



# Getting Started with SuiteFlex

March 18, 2009  
Release 2009 Version 1

**Copyright NetSuite, Inc. 2009 All rights reserved.**

Getting Started with SuiteFlex

This document is the property of NetSuite, Inc., and may not be reproduced in whole or in part without prior written approval of NetSuite, Inc.

**Trademarks**

NetSuite, NetSuite Basic, NetSuite CRM+ and NetSuite CRM are provided by NetSuite, Inc. and NetSuite is a trademark of NetSuite, Inc.

Oracle is a registered trademark of Oracle Corporation.

Other product names mentioned in this document may be trademarks, servicemarks, or tradenames of their respective companies and are hereby acknowledged.

# Contents

## SuiteFlex Overview

What is SuiteFlex? .....	5
--------------------------	---

## SuiteBuilder

## SuiteScript

<b>Client Scripts</b> .....	<b>8</b>
<i>Writing Your First Client Script</i> .....	9
<b>User Event Scripts</b> .....	<b>12</b>
<i>The Type Argument</i> .....	12
<i>Business Process Automation with User Event Scripts</i> .....	15
<i>Using User Event Scripts to Enhance Forms</i> .....	18
<b>Scheduled Scripts</b> .....	<b>20</b>
<b>SuiteScript UI Objects</b> .....	<b>23</b>
<i>Native Look-and-Feel</i> .....	23
<i>InlineHTML UI Objects</i> .....	24
<b>Suitelets</b> .....	<b>26</b>
<i>Building Custom Workflow with Suitelets</i> .....	29
<i>Backend Suitelets</i> .....	30
<b>Portlet Scripts</b> .....	<b>32</b>
<b>SuiteScript D-bug</b> .....	<b>32</b>

## SuiteBundler

<b>SuiteBundler Basics</b> .....	<b>36</b>
<i>Creating SuiteBundles</i> .....	37
<i>Installing and Updating SuiteBundles</i> .....	42
<b>Installing, Maintaining and Supporting Vertical Solutions Using SuiteBundler</b>	<b>43</b>
<i>Direct Bundle Install</i> .....	43
<i>Installing bundles from the Repository</i> .....	45
<i>Wrapper Bundle Deployment Model</i> .....	47
<i>Advanced Variations of the Wrapper Bundle Model</i> .....	51

## SuiteTalk Web Services

<b>Record Types and Metadata Supported</b> .....	<b>55</b>
<b>SuiteTalk Operations</b> .....	<b>56</b>
<b>How to Write a SuiteTalk Application</b> .....	<b>56</b>
<i>Platform Specific Considerations</i> .....	57

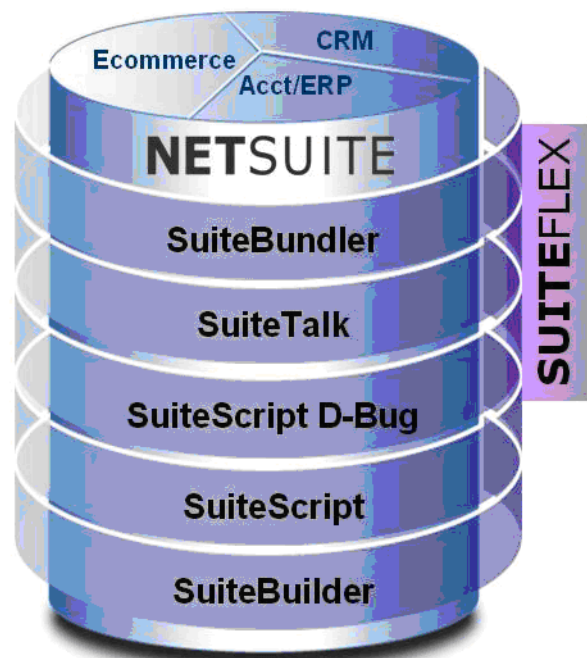
*Roles and Permissions Considerations* ..... 58  
*License and Concurrency Considerations* ..... 59

# SuiteFlex Overview

The *Getting Started with SuiteFlex Guide* is meant to enable ISVs and developers new to NetSuite and NS BOS to quickly get up to speed. The intended audience is developers and application designers, although IT staff of NetSuite customers will also find this material helpful. Due to the technical nature of the content, readers should possess prior software development and/or system analysis experience. Knowledge of JavaScript and Object Oriented development are helpful but not required.

## What is SuiteFlex?

SuiteFlex is the technology toolkit for customization, business process automation and data integration within the [NetSuite BOS platform](#). For developers, SuiteFlex provides a complete suite of tools and APIs for verticalizing NetSuite.



- **SuiteBuilder.** The SuiteBuilder is a set of UI-driven, point-and-click tools to configure NetSuite. It allows users to easily customize NetSuite by adding custom fields, custom lists, custom record types and custom forms. Customizations made with the SuiteBuilder take effect immediately and are carried forward seamlessly across NetSuite version upgrades.
- **SuiteScript.** SuiteScript is an extension of the JavaScript APIs that gives developers the ability to extend NetSuite beyond the capabilities offered by SuiteBuilder. Developers

can use SuiteScripts to extend NetSuite, automate business processes, or build new applications atop of NetSuite.

- **SuiteScript D-bug.** SuiteScript D-Bug is a debugging tool to aid developers in troubleshooting server side SuiteScripts.
- **SuiteBundler.** The SuiteBundler allows vertical solutions to be quickly and easily deployed to multiple NetSuite accounts with little to no migration resources needed from the solution provider.
- **SuiteTalk Web Services.** SuiteTalk exposes NetSuite as a web service to make integration with legacy systems and third party vertical applications easy. SuiteTalk relies on industry standards (SOAP, XML) to data communication with external hosts. The platform neutrality nature allows developers to choose their preferred programming language to develop SuiteTalk applications. (Note: Currently only .Net, Java, and PHP SuiteTalk applications are supported.)

# SuiteBuilder

As a hosted, integrated business application suite, NetSuite offers a rich set of records for ERP, CRM and HR purposes – all in a single coherent package. The most commonly used objects/records for all businesses are provided. These records include customers, employees and sales orders.

In addition to the standard records and their respective forms, NetSuite also provides a set of customization tools for administrators and IT staff to tailor their NetSuite accounts. These customization tools, named SuiteBuilder, are UI-driven and designed to be easy to use. Customizations made on an account take effect immediately. They are preserved between NetSuite versions, which means they continue to work on the account after the application is upgraded to a new version.

The SuiteBuilder allows standard records to be customized to hold additional data; custom forms can be created to give specific users access to specific fields; custom record types can be created to be used as objects unique to a vertical.

For details on SuiteBuilder's features and capabilities, please refer to the *SuiteBuilder Guide* in the NetSuite Help Center.

# SuiteScript

SuiteScript is an extension of the JavaScript APIs that gives developers the ability to extend NetSuite beyond the capabilities offered through point-and-click customization. Developers can use SuiteScripts to validate form data, automate business processes, build new functions, or even build new applications atop NetSuite.

SuiteScripts can run in a browser or on a NetSuite server. Some SuiteScript APIs are designed to run in both a browser and a NetSuite server side. These are referred to as Core SuiteScript APIs. Other APIs execute browser side only, while others execute only on a NetSuite server. Please refer to the *SuiteScript Developer Guide* for more details.

There are five types of SuiteScripts:

- [Client Scripts](#) (form level and record level)
- [User Event Scripts](#)
- [Scheduled Scripts](#)
- [SuiteScript UI Objects and Suitelets](#)
- [Portlet Scripts](#)

The majority of NetSuite forms, standard records, customization objects and their event points are exposed to SuiteScript APIs. By using the different types of SuiteScript, developers can develop sophisticated SuiteFlex-based vertical solutions

**Tip:** *Since all records and fields are referenced by their internal IDs in SuiteScripts, it is recommended developers turn on the "Show Internal ID" (Home → Set Preferences) preference. This preference displays an Internal ID column for all record lists, and displays the field IDs for field level help pop-up boxes.*

## Client Scripts

Client scripts are SuiteScripts executed in the browser. They are used on most standard records, custom record types and custom NetSuite pages (for example, Suitelets). Generally, client scripts are used to validate user-entered data and auto-populating fields/sublists at various form events – such as initialization, field changed, save record. Please refer to the *client script documentation* on the form events exposed to client scripts.

Another common use case for client scripts is to source data from an external data source to a field. This is accomplished by using the API `nlapiRequestURL`.

In the overall context of a SuiteFlex based application, client scripts are used to enforce data in the application's UI layer. Although developers can write client scripts that create/delete/update existing records, it is a practice that is discouraged because generally database logic should be performed by server-side scripts.

**Tip:** If there is a compelling use case for a client script to create/update/delete an existing record, then it can be done by delegating this task to a Suitelet. This design approach also has the added benefit of protecting the intellectual property because server scripts can be bundled with the source code hidden.

## Writing Your First Client Script

A great way to get started with client scripts is to deploy a very simple script that has a function on every exposed event. Consider the following client script file:

```
function myPageInit(type)
{
    alert('myPageInit');
    alert('type=' + type);
}

function mySaveRecord()
{
    alert('mySaveRecord');
    return true;
}

function myValidateField(type, name, linenum)
{
    if(name == 'custentity_my_custom_field')
    {
        alert('myValidateField');
        alert('type=' + type);
        alert('name=' + name);
        alert('linenum=' + linenum);
    }
    return true;
}

function myFieldChanged(type, name, linenum)
{
    alert('myFieldChanged');
    alert('type=' + type);
    alert('name=' + name);
    alert('linenum=' + linenum);
}

function myPostSourcing(type, name)
{
    alert('myPostSourcing');
    alert('type=' + type);
    alert('name=' + name);
}
```

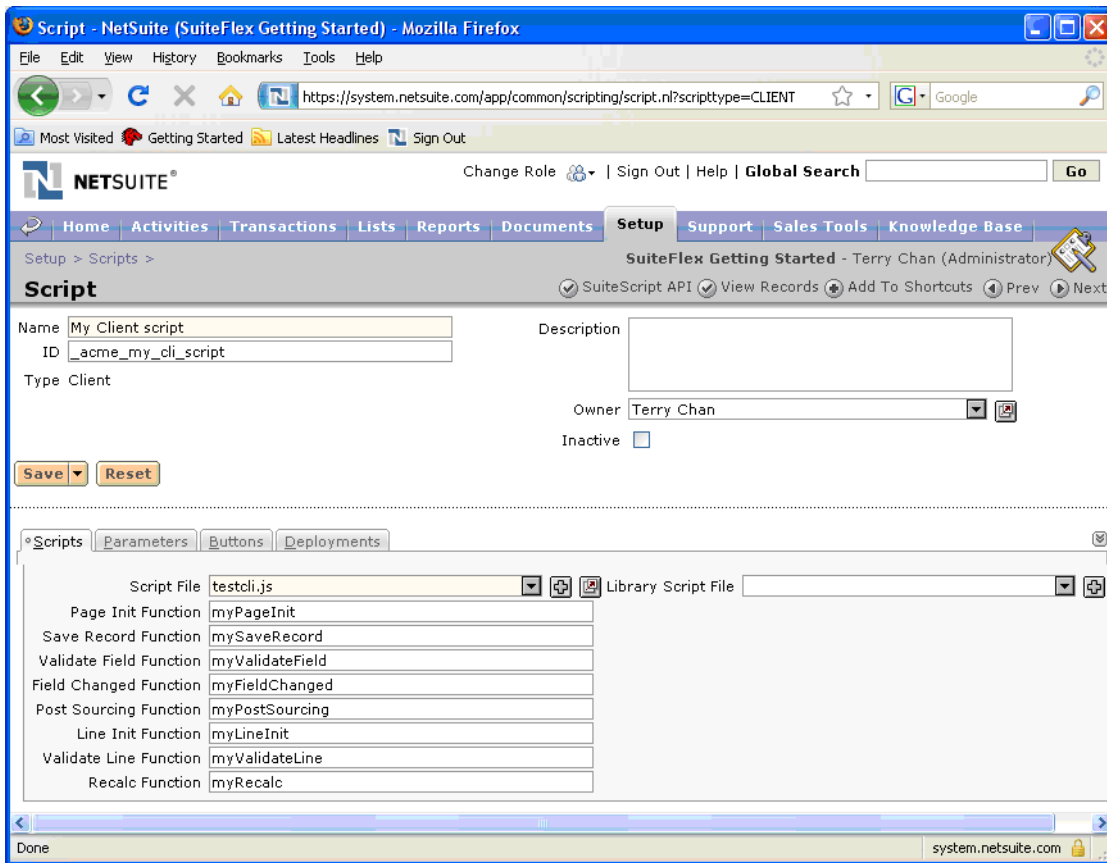
```
}  
  
function myLineInit(type)  
{  
    alert('myLineInit');  
    alert('type=' + type);  
}  
  
function myValidateLine(type)  
{  
    alert('myValidateLine');  
    alert('type=' + type);  
}  
  
function myRecalc(type)  
{  
    alert('myRecalc');  
    alert('type=' + type);  
}
```

This sample displays all the available arguments for every script-triggered event. Notice that some functions return a Boolean while some do not. Also note that some functions have `linenum` as one of the arguments. Sublist functions do not have a `linenum` argument because the event is confined to the specific line that triggered it. Please refer to the documentation for the details on all API arguments and return values.

The function `myValidateField` has an additional `if` block to check whether the event was invoked by a custom field with the id `custentity_my_custom_field`. This ensures the logic is executed only under the correct circumstances.

**Best Practice:** It is important to check the argument values to branch execution logic. This improves performance and avoids logic executed indiscriminately.

To obtain a better understanding on when these client script events are triggered and what the arguments contain, upload the JavaScript file to the File Cabinet's SuiteScript folder and deploy the script by specifying the functions in a script record (shown below).



**Best Practice:** Always specify a unique ID in the ID field when defining a script record. When saved, the system will automatically put in a **customscript** prefix. In the screenshot above, the final unique ID for this client script will be **customscript\_acme\_my\_cli\_script**.

The screen shot above demonstrates the definition of a record-level client script. This script is deployed globally to any records specified in the Deployments tab. A record level client script may also be used by custom NetSuite pages (Suitelets).

Form-level client scripts run against specific forms. They are defined on the form definition's Custom Code tab. Form-level client scripts cannot be used by Suitelets.

Below is a sample client script. It has a validate field function that ensure the value in the field with ID `custrecord_mustbe_uppercase` is always set to uppercase.

```
function validateFieldForceUppercase(name, type)
{
    if(type == 'custrecord_mustbe_uppercase')
    {
        //obtain the upper case value
        var upperCase =
        nlapiGetFieldValue('custrecord_mustbe_uppercase').toUpperCase();
    }
}
```

```
    //make sure it hasn't been set
    if(upperCase !=
nlapiGetFieldValue('custrecord_mustbe_uppercase'))
    {
        nlapiSetFieldValue('custrecord_mustbe_uppercase', upperCase,
false);
    }
    return true;
}
```

The above function is deployed on the validate field event. Since this function is invoked every time there is an attempt to move the focus way from a field, the first `if` block ensures the uppercase logic is executed only for the correct field. Since using the API `nlapiSetFieldValue` would also trigger events, the second `if` block is put in place to ensure the code will not get into an infinite loop. The final `return true` statement ensures the focus can be successfully taken away from the field.

**Tip:** The JavaScript alert statement is an easy way to debug client scripts. Alternatively, browser-specific JavaScript debuggers may also be used.

**Best Practice:** Getting and setting fields and sublists should be accomplished by using NetSuite's various `nlapi` APIs instead of directly accessing the DOM. The `nlapi` APIs provide a consistent coding interface across NetSuite versions, hence the scripts that use them are fully supported.

## User Event Scripts

User event scripts are SuiteScripts executed on the NetSuite server. They are triggered when users perform certain actions on records, such as create, load, update, copy, delete, or submit. User event scripts are used to implement business process automation, customize business rules, and to synchronize real-time data. They are also used as building blocks for highly customized workflows within NetSuite.

Most standard NetSuite records and custom record types are supported by user event scripts. The events exposed are:

- Before Load – event occurs when a read operation on a record takes place
- Before Submit – event occurs when a record is submitted, but before the changes are committed to the database
- After Submit – event occurs after the changes are committed to the database

### The Type Argument

All of the events have the common `type` argument which indicates the type of operation that invoked the event. This argument allows the script code to branch out to different logic depending on the operation type. For example, a script with “deltree” logic that deletes a record and all of its child records should only be invoked when `type` equals to “delete”. It is very important that user event scripts check the value of the `type` argument to avoid indiscriminate execution.

