SAFE Guide

SuiteApp Architectural Fundamentals & Examples

Version 2019.1
January, 2019

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

If this document is in public or private pre-General Availability status:
This documentation is in pre-General Availability status and is intended for demonstration and preliminary use only. It may not be specific to the hardware on which you are using the software. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

If this document is in private pre-General Availability status:
The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your pre-General Availability trial agreement only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle Partner Network Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Sample Code

Oracle may provide sample code in SuiteAnswers, the Help Center, User Guides, or elsewhere through help links. All such sample code is provided "as is” and “as available”, for use only with an authorized

NetSuite Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos.

Oracle may modify or remove sample code at any time without notice.

No Excessive Use of the Service

As the Service is a multi-tenant service offering on shared databases, Customer may not use the Service in excess of limits or thresholds that Oracle considers commercially reasonable for the Service. If Oracle reasonably concludes that a Customer's use is excessive and/or will cause immediate or ongoing performance issues for one or more of Oracle's other customers, Oracle may slow down or throttle Customer's excess use until such time that Customer's use stays within reasonable limits. If Customer's particular usage pattern requires a higher limit or threshold, then the Customer should procure a subscription to the Service that accommodates a higher limit and/or threshold that more effectively aligns with the Customer's actual usage pattern.

Beta Features

Oracle may make available to Customer certain features that are labeled "beta" that are not yet generally available. To use such features, Customer acknowledges and agrees that such beta features are subject to the terms and conditions accepted by Customer upon activation of the feature, or in the absence of such terms, subject to the limitations for the feature described in the User Guide and as follows: The beta feature is a prototype or beta version only and is not error or bug free and Customer agrees that it will use the beta feature carefully and will not use it in any way which might result in any loss, corruption or unauthorized access of or to its or any third party's property or information. Customer must promptly report to Oracle any defects, errors or other problems in beta features to support@netsuite.com or other designated contact for the specific beta feature. Oracle cannot guarantee the continued availability of such beta features and may substantially modify or cease providing such beta features without entitling Customer to any refund, credit, or other compensation. Oracle makes no representations or warranties regarding functionality or use of beta features and Oracle shall have no liability for any lost data, incomplete data, re-run time, inaccurate input, work delay, lost profits or adverse effect on the performance of the Service resulting from the use of beta features. Oracle's standard service levels, warranties and related commitments regarding the Service shall not apply to beta features and they may not be fully supported by Oracle's customer support. These limitations and exclusions shall apply until the date that Oracle at its sole option makes a beta feature generally available to its customers and partners as part of the Service without a "beta" label.

Integration with Third Party Applications

Oracle may make available to Customer certain features designed to interoperate with third party applications. To use such features, Customer may be required to obtain access to such third party applications from their providers, and may be required to grant Oracle access to Customer's account(s) on such third party applications. Oracle cannot guarantee the continued availability of such Service features or integration, and may cease providing them without entitling Customer to any refund, credit, or other compensation, if for example and without limitation, the provider of a third party application ceases to make such third party application generally available or available for interoperation with the corresponding Service features or integration in a manner acceptable to Oracle.

Copyright

NOTE: Section with titles highlighted in yellow are those sections updated for the 2019.1 version.

Contents

NOTE: Section with titles highlighted in yellow are those sections updated for the 2019.1 version.

Introduction

The development principles described in this guide (“Principles”) are for ISVs (independent software vendors) that want to leverage the NetSuite platform and NetSuite’s development tools to build custom SuiteApp solutions for deployment into customer accounts.

SuiteApp developers must design, develop, test, and deploy their SuiteApps according to these Principles. In many cases, if the Principles are not followed, your SuiteApps will not run. In other cases, they may still run, but may result in data-related, performance, or user experience issues.

- **Principle 1:** Understand NetSuite Features and Data Schema
- **Principle 2:** Manage SuiteScript Usage Unit Consumption
- **Principle 3:** Optimize Your SuiteApps to Conserve Shared Resources
- **Principle 4:** Understand Your SuiteApp May Be One of Many in an Account
- **Principle 5:** Design for Security and Privacy
- **Principle 6:** Test Your SuiteApps
- **Principle 7:** Consider Deploying Your SuiteApps as Managed Bundles
- **Principle 8:** Maintain Your SuiteApps

Important: This guide assumes you are familiar with the SuiteCloud platform. The Principles are not intended to replace SuiteScript or SuiteTalk Web Services training. Note that while all SDN members are encouraged to review the Principles, only developers that participate at the Select and Premier levels are eligible to self-validate their SuiteApps to receive the Built for NetSuite badge.

A Note on Technical Support

As a member of the SuiteCloud Developer Network (SDN) at the Select or Premier levels, you are entitled to technical support from NetSuite. If you participate at either of these levels, please log your support cases through the SDN portal’s **SDN Support Request** tab.

Be aware that technical support from NetSuite is not a substitute for reading the documentation in the NetSuite Help Center or guides such as this one. NetSuite strongly recommends attending NetSuite Essentials, SuiteScript, SuiteTalk Web Services, and other NetSuite training.

Using the SAFE Worksheet

Included at the end of this document is the Appendix: SAFE Worksheet. The SAFE Worksheet is provided for the benefit of any Select- or Premier-level SDN partners who are preparing the submission of their responses to the questions in the verification process.

Some of the questions in the SAFE Worksheet include educational notes to aid partners with their responses to some of the verification questions. The questions in the SAFE Worksheet are provided only to aid partners with their preparations for the verification process. When engaging in the verification process, partners will submit their responses through the online tools provided by SDN.

Important: The questions in the SAFE Worksheet are similar to those used in the verification process. However, the questions in the SAFE Worksheet may not be an exact copy of the questions that must be answered during the verification process.

Further Reading

If you are unfamiliar with any of the concepts in this guide, see "Where do I go to learn more about the concepts discussed in this guide?" This section provides information on using the NetSuite Help Center and SuiteAnswers to further research any topic.

Additionally, there is a "Further Reading" section at the end of most of the Principles discussed in this guide. The topic titles listed under each "Further Reading" section are the exact titles you should use when searching in the Help Center or SuiteAnswers. If you choose, you can simply copy and paste the "Further Reading" topic titles into the Search fields in either the NetSuite Help Center or SuiteAnswers.

A Note on Document Versioning

As of the NetSuite 2014.2 product release cycle, the version number of the SAFE Guide has been changed from its sequential numbering convention, such as Version 1.2, to a convention that matches the NetSuite product release cycle.

The version number of this document is 2019.1. It completely replaces version 2018.1, which also served 2018.2. If there is a mid-release version, it will be 2019.1.1. Otherwise, the next version will be 2019.2.

1 Understand NetSuite Features and Data Schema

As a SuiteApp developer, you must invest time learning about the core NetSuite ERP/CRM features you want to extend with your SuiteApps. An understanding of NetSuite business processes is imperative to identifying SuiteApp integration points.

Feature knowledge also allows you to focus on building value-added SuiteApps that extend the platform's features and avoid duplicating them. For example, NetSuite provides order fulfillment functionality; SuiteApp developers may extend this functionality in NetSuite by customizing the Order Fulfillment object, but must not build an equivalent module that duplicates existing, built-in NetSuite order fulfillment functionality.

You must understand the features you want to extend before you start accessing and using NetSuite programming APIs.

SuiteApp developers and architects must also be aware that upon enabling certain features (at Set Up > Company > Enable Features), some work flows or data schema elements will be activated. Some examples are the "Pick, Pack, and Ship" feature, and the "Multiple Shipping Routes" feature. The former adds extra steps to the sales order fulfillment process; the latter activates additional shipping addresses in the sales order line items level. Therefore, it is important to experiment with features that are relevant to the SuiteApp you plan on building in order to help anticipate the wide variety of customer NetSuite accounts with different combinations of features enabled.

Example

If you want to build a SuiteApp shipping solution, you must learn how order fulfillment and shipping work in NetSuite. You must include hands-on exercises with the NetSuite modules that are relevant to your product in your learning process.

Additionally, you must read any NetSuite Help Center documentation related to possible accounting impacts of your design. For a shipping solution, you must familiarize yourself with the order entry process and the order fulfillment process, including "pick, pack, ship," how to set up shipping items, and the inventory and general ledger impact of transactions.

Depending on how broadly your SuiteApp extends NetSuite ERP/CRM functionality, there may also be secondary records and operations to consider. For a shipping solution, the Reallocating Item operation is something that can impact quantity of committed items to be shipped. Therefore, this operation warrants consideration as a potential integration point.

1.1 Solutions Must Work With Existing Processes

As a SuiteApp developer providing value-add through augmenting existing ERP/CRM business logic, you should reuse data schema provided by the NetSuite platform. (See Data and Record Types Considerations for additional information.) You must also ensure your SuiteApp works well with built-in NetSuite business logic.

