to-Case.

1. From the object management settings for cases. go to Triggers.
2. Click **New**.
3. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger CheckFieldValues on Case (before insert) {
   for (Case c : Trigger.new) {
    if (c.Reason == 'Feedback')
     c.Priority = 'Low';
    }
}
```

4. Click **Save**.

Synchronize Case Statuses in Case Hierarchies

Case hierarchies seem unreliable when the `Status` of parent and child cases aren't synchronized, meaning that a parent is marked Closed when it has a child marked Open, or all the children are marked Closed and the parent is marked Open. You can add an Apex trigger that automatically synchronizes the `Status` of child and parent cases.

Prerequisites

Before you use this recipe, you must have:

- Case hierarchies set up in your organization. See "Related Cases" in the Salesforce Help.
- Familiarity defining Apex triggers and classes. See "Define Apex Triggers" and " Define Apex Classes" in the Salesforce Help.

Implementation

Here's how you add an Apex trigger that automatically synchronizes the `Status` of child and parent cases in hierarchies.

Note: Your organization might have custom values for case `Status`, but the sample code below uses Open and Closed.

1. From the object management settings for cases, go to Triggers.
2. Click **New**.
3. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger AutoCloseParentCase on Case (after insert, after update) {
   Set<Id> parentCaseIds = new Set<Id>();

   //Gather up a set of the parent cases.
   //We use Set instead of List because Set will prevent duplicates.
   for (Case c:Trigger.new) {
        if (c.ParentId!=null) {
            parentCaseIds.add(c.ParentId);
        }
   }

   if (parentCaseIds.size()>0) {
       List<Case> parentCases = [Select c.IsClosed,
                                     (Select IsClosed From Cases)
                                      From Case c
                                      Where c.Id in :parentCaseIds];

       List<Case> parentCasesToUpdate = new List<Case>();

       for (Case parent:parentCases) {
            Boolean allChildrenClosed = true;
            //Look through each of the children: If they're all closed,
            //then close the parent, otherwise keep it open.
            for (Case childCase:parent.Cases) {
                //If any child case is not closed, allChildrenClosed will flip to false

                 //and stay that way.
                 allChildrenClosed = allChildrenClosed && childCase.IsClosed;
            }

            //If the parent's closed status doesn't match its children's, then change
it.
            if (parent.IsClosed!=allChildrenClosed) {
                //INSERT VALID STATUSES FOR YOUR ORG HERE
                parent.Status = allChildrenClosed?'Closed':'Open';
                parentCasesToUpdate.add(parent);
            }
      }

      //Now we do our bulk update.
      if (parentCasesToUpdate.size()>0) {
          update parentCasesToUpdate;
      }
```

```
    }
}
```

4. Click **Save**.
5. From Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes** and click **New**.
6. In the class editor, add this test class definition, and then click **Save**.

```
@isTest
private class TestAutoCloseParentCase {
   public static List<Case> createCaseHierarchy(String startingStatus) {
     List<Case> cases = new List<Case>();

       Case parent = new Case(Subject='foo',Origin='Phone',Status=startingStatus);
       insert parent;
       cases.add(parent);

       Case child1 = new
   Case(Subject='child1',Origin='Phone',Status=startingStatus,ParentId=parent.Id);
       insert child1;
       cases.add(child1);

       Case child2 = new
   Case(Subject='child2',Origin='Phone',Status=startingStatus,ParentId=parent.Id);
       insert child2;
       cases.add(child2);

       return cases;
   }

   public static testMethod void testCloseParent() {
       List<Case> cases = createCaseHierarchy('New');

       String parentId = null;

       //Update one of the children and see that the parent stays open.
       for (Case c:cases) {
            if (c.ParentId!=null) {
                parentId = c.ParentId;
                c.Status = 'Closed';
                update c;
                break;
            }
       }

       //Shouldn't be closed yet.
       Case parent = [select IsClosed from Case where Id=:parentId];
       System.assert(parent.IsClosed==false);

       for (Case c:cases) {
            if (c.ParentId!=null && c.Status!='Closed') {
                parentId = c.ParentId;
                c.Status = 'Closed';
                update c;
```

```
                break;
            }
        }

        //Now the parent should be closed.
        parent = [select IsClosed from Case where Id=:parentId];
        System.assert(parent.IsClosed==true);

        //Now let's reopen one of the children.
        for (Case c:cases) {
             if (c.ParentId!=null && c.Status=='Closed') {
             parentId = c.ParentId;
             c.Status = 'New';
             update c;
             break;
        }
    }

   //Now the parent should be reopened again.
   parent = [select IsClosed from Case where Id=:parentId];
   System.assert(parent.IsClosed!=true);
   }
}
```

Validate Case Fields During Creation

Every customer case your support team receives should include a `Subject` for easier identification and faster resolution. You can add an Apex trigger that prevents new cases from being created without a `Subject`.