The following sample demonstrates how to check the value of the `type` for each event. Please refer to the SuiteScript documentation for the valid `type` argument values for each event.

```
function myBeforeLoadUE(type)
{
  if(type == 'create')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is create');
  }

  if(type == 'view')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is view');
  }

  if(type == 'edit')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
  }
}

function myBeforeSubmitUE(type)
{
  if(type == 'create')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is create');
  }

  if(type == 'delete')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is delete');
  }

  if(type == 'edit')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
  }

  if(type == 'cancel')
  {
    nlapiLogExecution('DEBUG', 'type argument', 'type is cancel');
  }
}

function myAfterSubmitUE(type)
{
```

```
if(type == 'create')
{
  nlapiLogExecution('DEBUG', 'type argument', 'type is create');
}

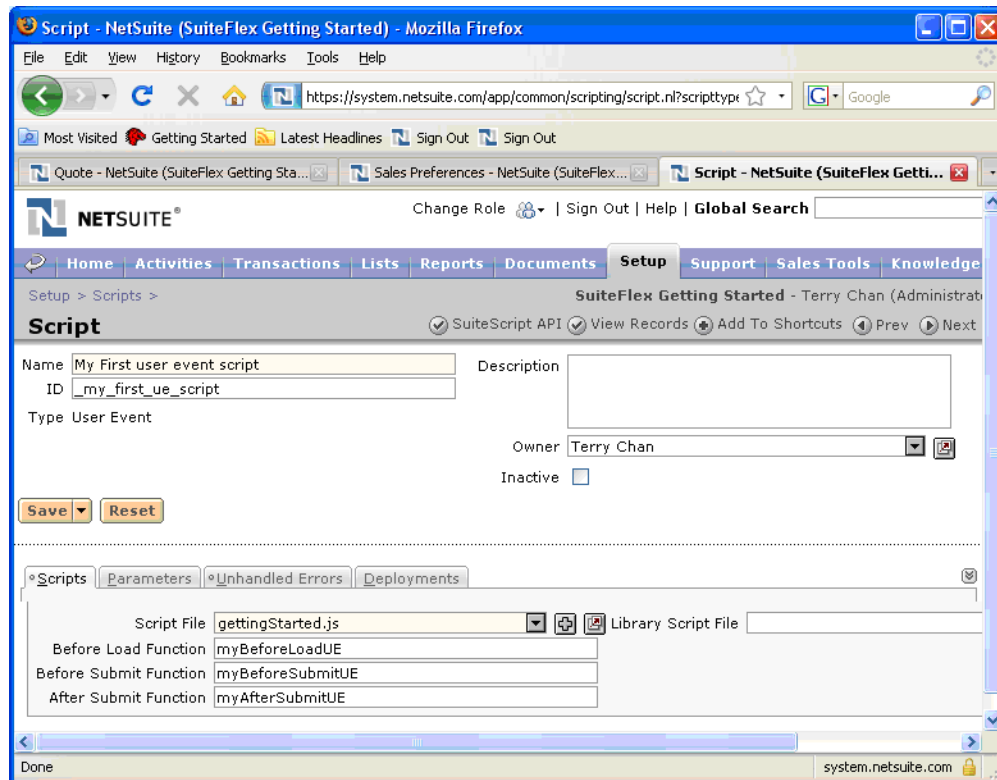
if(type == 'delete')
{
  nlapiLogExecution('DEBUG', 'type argument', 'type is delete');
}

if(type == 'edit')
{
  nlapiLogExecution('DEBUG', 'type argument', 'type is edit');
}

if(type == 'approve')
{
  nlapiLogExecution('DEBUG', 'type argument', 'type is approve');
}
}
```

**Tip:** Logging done with `nlapiLogExecution` may be classified into 4 types: DEBUG, AUDIT, ERROR, and EMERGENCY. The source code should correctly set the logging type. Log type filtering may be set during runtime to give concise and useful logged information.

After uploading the source code file to the SuiteScript folder in the File Cabinet, a user event script record is defined by going to Set Up → Customization → Scripts → New → User Event. The following shows the script record definition page.



## Business Process Automation with User Event Scripts

By exposing the key events of the records, SuiteScript APIs provide developers the ability to automate business processes.

A simple CRM use case that could be addressed with user event scripts is creating a follow-up phone call record for every newly created customer record. The solution is to deploy a script on the customer record's *after submit* event that will create the phone call record. See the following sample code:

```
function followUpCall_CustomerAfterSubmit(type)
{
    //Only execute the logic if a new customer is created
    if(type == 'create')
    {
        //Obtain a handle to the newly created customer record
        var custRec = nlapiGetNewRecord();

        if(custRec.getFieldValue('salesrep') != null)
        {
            //Create a new blank instance of a PhoneCall
            var call = nlapiCreateRecord("phonecall");
        }
    }
}
```

```
//Setting the title field on the PhoneCall record
call.setFieldValue('title', 'Make follow-up call to new
customer');

//Setting the assigned field to the sales rep of the
//new customer
call.setFieldValue('assigned',
custRec.getFieldValue('salesrep'));

//Use the library function to obtain a date object
//that represents tomorrow
var today = new Date();
var tomorrow = nlapiAddDays(today, 1);
call.setFieldValue('statedate', nlapiDateToString(tomorrow));

//Setting the phone field to the phone of the
//new customer
call.setFieldValue('phone', custRec.getFieldValue('phone'));

try
{
    //committing the phone call record to the database
    var callId = nlapiSubmitRecord(call, true);
    nlapiLogExecution('DEBUG', 'call record created successfully',
'ID = ' + callId);
}
catch(e)
{
    nlapiLogExecution('ERROR', e.getCode(), e.getDetails());
}
}
```

**Best Practice:** APIs such as `nlapiSubmitRecord` that access the database should be wrapped in try-catch blocks.

In the above use case, *after submit* is a better event to handle the logic than *before submit*. In the *before submit* event, the customer data has not yet been committed to the database. Hence, putting the phone call logic in the *after submit* event guarantees there will not be an orphan phone call record.

**Tip:** During design time, developers should carefully consider in which event to implement their server logic.

The phone call use case may be further enhanced by redirecting the user to the Phone Call page once it is created. This is accomplished by putting in redirect logic after the phone call record is submitted.

```
try
{
    //committing the phone call record to the database
    var callId = nlapiSubmitRecord(call, true);
    nlapiLogExecution('DEBUG', 'call record created successfully', 'ID
= ' + callId);

    //Redirect the user to the newly created phone call
    nlapiSetRedirectURL('RECORD', 'phonecall', callId, false, null);
}
catch(e)
{
    nlapiLogExecution('ERROR', e.getCode(), e.getDetails());
}
```

User event scripts are not only triggered as a result of user actions carried out through the browser, they are also triggered by other means as well (for example, CSV, Web Services, offline client). Remember in this case, could be any kind of client, not just a web browser. Examples: Using CSV to import records triggers *before submit* and *after submit* events; using the SuiteTalk “GET” operation to retrieve an existing record would trigger its *before load* event. Sometimes these events invoke scripts not designed to be executed in that manner and bring undesirable results. To prevent a script from getting executed by the wrong execution context, use the `nlobjContext` object as a filter.

For example, in order to ensure a *before load* user event script is executed only when a record is created using the browser interface, the script must check both the `type` argument and the execution context as (as shown below):

```
function myBeforeLoadUE(type)
{
    //obtain the context object
    var context = nlapiGetContext();
    if(type == 'create' && context.getExecutionContext ==
'userinterface')
    {
        .
        .
        .
    }
}
```

Note that the API `nlapiGetContext` is not exclusive to user event scripts. It can also be used by client scripts and Suitelets. Please refer to the SuiteScript documentation for details on `nlapiGetContext` and the `nlobjContext` object.

**Tip:** The `nlobjContext` object provides metadata of the script’s context. Use this information to help implement fine-grained control logic in SuiteScript.

## Using User Event Scripts to Enhance Forms

Another common use of User Event scripts is to dynamically customize, or enhance, entry forms and transactions forms. This approach gives NetSuite forms the ability to customize themselves in runtime – something that cannot be done with pre-configured roles-based forms.

In NetSuite, entry forms and transaction forms are customized by the administrator(s). The placement of UI elements (fields, tabs, sublists) on a form can be arranged, or be made inline or hidden depending on the business needs of the end users. Multiple forms can be create for a record type and assigned to specific roles. Typically these kinds of customization are done during design time. Custom forms are confined to specific roles and do not allow for a lot of runtime customization. A User Event script on a record's *before load* event could provide some flexibility to runtime customization.

The key to using User Event scripts to customize a form during runtime is a second argument named `form` in the *before load* event. This optional argument is the reference to the entry/transaction form. Developers may use this to dynamically change existing UI elements, or add new ones. The UI elements are added using [SuiteScript UI Objects](#).

A use case for this scripting capability is in order to improve month-end sales, a company introduces an end-of-month promotion that is only active for the last five days of the month. All sales order forms would have a field labelled “Eligible EOM Promotion” on the last five days of the month. The following is the User Event script code that is meant to be deployed on the *before load* event of the sales order record.

```
/*  
 * This function is a module to implement at end of  
 * month (last 5 days of month) promotion for sales  
 * orders. It is meant to be deployed on the before  
 * load event of the sales order record.  
 */  
function customizeUI_SalesOrderBeforeLoad(type, form)  
{  
    var currentContext = nlapiGetContext();  
  
    //Execute the logic only when creating a sales order with the  
    browser UI  
    if(type == 'create' && currentContext.getExecutionContext() ==  
    'userinterface')  
    {  
        var fieldId = 'custpage_eom_promotion';  
        var fieldLabel = 'Eligible EOM promotion';  
        var today = new Date();  
        var month = today.getMonth();  
        var date = today.getDate();  
        nlapiLogExecution('DEBUG', 'month date', month + ' ' + date);  
    }  
}
```

```
//February
if(month==1)
{
    if( date==24 | date==25 | date==26 | date==27 | date==28 |
date==29)
        form.addField(fieldId, 'checkbox', fieldLabel);
    }
    //31-day months
    else if(month==0 | month==2 | month ==4 | month==6 | month==7 |
month==9 | month==11)
    {
        if( date==27 | date==28 | date==29 | date==30 | date==31)
        form.addField(fieldId, 'checkbox', fieldLabel);
    }
    //30-day months
    else
    {
        if( date==26 | date==27 | date==28 | date==29 | date==30)
        form.addField(fieldId, 'checkbox', fieldLabel);
    }
}
}
```

When the script is deployed, all sales order forms will have the Eligible EOM Promotion checkbox only during the last five days of the month as shown below.

The screenshot displays the NetSuite 'Sales Order' form. The browser title is 'Sales Order - NetSuite (SuiteFlex Getting Started) - Mozilla Firefox'. The URL is 'https://system.netsuite.com/app/accounting/transactions/salesord.nl?whence='. The form is for a 'Standard Sales...er - Cash Sale' with a date of '10/10/2008' and order number '182'. The status is 'Pending Fulfillment'. A red box highlights the 'Eligible EOM promotion' checkbox, which is unchecked. The form includes buttons for 'Save', 'Auto Fill', 'Reset', 'Memorize', 'Clear Splits', 'Customize', and 'New Body Field'. The bottom of the form shows a table for items with columns for Item, Quantity, Units, Description, Price Level, Rate, Amount, Commit, Tax, Options, Create PO, Rev. Rec. Schedule, Rev. Rec. Start Date, Rev. Rec. End Date, and Closed.

Note that since these UI elements are created dynamically, they are superficial and do not have supporting back end data models. There is a disconnect between the UI and backend data, hence the script-created fields' values will not be saved.

UI elements created with User Event scripts and SuiteScript UI Objects, such as the Eligible EOM promotion example, are scriptable by client script APIs. A remedy to the disconnect problem is linking the script-created field to a real field (with back end data support) via a client script. The value of the real field, which might be made hidden or inline on the form definition, is driven by the value entered in the script-created field. Hence the real fields are populated and the data is saved.

## Scheduled Scripts

Scheduled Scripts are SuiteScripts that run on the NetSuite server. Compared to User Event scripts, Scheduled scripts are allotted longer execution time, and are given a higher limit of usage governance. Hence they are ideal for long running tasks and batch jobs. Please refer to the SuiteScript documentation for details on SuiteScript governance.

Scheduled scripts may be invoked in the following contexts:

- Invoked at regularly scheduled intervals

- Invoked by the user interface – this is useful for adhoc operations or for testing and debugging scripts
- Invoked on-demand by other server scripts – this method is used when the originating scripts' usage governance does not allow them to finish the tasks, hence they would delegate them to a scheduled script

The `type` argument for the scheduled script function provides the context in which the script is invoked.

A use case for a scheduled script is to send thank you notes to valued, repeated customers. A scheduled script may be executed to perform a daily searches for sales orders that are placed today and within the last 30 days from the same customer. After retrieving the results, the scheduled script then sends these customers an email on behalf of the sales rep to thank them for their repeated business. The following is the source code for such a script.

```
/*  
 * This scheduled script looks for customers that  
 * have placed multiple orders in the last 30 days.  
 * It will send a thank you email to these customers  
 * on behalf of their sales reps.  
 */  
function findHotCustomerScheduled(type)  
{  
    //Invoke only when it is scheduled  
    if(type == 'scheduled')  
    {  
        //Obtaining the context object and logging the remaining usage  
        available  
        var context = nlapiGetContext();  
        nlapiLogExecution('DEBUG', 'Remaining usage at script beginning',  
context.getRemainingUsage());  
  
        //Setting up filters to search for sales orders  
        //with trandate of today.  
        var todaySOFilters = new Array();  
        todaySOFilters[0] = new nlobjSearchFilter('trandate', null, 'on',  
'today');  
  
        //Setting up the columns. Note the join entity.salesrep column.  
        var todaySOColumns = new Array();  
        todaySOColumns[0] = new nlobjSearchColumn('tranid', null, null);  
        todaySOColumns[1] = new nlobjSearchColumn('entity', null, null);  
        todaySOColumns[2] = new nlobjSearchColumn('salesrep', 'entity',  
null);  
  
        //Search for the sales orders with trandate of today
```

```
var todaySO = nlapiSearchRecord('salesorder', null,
todaySOFilters, todaySOColumns);
nlapiLogExecution('DEBUG', 'Remaining usage after searching sales
orders from today', context.getRemainingUsage());

//Looping through each result found
for(var i = 0; todaySO != null && i < todaySO.length; i++)
{
    //obtain a result
    var so = todaySO[i];

    //Setting up the filters for another sales order search
    //that are of the same customer and have trandate within
    //the last 30 days
    var oldSOFilters = new Array();
    var thirtyDaysAgo = nlapiAddDays(new Date(), -30);
    oldSOFilters[0] = new nlobjSearchFilter('trandate', null,
'onorafter', thirtyDaysAgo);
    oldSOFilters[1] = new nlobjSearchFilter('entity', null, 'is',
so.getValue('entity'));
    oldSOFilters[2] = new nlobjSearchFilter('tranid', null, 'isnot',
so.getValue('tranid'));

    //Search for for the repeated sales in the last 30 days
    var oldSO = nlapiSearchRecord('salesorder', null, oldSOFilters,
null);
    nlapiLogExecution('DEBUG', 'Remaining usage after in for loop,
i=' + i, context.getRemainingUsage());

    //If results are found, send a thank you email
    if(oldSO != null)
    {
        //Setting up the subject and body of the email
        var subject = 'Thank you!';
        var body = 'Dear ' + so.getText('entity') + ', thank you for
your repeated business in the last 30 days.';

        //Sending the thank you email to the customer on behalf of the
sales rep
        //Note the code to obtain the join column entity.salesrep
        nlapiSendEmail(so.getValue('salesrep', 'entity'),
so.getValue('entity'), subject, body);
        nlapiLogExecution('DEBUG', 'Remaining usage after sending
thank you email', context.getRemainingUsage());
    }
}
```

```

    }
}

```

Notice there are a number of `nlobjContext.getRemainingUsage()` API calls in the sample code. This API provides the remaining SuiteScript usage to help scripts monitor how close they are to running into SuiteScript usage governance.

SuiteScript governance is a mechanism to optimize server and database performance. Please refer to the documentation for details on SuiteScript metering and governance.

**Tip:** Since scheduled scripts also trigger user event scripts, developers may need to revisit the design of their user event scripts to ensure they will be invoked by the correct execution contexts

## SuiteScript UI Objects

SuiteScript UI objects is a UI toolkit for server scripts such as Suitelets and user event scripts. SuiteScript UI objects are UI components **generated on the server** as HTML. They are **displayed in the browser side** and are accessible through client scripts.

SuiteScript UI objects are used primarily by Suitelets and user event scripts. [Suitelets](#) allow developers to programmatically build custom NetSuite pages; user event scripts expose entry forms and transaction forms for customization. This is made possible by using SuiteScript UI objects to create and manipulate the native NetSuite UI components (forms, fields, sublists etc), thus giving users the familiar NetSuite look-and-feel for scripted UI customization.