Example

Consider a warehouse management system (WMS) that provides additional business logic and data management specific to the wholesale distribution vertical. The WMS may augment the Purchase Order (PO) receipt process by providing a custom process for receiving purchase orders into a warehouse.

The WMS must honor NetSuite as the system of record by using the NetSuite platform's transaction and item records as its backbone. These record types include the Purchase Order, Item Receipt, and Inventory Item record types. A SuiteApp may provide a series of Suitelet pages to build a vertical-specific PO receipt process, a special mechanism to trigger the process, and custom record types (not built in to NetSuite) to represent specific business objects in a warehouse environment.

You must also consider how to handle the standard PO receipt process provided by the NetSuite platform. This process is usually started by clicking the Receive button on a PO that has a status set to Pending Receipt. You must decide whether clicking Receive should be allowed, and if so, under what circumstances.

Because clicking the Receive button on the standard NetSuite PO form to directly receive it does not invoke the custom WMS logic, receiving a PO using the standard Receive button may bypass the WMS Suitelet pages. This bypass may cause WMS-specific custom records to be out of sync with their related standard records. In this case, the Receive button on the custom PO form, and possibly the ability to directly receive a PO in NetSuite, should be disabled for items under the control of the WMS.

Conversely, the WMS must be flexible enough to allow the direct receipt of a PO for items that are not stocked in the inventory for sale, and, therefore, should not be exempted from the WMS logic. For example, replenishing office supplies is usually not controlled by a WMS, thus the direct receipt of these supplies should be allowed.

Note that POs for office supplies and other non-warehouse items may be entered by non-warehouse staff using roles unrelated to warehousing. Therefore, role considerations must also be factored into the WMS testing process.

1.2 Data and Record Types Considerations

As a SuiteApp developer, you should use standard NetSuite records when it makes sense, and simply extend them with custom fields when necessary.

You must thoroughly identify the records or fields that are already available. NetSuite is a deep and comprehensive application. Most of the fields you will need for your SuiteApp are already built into the system.

Note: Refer to the SuiteScript Records Browser and SuiteTalk Schema Browser (both are available in the NetSuite Help Center) for a comprehensive list of records and fields supported by SuiteScript and SuiteTalk, respectively.

In cases where you need to record information based on specific business needs, and there are no equivalent built-in fields in which to store your data, you should create custom fields to hold your information.

1.2.1 Overextending Standard Records

Do not “stretch” the use of standard NetSuite records. Doing so may cause problems with other SuiteApps or integrations that use the same standard records. When necessary, create custom records to meet the needs specific to your business. By doing so, you avoid misusing standard NetSuite records.

Example

Consider a car rental module SuiteApp. You might think you can use the NetSuite CalendarEvent object for operations such as managing rental periods. By doing this, however, you are actually stretching the functionality of the CalendarEvent object, as the CalendarEvent object was not designed to perform operations related to rental periods. NetSuite designed the CalendarEvent object to handle calendar events and tasks. Consequently, if you use the CalendarEvent object incorrectly and attempt to sync the car rental SuiteApp calendar with the existing NetSuite calendar, rental events may be mistakenly identified as calendar events.

1.2.2 Creating Custom Records

Create and use custom records to represent unique business objects that are not appropriately aligned with standard objects.

In the rental car example above, instead of using the CalendarEvent object, you should create a Rental Agreements custom record, along with any necessary scripts for handling operations specific to this record type.

Fundamentally, calendar events and car rental agreements are different business objects. Therefore, to avoid data collisions with other SuiteApps that might integrate with calendar events, you should create a new custom record type to represent the car rental agreement.

1.2.3 Verifying Support for Your Technical Framework

Using the correct technical framework supported by the SuiteCloud platform is vital to ensure the ongoing success of your SuiteApp. Before you start developing your SuiteApp, it is important to find out whether the framework you plan on using is fully supported. This framework can vary depending on the nature of your SuiteApp and the APIs it uses. For example, developers of SuiteTalk Web services integrations should ensure the version of their SOAP runtime platform (such as Java Apache Axis and .NET) is supported. Integrated SuiteApps that use SuiteTalk Web Services must also use a WSDL that is supported by NetSuite (a 3-year window). For example, for the version 2019.1 release, the oldest supported WSDL is version 2016.1. Any integrated SuiteApps that use version 2015.2 or older WSDLs will not meet the requirements set by the SDN quality program. This requirement ensures integrated SuiteApps will be fully functional and supported for production usage.

SuiteScripts, especially those with inline HTML UI components, are developed on supported combinations of OS and browsers. The details regarding the list of supported technical frameworks can be found in the NetSuite product documentation or SuiteAnswers by searching for the appropriate keywords such as “Apache Axis” or “supported browsers”.

1.2.4 Harmonizing Data Structures in your Integration

When building integrated SuiteApps, the pertinent elements in the external application’s data structure should match those in NetSuite as much as possible. This avoids any data truncation/integrity issues or unexpected validation errors when adding or synching data into NetSuite and vice versa. For example, the field size for customer names or the field type and format of the phone number from the external application should match with NetSuite. If structural mismatches cannot be avoided, adding some logic to properly truncate or format the information for the affected fields will help improve data integrity.

1.3 Developing for a Distributed Environment

NetSuite is a cloud application with servers hosted in multiple data centers. Each data center has its own domain. Once you develop your SuiteApp, it will be deployed on various customer accounts. Some accounts might be hosted on one data center, while others might be hosted on a different data center. For this reason, you must develop your application so that it can be used by any customer, regardless of which data center hosts the account. At the time of this writing, NetSuite currently has six datacenters in production with the following domains:

- system.netsuite.com
- system.na1.netsuite.com
- system.na2.netsuite.com
- system.na3.netsuite.com
- system.eu1.netsuite.com
- system.eu2.netsuite.com

More datacenters will become online as NetSuite expanses internationally. Therefore, this list is not exhaustive and may not be up to date. For SuiteApp developers, this is important to consider various scenarios and identify the cases in which a SuiteApp needs to identify the data center, and those cases in which it does not.

1.3.1 Native SuiteApps Built on the SuiteCloud Platform and Accessed from a Browser

In some cases, your SuiteApp may be built on the platform using native SuiteCloud technologies, such as SuiteBuilder, SuiteScript, and SuiteFlow. These SuiteApps are categorized as Native SuiteApps because they operate “natively” inside NetSuite without direct communication with any other external systems. In this scenario, when users access the application from a browser, they are automatically redirected to the appropriate URL. Therefore, if your SuiteApp has UI elements accessed via the browser (elements such as custom records, custom fields, custom forms, and dashboards), you do not need to do any special programming for these elements. Your application does not need to be aware of the specific data center that hosts your customer’s account.

1.3.2 Integrated SuiteApps that Use Web Services and/or RESTlets

When your Integrated SuiteApp interacts with NetSuite with SuiteTalk Web Services or RESTlets, you need to connect to the Web Services endpoint or RESTlet URL on the correct datacenter.

Beginning in version 2017.2, all customer production and sandbox accounts have Web Services endpoints and RESTlet domains that are unique to their account IDs. For example, an account with account ID 123456 will have the Web Services endpoint domain https://123456.suitetalk.api.netsuite.com, while its RESTlet URLs will have domain https://123456.restlets.api.netsuite.com. Integrations built prior to 2017.2 that rely on datacenter-specific domains will continue to work.

This feature allows customers to make these integration API calls without explicitly specifying the datacenter of their accounts as needed in the past – all they need to do is to include their account ID into the domain as shown in the example above. Since a customer’s custom integration code is not meant to be run on another other NetSuite accounts except their own, this is an acceptable practice because the domains will stay the same regardless of the datacenter of the account.

However, this practice does not work for ISVs of Integrated SuiteApps for a simple reason: this feature is not available for SDN accounts. For SDN accounts, their Web Services endpoints and RESTlets URLs are still dependent on the datacenter. For example, an SDN account with account ID TSTDRV123456 (all SDN accounts IDs have the “TSTDRV” prefix) will have the Web Services endpoint domain https://webservices.na1.netsuite.com, and its RESTlet URLs will have domain https://rest.na1.netsuite.com. Notice neither domains have the account ID in them.

Given this limitation, Integrated SuiteApps from SDN partners must dynamically determine an account’s integration domains. When built this way, these integrated SuiteApps will work during development and QA time when the SDN accounts are used (which only support datacenter-specific domains) and when deployed to customer accounts.

The easiest way to programmatically determine the correct domains to use is to use the getDataCenterUrls API. This is an API that does not require authentication, and only has the account ID as a parameter. Its response includes the system domain, Web Services domain, and RESTlet domain. The system domain will always be datacenter-specific because it is meant to indicate the datacenter; the latter two API domains are account ID-specific for customer production accounts and sandbox account, and datacenter-specific for SDN accounts.