Prerequisites

Before you use this recipe, you should have familiarity defining Apex triggers.

See "Define Apex Triggers" in the SalesforceHelp.

Implementation

Here's how you add an Apex trigger to validate that cases have a `Subject`. Validation occurs whether cases are created manually by support agents or automatically by Email-to-Case or Web-to-Case.

1. From the object management settings for cases, go to Triggers.
2. Click **New**.
3. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger CheckFieldValues on Case (before insert) {
   for (Case c : Trigger.new) {
    if (c.Subject == null) {
     c.addError('Subject cannot be blank!');
      }
   }
}
```

4. Click **Save**.

Auto-Create Contacts from Web-to-Case or Email-to-Case

Web-to-Case and Email-to-Case automatically add new cases to existing contacts, but you might want to create new contacts from these cases. You can add an Apex trigger that creates contacts from Web or email cases that don't have a contact.

Prerequisites

Before you use this recipe, you must have:

- Web-to-Case or Email-to-Case set up in your organization. See "Set Up Web-to-Case," "Set Up Email-to-Case," and "Set Up On-Demand Email-to-Case," in the Salesforce Help.
- Familiarity defining Apex triggers and classes. See "Define Apex Triggers" and " Define Apex Classes" in the Salesforce Help.

Implementation

Here's how you add an Apex trigger and class that creates contacts from Web or email cases that don't have a contact.

1. From the object management settings for cases, go to Triggers.
2. Click **New**.
3. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger CaseAutocreateContact on Case (before insert) {
   List<String> emailAddresses = new List<String>();
   //First exclude any cases where the contact is set
   for (Case caseObj:Trigger.new) {
       if (caseObj.ContactId==null &&
           caseObj.SuppliedEmail!='')
       {
           emailAddresses.add(caseObj.SuppliedEmail);
       }
   }

   //Now we have a nice list of all the email addresses.
   //Let's query on it and see how manycontacts already exist.
   List<Contact> listContacts = [Select Id,
   Email From Contact Where Email in :emailAddresses];
       Set<String> takenEmails = new Set<String>();
       for (Contact c:listContacts) {
        takenEmails.add(c.Email);
   }

   Map<String,Contact> emailToContactMap = new Map<String,Contact>();
   List<Case> casesToUpdate = new List<Case>();

   for (Case caseObj:Trigger.new) {
        if (caseObj.ContactId==null &&
            caseObj.SuppliedName!=null &&
            caseObj.SuppliedEmail!=null &&
            caseObj.SuppliedName!='' &&
            !caseObj.SuppliedName.contains('@') &&
```

```
            caseObj.SuppliedEmail!='' &&
            !takenEmails.contains(caseObj.SuppliedEmail))
        {
            //The case was created with a null contact
            //Let's make a contact for it
            String[] nameParts = caseObj.SuppliedName.split(' ',2);
            if (nameParts.size() == 2)
            {
                Contact cont = new Contact(FirstName=nameParts[0],
                                            LastName=nameParts[1],
                                            Email=caseObj.SuppliedEmail,
                                            Autocreated__c=true);
                emailToContactMap.put(caseObj.SuppliedEmail,cont);
                casesToUpdate.add(caseObj);
            }
         }
    }

    List<Contact> newContacts = emailToContactMap.values();
    insert newContacts;

    for (Case caseObj:casesToUpdate) {
         Contact newContact = emailToContactMap.get(caseObj.SuppliedEmail);

         caseObj.ContactId = newContact.Id;
    }
}
```