While SuiteScript UI objects give developers a lot of control over the characteristics, placement, and behaviours of UI elements - while retaining the NetSuite look-and-feel, resources need to be spent to develop and maintain them. During design time, application architects should carefully weigh the trade off between customizing the NetSuite UI with SuiteBuilder, versus programmatically customizing it with SuiteScript UI objects.

**Tip:** Sometimes the best solution for a customized workflow with multiple pages is a hybrid UI design, which encompasses both customized entry forms as well as Suitelets built with UI objects

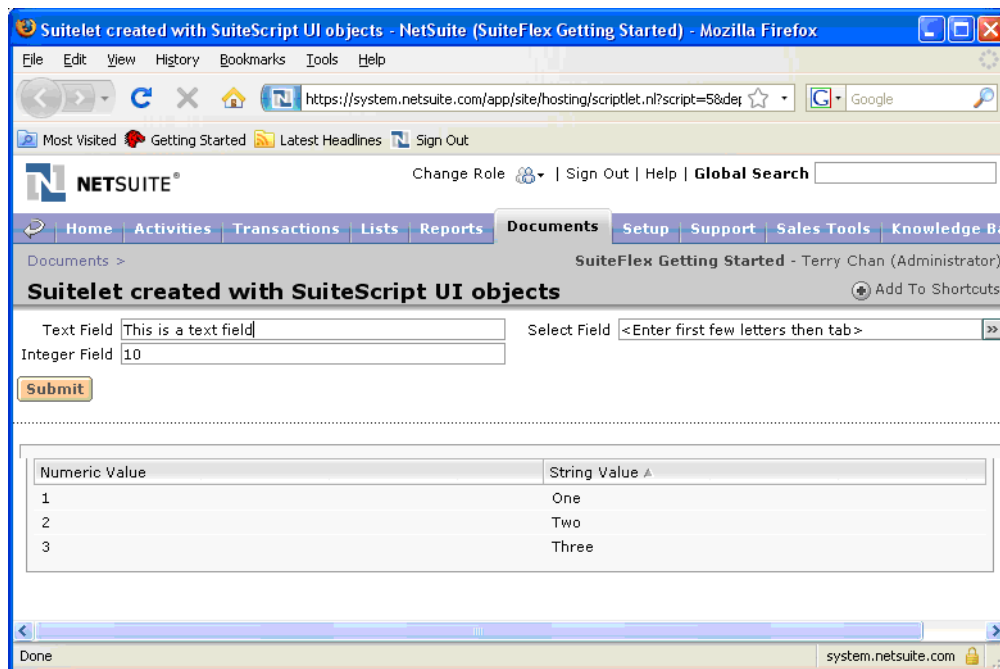
### Native Look-and-Feel

Depending on the design and purpose of the custom UI, developers may use either the `nlobjForm` object or `nlobjList` object as the basis. These objects encapsulate a scriptable NetSuite form and NetSuite list, respectively. Developers may add a wide selection of scriptable UI elements to these objects to adopt the NetSuite look-and-feel.

On entry forms and transaction forms, the `nlobjForm` object is accessed in user events scripts on which new fields are added (see [Using User Event Scripts to Enhance Forms](#)) on the server before the pages are sent to the browser.

In Suitelets, a blank `nlobjForm` object is created with `nlapiCreateForm`. On the server, the Suitelet code adds fields, tabs, buttons and sub-lists to the `nlobjForm` object. The server defines the client script (if applicable) and sends the page to the browser. When the page is submitted, the values in these UI Objects become part of the request and available to aid logic branching in the code.

Below is an example of a custom NetSuite page built with a Suitelet using SuiteScript UI objects.

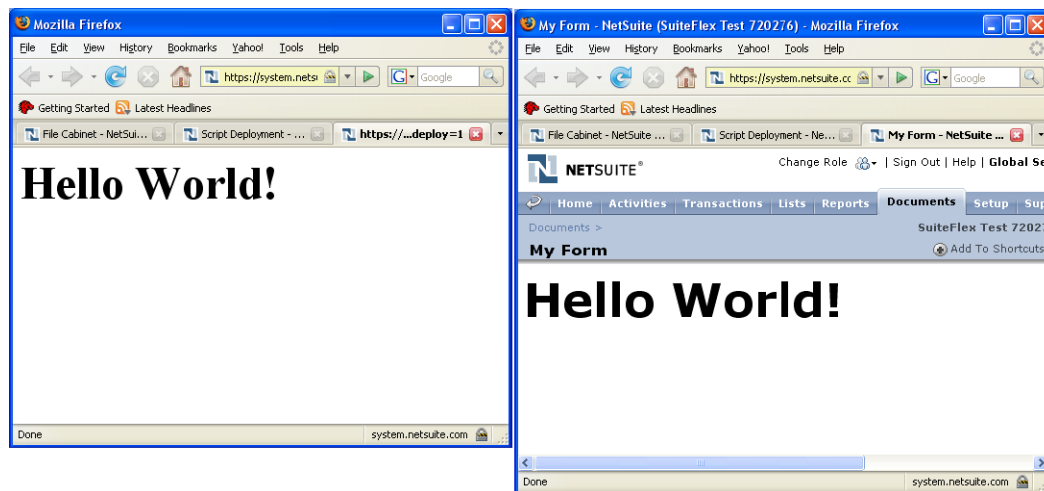


## InlineHTML UI Objects

The SuiteScript UI objects make most of the NetSuite UI elements scriptable. However, they still may not lend themselves well to certain use cases. In these circumstances, developers can develop custom UI elements by providing HTML that SuiteScript can render on a NetSuite page. These UI elements are known as "InlineHTML".

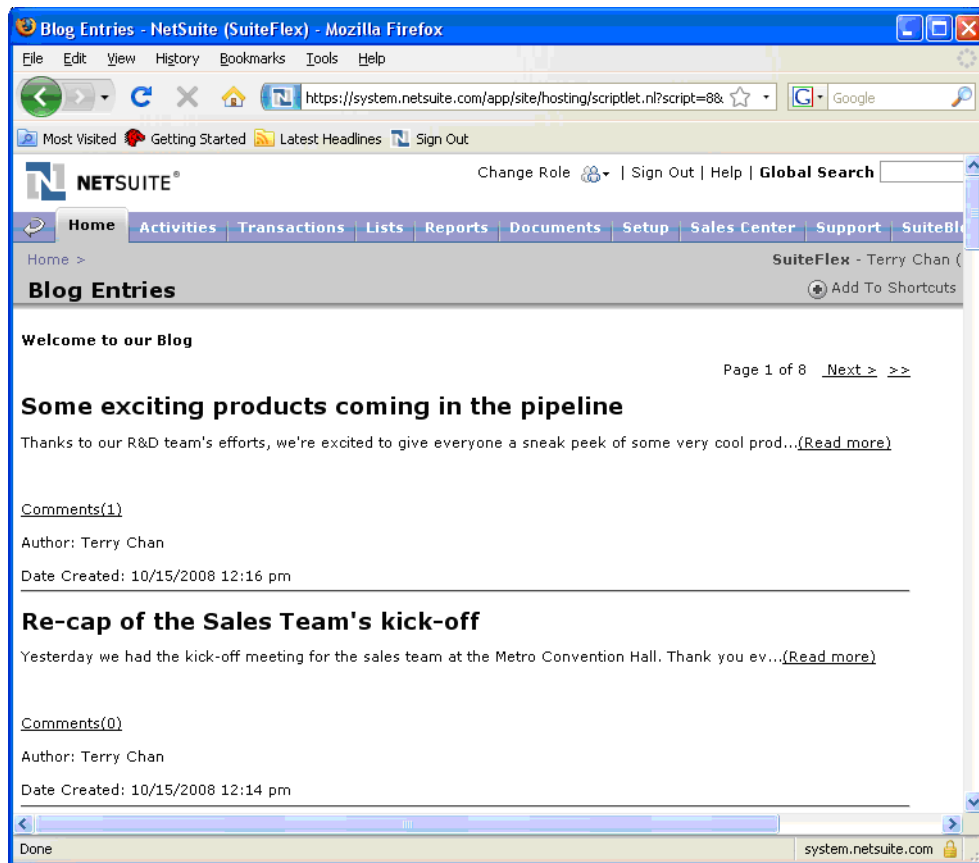
InlineHTML can be implemented in two ways:

- Pure custom HTML with no SuiteScript UI Objects
- Hybrid of custom HTML and SuiteScript UI Objects



Approach #1, as shown on the left side, requires developers to provide all the HTML code they want to appear on the page, as if performing web designing on a blank canvas. Approach #2, shown on the right side allows custom HTML to be embedded in a NetSuite page.

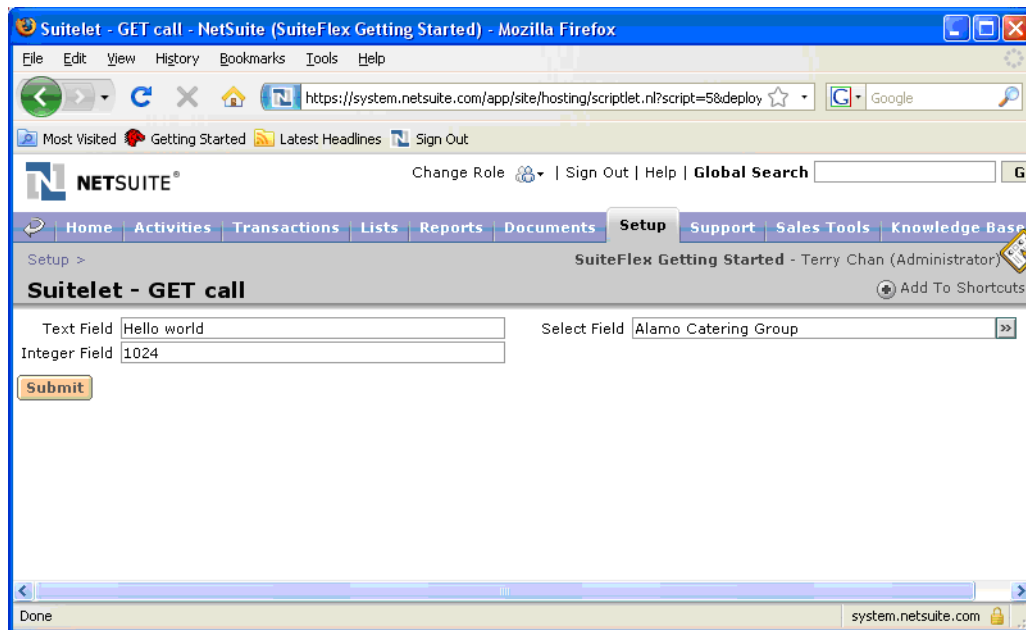
An example of an application implemented with a hybrid of inlineHTML and UI Objects is a blog hosted within NetSuite. A blog page displays blog entries in descending chronological order with “Read More” hyperlinks. Due to the potentially large number of entries, pagination is required. Readers should also be able to read and leave comments. These requirements cannot be easily satisfied by standard NetSuite UI elements in a reader-friendly manner. However, rendering the blog entries’ data (stored as custom records) in HTML and displaying it in SuiteScript UI objects as inlineHTML would satisfy this use case.



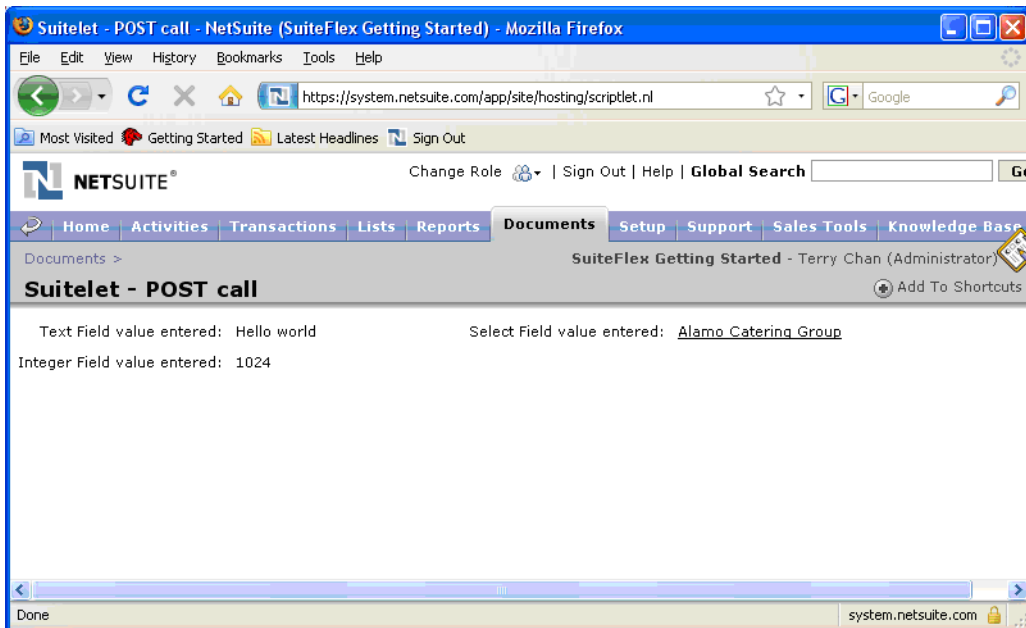
## Suitelets

Suitelets are extensions of SuiteScript that give developers the ability to write custom NetSuite pages and backend logic. Suitelets are server side scripts that operate in a request-response model. They are invoked by HTTP GET or POST requests to system generated URLs.

Shown below are screenshots of a simple Suitelet with a few fields. The Suitelet is invoked by making a GET request from the browser. Notice that this Suitelet is built with SuiteScript UI Objects.



When the Submit button is clicked, the same Suitelet is invoked again with a HTTP POST event. The values entered in the previous screen are displayed in inline (read-only) mode.



Below is the source code for this Suitelet. It is executed on the server, which generates HTML and sends it to the browser.

```
function gettingStartedSuitelet(request, response)
{
    if(request.getMethod() == 'GET')
    {
```

```

    //Creating the form and adding fields to it
    var form = nlapiCreateForm("Suitelet - GET call");
    form.addField('custpage_field1', 'text', 'Text
Field').setDefaultValue('This is a text field');
    form.addField('custpage_field2', 'integer', 'Integer
Field').setDefaultValue(10);
    form.addField('custpage_field3', 'select', 'Select Field',
'customer');

    form.addSubmitButton('Submit');

    response.writePage(form);
    }
    //POST call
else
{
    var form = nlapiCreateForm("Suitelet - POST call");

    //creating the fields on the form and populating
    //them with values from the previous screen
    var resultField1 = form.addField('custpage_res1', 'text', 'Text
Field value entered: ');
    resultField1.setDefaultValue(request
.getParameter('custpage_field1'));
    resultField1.setDisplayType('inline');

    var resultField2 = form.addField('custpage_res2', 'integer',
'Integer Field value entered: ');
    resultField2.setDefaultValue(request
.getParameter('custpage_field2'));
    resultField2.setDisplayType('inline');

    var resultField3 = form.addField('custpage_res3', 'select',
'Select Field value entered: ', 'customer');
    resultField3.setDefaultValue(request
.getParameter('custpage_field3'));
    resultField3.setDisplayType('inline');

    response.writePage(form);
    }
}

```

The entry point of the function has two mandatory arguments: `request` and `response`. These arguments are instances of `nlobjRequest` and `nlobjResponse`, respectively.

Typically, invoking a Suitelet via a browser would make a HTTP GET call. The type of HTTP call is determined by the `nlobjRequest.getMethod()` API. The code creates an

`nlobjForm` object and populates it with UI objects. The populated form is sent to the response object via the `nlobjResponse.writePage()` API.

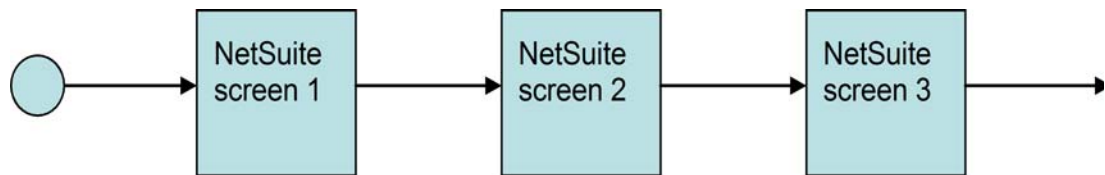
When the user clicks the Submit button, an HTTP POST call is made. The code's `else` block obtain the values entered in the first page from the `request` object and populates them into another `nlobjForm` object and sends it to `response.writePage()`.

For more sample Suitelets, please refer to the *SuiteScript Developer Guide* and *SuiteScript Reference Guide* in the NetSuite Help Center.