The following steps should be followed for an integrated SuiteApp to work on all types of accounts (please refer to chapter 6 for details on sandbox accounts):

1) Use getDataCenterUrls to obtain the correct domains for a given account ID
2) Append the needed strings to the domains to construct the complete URLs for API calls; for Web Services, this means the endpoint URL including the version; for RESTlets, this means the script ID and deployment ID
3) Invoke the integration API calls using the URLs constructed from step 2
4) Optionally, store the domains to avoid making un-necessary getDataCenterUrls calls in the future; this information should be overridable by the users or administrators should the need arises

For reference, the following is a sample invocation of getDataCenterURLs for a SDN account with a TSTDRV account ID. Note the returned domains are datacenter specific:

```
<soapenv:Envelope
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:platformMsgs='urn:messages_2017_1.platform.webservices.netsuite.com'>
    <soapenv:Header/>
    <soapenv:Body>
        <getDataCenterUrls xsi:type='platformMsgs:GetDataCenterUrlsRequest'>
            <account xsi:type='xsd:string'>TSTDRV935086</account>
        </getDataCenterUrls>
    </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <platformMsgs:documentInfo
            xmlns:platformMsgs="urn:messages_2017_1.platform.webservices.netsuite.com">
```

```
<platformMsgs:nsId>WEBSERVICES__1107201717366790491983539797_523abcc1d7be0294b9</platformMsgs:nsId>
        </platformMsgs:documentInfo>
    </soapenv:Header>
    <soapenv:Body>
        <getDataCenterUrlsResponse
            xmlns="">
            <platformCore:getDataCenterUrlsResult
                xmlns:platformCore="urn:core_2017_1.platform.webservices.netsuite.com">
                <platformCore:status isSuccess="true"></platformCore:status>
                <platformCore:dataCenterUrls>
                    <platformCore:restDomain>https://rest.na1.netsuite.com</platformCore:restDomain>

<platformCore:webservicesDomain>https://webservices.na1.netsuite.com</platformCore:webservicesDomain>

<platformCore:systemDomain>https://system.na1.netsuite.com</platformCore:systemDomain>
                </platformCore:dataCenterUrls>
            </platformCore:getDataCenterUrlsResult>
        </getDataCenterUrlsResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

The following is a sample invocation of getDataCenterURLs of a customer production account. Note the returned API domains are account ID specific:

```
<soapenv:Envelope
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:platformMsgs='urn:messages_2017_1.platform.webservices.netsuite.com'>
    <soapenv:Header/>
    <soapenv:Body>
        <getDataCenterUrls xsi:type='platformMsgs:GetDataCenterUrlsRequest'>
            <account xsi:type='xsd:string'>1863787</account>
        </getDataCenterUrls>
    </soapenv:Body>
```

```
</soapenv:Envelope>


<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <platformMsgs:documentInfo
            xmlns:platformMsgs="urn:messages_2017_1.platform.webservices.netsuite.com">

<platformMsgs:nsId>WEBSERVICES__1107201717340394951251388693_c2a480a79e9e154429</platformMsgs:nsId>
        </platformMsgs:documentInfo>
    </soapenv:Header>
    <soapenv:Body>
        <getDataCenterUrlsResponse
            xmlns="">
            <platformCore:getDataCenterUrlsResult
                xmlns:platformCore="urn:core_2017_1.platform.webservices.netsuite.com">
                <platformCore:status isSuccess="true"></platformCore:status>
                <platformCore:dataCenterUrls>

<platformCore:restDomain>https://1863787.restlets.api.netsuite.com</platformCore:restDomain>

<platformCore:webservicesDomain>https://1863787.suitetalk.api.netsuite.com</platformCore:webservicesD
omain>

<platformCore:systemDomain>https://system.na3.netsuite.com</platformCore:systemDomain>
                </platformCore:dataCenterUrls>
            </platformCore:getDataCenterUrlsResult>
        </getDataCenterUrlsResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

1.4 Considerations when Using SuiteSignOn

To identify which domain to connect to when using SuiteSignOn (outbound single sign-on), NetSuite has added the **dc** parameter. The dc parameter provides a data center ID that can be used to map to the correct domain for NetSuite access. So, when developing your SuiteApp, do not hard-code the domain. Instead, use the dc parameter.

Note that NetSuite uses another variable, **env**, to identify whether a request is coming from a production, release preview, or sandbox NetSuite environment.

When developing your SuiteApp, your SuiteSignOn integration code should parse the parameters included in NetSuite's initial handshake outbound HTTP calls to the external application and populate the domains dynamically based on the value of the dc and env parameters.

The mappings of the dc parameter to the available production domains are as shown in the table examples below.

dc Parameter (Data Center ID)	env Parameter	NetSuite Domain	Web Services Domain	REST Domain
001	PRODUCTION	system.netsuite.com	webservices.netsuite.com	rest.netsuite.com
002	PRODUCTION	system.na1.netsuite.com	webservices.na1.netsuite.com	rest.na1.netsuite.com

For more details about these parameters, see "Mappings of dc and env URL Parameter Values" in the NetSuite Help Center:

https://system.netsuite.com/app/help/helpcenter.nl?fid=section_N3814593.html

1.5 Avoid Using External Suitelets

Use of External Suitelets (Suitelets with the "Available without Login" setting enabled) is a legacy method for NetSuite administrators to build ad hoc integration points or to build custom web site pages with dynamic content. However, commercially available SuiteApps must not include these external Suitelets. Instead of using external Suitelets to display public-facing information with dynamic content, consider using SuiteCommerce Advanced or SiteBuilder-based technologies such as SuiteScript Server Pages (SSP).

NetSuite has provided a free sample SuiteApp that demonstrates how to do this. It is named "SDN Sample – Replace External Suitelets" and can be downloaded using the SuiteBundler (bundle ID 40595).

For the use cases of ad hoc integration points, use either SuiteTalk web services or RESTlets.

1.6 OneWorld Considerations

NetSuite OneWorld is a version of NetSuite that supports multiple subsidiaries. It addresses complex multinational and multi-company needs by allowing companies to adjust for currency, taxation and legal compliance differences at the local level, with regional and global business consolidations roll-up.

One of the first questions developers new to the SuiteCloud platform are confronted with is whether they should build a common SuiteApp for both NetSuite OneWorld and regular NetSuite (informally known as "single instance"), or separate SuiteApps for each version. In general, it is a good idea to build a common SuiteApp that supports both. By employing this unified solution approach, ISVs can target a larger market of existing single instance customers and a rapidly growing market of enterprise customers using NetSuite OneWorld. With a common solution, there is usually less code to maintain, and therefore QA time and code fix delivery times are shortened. The following are technical factors to consider when designing a common solution for NetSuite OneWorld and single instance NetSuite.

1.6.1 The Subsidiary Field

In NetSuite OneWorld, the Subsidiary select field is a mandatory field for most standard records. Sometimes, the Subsidiary field acts as a filter to other select fields. For example, the Subsidiary field in the Sales Order record might filter the available selections in the Item select field in the Items sublist. Any SuiteScript or SuiteTalk web services API calls will need to follow the same filtering logic because they both mimic the behavior of the NetSuite UI.

1.6.2 Records Specific to NetSuite OneWorld

There are records that are unique to NetSuite OneWorld. An example is the Intercompany Journal Entry record, which is a specialized type of journal to record debits and credits to be posted to ledger accounts for transactions between two subsidiaries.

Depending on a SuiteApp's functional requirements, it may need to operate on these records that are unique to NetSuite OneWorld. If a SuiteApp is meant to be used for both NetSuite OneWorld and single instance NetSuite accounts, then it might need to have separate sets of I/O logic. For example: when operating on a OneWorld account, a SuiteApp might need to determine if a transaction is between different subsidiaries, and therefore needs to decide whether to programmatically create a Journal Entry or Intercompany Journal Entry record. The same SuiteApp, when operating on a single instance of NetSuite, does not need to make that decision and will only create a Journal Entry record for the same use case.

1.7 Creating extensions for SuiteCommerce Advanced

SuitCommerce Advanced provides with an Extensibility Layer Framework for developing extensions.

Partners should always make use of this framework when developing extensions that will be distributed to multiple customers.

These extensions should adhere to the Best Practices and standards established by this framework (https://developers.suitecommerce.com) and they should be conceived under the premise that multiple other extensions could also be deployed on the same account. Therefore, performance optimization and exception controls are crucial in this sort of suiteapps.

1.8 Connector SuiteApps Considerations

Most integration SuiteApps provide functional capabilities to the end users while using the SuiteCloud integration interfaces as a mean for data communication with NetSuite. There is also another breed of SuiteApps that act as a connector platform to NetSuite (and other systems), but otherwise do not provide functional capabilities. These are called connector SuiteApps.

The value of Connector SuiteApps is in providing an abstract layer above the underlying SuiteCloud integration interfaces (RESTlets and/or SuiteTalk web services). This abstract layer presents the connector users with an easy to use interface for consuming NetSuite REST or SOAP web services, thereby allowing them to spend more resource on business logic development instead of in API invocation.