4. Click **Save**.
5. From Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes** and click **New**.
6. In the class editor, add this test class definition, and then click **Save**.

```
public class CaseAutocreateContactTest {

    public static testMethod void testBulkContactsGetCreated() {
    List<Case> newCases = new List<Case>();
    for (Integer i = 0; i<100; i++) {
         Case c = new Case(SuppliedEmail='jdoe_test_test@doe.com' + i,
                           SuppliedName='John Doe' + i,
                           Subject='Feedback - Something' + i);

         newCases.add(c);
    }
    insert newCases;
    System.debug('here');
    List<Id> newCaseIds = new List<Id>();
    for (Case caseObj:newCases) {
         newCaseIds.add(caseObj.Id);
    }

    List<Case> updatedCases = [Select ContactId From Case Where Id in :newCaseIds];

    for (Case caseObj:updatedCases) {
```

```
        System.debug(caseObj.Id + ' ' + caseObj.ContactId);
        System.assert(caseObj.ContactId!=null,'There should be no null contacts');
      }
    }

    public static testMethod void testContactGetsCreated() {
       Case c = new Case(SuppliedEmail='jdoe_test_test@doe.com',
                           SuppliedName='John Doe',
                           Subject='Feedback - Something');
       insert c;

       List<Contact> johnDoes = [select Id from
       Contact where Email='jdoe_test_test@doe.com'];

       //There should be only 1: The trigger should not have created another.
       System.assert(johnDoes.size()==1, 'There should be one John Doe!');

       Case caseObj = [select ContactId from Case where Id=:c.Id];
       System.assert(caseObj.ContactId!=null,'There should be no
       null contact on the case');
    }

    public static testMethod void testNoDupesAreCreated() {
       Contact cnt1 = new Contact(FirstName = 'John',
                           LastName = 'Doe',
                           Email='jdoe_test_test@doe.com');

       insert cnt1;

       Case case1 = new Case(SuppliedEmail='jdoe_test_test@doe.com',
                           SuppliedName='John Doe',
                           Subject='Feedback - Something');

       insert case1;

       List<Contact> johnDoes = [select Id
       from Contact where Email='jdoe_test_test@doe.com'];

       //There should be only 1: The trigger should not have created another.
       System.assert(johnDoes.size()==1, 'There should be only one John Doe!');
    }

    public static testMethod void testEmailNameDoesntGetCreated() {
        Case c = new Case(SuppliedEmail='testEmailNameDoesntGetCreated@doe.com',
                            SuppliedName='testEmailNameDoesntGetCreated@doe.com',
                            Subject='Feedback - Something');
        insert c;

        List<Contact> johnDoes = [select Id
        from Contact where Email='testEmailNameDoesntGetCreated@doe.com'];

        //there should be only 1 -- the trigger should not have created another
        System.assert(johnDoes.size()==0, 'There should be no John Does!');
```

```
        }
}
```

Salesforce Console Recipes

The Salesforce console is your agents' one-stop-shop for assisting customers with support cases. The console is pretty incredible right out of the box, incorporating powerful features like Case Feed, Live Agent, Knowledge, macros, and a host of other goodies all in one place. But guess what—you can make it even more powerful with code!

USER PERMISSIONS

To set up a for Service: Salesforce Console
- "Customize Application"

IN THIS SECTION:

Create a Marquee Console Footer Component
Often it's challenging to keep support agents informed of important announcements or team-wide issues. You can create a Marquee, or scrolling-text tool, in the console footer by adding a Visualforce page to a console component.

Create a My Notes Console Footer Component
Make it easier for support agents to capture notes quickly in a Salesforce console. You can create a My Notes tool in the footer by adding a Visualforce page and Apex class to a console component.

Create a Log Out Keyboard Shortcut for a Console
Reduce the amount of time and clicks in takes to perform actions in a Salesforce console by creating keyboard shortcuts, such as one for logging out. You can add a Visualforce page with an Apex controller to a console component to let support agents log out with the press of a few buttons.

Create a *Marquee* Console Footer Component

Often it's challenging to keep support agents informed of important announcements or team-wide issues. You can create a Marquee, or scrolling-text tool, in the console footer by adding a Visualforce page to a console component.

Prerequisites

Before you use this recipe, you must have:

- A Chatter feed set up in your organization in which to include announcements or scrolling text for a marquee. See "Chatter" in the Salesforce Help.
- A Salesforce console set up in your organization. See "Set Up a Salesforce Console for Service" in the Salesforce Help.
- Familiarity with custom console components and Visualforce pages. See " Add Console Components to Apps" and " Defining Visualforce Pages" in the Salesforce Help.

Implementation

Here's how you add a Visualforce page to a console component to create a Marquee tool in the footer.

1. From Setup, enter `Visualforce Pages` in the `Quick Find` box, then select **Visualforce Pages**, and click **New**.
2. In the page editor, add this page definition, and then click **Save**.

```
<apex:page showHeader="false">
   <style>
```

```
        body {
          padding: 10px;
          }
    </style>
     <div style="margin-left:auto; margin-right:auto;width:50%;">
        <apex:form>
            <apex:commandButton value="Start" id="startBtn" onclick="scrollButtonText();

              return false;" style="margin-left:10px"/>
             <apex:commandButton value="Stop" id="stopBtn" onclick="setButtonText();
              return false;" style="margin-left:10px"/>
        </apex:form>
     </div>

     <apex:includeScript value="/support/console/29.0/integration.js"/>
     //Here's the unique ID of a Chatter feed from the feed's URL.
     <chatter:feed entityId="0F9D00000000qjR" rendered="true"/>
     <script>
       function srcUp(url) {
          sforce.console.openPrimaryTab(null, url, true);
        }
       setInterval(function(){window.location.href = window.location.href;},60000);
         function getFeedItemBody() {
           var feeds = [];
           var elements = document.getElementsByClassName('feeditemtext');
           for (var i=0; i<elements.length; i++) {
             if (elements[i].innerHTML) feeds.push(elements[i].innerHTML);
           }
           return feeds.join(" | ");
         }

         var feedItems = getFeedItemBody();
         scrollButtonText();
         function scrollButtonText() {
           if (! feedItems)
               setButtonText();
           else {
             sforce.console.setCustomConsoleComponentButtonText(feedItems, function() {
               sforce.console.scrollCustomConsoleComponentButtonText(150, 5,
               true, function(result){});
             });
         }
     }
      function setButtonText() {
       sforce.console.setCustomConsoleComponentButtonText('Alerts');
      }
     </script>
</apex:page>
```