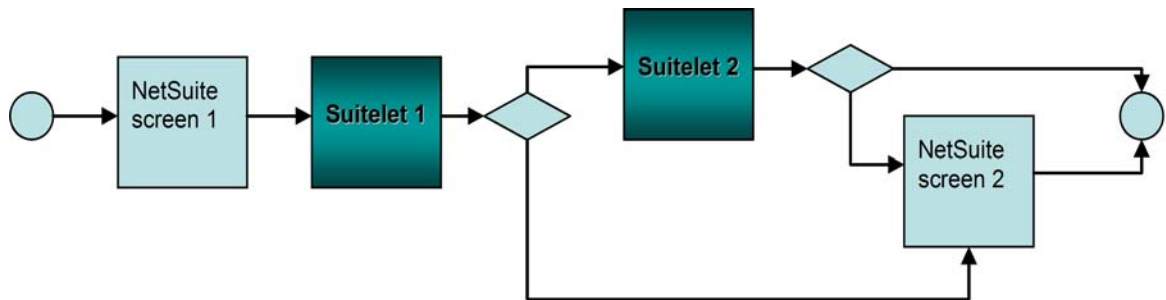
## Building Custom Workflow with Suitelets

Prior to NetSuite version 11, workflows in NetSuite were predefined sequences of standard NetSuite screens. For example: enter quote → create sales order → fulfill order → bill customer.

SuiteBuilder and client scripts were helpful for customizing individual pages, but there was little room for customizing the workflows. Since there was no way to build standalone NetSuite screens, ISV customization work was confined to client SuiteScript and custom record types with little to no provision to build new functionalities (see diagram below).



With the advent of user event scripts, Suitelets and UI Objects, application designers may create custom workflows by chaining together standard and/or custom NetSuite screens. These workflows may be complemented by custom backend logic. Since standard and custom records are scriptable, ISVs may develop new functionalities atop the NetSuite platform. The contexts within a workflow can be maintained between pages to aid logic branching (diagram below).



## Backend Suitelets

Suitelets give developers the ability to build custom NetSuite pages. However, developers should not overlook using backend Suitelets – Suitelets that do not generate any UI elements – to create powerful applications.

A backend Suitelet is a Suitelet that only performs backend logic. Just like a Suitelet that builds NetSuite pages, a backend Suitelet is invoked by making HTTP GET or POST calls to a URL.

The following are good uses of backend Suitelets:

- Providing a service for backend logic to other SuiteScripts, or to other external hosts outside of NetSuite
- Offloading server logic from client scripts to a backend Suitelet shipped without source code to protect sensitive intellectual property

A use case of a backend Suitelet is a service that provides customer information based on a phone number. The following is the code for a Suitelet that returns customer entity IDs (for records with matching phone numbers) separated by the | character.

```

/*****
 * This function searches for customer records
 * that match a supplied parameter custparam_phone
 * and return the results in a string separated
 * by the | character.
 */
function lookupPhoneBackendSuitelet(request, response)
{
    if(request.getMethod() == 'GET')
    {
        //null check on the required parameter
        if(request.getParameter('custparam_phone') != null)
        {
            //Setting up the filters and columns
            var filters = new Array();
            var columns = new Array();

```

```
//Use the supplied custparam_phone value as filter
filters[0] = new nlobjSearchFilter('phone', null, 'is',
request.getParameter('custparam_phone'));
columns[0] = new nlobjSearchColumn('entityid', null, null);

//Search for customer records that match the filters
var results = nlapiSearchRecord('customer', null, filters,
columns);

if(results != null)
{
    var resultString = '';
    //Loop through the results
    for(var i = 0; i < results.length; i++)
    {
        //constructing the result string
        var result = results[i];
        resultString = resultString + result.getValue('entityid');

        //adding the | separator
        if (i != parseInt(results.length - 1))
        {
            resultString = resultString + '|';
        }
        nlapiLogExecution('DEBUG', 'resultString', resultString);
    }
    response.write(resultString);
}
else
{
    response.write('none found');
}
}
}
```

Notice that this Suitelet does not use any UI Object APIs. Communication with the Suitelet is done strictly with the `request` and `response` objects. NetSuite generates a URL to invoke this Suitelet. To correctly invoke it, the `custparam_phone` value (bold) needs to be appended at the end of the invoking URL:

[https://system.netsuite.com/app/site/hosting/scriptlet.nl?script=6&deploy=1&custparam\\_phone=\(123\)-456-7890](https://system.netsuite.com/app/site/hosting/scriptlet.nl?script=6&deploy=1&custparam_phone=(123)-456-7890)

The code that calls this backend Suitelet needs to do the following:

1. Use `nlapiResolveURL` to dynamically obtain the invoking URL
2. Supply required parameters
3. Process the returned results

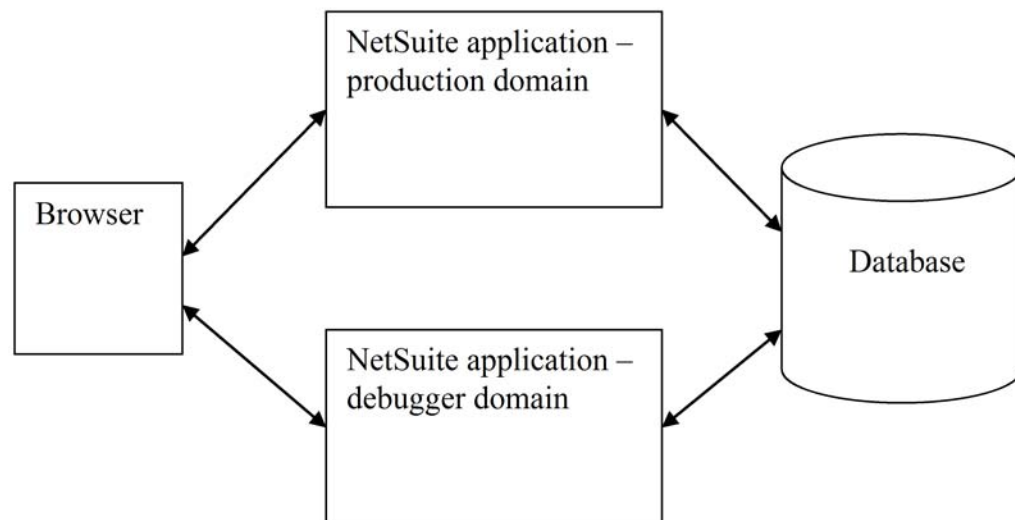
**Note:** Backend Suitelets should not be used to get around SuiteScript usage governance. Suitelets designed with this intention are considered abusive by NetSuite.

## Portlet Scripts

Portlet scripts are server side SuiteScripts that can generate content to be displayed in dashboard portlets. Developers may put forms, lists, rows, or custom HTML in portlet scripts. For more information and sample code of Portlet scripts, refer to the *SuiteScript Developer Guide* and *SuiteScript Reference Guide* in the NetSuite Help Center.

## SuiteScript D-bug

SuiteScript D-bug is an interactive debugger for server side SuiteScript of all types (User Event, Suitelet, Scheduled, Portlet). It is functionally similar to other interactive debuggers commonly found in popular IDEs; hence developers will find this tool familiar and easy to use. The biggest difference between SuiteScript D-bug and a traditional interactive debugger is that SuiteScript D-bug is an online tool.

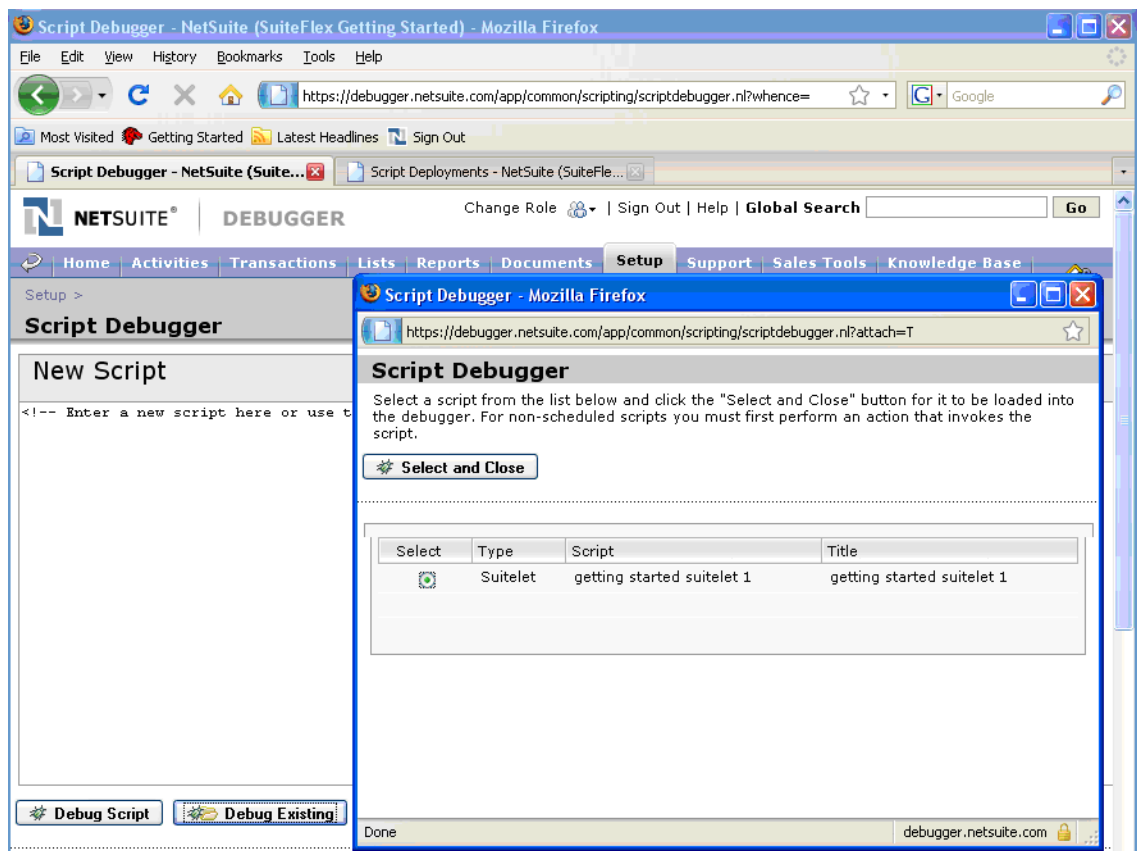


SuiteScript D-bug is a 3-tier web application not unlike NetSuite itself. The tiers are browser, application, and database.

The NetSuite application in the debugger domain is functionally identical to the one in the production domain with the exception of added debugging capability. Note that both domains share a common database in the above architecture. This means **data changes performed using the debugger domain impact the production domain.**

To use SuiteScript D-bug, a developer should log into the debugger domain at <https://debugger.netsuite.com>. Edit the deployment record of the script to be debugged and ensure its status is set to “Testing” and the owner is set to the currently logged in user.

To start the Debugger, go to Set Up → Customization → Script Debugger. Click the Debug Existing button to select the script to debug as shown below. Note the “DEBUGGER” label on the top of the page to act as a visual cue to users they’re logged in to the debugger domain.



To start debugging, invoke the script. The Debugger interrupts the execution and displays the source code.

**Script Debugger**

Debugging Suitelet: getting started suitelet 1

```
24     response.writePage(form);
25   }
26   //POST call
27   else
28   {
29     var form = nlapiCreateForm("Suitelet - POST call");
30
31     //creating the fields on the form and populating
32     //them with values from the previous screen
33     var resultField1 = form.addField('custpage_res1', 'text', 'Text Field value entered: ');
34     resultField1.setDefaultValues(request.getParameter('custpage_field1'));
35     resultField1.setDisplayType('inline');
36
37     var resultField2 = form.addField('custpage_res2', 'integer', 'Integer Field value entered: ');
38     resultField2.setDefaultValues(request.getParameter('custpage_field2'));
39     resultField2.setDisplayType('inline');
40
41     var resultField3 = form.addField('custpage_res3', 'select', 'Select Field value entered: ', 'customer');
42     resultField3.setDefaultValues(request.getParameter('custpage_field3'));
43     resultField3.setDisplayType('inline');
44
45     response.writePage(form);
```

Debug Script   Debug Existing  

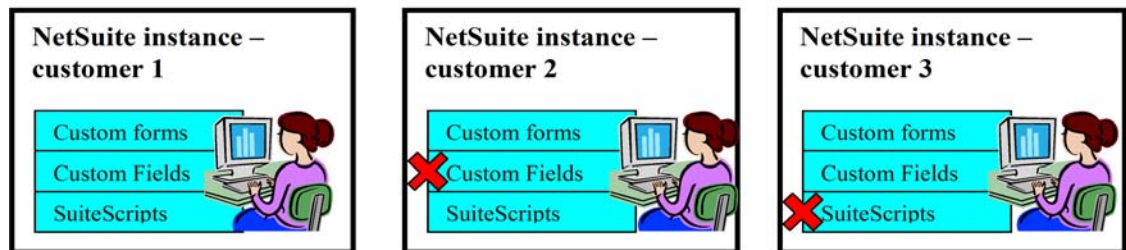
Developers can control the step-by-step execution, set break points, and inspect variables within the debugger. Once a script has finished execution or has been terminated, the developer needs to click the Re-run Script button in the debugger to debug it again. Note that the GET and POST events of a Suitelet are considered distinction execution sessions, and may need to be debugged twice.

# SuiteBundler

The NetSuite platform provides a very rich accounting, ERP, CRM, ecommerce , and HR feature set in a single application suite. NetSuite is optimized for a number of verticals, and the ability to customize every facet of the product makes it extremely versatile in addressing unique business needs of different verticals.

The tools and APIs in NS BOS (Suitebuilder, SuiteScript, SuiteTalk) provide everything ISVs require to build powerful vertical or horizontal application on an application suite delivered in the SaaS model. The NetSuite platform features and SaaS delivery model free ISVs from developing the underpinnings and system infrastructure; hence allowing them to focus on delivering value-added features atop the NetSuite platform for vertical customers.

These vertical solutions may consist of a number of customization objects – custom fields, custom record types, Center Tabs, SuiteScripts etc. Deploying this collection of customization on a NetSuite instance requires every component to be recreated manually on the customer instance.

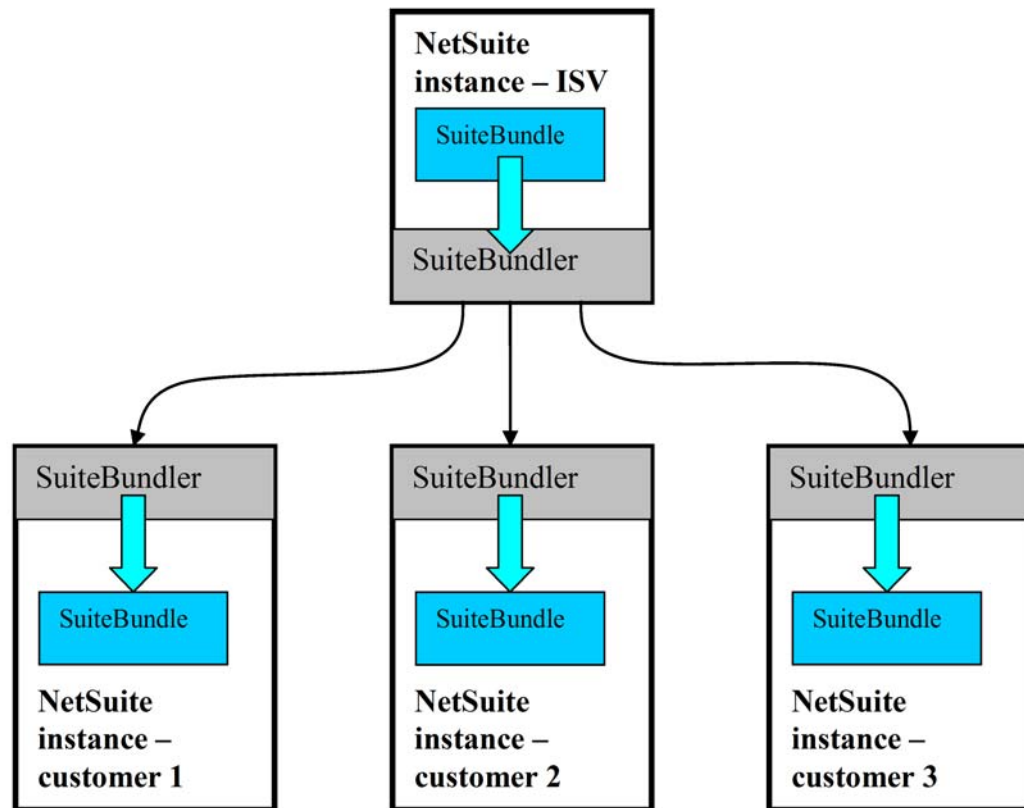


*Manually deploying a vertical solution by recreating components is time consuming and error prone.*

For example, a vertical solution with 50 custom fields, 10 custom record types and 15 SuiteScripts may take an ISV 5 days to develop. To deploy it on a customer's NetSuite account, the implementation team needs to use the NetSuite UI to recreate the identical fields, custom record types and SuiteScript records. The time to complete this implementation would be slightly less than 5 days - the only time saved is the development of the SuiteScript source code which is an investment that needs to be made just once. This deployment model is extremely time-consuming and error prone thus significantly increases the resources required for the ISV, lengthens implementation time, delay go-live date, and negatively impacts customer satisfaction. Upgrading the solution involves manually making changes to all the impacted components. Again, this is another time-consuming and inefficient proposition.