The most common connector SuiteApp users are IT staff who build custom integrations, and SuiteApp developers who want to use a connector as part of the infrastructure of their integration applications.

Connector SuiteApps that rely on SuiteTalk web services can support multiple versions of the WSDL. It is important to support at least one WSDL that is less than 3 years old. WSDLs older than three years are considered unsupported by the NetSuite support team and the Engineering team. Even though these older WSDLs might continue to work, they must be updated regularly.

Partially supporting a WSDL is no longer allowed by the SDN quality initiative. For any given version of WSDL that a connector SuiteApp supports, it must support all of the operations within that version in order to provide users with the complete functionality of the version. In particular, this is includes all the synchronous and asynchronous operations, and all authentication methods. This is not to be confused with support for all the record types in the WSDL. Users of the Connector SuiteApp must be afforded all usage options, and not be denied any option available in the underlying technology, even if it is not a current requirement by the user. Note that this requirement is not applicable to connector SuiteApps that rely on the RESTlet interface.

SuiteApp developers that choose to use a connector SuiteApp as part of their infrastructure must ensure they choose one that adheres to all the SAFE security and performance guidelines. Failure to do so could result in the SuiteApp not meeting SDN quality verification process.

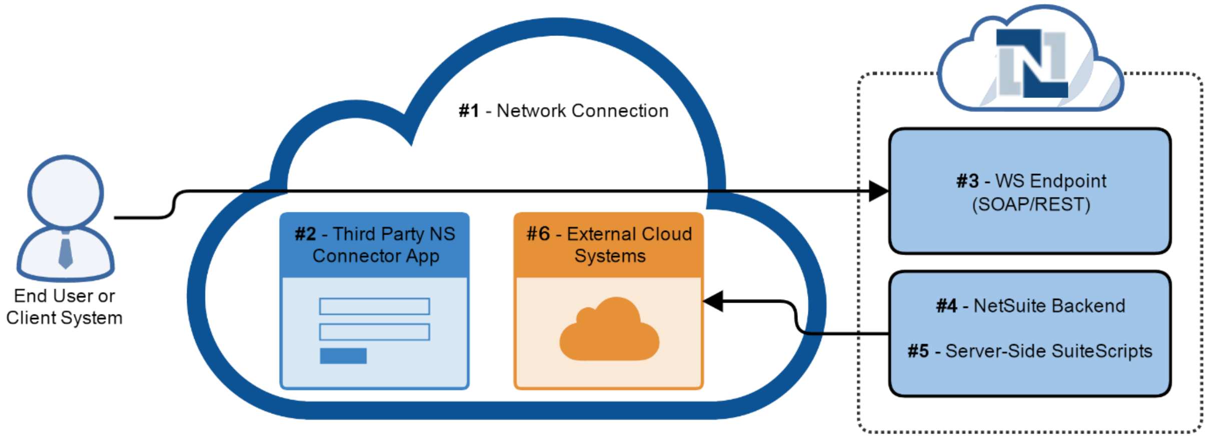
1.9 Building Fault Tolerant SuiteApps

Fault tolerance is the key attribute in enterprise applications that ensure they continue to operate correctly in the event of failures in one or multiple of its components. In these exceptional situations, the enterprise application may suffer degradation in its quality or performance, but the degradation should be proportional to the severity of the components' failure.

The complete coverage of system fault tolerance design is out of the scope of this document. However, this section will focus on the potential failure points and design considerations that are specific to SuiteApps.

Potential Failure Points

Depending on the type of SuiteApp (Integrated SuiteApp, Hybrid SuiteApp, Native SuiteApp), there can be potential failure points that are outside of NetSuite or resident inside NetSuite. The following diagram illustrated the most common failure points.


#1 - Network Connection
End User or Client System
#2 - Third Party NS Connector App
#6 - External Cloud Systems
#3 - WS Endpoint (SOAP/REST)
#4 - NetSuite Backend
#5 - Server-Side SuiteScripts

Potential failure points:

1 **Network connection** – the network component denoted can be any node between the source (end user or client system) and the target NetSuite account, including any connector SuiteApps that the client system may deploy. During network outages, integrated SuiteApps should log the failure and execute re-try logic. Users may also be notified, if applicable.

2 **NS Connector SuiteApps** – Connector SuiteApps provide an abstract layer above the underlying SuiteCloud integration interfaces (see section 1.7 for details). These SuiteApps can be used as part of the infrastructure for integration SuiteApps. Only those Connector SuiteApps that have obtained the Build for NetSuite validation badge are recommended. Please refer to the Connector SuiteApp vendors on the fault tolerance and robustness built into their products, and the best practices in deploying them.

3 **Web Services Endpoint** – depending on the integration interface an integrated SuiteApp uses, the web services endpoint can be either the SuiteTalk endpoint or a RESTlet's system-generated URL. Both types of endpoints can be potential failure points, and NetSuite will return correct error codes and messages when they are unavailable. While unplanned outages of these endpoints are rare, it is important to anticipate and handle these exceptions gracefully.

4 **NetSuite backend** – since planned and unplanned NetSuite server outages can impact integrated SuiteApps, these apps must be designed to handle system faults. Fault tolerance design usually includes logging, retry logic, and user/administrator notification if needed. However, retry logic must not assume the absence of NetSuite server response equates the complete failure of API calls. This is especially important for the API calls that insert or update data because those API calls are not inherently idempotent. Please refer to the section named "Using the External ID Field When Importing Critical Business Data" for details.

5 **Server Side SuiteScripts** – events such as NetSuite upgrades and unplanned system outages will impact all server side SuiteScripts. Scheduled scripts tend to be impacted most because they are usually longer running, and can execute during off hours which can coincide with planned outages. The two keys to minimizing data loss and data integrity issues caused by these system disruptions are the following:

a. **Examine the type argument** – upon entry of a scheduled script, the system-generated type argument must be examined to determine the circumstances that caused the invocation of the script. The type values that are pertinent to the topic of fault tolerance are "aborted" and skipped". The "aborted" value indicates the script was re-executed after an aborted execution caused by a system failure, and a recovery point was not set (see point in the next paragraph). The lack of a recovery point means the script may be repeating logic already executed. Therefore, the script's logic must take that into account and either proceed cautiously and/or notify administrators that manual intervention may be needed. On the other hand, the "skipped" value indicates the execution was postponed due to system downtime, but was otherwise not aborted prior. For the full list of valid values in the type argument, please refer to the scheduled script documentation in NetSuite Help Center.

b. **Setting recovery points** – to minimize issues caused by unplanned system failures, developers should preserve a scheduled script's state of execution by strategically making nlapiSetRecoveryPoint API calls. In the event of a system failure, a halted scheduled script will resume from the state where the recovery point was last set, thereby regaining execution continuity and context. Note that a script will consume 100 SuiteScript usage points when invoking nlapiSetRecoveryPoint, therefore, it is important to make this API call judiciously.

6 **External Cloud Systems** – SuiteApps that connect to external cloud systems typically do so by using the nlapiRequestURL API (or the credentials-focused API, nlapiRequestURLWithCredentials). If the external system in question is unavailable, or takes too long to respond, then the API will throw an error (SSS_CONNECTION_CLOSED or SSS_CONNECTION_TIME_OUT). If these errors are not handled gracefully, it can appear the SuiteApp itself has failed even though it is not a problem caused by the SuiteApp or the SuiteCloud platform. Therefore, it is important to anticipate these external system failures and gracefully handle them in the SuiteApp's code.

1.10 Considerations when Building with SuiteSuccess

1.10.1 The SuiteSuccess Initiative

SuiteSuccess is NetSuite's go to market model, now in its second year of production with over 500 live customers. By the end of 2018, it is expected that over 90% of customers will be sold and implemented using SuiteSuccess. It is made up of four major components:

- End-to-End customer engagement model which focuses on business results, processes and a stairway of growth.
- Verticalized solution with pre-configured leading practices delivered as SuiteApps with SKUs
- A more consultative approach to selling – leading customers with experience versus asking what they want the system to do
- A revamped delivery model which accelerates time to value – going live in around 100 days.

For partners, the impacts are many, but for the purposes of SAFE and BfN 2017.2, we will focus on the first technical effects of SuiteSuccess upon the SAFE guidelines. These can be found in Section 6 – Test your SuiteApps.

Part of the SuiteSuccess process is manifested in the new, standard, baseline Netsuite account(s). These baseline accounts are now provisioned with vertically specific customizations, preconfigured, or pre-installed, in the account. The method of customization is via the SuiteCloud platform, and ultimately the SuiteBundler. These bundles represent vertically specific customizations for the users in a vertical market. The result is a SuiteSuccess "master" account that can be reused as a vertical template.