3. In Setup, enter `Custom Console Components` in the `Quick Find` box, then select **Custom Console Components**.
4. Click **New** to create a custom console component that points to the Visualforce page above, and click **Save**.
5. Assign the custom console component to each console where you want users to access it. See "Assign a Console Component to an App" in the Salesforce Help.

Create a *My Notes* Console Footer Component

Make it easier for support agents to capture notes quickly in a Salesforce console. You can create a My Notes tool in the footer by adding a Visualforce page and Apex class to a console component.

Prerequisites

Before you use this recipe, you must have:

- A My_Notes custom object with a Notes rich text field set up in your organization. See "Define a Custom Object" and "Create Custom Fields" in the Salesforce Help.
- A Salesforce console set up in your organization. See "Set Up a Salesforce Console for Service" in the Salesforce Help.
- Familiarity with custom console components, Apex classes, and Visualforce pages. See " Add Console Components to Apps," " Define Apex Classes," and " Defining Visualforce Pages" in the Salesforce Help.

Implementation

Here's how you add an Apex class and a Visualforce page to a console component to create a My Notes tool in the footer.

1. From Setup, enter `Apex Classes` in the `Quick Find` box, then select **Apex Classes**, and click **New**.
2. In the class editor, enter this class definition, and then click **Save**.

```
public class MyNotesController {
   public void setNotes() {
       myNotepad.note__c = Notes;
       update myNotepad;
   }

   My_Notes__c myNotepad;
   Integer notesCount;
   public String Notes{ get; set;}

   public MyNotesController() {
     notesCount = [Select COUNT() from My_Notes__c where ownerId =
     :UserInfo.getUserId()];
   }

   public void init() {
      My_Notes__c  notesObject;
      if (notesCount == 0) {
         notesObject = new My_Notes__c();
         notesObject.ownerId = UserInfo.getUserId();
         insert notesObject;
      } else {
         notesObject = [Select Note__c from My_Notes__c where ownerId =
         :UserInfo.getUserId()LIMIT 1];
      }
      myNotepad = notesObject;
      Notes = notesObject.note__c;
   }
}
```

3. From Setup, enter `Visualforce Pages` in the `Quick Find` box, then select **Visualforce Pages**, and click **New**.

4. In the page editor, add this page definition, and then click **Save**.

```
<apex:page controller="MyNotesController" action="{!init}">
   <apex:includeScript value="/support/console/25.0/integration.js"/>
    <style>
      body { margin: 0; padding: 0; overflow: hidden; }
      html { height:100%;}
      html body.sfdcBody {
          //Background-color: #FFFFCC;
          padding:0px;
          margin: 0px;
          height:100%;
       }
    </style>

    <textarea id="notesInput" style="border:none;background-color:#FFFFCC;
    width:100%; height:100%;" onkeyup="saveNotes();"
    >
      {!Notes}
    </textarea>

    <apex:form >
       <apex:actionFunction action="{!setNotes}" name="setNotesJS" reRender="">
            <apex:param name="note" assignTo="{!Notes}" value=""/>
       </apex:actionFunction>
    </apex:form>

    <script>
       var notesTextArea = document.getElementById('notesInput');
        function saveNotes() {
           setNotesJS(notesTextArea.value);
        }

        var listener = function (result) {
           sforce.console.setCustomConsoleComponentWindowVisible(true);
           document.getElementById('notesInput').innerHTML +=
           '\nMessage Received: ' + result.message;
        };

        //Add a listener for the 'SampleEvent' event type.
        sforce.console.addEventListener('updateMyNotes', listener);
    </script>
</apex:page>
```

5. In Setup, enter `Custom Console Components` in the `Quick Find` box, then select **Custom Console Components**.
6. Click **New** to create a custom console component that points to the Visualforce page above, and click **Save**.
7. Assign the custom console component to each console where you want users to access it. See "Assign a Console Component to an App" in the Salesforce Help.