The introduction of SuiteBundler makes the deployment of vertical solutions much faster and easier. ISVs may put all the components of the solution into a **SuiteBundle**, then deploy it into customer accounts. The customer account administrator installs the SuiteBundle, which

brings in all the components into the account. All of this can take place within minutes and does not require any direct assistance from the NetSuite operations team.



Using the SuiteBundler to deliver the previous sample solution takes minutes instead of days. In addition to the time saved, the quality of the deployment is dramatically improved as the tasks of recreating customization objects are delegated to the SuiteBundler.

## SuiteBundler Basics

The unprecedented level of ease in deploying customizations introduced by the SuiteBundler benefits ISVs and vertical solution providers greatly. Before an ISV starts using the SuiteBundler and designing support and maintenance processes around it, it should become familiar with basic SuiteBundler basic concepts and terminology.

- **SuiteBundle** – a SuiteBundle is a collection of customization objects on a NetSuite account. Only an administrator may create SuiteBundles
- **Bundle Author** – a bundle author is an administrator who has created a SuiteBundle
- **Source Account** – from a SuiteBundle’s perspective, a source account is the NetSuite account on which it was created. Each SuiteBundle can only have one source account. A source account may edit and delete its bundles

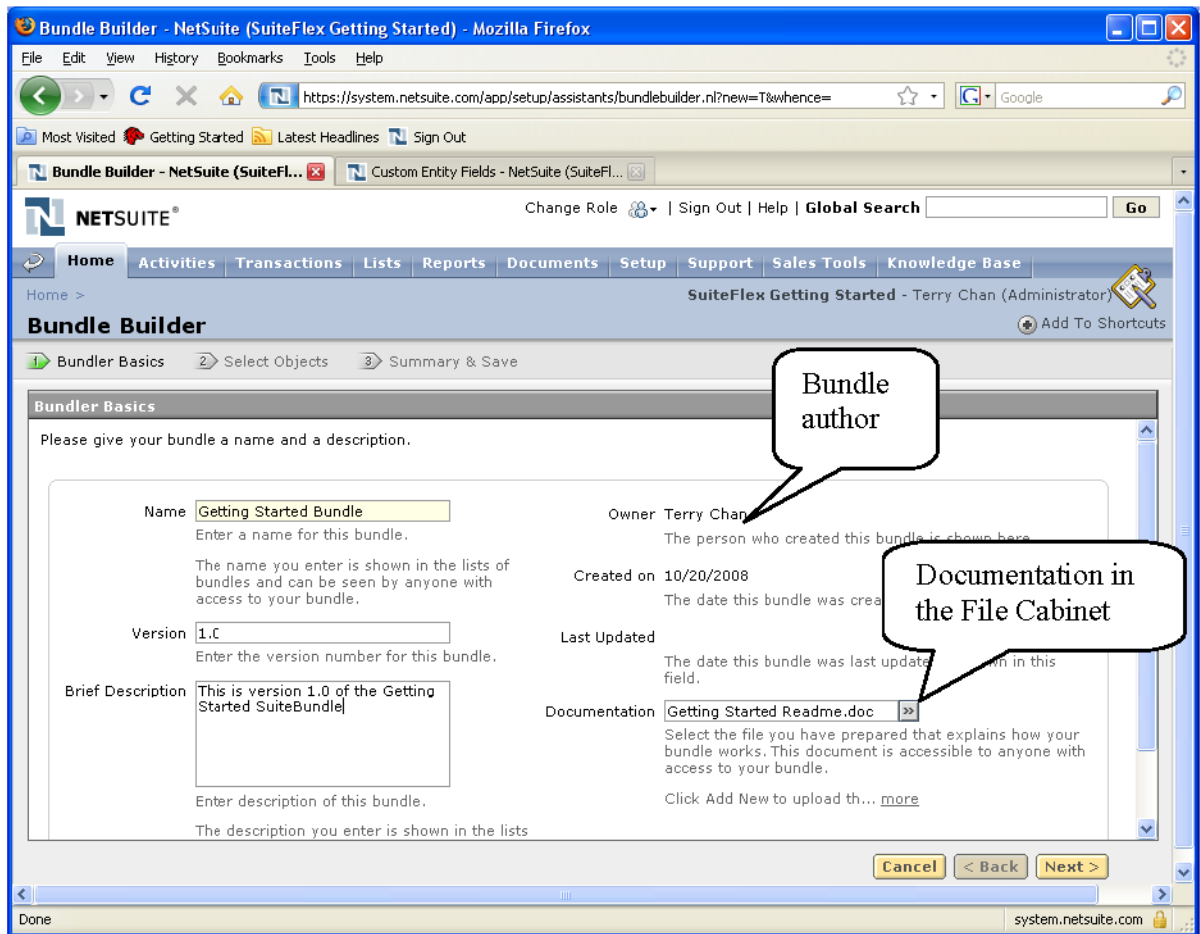
- **Target Account** - from a SuiteBundle's perspective, a target account is a NetSuite to which it is installed. Each SuiteBundle may be installed on 0 to many target accounts
- **Bundle Availability Level** – the bundle availability level controls the access level of a SuiteBundle. A bundle author may configure a SuiteBundle to be Private, Shared (with 0 to many other target accounts) or Public
- **SuiteSource Repository** – a common repository for all NetSuite accounts to store their bundles, or to install bundles from.

It should be noted that bundle versioning is different from traditional versioning concepts of the client-server environment. Throughout its life time, a bundle evolves from version to version. Its content may change across versions, but the bundle itself remains the same physical bundle. Once updated, a bundle cannot be rolled back to an older version of itself. This is different from versioning of on-premise software where version upgrading involves uninstalling and/or replacing of binary files, and is consistent with the SaaS delivery model.

For detailed information about SuiteBundler, please see the SuiteBundler section in the NetSuite Help Center.

## Creating SuiteBundles

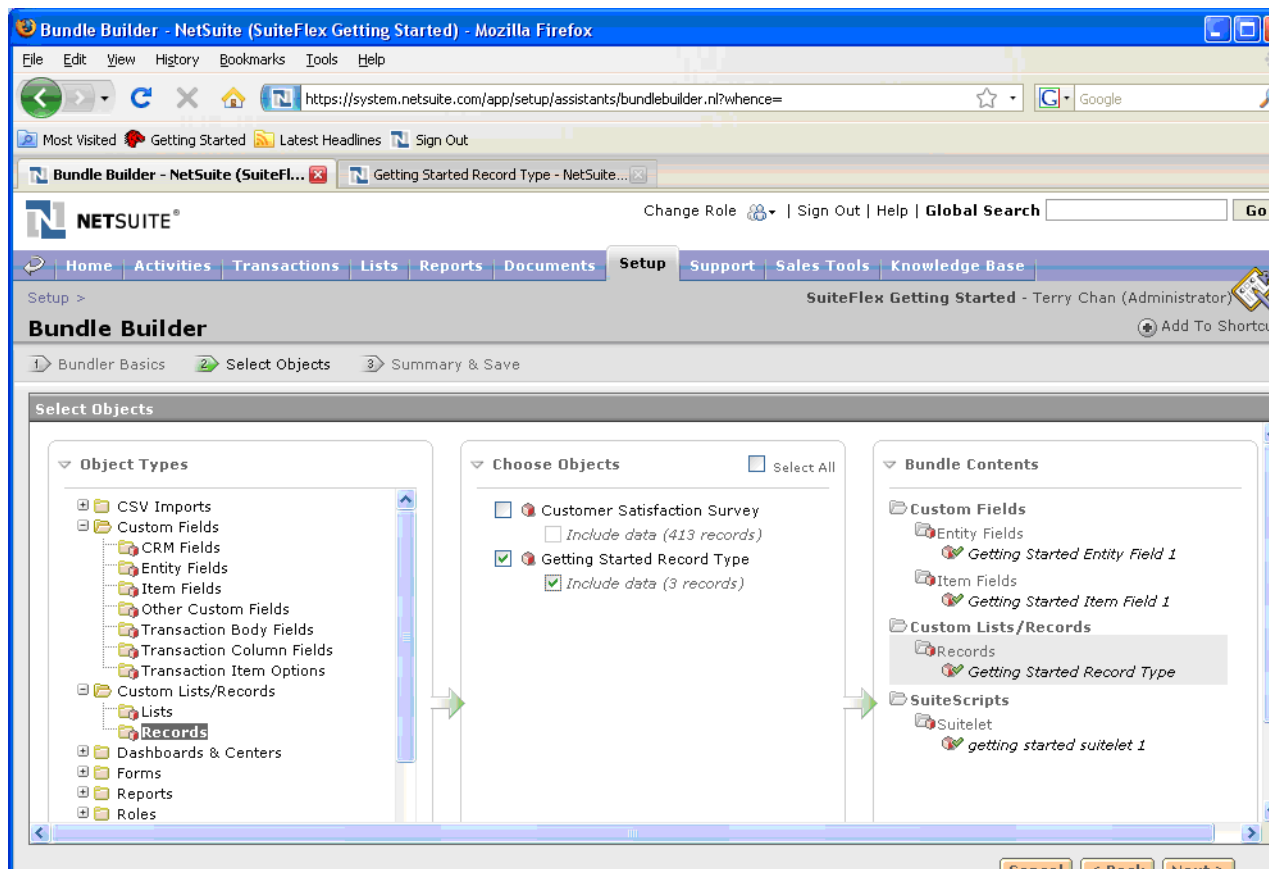
Before a SuiteBundle can be created, the administrator has to enable the SuiteBundler feature. Once enabled, the SuiteBundler may be accessed by going to Set Up → Customization → Create Bundle.



The bundle author provides basic information on the first screen.

**Best Practice:** Documentation should be shipped with bundles to provide support and to answer common how-to questions. This is especially important for bundles that contain SuiteScripts.\

The second page is where the bundle author defines the content of the bundle. A wide variety of customization objects can be added to a bundle.



**Best Practice:** Check if the member objects you want to ship as your solution can be added to a bundle. Devise workarounds, such as using SuiteTalk Web Service import or CSV import, for the objects that cannot be included in bundles

Adding custom record types to a bundle means adding its definition. However bundle authors may also choose to include data as well. Checking the “Include Data” check box will include all instances of that custom record type into the bundle. **Those custom record types that target accounts will create instances of should not have this option enabled in the bundle to avoid inadvertent data deletion during bundle updates.** The alternatives to ship custom record data as part of the solution are:

- Include CSV import map in the bundle
- Use SuiteTalk Web Services to import data

Before the bundle is saved, the summary of the components is shown in the next screen.

The screenshot shows the NetSuite SuiteBundler interface in a Mozilla Firefox browser window. The browser address bar shows the URL: <https://system.netsuite.com/app/setup/assistants/bundlebuilder.nl?whence=>. The page title is "Bundle Builder - NetSuite (SuiteFlex Getting Started) - Mozilla Firefox". The Netsuite logo is visible in the top left, and the user is identified as "Terry Chan (Administrator)".

The main navigation bar includes: Home, Activities, Transactions, Lists, Reports, Documents, Setup, Support, Sales Tools, Knowledge Base. The current page is "Setup > Bundle Builder".

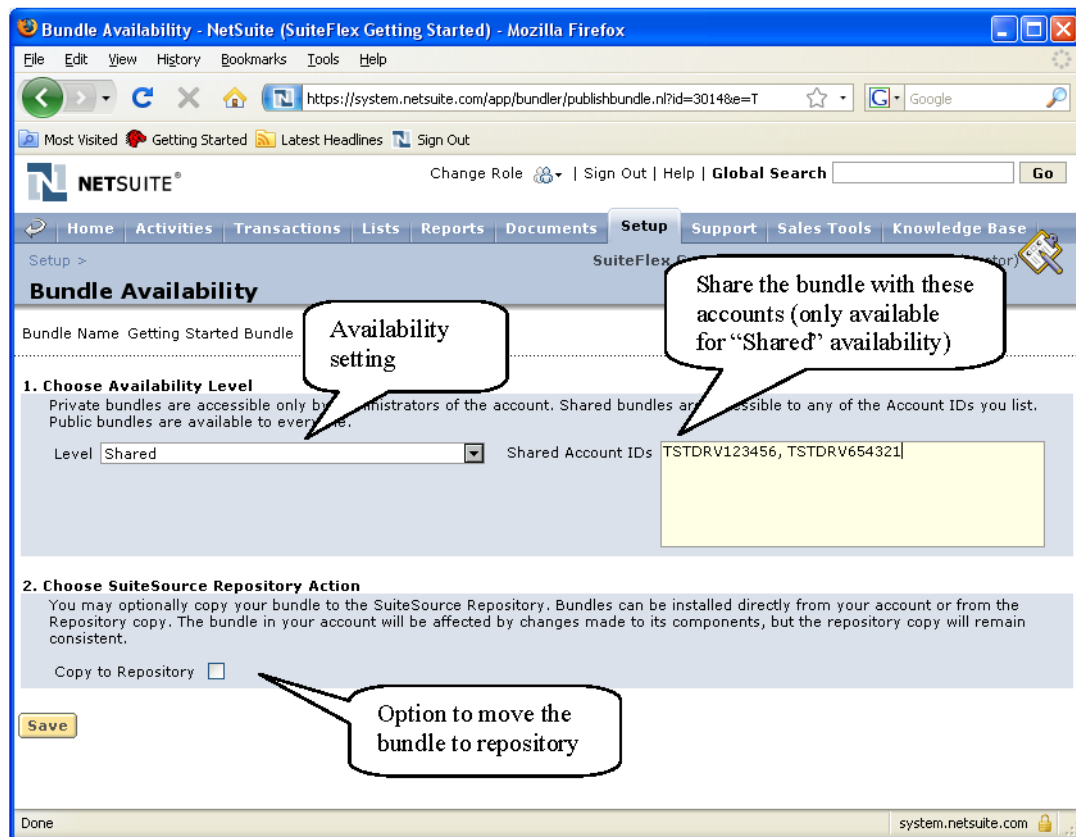
The "Summary & Save" section contains the following text: "Please review the contents of your bundle. Note that there may be more objects than you selected in the previous step due to dependencies that exist on the objects you selected, as shown by the link icon below."

Name	ID	Referenced By
<b>Custom Fields</b>		
<b>Entity Fields</b>		
Getting Started Entity Field 1	custentity_getting_started_ent1	
<b>Item Fields</b>		
Getting Started Item Field 1	custitem_getting_started1	
<b>Custom Lists/Records</b>		
<b>Lists</b>		
Customer contact type (4 records)	customlist16	Customer Satisfaction Survey
Relevant/Irrelevant (4 records)	customlist17	Customer Satisfaction Survey
Number of support contacts (4 records)	customlist20	Customer Satisfaction Survey
Satisfaction Level (5 records)	customlist13	Customer Satisfaction Survey
YesNo (2 records)	customlist12	Customer Satisfaction Survey

Buttons at the bottom: Cancel, < Back, Save.

Notice the “Referenced By” column on the right-hand side. The SuiteBundler would include objects in a bundle that are referenced by those explicitly added by the bundle author in order to maintain referential integrity. In the screen show above, some of the objects added by the author reference the custom record type “Customer Satisfaction Survey”, hence that custom record type as well as all other customization objects it references are added to the bundle automatically.

After saving the bundle, configure its availability by clicking the “Modify availability of your new bundle” hyperlink.



The Level drop-down menu determines the access control level of the bundle.