In the context of the SuiteSuccess process, these vertical templates provide an ability in the sales cycle to demonstrate much more of the capabilities of the NetSuite solution than the standard, blank NetSuite account. This vertically specific account enables a sales rep or sales consultant to demonstrate the NetSuite solution in way that fits much better with the vertical requirements of a prospect, than the baseline NetSuite product.

Furthermore, during the implementation cycle, starting with a SuiteSuccess vertical template provides the implementation team with tremendous leap ahead in the cycle. Given the pre-configured tailoring of the baseline solution, the implementation can also deliver what was sold to the new customer initially.

What does all of this mean for the SuiteApp developer? First, the SuiteApp developer must be aware that the SuiteSuccess account just provisioned or now being implemented contains as many as 20 or more SuiteBundles.

The developer <u>must</u> be aware that the ability to customize these customizations has not fully evolved and should be avoided. Do not attempt to add, change, delete or make reference to objects contained in a pre-existing bundle residing in a SuiteSuccess account.

The initial set of principles documented in the SAFE guide for SuiteApp developers address some of these best practice avoidances. For this reason, the SAFE guide now mandates that SuiteApps should be developed in a non-SuiteSuccess account. This will ensure that no SuiteSuccess objects are added, changed, deleted nor referenced during development. This is intended to prevent the development of dependent objects and to avoid potential collisions or conflicts between custom objects. These relationships will be uncovered during testing only in a SuiteSuccess account. For further information, please refer to the guidance offered in Section 6 of this SAFE guide.

1.11 Working with NetSuite Technical Teams

As a “Select” tier ISV partner of the SuiteCloud Developer Network (SDN), you will be working with various teams in NetSuite frequently in the lifecycle of your SuiteApps. Some of these teams are technical in nature who will help you in the design, development and post-deployment support phases of your SuiteApps. This section focuses on explaining the role of these teams and how best to get the most out of your relationships with them.

The SDN Solutions Engineering Team

The Solutions Engineering team provides technical enablement support to SDN partners in their design and development of their SuiteApps. They provide architectural and API level guidance on the SuiteCloud and SuiteCommerce platform tools and APIs with SAFE principal adherence and Built for NetSuite validation as the end goal. When the need arises, they also bring NetSuite product management teams into technical discussions with partners, such as roadmap sessions and integration design sessions. This team also produces the self-help technical content for partners’ consumption, including technical documents, sample apps, and videos.

The SDN Solutions Engineering team is your main technical contact. They can guide you on whether your technical inquiries should be answered by existing self-help content, or the NetSuite technical support team, or whether dedicated design sessions are needed.

The SDN Quality Assurance Team

There are primarily two roles fulfilled by the SDN QA team. The first role is the administration of the BfN, or Built for NetSuite, program. This is the SuiteApp Verification process. SDN QA team members can also help guide SDN partners during the BfN review process from any point through to its completion. SDN QA team members receive email notifications for every BfN questionnaire that is submitted for review, which essentially initiates the process. If additional guidance is required before or after the review process, please send email to SDNQA@netsuite.com.

The second role is the offering of guidance and support for the QA testing of your SuiteApp. The SDN QA team is prepared to assist with the development of automated QA testing at any time. Again, if you have questions, or additional guidance is required, please send email to SDNQA@netsuite.com.

The NetSuite Product Management Teams

The role of the Product Management teams (PM) is to formulate the product feature roadmap for various NetSuite components, and prioritize and design those features for release. Some NetSuite products as used extensively by SDN partners (SuiteCloud, SuiteCommerce, ERP), therefore those PMs may work with this core group of their constituents, with SDN as the liaison. The most common way for PMs and SDN partners to meet is during the pre-release webinars (usually in the days leading up to every NetSuite release), and during the annual SuiteWorld user and partner conference.

Please let the SDN team know if you have product management related questions, they will determine the best course of action. Note that having direct one-on-one sessions between SDN partners and PMs are up to discretion of the SDN team.

The NetSuite Technical Support Team

The Technical Support team in NetSuite is skilled in all aspects of the product, and its members service both customers and partners. SDN Select tier partners may contact this team by opening support cases using the Advanced Partner Center (APC) portal.

The Technical Support team is tasked with providing answers to how-to questions on the NetSuite product (including SuiteCloud and SuiteCommerce platforms), troubleshooting product error messages, and opening defects and enhancement requests. Please note that aiding SDN partners in SuiteApp architecture and designs is out of the technical support team's scope – this is a task for the SDN Solutions Engineering team.

1.12 Multiple administrators for DEV/QA/Deployment accounts

When SDN partners are onboarded, they are provisioned with 3 separate SDN trailing accounts for development, QA, and deployment purposes (see Chapter 8.1). By default, a user with administrative permission is created for each of these accounts using the requestor's email address by which he/she initially becomes the sole administrator of the accounts. As a best practice, new users with administrative permission should be created as soon as possible to the newly provisioned accounts. Partners, in some cases, rely on the lone administrator access to maintain all their accounts without a contingency plan. In the event the employee leaves the company, this will lead to problems including the inability to: create another user with administrative permission; development/maintenance/update of the SuiteApps. If this occurred in a deployment account, this could have huge implications to the partner and their customers particularly during phased release. Since the SuiteApp can suddenly fail due to undetected problems during phased release testing and updates in the NetSuite platform, desired changes and necessary fixes cannot be pushed to the customer install base accordingly.

1.13 Further Reading

- *NetSuite Documentation Overview*
- *Understanding Accounting-Related Features*

- *Understanding General Ledger Impact of Transactions*
- *Inventory Management*
- *Understanding NetSuite OneWorld*
- *Understanding NetSuite Features in Web Services*

2 Manage SuiteScript Usage Unit Consumption

Managing the consumption of SuiteScript usage governance must be a critical part of your application design if high I/O volume is expected.

Server SuiteScript allows you to execute your own custom logic on NetSuite servers. However, executing custom logic in a cloud-based, multi-tenant environment introduces problems associated with poorly-written scripts executing in a single NetSuite account. Poorly-written scripts can jeopardize the performance of other NetSuite user accounts.

The SuiteScript usage governance described below minimizes problems caused by resource-consuming, poorly-written scripts. Most of the SuiteScript usage governance constraints are placed on I/O. Note, however, NetSuite also applies governance to other potentially resource-hungry and time-consuming tasks.

Important: When developing SuiteScripts it is important to use only supported and documented APIs. Supported APIs can be found in NetSuite Help. Additionally, DOM references in SuiteScripts are not supported.

2.1 Governance-related Issues

Many developers new to SuiteCloud development come from a background of working on platforms that have little or no limits on I/O and/or CPU access. A common hurdle that developers face when developing on the NetSuite platform is coming up with designs that are compatible with SuiteScript usage governance. This hurdle is especially problematic for applications that are I/O intensive, such as those that need to create or update hundreds of records in a synchronous manner.

Applications designed with little consideration for SuiteScript usage governance may run into SuiteScript usage limit errors later during QA cycles, or in production, when SuiteApps are deployed by customers. When a script exceeds governance limits, the system throws a usage limit error, and the script is halted by the platform. The script cannot be resumed at the point of failure. As a result, applications that rely on a series of I/O to be completed can potentially be terminated in mid-stream, thus threatening data integrity upon which all downstream tasks and transactions rely.

Example

An example of this failure is a third-party shipping application built on the item fulfillment transaction. During the before save (beforeSubmit) event on the server for the item fulfillment, the script needs to perform a series of searches on the items, bins, skids, and serial numbers, then create multiple custom records that represent the complex set of rules for shipping and inventory management. If such a script is not designed with SuiteScript usage governance in mind, the script can reach usage limits (depending on how much I/O is needed) while in the middle of processing and can be terminated by NetSuite. To resolve problems caused by scripts being terminated in mid-execution, data needs to be manually reconciled by examining the audit trail (through system notes) on the records.

2.2 Governance Considerations Early in the Design Phase

When discovered late in development or QA cycles, it is difficult to resolve governance-related design flaws, as they often require a significant redesign of the scripts. The remedy usually involves segregating a script into multiple scripts – some with lower usage limits to handle the presentation layer, some with higher usage limits to handle the bulk of I/O. In practice, this remedy means moving a significant amount of I/O logic that is already written from one type of script to another, as well as designing the mechanism to handle the safe transition of the data between the script types.

Because a redesigned script may contain logic executed asynchronously, the user workflow may also need to be redesigned. Due to the amount of redesign work involved, as well as the potential customer downtime introduced, you must consider SuiteScript usage governance early in the SuiteApp design phase.

Note: When reading and updating a single record, consider using nlapiLookupField and nlapiSubmitField instead of using the searching and record submitting APIs. Generally, these two APIs consume less SuiteScript usage and yield better performance than searching and record submitting APIs.

2.3 Script Designs for Managing Governance

The following are NetSuite-recommended script designs for handling higher I/O volume. These designs are implemented in user event scripts (beforeSubmit and/or afterSubmit) and the POST event of Suitelets.