Create a Log Out Keyboard Shortcut for a Console

Reduce the amount of time and clicks in takes to perform actions in a Salesforce console by creating keyboard shortcuts, such as one for logging out. You can add a Visualforce page with an Apex controller to a console component to let support agents log out with the press of a few buttons.

Prerequisites

Before you use this recipe, you must have:

- A Salesforce console set up in your organization. See "Set Up a Salesforce Console for Service" in the Salesforce Help.
- A custom keyboard shortcut defined for a console that launches an event called `customShortcut.Logout`. See "Customize Keyboard Shortcuts for a Salesforce Console" in the Salesforce Help.
- Familiarity with custom console components, Apex classes, Visualforce pages, and the Salesforce Console Integration Toolkit. See " Add Console Components to Apps," " Define Apex Classes," and " Defining Visualforce Pages" in the Salesforce Help; see *Salesforce Console Integration Toolkit Developer's Guide* in the Salesforce Developer Library.

Implementation

Here's how you add a Visualforce page with an Apex controller to a console component to let support agents log out with a keyboard shortcut.

1. From Setup, enter *`Visualforce Pages`* in the `Quick Find` box, then select **Visualforce Pages**, and click **New**.
2. In the page editor, add this page definition, and then click **Save**.

```
<apex:page>
<apex:includeScript value="/support/console/33.0/integration.js"/>
   <script>
       var logoutEventHandler = function()  {
          top.location = '/secur/logout.jsp';
      }
       sforce.console.addEventListener('customShortcut.Logout', logoutEventHandler);
    </script>
</apex:page>
```

3. In Setup, enter *`Custom Console Components`* in the `Quick Find` box, then select **Custom Console Components**.
4. Click **New** to create a custom console component that points to the Visualforce page above.
5. Click `Hide` since there's no need to display the keyboard shortcut in a console's footer, and click **Save**.
6. Assign the custom console component to each console where you want users to access it. See "Assign a Console Component to an App" in the Salesforce Help.
7. Enable keyboard shortcuts for each console where you want the new shortcut available, and add the following details for the shortcut, then click **Save**. See "Customize Keyboard Shortcuts for a Salesforce Console" in the Salesforce Help.

Console Action:	Logout
Console Event Name:	`customShortcut.Logout`
Key Command:	CTRL+SHIFT+L

Organization Sync Recipes

Organization Sync lets you access your Salesforce data at a moment's notice, even during downtime and maintenance periods. With Organization Sync, you can set up a secondary, synced organization where users can work whenever your primary organization is unavailable. Pretty neat, huh? With some simple code, you can streamline your agents' Organization Sync experience even further.

USER PERMISSIONS

To set up a replication connection:

- "Manage Connections"

IN THIS SECTION:

Sync Case Numbers between Organizations
If your organization use Organization Sync to let your users access Salesforce data when Salesforce is down for maintenance, you might have noticed that case numbers aren't consistent between your primary and secondary organizations. That makes it harder for your users to look up information from one organization to another. But never fear! With a simple Apex trigger, you can sync case numbers to be consistent between your primary and secondary organizations.

Sync Case Numbers between Organizations

If your organization use Organization Sync to let your users access Salesforce data when Salesforce is down for maintenance, you might have noticed that case numbers aren't consistent between your primary and secondary organizations. That makes it harder for your users to look up information from one organization to another. But never fear! With a simple Apex trigger, you can sync case numbers to be consistent between your primary and secondary organizations.

To fix this problem, we first create a custom text field on case records. Next, we set up a trigger that copies the original case number into our new field in your primary organization. When these cases are copied to your secondary organization, the custom field and the case number go along for the ride. That way, case numbers match between the two organizations. Let's get to it!

Prerequisites

Before we implement our trigger, let's update a few settings in your primary and secondary organizations so the implementation works correctly.

Before we get to the fun stuff—the code, that is—make sure that you make these little tweaks to *both* your primary and secondary organizations. If both organizations aren't updated properly, the code doesn't work correctly.

1. Add a custom text field called `Synced Case Number` to your case pages. See "Create Custom Fields" in the Salesforce Help.
2. Update the page layouts of the case pages that you added your new field to. Add your Synced Case Number field, and then remove or hide the system-generated Case Number field. See "Edit Page Layouts for Standard Objects" in the Salesforce Help.

Implementation

With just a few easy setup steps and a little Apex code, we can get both of your organizations ready to go in no time.

Create Your Trigger

Now that we've set up our case page layouts, let's implement a trigger that does all the heavy lifting for us. You'll create your trigger in both your primary and secondary organizations.