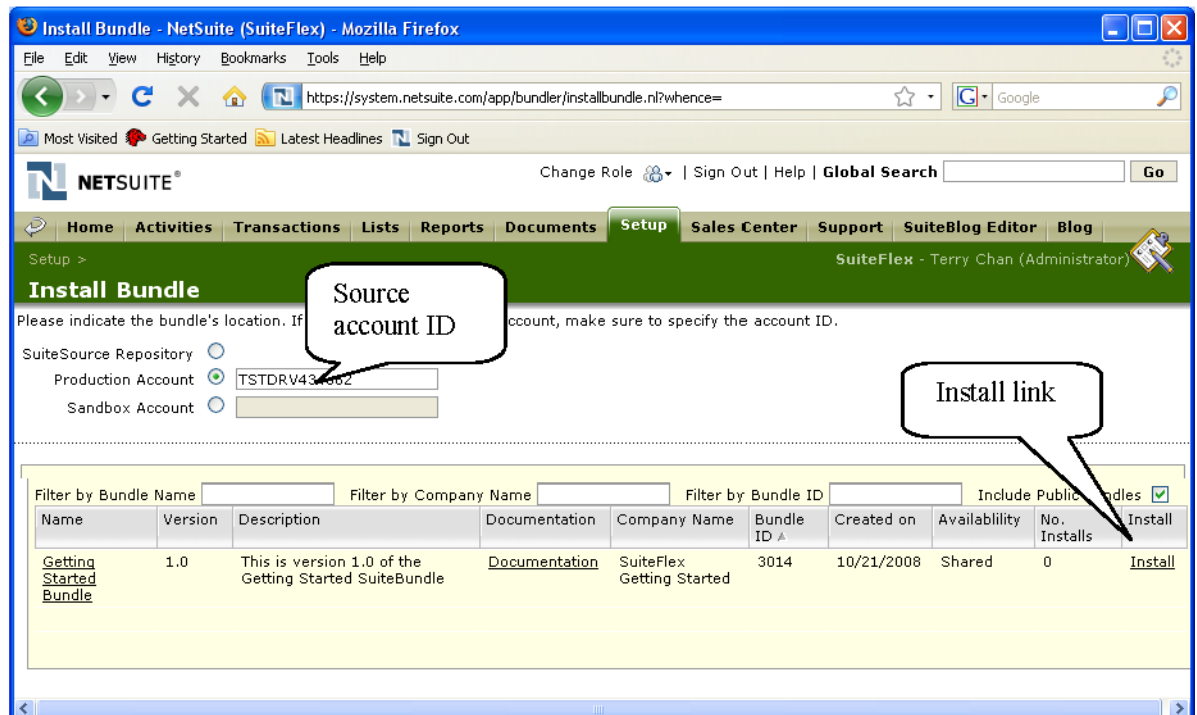
- **Private** – only the account that created the bundle may install it (hence it is both the source and target account). Generally, bundles with this level of availability are meant for an account's own use.
- **Shared** – shared bundles may be installed by the list of accounts specified in the "Shared Account IDs" field. ISVs that wish to charge their customers may set their bundles to this availability level and specify the customers' account IDs.
- **Public** – any NetSuite account may install a public bundle. ISVs that wish to use bundles to attract potential customers to their offerings that complement NetSuite (such as another SaaS) may use this availability level.

Bundle authors may choose to copy the bundle to the repository. Bundles copied to the repository remains consistent even when changes have been made to it in the source account. Hence they may be used as stable, generally available versions to be installed by target accounts.

**Best Practice:** customization objects pertinent to a vertical solution should be grouped into a monolithic bundle instead of multiple logical bundles. This eliminates the inter-dependence that multiple bundles would introduce

## Installing and Updating SuiteBundles

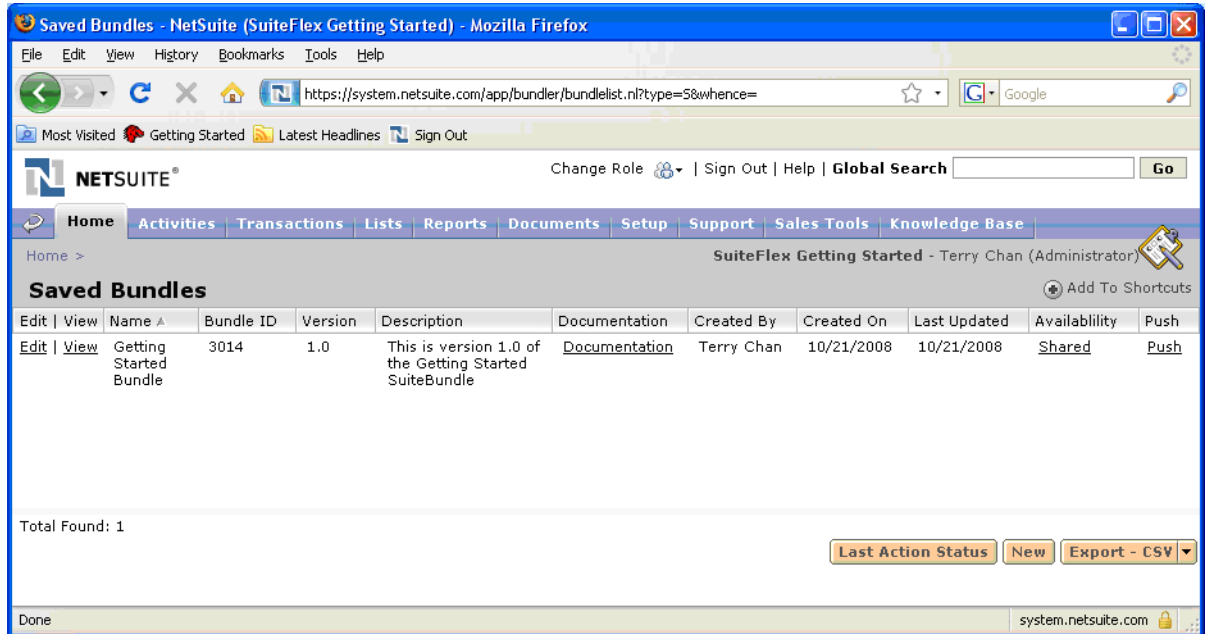
In order to install a SuiteBundle, the administrator of the target account should navigate to Set Up → Customization → Install Bundle.



To install a bundle from the repository, select the SuiteSource Repository radio button and hit the TAB key. To install a shared bundle directly from the source account, select the Production Account radio button, enter the source account ID and hit tab. Hit the Install hyperlink to install the bundle to the target account.

Go to Set Up → Customization → Install Bundle → List to view the list of bundles currently installed on the account. For each installed bundle, there is an “Update” link to update it. ISVs and vertical solution providers may notify their customers to update their installed bundles by going to this screen.

Starting with version 2008.2, bundles may also be installed and updated via a “Push” model – pushing bundle content from the source account to target accounts. In the source account, go to Set Up → Customization → Create Bundle → List. For each bundle, there is a Push link



Hit the Push link to install the bundle to new target accounts, or to update the bundle already installed on the current target accounts. Note that bundles may only be pushed to accounts of which the currently logged in user at the source account also has administrator access.

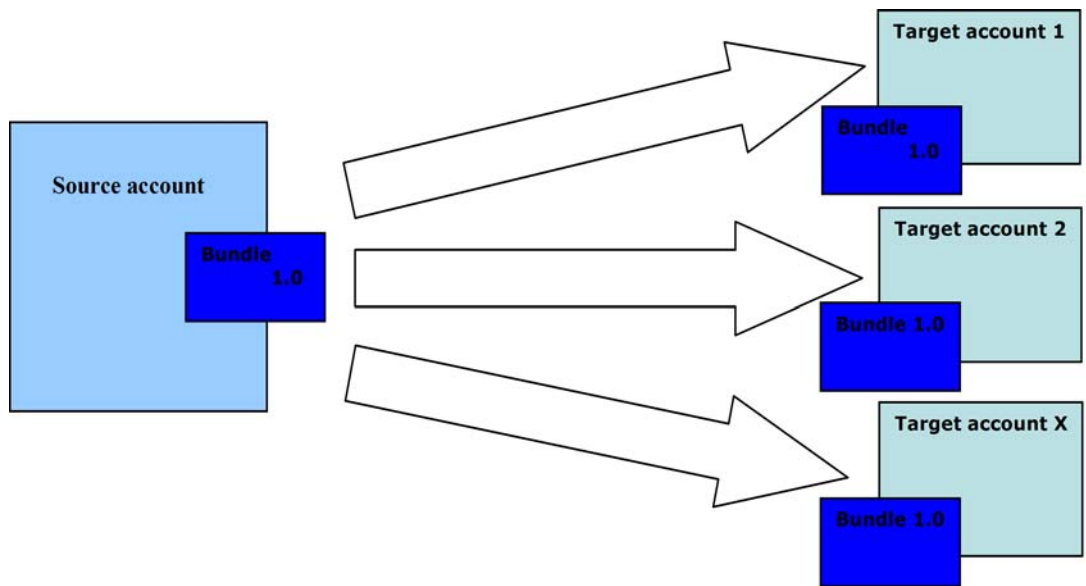
## Installing, Maintaining and Supporting Vertical Solutions Using SuiteBundler

SuiteBundler provides the basis to easily deploy and maintain vertical solutions for multiple NetSuite accounts. It provides basic features to satisfy the testing needs for regular NetSuite users as well as for solution providers.

ISVs may also use the SuiteBundler to deploy their SaaS based vertical solutions. SuiteBundles could be mission critical applications atop the NetSuite platform that are themselves rich in features and highly customizable. With this level of sophistication come the traditional software development requirements and customer support requirements, as well as SaaS specific release requirements. Multiple practices to satisfy these requirements using the SuiteBundler will be discussed.

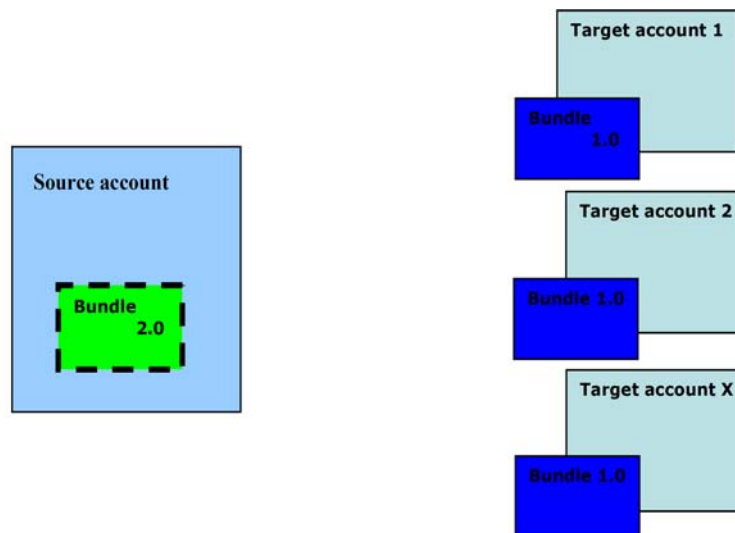
### Direct Bundle Install

Installing a bundle directly from the source account to the target account (described in [SuiteBundler Basics](#)) is the simplest way to deploy and maintain vertical solutions. The following diagram displays this deployment method.



This deployment model works well for simple bundles that contain only a few, if any, SuiteScripts. Any changes and updates made to the bundle could be easily propagated to the target accounts by notifying their administrators to update or by pushing updates to target account.

However, this model is not ideal for more complex bundles, or bundles that contain a lot of SuiteScripts, due to the lack of de-coupling between the current production version and the next version that is under development. Consider the scenario when bundle 1.0 has been deployed in production, and version 2.0 of the bundle (shown as green box with dotted border) is still under development as shown below.



Note that bundle 2.0 on the source account is still the same physical bundle as bundle 1.0, except it is now in a transition period because of developmental changes.

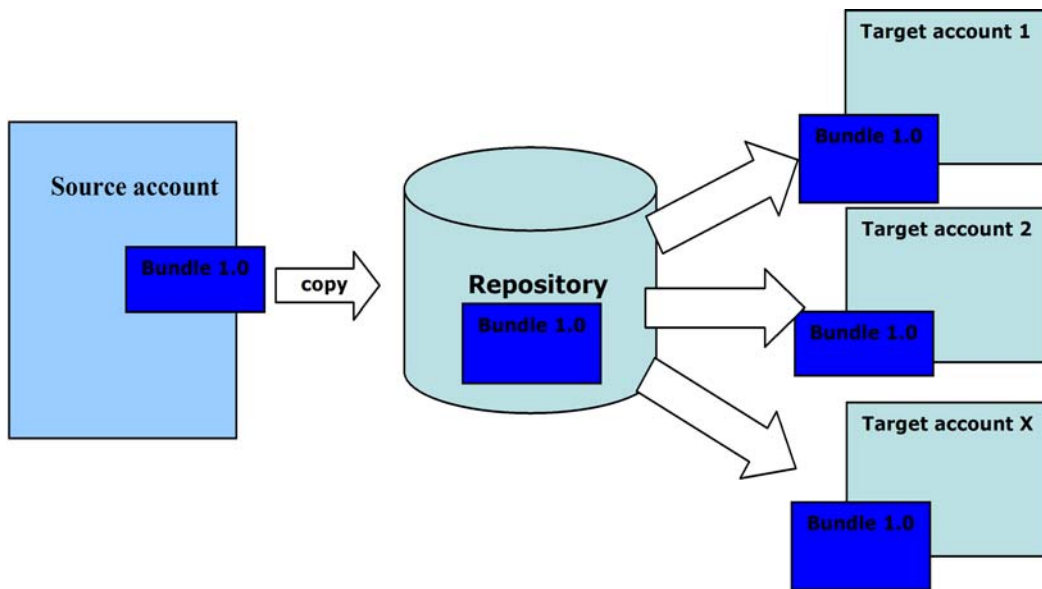
As developers actively work on bundle version 2.0, the target accounts become difficult to support because they are now out of sync with the source account's bundle content. Bug releases for version 1.0 needs to be done in the error-prone fashion of manually making changes to the bundle contents on the target accounts. Target account administrators may inadvertently update the installed bundles, thus overwriting them with a version that is not yet ready.

Due to the lack of decoupling and the danger of production bundles getting overwritten, ISVs should carefully consider its limitations when choosing their bundle deployment methods. The advantages and disadvantages of the direct install model are listed in the table below.

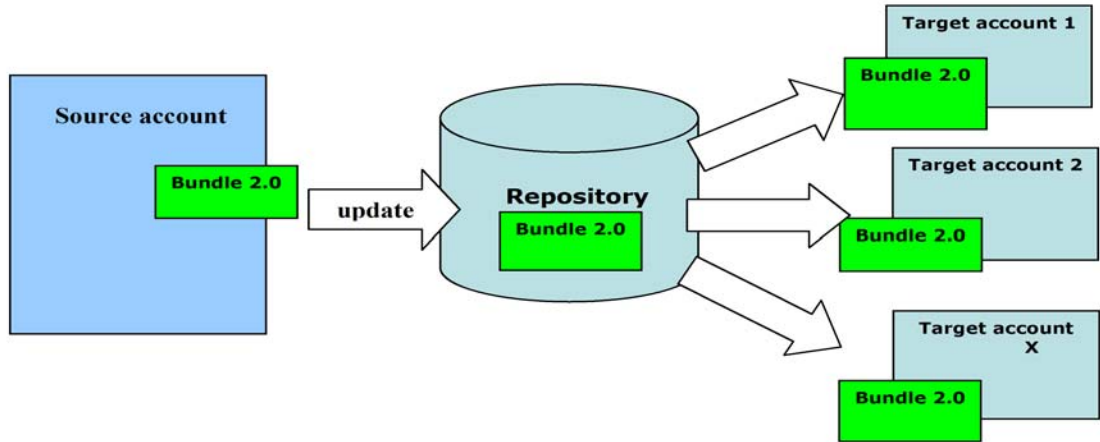
Pros	Cons
Easy bundle deployment of production ready bundles	Difficult to release bug fixes to production bundles
Easy to update bundles	Cannot prevent production bundles from being overwritten by un-intended bundle updates
	No decoupling of production ready bundle and bundle under development
	No consistent copy of production ready bundle

### Installing bundles from the Repository

Bundle authors may copy their bundles to the repository. The repository provides the needed separation between the production bundle version and new bundle version that is under development. Bundles copied to the repository stay consistent even when changes have been made to the original bundle in the source account. Target accounts may install bundles from the repository instead of directly from the source account.

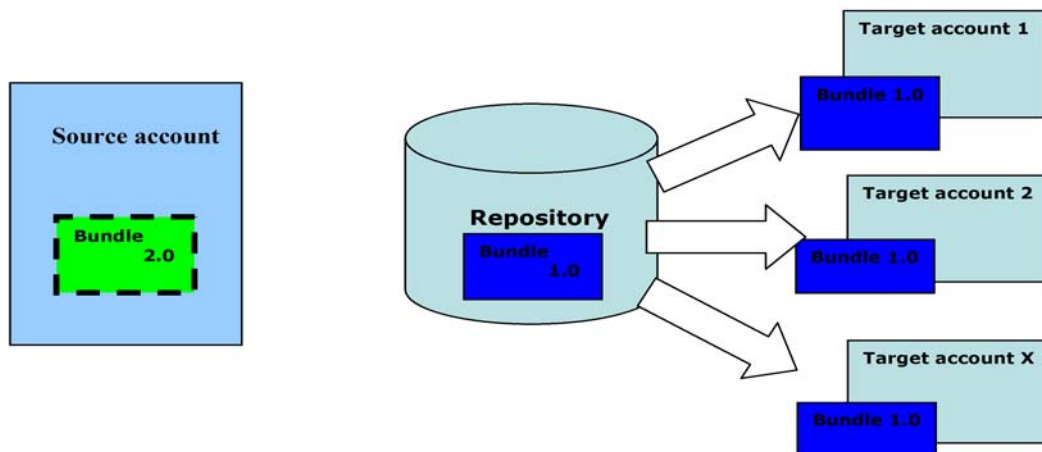


When the new version is ready to be released, the bundle author would update the repository copy from the last version to the new version. Subsequently, the target account administrators would update their bundles from the repository as shown below.