- User Event Scripts and Suitelets
- Delegating I/O to Scheduled Scripts
- Using Parent-Child Relationships to Perform Mass Update/Create

Note: Scripting approaches listed above are not implemented in beforeLoad user event scripts or in the GET event on Suitelets. Typically, only data-read operations occur in beforeLoad and GET events.

Note: For additional code-level best practices and optimization techniques, see *Principle 3: Optimize Your SuiteApps to Conserve Shared Resources*.

2.3.1 User Event Scripts and Suitelets

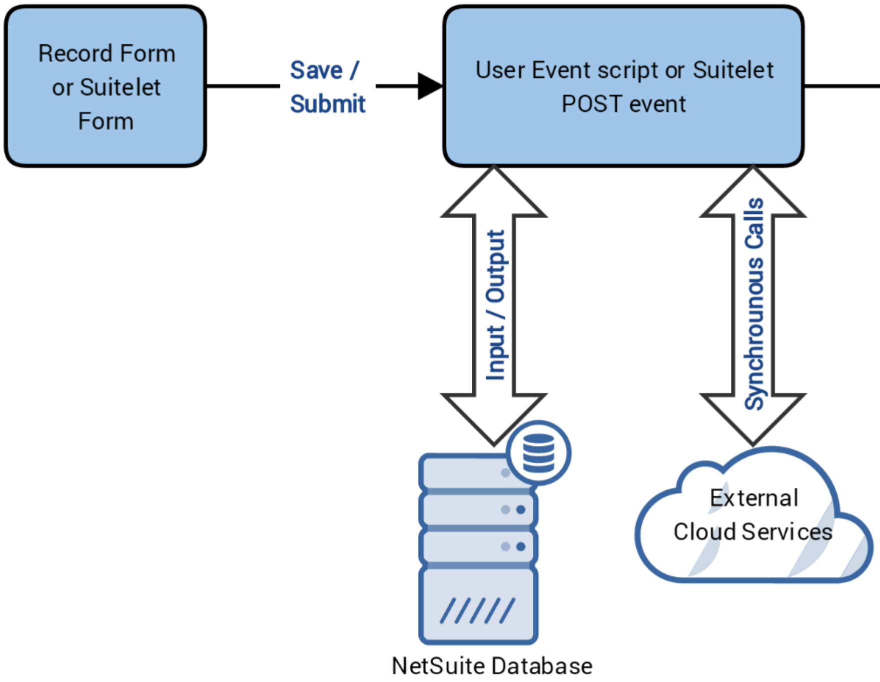
The first design approach for managing higher I/O volume is to use user event scripts and Suitelets. It is common to put all usage-consuming API calls, including I/O API calls and calls to external cloud-based services, directly in user event scripts and Suitelets. User event scripts and Suitelets are often invoked when users save a record or, in the case of Suitelets, when users click the Submit button. In most cases, writing your business logic in user event scripts and Suitelets works well, especially for use cases that do not require a large number of API calls.

When writing user event scripts and Suitelets, you should respect the original design intent for both of these script types. NetSuite intends for user event scripts and Suitelets to be short-lived and lightweight, in order to provide the best user experience. Logic placed in user event scripts and the POST blocks of Suitelets is executed between the time a page is submitted and the next page finishes loading. The more logic and I/O placed into these scripts, the longer users may have to wait for pages to finish loading, and their experience of using the SuiteApp may suffer. Therefore, developers should put logic in these scripts only when it is critical to align its execution with the saving of a record or the submitting of a Suitelet page. Otherwise the logic should be executed asynchronously (as described in the next section). A general rule of thumb is: if the logic takes longer than 5 seconds to run synchronously, then it should be moved to a scheduled script where it can be executed asynchronously.

When using either of these script types, the design goal should be to provide good user experience by writing code that executes the simplest logic possible, with a minimum amount of I/O.

Even within the allotted usage limit, as the number of API calls increase, the execution time for user event scripts and Suitelets increases. The increase in execution time can adversely affect user experience, making users wait for the NetSuite server while script logic is processing. Since user event scripts and Suitelets have the lowest allotted amount of usage units (1,000 units per script type), units can be easily exhausted, resulting in “usage limit exceeded” errors. When “usage limit exceeded” errors are thrown, NetSuite terminates the execution of the script. (See Governance-related Issues for additional information.)

Also note that the API to invoke external cloud-based services (nlapiRequestURL) executes synchronously. Consequently, invoking external services from within user event scripts and Suitelets can also threaten the goal of keeping these scripts short-lived.



Using User Event Scripts and Suitelets to Perform Simple Programmatic Tasks

Employing user event scripts and Suitelets to execute custom business logic has the following pros and cons:

Pros

- User event scripts and Suitelets are easy to implement.
- Synchronous invocation of these script types means logic is triggered immediately.

Cons

- Synchronous invocation means user experience may suffer if an excessive number of API calls is made.
- Lower allotted units for user event scripts and Suitelets means "usage limit exceeded" errors are thrown more often.
- User event scripts may not execute as expected under some circumstances. For example, user event scripts do not get executed in a nested manner.

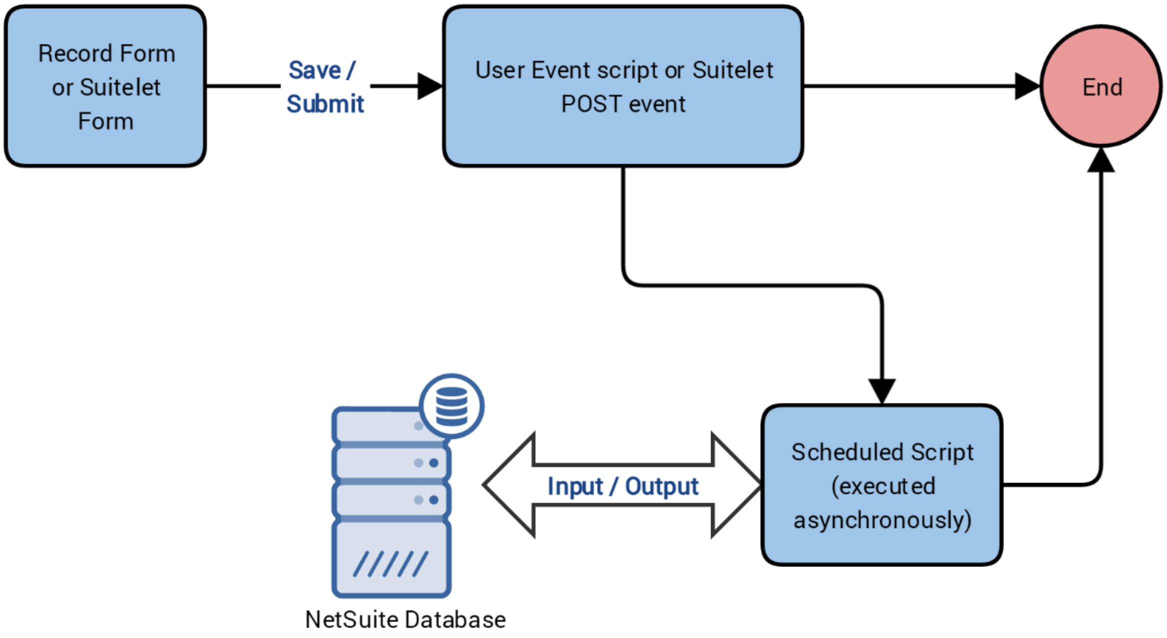
2.3.2 Delegating I/O to Scheduled Scripts

The second design approach leverages the high governance units allotted to NetSuite scheduled scripts. Scheduled scripts are allotted the highest amount of unit usage consumption (10,000 units per script). Therefore, scheduled scripts help address governance restrictions associated with user event scripts and Suitelets.

As a script developer, you can write user event scripts or Suitelets that delegate the "heavy lifting" of I/O API calls to a scheduled script. The caller script involves a scheduled script designed to handle the bulk of I/O API calls by using the nlapiScheduleScript API. As soon as the API call is made, the control returns back to the caller script, thereby improving user experience. The user does not need to wait for API logic to finish executing. The invoked scheduled script is placed in a queue and executed asynchronously. When the scheduled script is close to reaching the usage limit, the script may call nlapiScheduleScript to invoke itself again. To maintain continuity among scripts, you must write logic to pass the state (the unique configuration of data of your script at an arbitrary point) from the calling script to the called (asynchronous) scheduled script.

Alternatively, the state-passing logic can be completely bypassed if a script continuously checks its remaining usage, and uses nlapiYieldScript to reset the usage back to full and place itself at the end of the queue again. (For specific details on using nlapiScheduleScript and nlapiYieldScript, see *nlapiScheduleScript* and *nlapiYieldScript* in the NetSuite Help Center.)