The trigger copies the original, system-generated case number to our new Synced Case Number field on new and existing cases. Then, when your primary organization syncs information to your secondary organization, the values of the Synced Case Number field matches. Here's the code for our trigger:

```
trigger UpdateCaseNumber on Case (after insert) {
    List<Case> cases = new List<Case>();
        for (Case c : Trigger.new) {
          if (c.CustomCaseNumber__c == null) {
            Case caseToUpdate = new Case(Id=c.Id);
            caseToUpdate.CustomCaseNumber__c = c.CaseNumber;
            cases.add(caseToUpdate);
    }
  }
  database.update(cases);
}
```

Let's look at what's going on in our code. For all cases, this trigger copies the system-generated case number—that is, the number that Salesforce assigns to the case when it's created. Then, our trigger inserts that number into our `Synced Case Number` field and updates the case. When your primary organization syncs with your secondary organization, the case number in the `Synced Case Number` field is the same in both organizations. Easy as pie (and almost as tasty)!

Finishing Touches

We're almost done! All we have to do now is change how the system-generated case numbers are displayed in your secondary organization. That way, any new system-generated case numbers remain distinct from any existing case numbers in your primary organization.

But don't worry: The system-generated `Case Number` field isn't visible from your case layouts (Remember when we removed it?), and your agents can search for the Synced Case Number instead. The Synced Case Number is the same between both organizations, so agents can find corresponding cases.

1. In your secondary organization *only*, append `2-` to the beginning of the default value in the `Case Number` field. To do so, edit the `Case Number` field, and then change the value of the `Display Format` field to *`2-{00000000}`*.

Live Agent Recipes

Live Agent is a chat product that's fully integrated with Salesforce and enables service organizations to connect with customers in real time with Web-based chats. Live Agent is a powerful customer service tool for your agents and supervisors, but we can make it even better with a couple of simple customizations and a bit of code!

EDITIONS

Available in: Salesforce Classic

Live Agent is available in: **Performance** Editions and **Developer** Edition organizations that were created after June 14, 2012

Live Agent is available for an additional cost in: **Enterprise** and **Unlimited** Editions

IN THIS SECTION:

Automatically Send a Custom Message to Customers when They're Transferred to a New Agent
Does your organization use automatic greeting messages to greet customers when they join a new chat? Ever notice how customers receive the same greeting message again when they're transferred to a new agent? Improve your customers' chat experience by setting up a custom component that will send them a customized transfer message instead, all in a couple of easy steps.

Display Customer Names in the Agent Chat Window
Your customers are people too! With a bit of coding, you can display your chat customers' names to your agents in the agent chat window to make your agents' chat experience more personal and efficient.

Automatically Send a Custom Message to Customers when They're Transferred to a New Agent

Does your organization use automatic greeting messages to greet customers when they join a new chat? Ever notice how customers receive the same greeting message again when they're transferred to a new agent? Improve your customers' chat experience by setting up a custom component that will send them a customized transfer message instead, all in a couple of easy steps.

To fix this problem, we'll create a hidden component that works in the console to intercept transferred chats. The component detects that an agent transfers a chat to another agent. When the second agent accepts the transfer, the component automatically sends a new, custom transfer message to the customer instead of repeating the automatic greeting message.

Prerequisites

Before we get started, verify that your environment is set up correctly.

To use this recipe, make sure that:

- Live Agent is enabled and configured in the Salesforce console for your organization. See "Add Live Agent to the Salesforce Console" in the Salesforce Help.
- Your organization sends automatic greeting messages when an agent accepts a chat with a customer. See "Live Agent Configuration Settings" in the Salesforce Help.

Implementation

With some simple JavaScript code and a few setup steps, you'll have your customized transfer message configured in no time.

Create a Visualforce Page for Your Custom Console Component

First, we have to create a Visualforce page that will be the basis for our custom console component.

For this example, let's name this Visualforce page "Customize Transfer Greeting."

Here's the code for our Visualforce page:

```
<apex:page>
 <apex:includeScript value="/support/console/32.0/integration.js"/>
 <script type="text/javascript">
        sforce.console.chat.onChatStarted(function(result) {
            sforce.console.chat.onAgentSend(result.chatKey,
            function(resOnAgentSend) {
                /*Monitors every chat message sent by the agent to verify
                that the chat has not been transferred.*/
                if(!(resOnAgentSend.isAutoGreeting && resOnAgentSend.isTransferred)) {
                    sforce.console.chat.sendMessage(result.chatKey,
                    resOnAgentSend.content, function(resSendMessage) {
                        if(!resSendMessage.success){
                            return;
                        }
                    });
                }
                else {
                    /*Sends your customized transfer message to the customer
                    if the chat has been transferred to a new agent. Replace
                    the text in quotation marks to whatever you want your transfer
```

```
                     message to say.*/
                     sforce.console.chat.sendMessage(result.chatKey, "Hi! Thank you
                     for your patience. Give me one second while I review your chat." ,
                         function(resSendMessage) {
                         if(!resSendMessage.success){
                             return;
                     }
                 });
             }
         });
     });
</script>
</apex:page>
```

After a chat begins, this code monitors every message that the agent sends to the customer. If the agent sends a transfer request at any point during the chat, the component "catches" the transfer request and sends a custom message to the customer when a new agent takes over the chat.