The repository offers a number of advantages over the direct install model. The repository makes deployment easy by providing a consistent copy of the bundle. Target accounts that install the bundle from the repository at different times will receive the same content; inadvertent overwriting of bundle content will not happen. The separation of the production version bundle and new version addresses the needs for software development teams to maintain separate code bases.

However, there is still a serviceability issue when using the repository to deploy bundles. Consider the scenario where bug fixes for the production bundle need to be provided to the target accounts when the source account is actively working on the new version (green box with dotted border).



Bug fixes cannot be directly introduced into bundle version 2.0 because it is still under development and not ready to be moved to the repository. The repository is not a NetSuite account; hence the bundle 1.0 stored there cannot be edited to put in bug fixes.

Using the repository to deploy SuiteBundles is well suited to bundles that are light in SuiteScripts and/or do not have active development work being done for new versions. The following table summarizes its strengths and weaknesses.

Pros	Cons
Easy bundle deployment of production-ready bundles	Difficult to release bug fixes to production bundles
Easy to update bundles	
Provides de-coupling of production ready bundle and bundle under development	
Consistent copy of production ready bundle available	
Prevents production bundles from being overwritten by un-intended bundle updates	

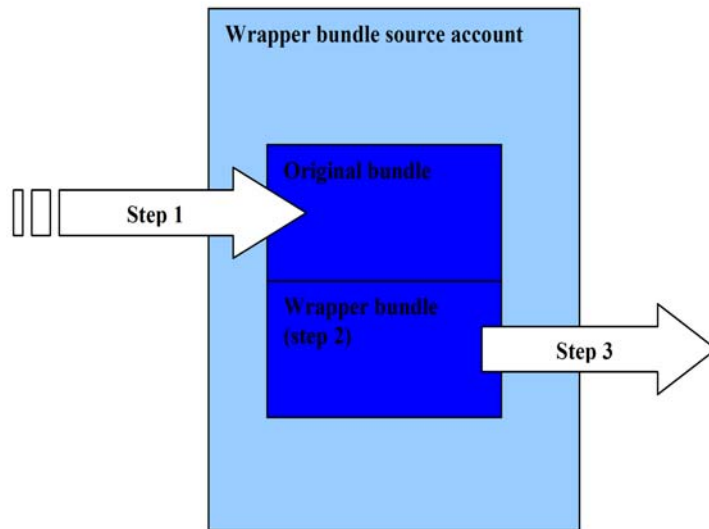
## Wrapper Bundle Deployment Model

The model that satisfies deployment requirements, maintenance requirements, software development requirements, and serviceability requirements is the wrapper bundle model. This design introduces two primary concepts: the deployment account and the wrapper bundle.

The deployment account is another NetSuite account that is owned by the same organization as the bundle source account. The deployment account is meant to be the platform from which to deploy the production bundle to target accounts. This design provides the code separation, and effectively turns the original source account into a development account used strictly for developing new bundle versions.

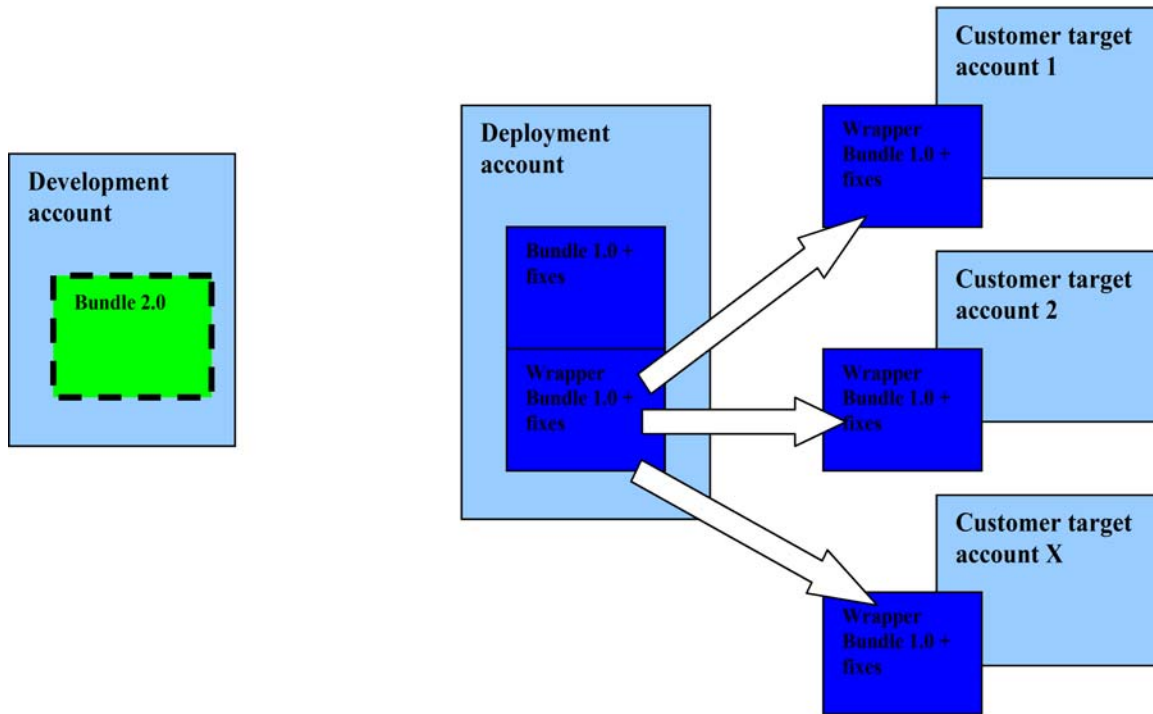
A wrapper bundle is a SuiteBundle that has content identical to those of another bundle installed from another source account. The steps to create a wrapper bundle are as follows.

1. Install a bundle to a target account
2. On the target account, create a new bundle that has the same content as the bundle installed in step 1. This bundle is the wrapper bundle - it “wraps” the original bundle
3. Deploy the wrapper bundle to other target accounts



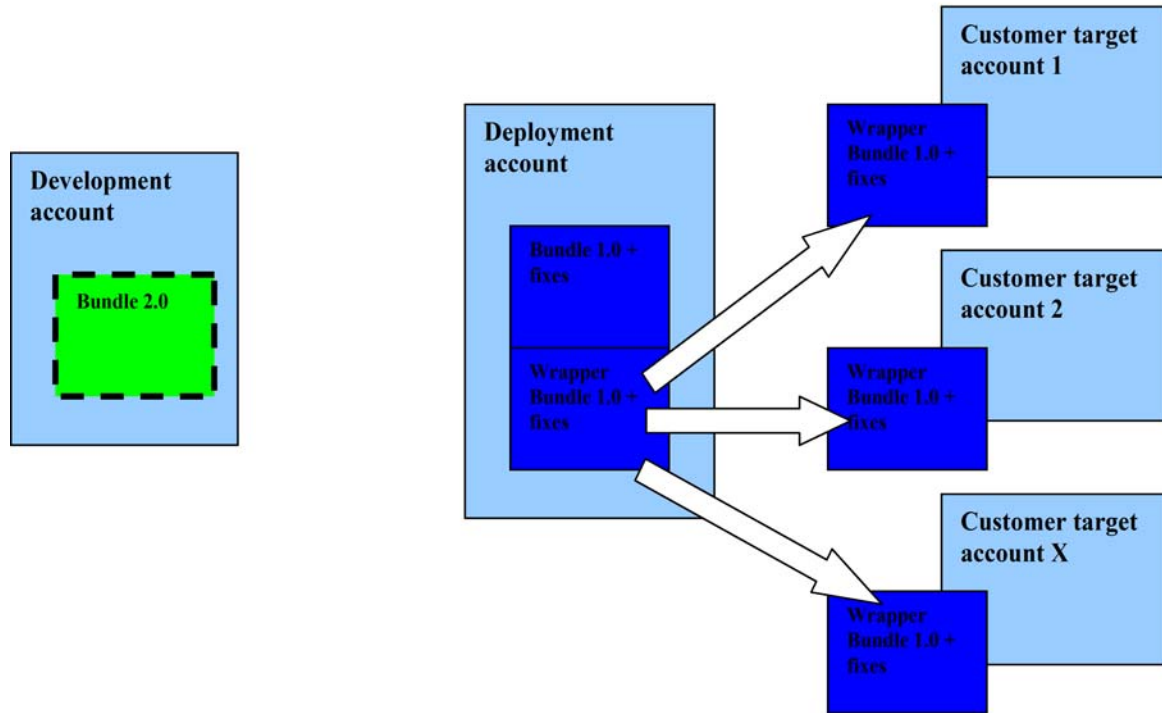
The diagram above shows an account that is simultaneously the target account for the original bundle as well as the source account of the wrapper bundle. The two bundles are two separate physical bundles, but they share the same objects. This gives the wrapper bundle the unique attribute of its content is updated automatically when the original bundle is updated.

The wrapper deployment model is a structure that combines the development account, deployment account, and the wrapper bundle. It is able to address the needs for a SaaS vendor's development team, release team and customer support team. The general structure is shown in the following diagram.

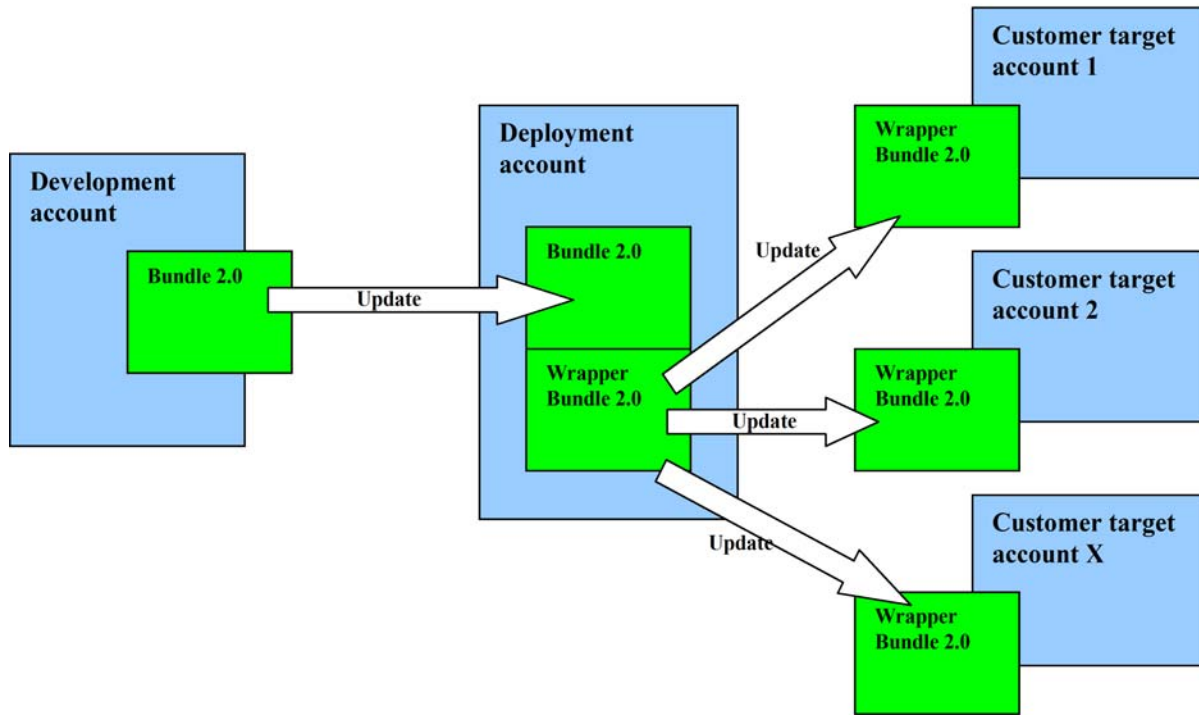


The deployment of the wrapper bundle to the customer target account is quick and easy just like the other approaches discussed. Since the deployment account is a NetSuite account, all the contents within the bundle could be edited, thus enabling bug fixes to be incorporated without impacting the development account. The development account would be dedicated to develop the new version of the bundle.

It should be noted that the de-coupling of production ready bundle and bundle under development introduces code branching, which means bug fixes for bundle 1.0 needs to be manually moved back to bundle 2.0 (if applicable). The following diagram shows the bug fixing, and new bundle development processes in the wrapper bundle model.



When bundle version 2.0 is ready to be released, it is pushed from the development account to the deployment account (diagram below). All the components in the wrapper bundle will be updated automatically. The wrapper bundle author still needs to manually add any new objects to the wrapper bundle that version 2.0 introduces. Wrapper bundle 2.0 is then deployed to the customer target accounts. The maintenance and new version development cycle begins again.



Here are the summary of advantages and disadvantages of the wrapper deployment model.

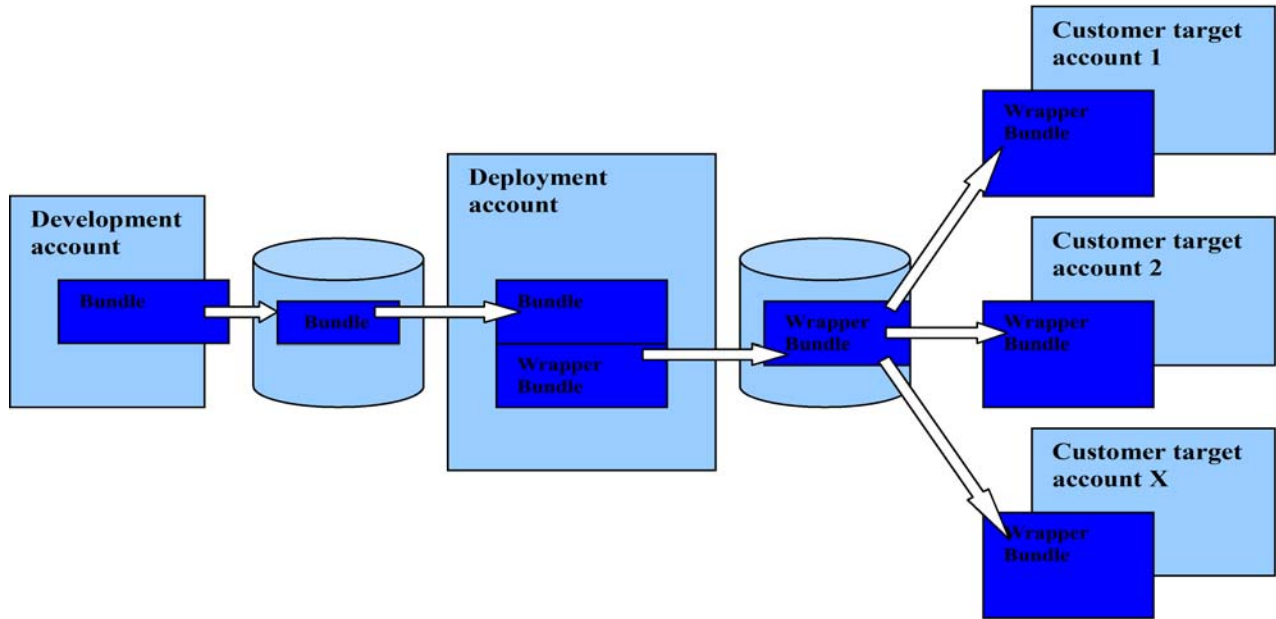
Pros	Cons
Easy bundle deployment of production-ready bundles	Difficult to release bug fixes to production bundles
Easy to update bundles	Cannot prevent production bundles from being overwritten by unintended bundle updates
Provides decoupling of production-ready bundle and bundle under development	
Consistent copy of production-ready bundle available	
Bug fixes for production bundles can be easily released	

Due to the numerous advantages offered by the wrapper bundle, ISVs with complex bundles that included a lot of SuiteScripts should seriously consider using this deployment model.