Note: If your SuiteApp contains a high number of scheduled scripts and/or the scheduled scripts are invoked often, the standard scheduled script queue may not be able to process these scripts fast enough. You may consider using the Multiple Scheduled Script Queue feature, which is included in the SuiteCloud Plus add-on module. The SuiteCloud Plus module can be added to SDN partner test accounts free of charge, and is available to customers for a fee. Some customers may have the Multiple Scheduled Script Queue feature enabled while some may not. In order to support all customers, your SuiteApp can query for the number of scheduled script work queues available in a customer account, and optimize scheduled script management according to whether the customer has one queue or multiple queues. In addition, you can use SuiteScript searching APIs to search for scheduled script instances and programmatically determine which scheduled script deployments are available to be invoked, thus allowing your SuiteApp to effectively use the multiple scheduled script queue feature during runtime to enhance execution throughput.


Record Form or Suitelet Form
Save / Submit
User Event script or Suitelet POST event
End
Scheduled Script (executed asynchronously)
Input / Output
NetSuite Database

Incorporating Scheduled Scripts into Script Design that Requires High Volume I/O

Employing scheduled scripts to execute your custom business logic has the following pros and cons:

Pros

- The large number of allotted units makes scheduled scripts well suited for high volume I/O.
- A scheduled script can call itself again (or another scheduled script) to continue execution of very high volume I/O until the task is completed.
- Users have a better experience, as time consuming I/O logic is executed asynchronously.

Cons

- Using scheduled scripts is more complex than the design approach leveraging user event scripts and Suitelets (see User Event Scripts and Suitelets).
- Asynchronous execution may not be feasible for some use cases. Some examples are user-centric applications that require immediate feedback to be given to users.

2.3.3 Map Reduce vs Scheduled Scripts

As previously explained Scheduled Scripts can be a good solution when delegating the "heavy lifting" I/O API calls, however the platform offers a different approach when proposing a solution for speeding up the process of large amounts of data. This is the **map/reduce** script type (only available in SS2.0).

This type of script is best suited for situations where the data can be divided into small, independent parts. So, when the script is executed, a structured framework automatically creates enough jobs to process all of these parts unbeknownst to the user. Another advantage of map/reduce is that these jobs can work in parallel and the level of parallelism can be chosen by the user upon deployment.

Like a scheduled script, a map/reduce script can be invoked manually or on a predefined schedule. However, map/reduce scripts offer several advantages over scheduled scripts. One advantage is that, if a map/reduce job violates certain aspects of NetSuite governance, the map/reduce framework automatically causes the job to yield and its work to be rescheduled for later, without disruption to the script.

When to use map/reduce

As a rule of thumb, Map/reduce could be used for any scenario where it is needed to process multiple records and the logic can be separated into relatively lightweight segments. In contrast, map/reduce is not as well suited to situations where you want to enact a long, complex function for each part of your data set. A complex series of steps might be one that includes the loading and saving of multiple records.

Searching for invoices that meet certain criteria and apply a discount on each one, identify a set of files in the File Cabinet, modify them and send them to an external server and search for duplicated customer records and apply certain business rule to duplicates could be feasible use cases for this sort of scripts.

When to avoid map/reduce

Even when it is possible to set a Map/reduce script not to run processes in parallel mode during deployment, the key reason for choosing this type of scripts is in fact when multithreading is needed.

Therefore, in scenarios where one of the parallel processes depends upon the update of another, map/reduce should not be the best choice because the order each process will be triggered cannot be determined beforehand. Please refer to section 4.7 for concurrency related issues such as race conditions, and the methods used to address them.

Comparison

Scheduled Script	Map/Reduce
Processes run in a single thread	Processes run in multiple threads
Available in SS1.0 and SS2.0	Available in SS2.0
Governance: The memory limit for a scheduled script is 50 MB. Before this limit is reached the nlapiYieldScript() funcition must be called.	Governance: Hard limits on total persisted data: <=200MB Hard limits on functions invocations • map: 1,000 usage units (the same as mass update scripts) • reduce: 5,000 usage units • getInputData: 10,000 usage units • summarize: 10,000 usage units • Soft limits on map reduce jobs • 10000 units
Manual yield	Automatic yield

Sample use of Map/Reduce

One of the most common scenarios when doing integrations involve processing transactions modified in a certain period of time.

The following script searches for transactions modified the previous day. The map logic updates the memo field and in the map script transactions are sorted by type and also invoked in the reuse script.

The reuse script groups by transaction types and sends an email to the script owner supervisor summarizing each transaction.

```
define(['N/search', 'N/record','N/runtime','N/email'],function(search,record,runtime,email) {

    function getInputData() {

        var today = new Date();

        var dd = today.getDate()-1;

        var mm = today.getMonth()+1;

        var yyyy = today.getFullYear();

        var yesterday = mm.toString() + '/' + dd.toString() + '/' + yyyy.toString();


        var myColumns = [{name:'internalid'},{name:'trandate'},{name:'recordtype'}];

        var myFilters =
[{name:'trandate',operator:'on',values:[yesterday]},{name:'mainline',operator:'is',values:['true']}
];
```

```
    return search.create({
      title: 'Sample Search',
      type: search.Type.TRANSACTION,
      columns: myColumns,
      filters: myFilters
    });
}

function map(context) {
    var searchResult = JSON.parse(context.value);
    var tranId = searchResult.id;
    var tranType = searchResult.values.recordtype;

    executeMapLogic(tranId,tranType);

    context.write({
        key: tranType,
        value: tranId
    });
}

function executeMapLogic(tranId, tranType) {
  var trans = record.load({
                        type: tranType,
                        id: tranId,
                        isDynamic: true
                      });

  trans.setText({
          fieldId: 'memo',
          text: 'Processed by MapReduce Script',
          ignoreFieldChange : false
        });
  trans.save();
}

function reduce(context) {
    var tranType = context.key;
    var tranId = context.values;
    var today = new Date();
    var dd = today.getDate()-1;
    var mm = today.getMonth()+1;
    var yyyy = today.getFullYear();
    var yesterday = mm.toString() + '/' + dd.toString() + '/' + yyyy.toString();
    var eBody = tranType + " transactions for " + yesterday + " : " + JSON.stringify(tranId);
    var eTitle = 'Map Reduce result ' + tranType + " transactions for " + yesterday

    var emailTo, emailFrom;
    var myColumns = [{name:'owner'},{name:'supervisor',join:'user'}];
    var myFilters = [{name:'scriptid',operator:'is',values:runtime.getCurrentScript().id}];

    var mySearch = search.create({
                      type: search.Type.MAP_REDUCE_SCRIPT,
                                columns: myColumns,
                                filters: myFilters
                    });

    mySearch.run().each(function(result) {
              rec++;
                          emailTo = result.getValue({name:'supervisor',join:'user'});
                          emailFrom = result.getValue({name:'owner'});
                });

    email.send({
                     author:emailFrom,
           recipients: emailTo,
           subject: eTitle,
           body: eBody
                 });
}


function summarize(summary) {
  log.debug('start SUMMARIZE');
}

return {
    getInputData: getInputData,
    map: map,
    reduce: reduce,
    summarize: summarize
};
});
```

2.3.4 Using Parent-Child Relationships to Perform Mass Create/Update

User event scripts and Suitelets that create and/or update a large number of records often exceed the governance limits placed on these script types. This excess is caused by the usage-unit consuming APIs these scripts use to perform mass updates and mass creates. When invoked repeatedly, the APIs can accumulate to a sum that exceeds the amount of usage allowed. The user experience of these scripts may also suffer because users have to wait for the pages to load while the scripts perform data-write operations.

A common design to alleviate these governance and performance issues is to delegate the data-write operations to scheduled scripts, as discussed in Delegating I/O to Scheduled Scripts. Processes off-loaded to scheduled scripts are executed asynchronously. However, this approach may not be suitable for use cases in which users expect immediate feedback. An alternative design that enables these high-volume I/O processes to execute synchronously is using parent-child relationships.

Any standard or custom record can be configured to have parent-child relationships. When viewed in the browser, these parent-child relationships are displayed as custom child record sublists in the parent record.

Note: See *Custom Child Record Sublists* in the NetSuite Help Center for details on setting up parent-child record relationships and manipulating them programmatically using SuiteScript.

The child records in parent-child relationships can be mass created and updated programmatically, with a very low SuiteScript usage consumption and little adverse performance impact. This outcome is accomplished using Sublist APIs to add child records to a parent record. When the parent record is submitted, so are all of its child records. When the operation is finished, only the API calls spent on submitting the parent record have consumed SuiteScript usage units. All the child records are created or edited, essentially without consuming any SuiteScript usage units.

Parent-child relationships can also help mass create/update records that do not inherently have functional parent-child relationships. To establish a parent-child relationship, create a custom record type that is to be used as the parent record. This parent record's sole purpose is to be used as a means to allow the mass create/update of child records. The parent record serves no other functional purpose and, therefore, contains no real application data or business data. The child records are the only records that have a functional purpose and contain real data.