But what if you don't want to send a new message when the chat is transferred? That's easy too. Just remove the `else` statement from the code. The new code sample looks like this:

```
<apex:page >
 <apex:includeScript value="/support/console/32.0/integration.js"/>
 <script type="text/javascript">
        sforce.console.chat.onChatStarted(function(result) {
            sforce.console.chat.onAgentSend(result.chatKey, function(resOnAgentSend) {
               if(!(resOnAgentSend.isAutoGreeting && resOnAgentSend.isTransferred)) {
                 sforce.console.chat.sendMessage(result.chatKey, resOnAgentSend.content,
                     function(resSendMessage) {
                     if(!resSendMessage.success){
                         return;
                       }
                    });
              }
        });
    });
</script>
</apex:page>
```

Create Your Custom Console Component

Next, we need to incorporate our Visualforce code into a custom console component before we can add the component to the console.

1. From Setup, enter *`Custom Console Components`* in the `Quick Find` box, then select **Custom Console Components**.
2. Click **New**.
3. Fill out the following settings.

   These are the only settings that you need to specify for this component, so you can ignore any of the settings that aren't listed here.

Setting Name	What should I set it to?
`Name`	The name of your component. For this example, let's call our component *`Customize_Transfer_Greeting`* so it matches the name of our Visualforce page.

Setting Name	What should I set it to?
`Hide`	Selected. This is important because you don't want your agents to see the component working in the console. Hiding the component lets the component run in the background whenever your agents are working, without taking up space on their screens.
`Button Name`	The name that appears on the footer widget when the component appears in the console. Because we hid the component, this won't be visible to users. However, because this is a required field when we create our component, let's call it *`Customize Transfer Greeting`*.
`Component`	Visualforce Page. In the lookup field, search for and select the Customize Transfer Greeting Visualforce page that you created earlier.

4. Click **Save**.

And that's it! Your customers can start enjoying their new transfer experience, and you can sit back and relax.

Add Your Custom Component to the Console

Great! We've created our component, but it won't start working until we add it to the console. Let's go ahead and add the Customize Transfer Greeting custom component to your Salesforce console so that it can start working for your agents.

1. In Setup, enter *`Apps`* in the `Quick Find` box, then select **Apps**.
2. Click **Edit** next to the Salesforce console that you want to add the component to.
3. In the Choose Console Components section, add the Customize Transfer Message component to the list of selected items.
4. Click **Save**.

Display Customer Names in the Agent Chat Window

Your customers are people too! With a bit of coding, you can display your chat customers' names to your agents in the agent chat window to make your agents' chat experience more personal and efficient.

In a normal Live Agent implementation, customers can see your agents' names in the chat log, but your agents can't see the names of the customers that they're chatting with. To fix this problem, we'll use a simple JavaScript function that works with the Pre-Chat API to retrieve the customer's name from a pre-chat form and publish it to the chat window. Let's get started!

Prerequisites

Before we get started, verify that your environment is set up correctly.

For this recipe to work, make sure that:

- Live Agent is enabled and configured in the Salesforce console for your organization. See "Add Live Agent to the Salesforce Console" in the Salesforce Help.

- Your Live Agent implementation uses a pre-chat form to collect your customer's name. See "Pre-Chat Forms and Post-Chat Pages" in the Salesforce Help.

Implementation

Luckily, it's easy to include the customer's name in the agent's chat window. All we need to do is to add some JavaScript code in your pre-chat page.

Create Your Pre-Chat Form

First, let's create a pre-chat form that we'll use to collect some information about the customer, such as the customer's name and email address.