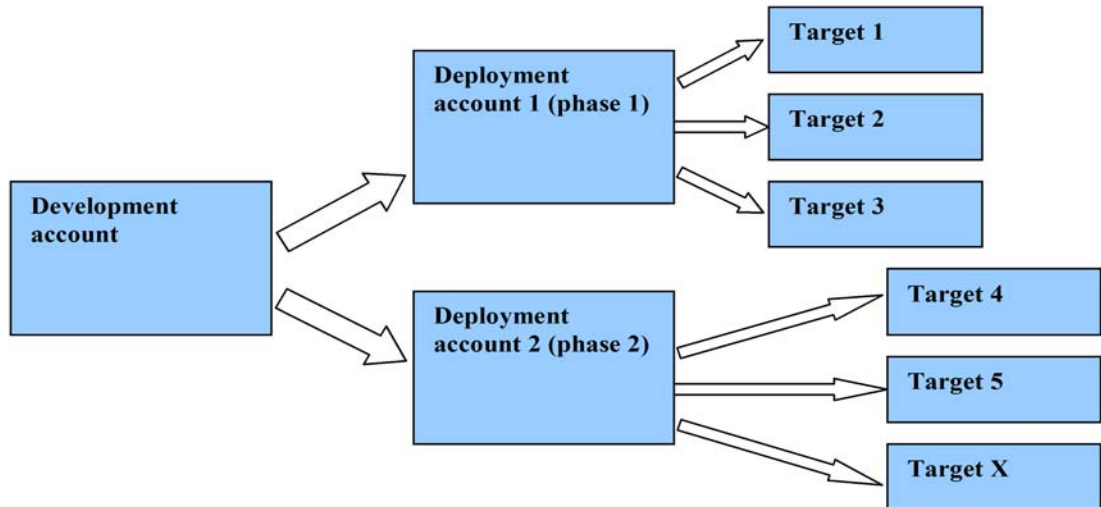
### Advanced Variations of the Wrapper Bundle Model

The core design of the wrapper bundle deployment model addresses most of the basic needs of maintenance, release and customer support. It could be further enhanced to address other common needs such as beta testing, phased releases, and customer support testing environment. The following advanced deployment models are all based on the basic wrapper model discussed earlier.

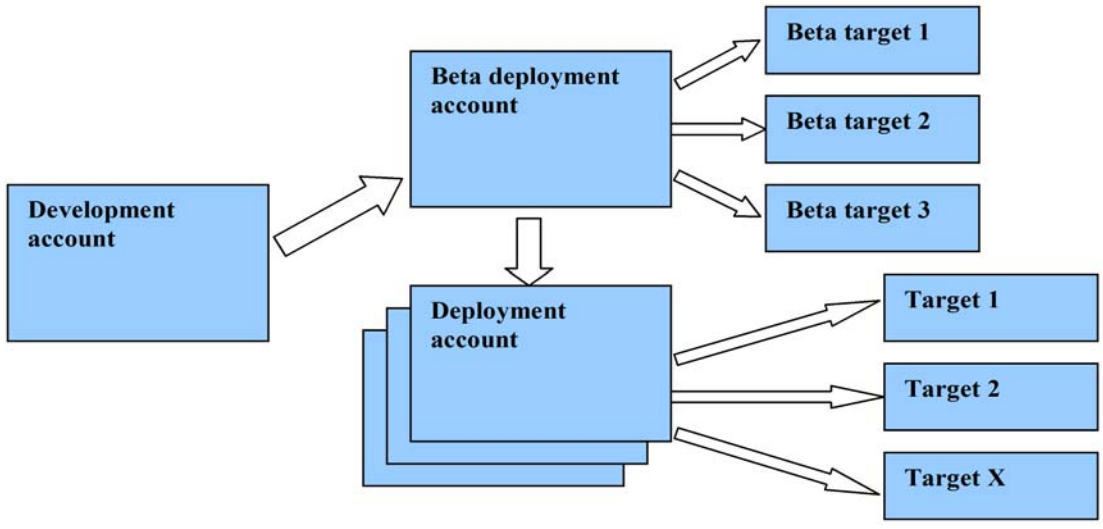
By utilizing the repository in the wrapper bundle model, the target accounts would receive a consistent, supported copy of the wrapper bundle as shown below.



A variation of the wrapper bundle to support phased releases is to implement multiple deployment accounts in parallel. In the diagram below, there are two deployment accounts for two different phases – a leading phase and a trailing phase. Since the deployment accounts are separate, they would maintain their own wrapper bundle to their respective target accounts. In phase 1 of a phased release, the development account would update deployment account 1, which in turns updates the target accounts. In phase 2, the development account updates deployment account 2.



The wrapper model with deployment accounts in series could be used to implement a beta framework. In the diagram below, a beta deployment account is set up serially with the deployment account. Beta target account testers obtain their bundles from the beta deployment account. When the beta is ready to be released, it is pushed to the deployment account. Note that multiple deployment accounts could be set up in parallel as shown.



# SuiteTalk Web Services

The SuiteTalk APIs expose NetSuite data and business process as a SOAP-based web service. Using these APIs, applications can programmatically access NetSuite without using the web browser. SuiteTalk's choice of adopting the SOAP protocol allows easy data communication with common development platforms that support the standard (example: Axis library for the Java platform).

Using the SuiteTalk APIs, data sources and applications outside of NetSuite can communicate with NetSuite. The following are some common scenarios for how ISVs and solution providers can use SuiteTalk to develop applications and services.

1. Desktop applications that uses a NetSuite account as the system of record
2. A SaaS vendor that integrates its service with NetSuite
3. An application that performs a nightly batch data synch between NetSuite and an external database
4. Importing historical data into NetSuite

## Record Types and Metadata Supported

Most standard NetSuite records are supported by SuiteTalk. The list of supported records spans all major facets of the NetSuite application from ERP to CRM to Customization. Please refer to the *SuiteTalk Records Guide* found in the NetSuite Help Center for the complete list of supported records and their fields.

In the SuiteTalk APIs, the `Record` class is the abstract super-class of all supported records. A supported record is always a concrete sub-class of `Record`, example: `Customer`, `SalesOrder`. Due to the neutrality required to be language agnostic, the SuiteTalk classes inheritance chain remains simplistic and does not implement language specific OO concepts such as multiple inheritance and interfaces.

A record's standard body fields are its attributes, examples: `Customer.email`, `SalesOrder.salesRep`. Composite attributes such as line items or sublists are structured as complex objects that contain arrays, examples: `SalesOrder.itemList`, `CalendarEvent.attendeeList`. Custom fields within a record (if available) are also structured as composite attributes, example: `Contact.customFieldList`.

**Tip:** A record attribute that ends with "List" is always a sublist.

In addition to standard records, SuiteTalk also supports custom objects and their metadata. An application external to NetSuite may query a NetSuite account to obtain the customizations done on it. This allows ISVs to ship off-the-shelf, generic applications that will work with any account. For example, a SuiteTalk POS application can be made to determine during runtime all the custom fields applied to the CashSale record so it can import CashSale records with the necessary custom fields set. For more information on this capability, please refer to the documentation for the `getCustomization` API.

## SuiteTalk Operations

SuiteTalk essentially exposes NetSuite as a data source for programmatic access, hence most of the data operations developers expect, such as insert, update, delete and select/search are supported. There are also a number of operations that are available as supporting operations for data read/write (examples: `initialize`, `getSelectValue`), provide metadata (example: `getCustomization`), or expose an application function to programmatic access (examples: `attach`, `detach`). Please refer to the *SuiteTalk Platform Guide* for detailed information on supported operations.

## How to Write a SuiteTalk Application

Before developers begin writing SuiteTalk applications, they should be aware that SuiteTalk operates on the same role-based permission structure as the browser interface. Since a SuiteTalk application needs a pair of sign-in credentials to login, its permission to various operations and records is subjected to the role it uses just like it would when it's used for a browser session. Example: a SuiteTalk application that logs in with the Accountant role will receive the same permissions as it will logging in to the browser interface using the same role.

Another characteristic of SuiteTalk is that its behaviour is very similar to that of the NetSuite UI. The workflow of a SuiteTalk application and its underlying SOAP exchange with NetSuite should tightly mimic the browser interface. For example:

1. A successful login operation must be performed to obtain a session in order to perform any subsequent operations
2. Some add operations are done in a tandem fashion such as loading an existing sales order to gain the context adding a new item fulfillment record. The loading of the sales order (using the `initialize` API) is the first operation and adding the item fulfillment record is the second
3. Restrictions and requirements on custom forms are honoured. Hence a SuiteTalk application's attempt to set a field that's hidden on a form results in a permission error. Likewise the SuiteTalk application must set fields that are required on the form it is using.

**Best Practice:** A SuiteTalk application should use the response object to handle any errors, if any, generated by a web service operation.

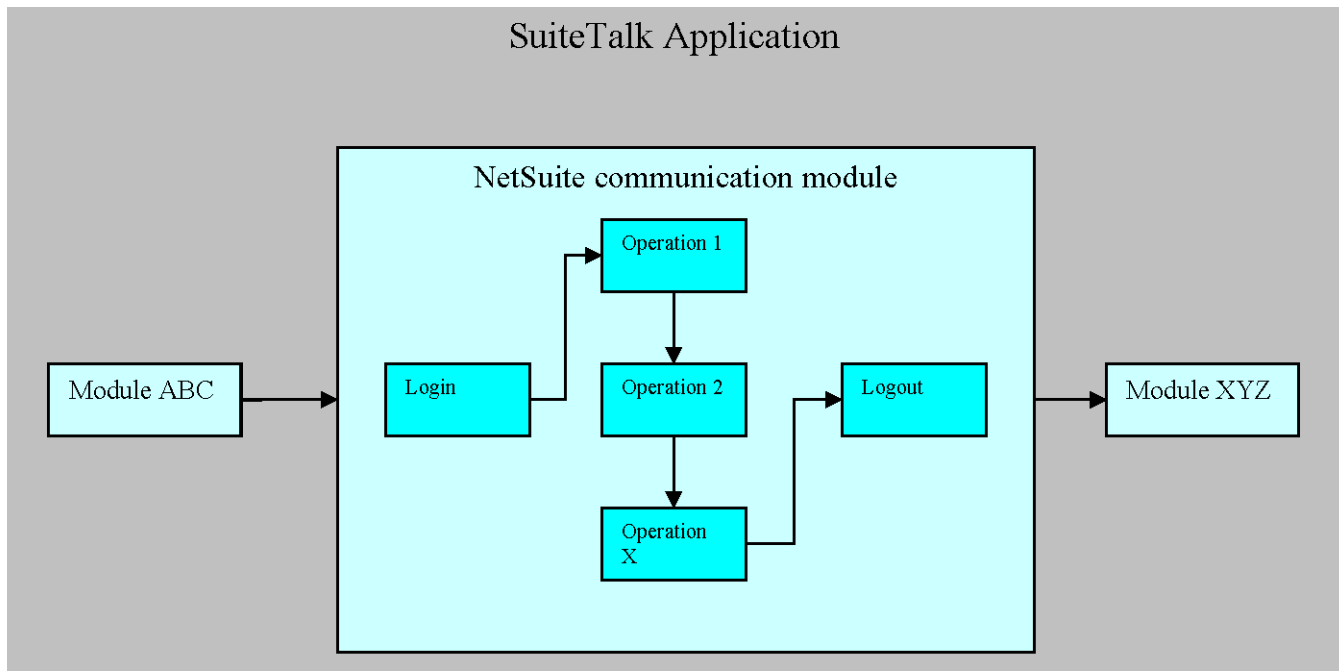
The combination of permission adherence and similar behavioural pattern between SuiteTalk and the NetSuite interface provide a consistent and predictable platform for developers. When

developing SuiteTalk applications, developers can be confident that if a certain workflow works in the UI, then its programmatic SuiteTalk equivalent should also work.

**Tip:** When unsure of how to achieve something with SuiteTalk, try observing how it's done in the browser UI, then replicate it programmatically.

Typically, a SuiteTalk application has modules that interact with NetSuite via SOAP, and other modules that deal with other components not directly related to NetSuite (examples: cash register of a POS system that uses NetSuite as backend). This section will focus only on the module that communicates with NetSuite.

The diagram below shows a SuiteTalk application that has a module dedicated to handling communication with NetSuite. Please note that this is an over-simplified design. In practice such a module would likely have a lot more interaction with other modules within the application.



When the requirements have been gathered, application designers need to consider the following when writing SuiteTalk applications:

1. Platform specific considerations
2. Roles and permission considerations
3. License and concurrency considerations

## Platform Specific Considerations

One of the advantages of SOAP-based web services such as SuiteTalk is that its reliance on industry standards provides platform neutrality - any programming language that supports the

SOAP protocol may use it with success. The most popular platforms for SuiteTalk applications are .Net, Java and PHP.

Microsoft .Net platform is the most popular platform for SuiteTalk developers. The Visual Studio .NET IDE has built-in web service tools, making WS client development easy. A common problem experienced by .Net developers is they neglect to set the matching “Specified” method when setting record fields that aren’t of type string. This results in the SOAP for those fields not being generated and causing user error messages. Please refer to the *SuiteScript FAQs* section in the NetSuite Help Center for details.

Java is another popular platform for SuiteTalk developers. Unlike .NET, Java applications can run on multiple operating systems, making it the ideal platform for heterogeneous enterprise IT environments. The most frequently used SOAP library for Java is the open source Apache Axis library. Due to a bug of Axis mishandling cookies, Java SuiteTalk applications sometimes have trouble maintaining sessions. NetSuite has provided a [patch](#) to alleviate this problem. Release engineers should ensure the deployed JAR file includes this patch to avoid session errors.

PHP is a web-based programming language frequently used by IT department staff. Some SaaS offerings are also implemented with PHP. Unlike .Net and Java, PHP does not have SOAP libraries that work well with SuiteTalk. Using PHP to make web service calls requires the developer to code logic to handle cookies to maintain session – something Java and .Net developers can rely on the platform to do. The NetSuite PHP toolkit introduced in NetSuite version 2008.2 alleviates this problem. The introduction of stateless web services in version 2008.2 WSDL is designed to remove the need for PHP developers to programmatically maintain sessions.

**Best Practice:** Regardless of the underlying programming platform, a SuiteTalk application should have the ability to log SOAP messages to and from NetSuite. This is useful for debugging purposes, or as a complementary tool to the Integration and Automation Usage Summary by Jobs report.

## Roles and Permissions Considerations

Due to SuiteTalk’s reliance on NetSuite’s role-based permissions, it is important for SuiteTalk developers to put that into considerations during the design phase to ensure smooth deployments.

It is common for developers to use the administrator role during development time because it gives them full permissions and access to all the records and operations. However, the target end users are likely to have less powerful roles that may not have access to the data the SuiteTalk application requires.

Another role related consideration is the preferred custom forms of some roles may not have access to certain fields or sublists that a SuiteTalk application requires. Hence the application’s attempts to set those fields will result in permission errors.

The solution to these problems is to define a custom role and custom forms for the SuiteTalk application. The custom role should have the correct access permissions and operations permissions that the SuiteTalk application needs. The custom forms should give access to fields and sublists that are relevant to the SuiteTalk application. All SuiteTalk supported records have a `CUSTOMFORM` field for the application to reference specific custom forms.

Custom forms and custom roles may be shipped as SuiteBundles, making them easy to deploy and update. Please refer to the SuiteBundler section for details.

**Best Practice:** During testing phase, QA engineers should test SuiteTalk applications using the intended role(s), in addition to the administrator role to catch permission related defects.

## License and Concurrency Considerations

Every NetSuite user is given one browser session and one SuiteTalk web services session. The browser session means the user can be logged in to NetSuite with only one browser at any given time. For example, a user that logs into NetSuite using FireFox while her Internet Explorer is still logged in would result in her IE session being terminated.

SuiteTalk user sessions work the same way. A SuiteTalk session is single-threaded, which means it cannot be used concurrently by multiple applications. Consider the following scenario of two applications trampling each other's sessions:

1. Application *ABC* logs in to NetSuite and establishes a session
2. *ABC* performs some data manipulation
3. Before *ABC* has a chance to log out, application *XYZ* logs in **using the same pair of sign-in credentials**
4. *ABC* attempts another data manipulation, but is unsuccessful because its session was terminated by *XYZ*
5. *ABC*'s retry logic logs in again, thus terminating *XYZ*'s session

As shown above, multiple applications using the same user sign-in credentials at the same time can result in session problems that are difficult to diagnose. These problems can often be resolved by using one of the following solutions:

- Ensure each pair of sign-in credentials are not used by multiple applications at any given time
- Use a separate license for each SuiteTalk application in use

Multi-threaded applications may also face the same session problem. Threads that access the SuiteTalk session simultaneously will trample each other the same way multiple applications do. To ensure a multi-threaded application can safely use a single WS session, try these solutions:

- Develop a pooling mechanism that serializes the access to the NetSuite WS session. When a thread gains access to the session, the access is locked until the thread finishes using it and releases it back to the pool. All other threads must wait for their turn to access the session. This mechanism avoids session problems by ensuring only one thread may use it at any given time.
- Consider purchasing the WS Plus module. The WS Plus module allows a user to be designated as a "concurrent web service user". A concurrent web service user's session can have up to 5 concurrent sessions at any time. This means the user can have up to

five logged in WS sessions, each with their own jsessionid. Note that threads sharing the same login session (hence sharing the same jsessionid) will still clobber each other.