We implemented this pre-chat form with the Pre-Chat API and some standard JavaScript. Let's take a look at the code:

```
1 <form action="https://liveagentrocks.com" id="preChatFormSubmit" method="post"
  onsubmit="setName()">
2 <input id="firstname" name="liveagent.prechat:FirstName" type="hidden" value="" />
3 <input id="lastname" name="liveagent.prechat:LastName" type="hidden" value="" />
4 <input id="email" name="liveagent.prechat:Email" type="hidden" value="" />
5 <input id="prechat_field" name="liveagent.prechat.name" type="hidden" value="" />
6
7 <input name="liveagent.prechat.query:Email" type="hidden" value="Contact,Contact.Email"
 />
8
9 <input name="liveagent.prechat.save:FirstName" type="hidden" value="FirstName__c" />
10 <input name="liveagent.prechat.save:LastName" type="hidden" value="LastName__c" />
11 <input name="liveagent.prechat.save:Email" type="hidden" value="eMail__c" />
12 </form>
```

The pre-chat form has three fields that are visible to the customer: `First Name`, `Last Name`, and `Email`. There's also a hidden field for the `liveagent.prechat.name` on line 5 that the customer can't see. Note, too, that we call the `setName()` function when the customer submits the form. We'll come back to that later.

Create Your JavaScript Function to Set the Customer's Name

First, let's create a pre-chat form that we'll use to collect some information about the customer, such as the customer's name and email address.

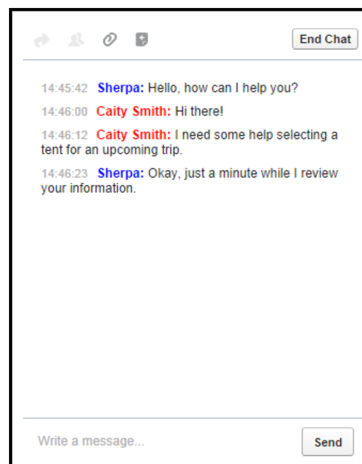
To change the customer's name in the chat window, we use the `liveagent.prechat.name` method in a JavaScript function that's called `setName()` to override the visitor name. We include the `setName()` function in our pre-chat page.

```
1 <script type="text/javascript">
2   function setName() {
3     document.getElementById("prechat_field").value =
4         document.getElementById("firstname").value + " " +
          document.getElementById("lastname").value;
5     return true;
6     }
7 </script>
```

Remember when we set our `onsubmit` event to call the `setName()` function in our pre-chat form code?

```
<form action="https://liveagentrocks.com" id="preChatFormSubmit" method="post"
onsubmit="setName()">
```

When the customer submits the form, the `setName()` function works its magic. The values that the customer provided in the `First Name` and `Last Name` fields appear to the agent as the visitor's name in the chat window.



And that's it! Happy coding!

APPENDIX A Resources

Below are some of the documents available to you online at the Salesforce Developer's Library and the Salesforce Success Community. Use these resources for further development on the Service Cloud.

Developer Documentation

- *SOAP API Developer's Guide*
- *Force.com REST API Developer's Guide*
- *Force.com Streaming API Developer's Guide*
- *Apex Code Developer's Guide*
- *Visualforce Developer's Guide*
- *Customizing Case Feed with Code*
- *Live Agent Developer's Guide*
- *Live Agent REST API Developer's Guide*
- *Open CTI Developer's Guide*
- *Salesforce Knowledge Developer's Guide*
- *Salesforce Console Integration Toolkit Developer's Guide*

Workbooks and Implementation Guides

- *Service Cloud Workbook*
- *Force.com Workbook*
- *Apex Workbook*
- *Visualforce Workbook*
- *Implementing Case Feed*
- *Live Chat for Administrators*
- *Case Management Implementation Guide*
- *Salesforce Console Implementation Guide*
- *Entitlement Management Implementation Guide*

INDEX

A

Audience 1

C

Case hierarchies
- Synchronize case statuses 4

Cases
- Auto-create contacts 8
- Field updates 4
- Synchronize cases statuses 4
- Synchronize statuses 5
- Update fields 4
- Validate 7
- Validation 7

D

Display customer names in the agent chat window
- create pre-chat form 21–22
- implementation 21
- prerequisites 20
- template 20

K

Keyboard shortcut
- Log out 15

L

Live Agent recipes 17

O

Organization Sync 16–17
Organization Sync recipes 3, 11, 16
Overview of service cloud development 1

P

Prerequisites 4, 7–8, 11, 13, 15
Purpose and scope 1

R

Recipe
- Case field updates 4
- Case validation 7
- Create a log out keyboard shortcut 15
- Create a marquee console component 11
- Create a notes console component 11, 13
- Create a scrolling-text console component 11
- Create contacts from Email-to-Case 8
- Create contacts from the Web-to-Case 8
- Synchronize cases statuses 4–5
- template 2

Recipes
- Summary 3

Resources
- Salesforce 23

S

Salesforce console
- Keyboard shortcut 15
- Marquee footer component 11
- Notes footer component 13
- Scrolling-text footer component 11

Send a custom chat transfer message
- add custom component to console 20
- create custom console component 19
- create Visualforce page 18
- implementation 18
- prerequisites 18
- template 18

Sync Auto-Generated Record Numbers Between Organizations 16–17

T

Template 2