When the script is invoked, it will create a new parent record. All the child records to be created/updated are added to the parent record. Tthe parent record is saved, which also saves all changes made to the child records. When this process completes, the parent record serves no further purpose, and can be deleted in batch processes (implemented using scheduled scripts).

Because the parent records contain no real data, their deletions do not impact data integrity. A child record can have multiple parent records, therefore this design approach is compatible with those records that naturally have parent-child relationships.

The following sample code shows how to use a parent-child relationship to mass create 200 child records.

```
var parent = _record.create({
   type: 'customrecord_sdn_parent',
   isDynamic: true
});

var sublist = 'recmachcustrecord_sdn_child_parent';
parent.selectNewLine({ sublistId: sublist });

for(var i = 0;i < 10; i++)
{
   parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
   'custrecord_sdn_child_fld1', value: 'aa'});
   parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
   'custrecord_sdn_child_fld2', value: 'ab'});
   parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
   'custrecord_sdn_child_fld3', value: 'ac'});
   parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
   'custrecord_sdn_child_fld4', value: 'ad'});
   parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
   'custrecord_sdn_child_fld5', value: 'ae'});

   parent.commitLine({ sublistId: sublist });
}

parent.save();
```

The following sample code shows how to use a parent-child relationship to mass update 200 child records.

```
var newparent = _record.create({
    type: 'customrecord_sdn_parent',
    isDynamic: true
});

var parentid = newparent.save();

var parent = _record.load({
    type: 'customrecord_sdn_parent',
    id: parentid,
    isDynamic: true
});

var sublist = 'recmachcustrecord_sdn_child_parent';
parent.selectNewLine({ sublistId: sublist });

for(var i = 0;i < 10; i++)
{
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId: 'id', value: i + 1});
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
    'custrecord_sdn_child_fld1', value: 'xa'});
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
    'custrecord_sdn_child_fld2', value: 'xb'});
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
    'custrecord_sdn_child_fld3', value: 'xc'});
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
    'custrecord_sdn_child_fld4', value: 'xd'});
    parent.setCurrentSublistValue({ sublistId: sublist, fieldId:
    'custrecord_sdn_child_fld5', value: 'xe'});
    parent.commitLine({ sublistId: sublist });
}

parent.save();
```

Employing parent-child relationships to perform mass create/update has the following pros and cons:

Pros

- Using parent-child relationships allows a large amount of data-write operations within a script’s allotted usage limit.
- The very fast record create or update for child records improves user experience.
- Synchronous data-write is ideal for use cases that require immediate results to be shown to users.
- Parent-child relationships have a built-in transactional capability. When the parent record fails to save, all changes made in the child records will be rolled back.

Cons

- Using parent-child relationships does not address mass data-read requirements.
- Additional scheduled script(s) are required to discard the parent records that no longer serve any functional purposes after the child record data-write operations are completed.
- User event scripts deployed on child records are not invoked when child records are created or updated

using parent-child record relationships.

2.3.5 Transient Record Controller

Most of the SuiteScript data-reading logic is performed using **nlapiLookupField** and the various searching APIs. Since developers can use the SuiteBuilder customization tools to link multiple standard and custom records (in addition to the various foreign key references in the standard schema), there is frequently the need for SuiteScript to traverse these linked record "chains." Since data-read APIs also consume SuiteScript usage, traversing these record chains to obtain field values can lead a script exceeding its governance limits.

The searching and lookup APIs have established usage governance best practices and built-in capabilities that allow traversing to linked records. Please refer to the Help documentation for this information.

An example of **nlapiLookupField** providing values in a linked record is the use case of obtaining the supervisor for a given customer's sales rep. Here is how the code would look like:

```
var customerSalesRepSupervisor = nlapiLookupField(‘customer’, customer_id,
‘salesrep.supervior’);
```

However, if there is a need to traverse further down the chain of records, then more APIs need to be invoked. An example is to enhance the use case above by also including the sales rep's email and the supervisor's location. Here is the code with the additional API calls to satisfy the new requirements:

```
var customerSalesRepSupervisor = nlapiLookupField(‘customer’, customer_id,
‘salesrep.supervior’);
var customerSalesRepEmail = nlapiLookupField(‘customer’, customer_id, ‘salesrep.email’);
var supervisorLocation = nlapiLookupField('employee', customerSalesRepSupervisor,
'location');
```

Note that each line above will consume SuiteScript usage units. While the listed example does not consume a lot of SuiteScript usage, each additional customer record and its record chain to be read will cause the usage to scale up linearly. If there are a lot of record chains to traverse and/or the chains are "long" (meaning many records linked together using select fields), then the total consumption of the API calls will inevitably cause the script to exceed its governance limit.

A potential solution for reading data from many record chains is using a design pattern called the Transient Record Controller. This design pattern relies on two key elements to drastically cut down on SuiteScript usage and provide fast data-read capabilities: the Controller custom record and SuiteScript dynamic mode. This design pattern employs the SuiteScript dynamic mode to manipulate a custom record that mimics the linked record schema in server side memory.

Constructing the Controller Custom Record Type

The first step of implementing the Transient Record Controller pattern is to build the Controller custom record type. This custom record type must mimic the linked record schema by including all the records and fields to be traversed. The linkages are built using Sourcing and Filtering.

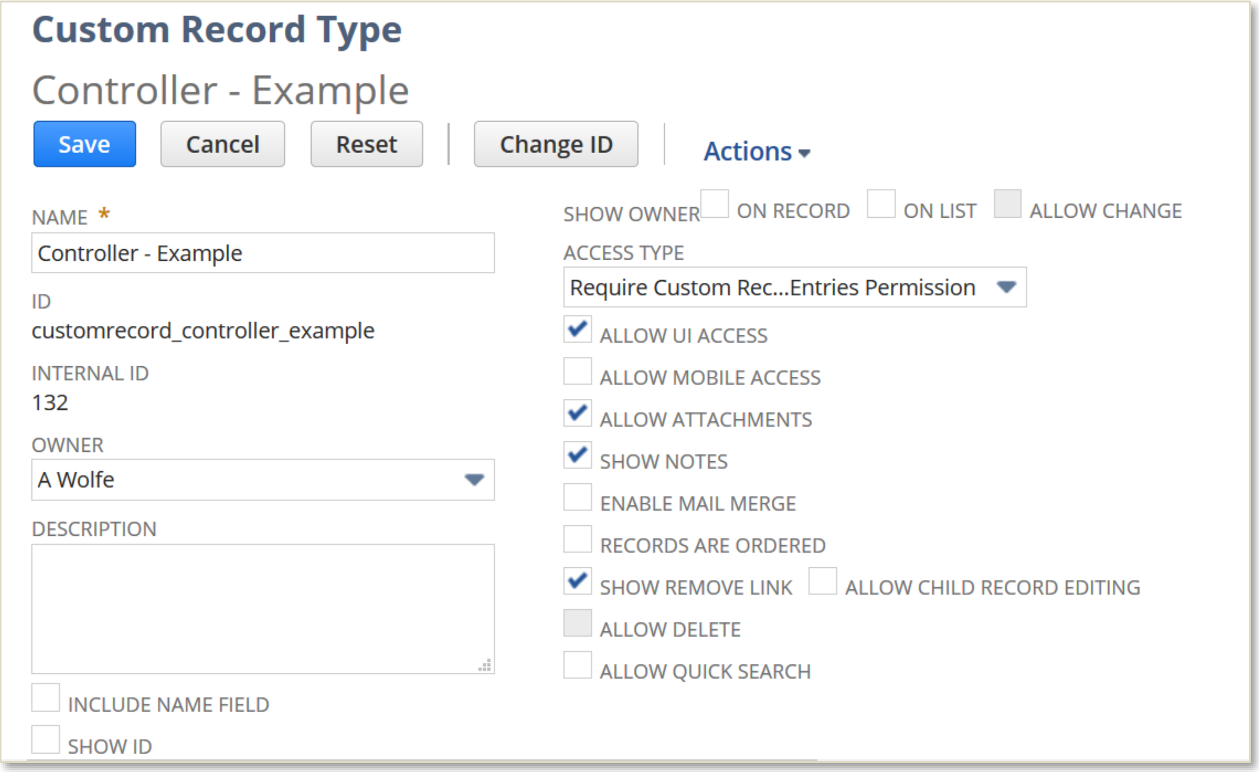
To begin building the Controller record for the use cases below, obtain the following:

1. The supervisor for a given customer’s sales rep

2. The email address for a given customer’s sales rep

3. The location of the supervisor for a given customer’s sales rep

First create the custom record type. Define a meaningful name and ID. Uncheck the Include Name Field setting. It is not necessary.



Custom Record Type

Controller - Example

NAME * Controller - Example

ID customrecord_controller_example

INTERNAL ID 132

OWNER A Wolfe

DESCRIPTION

□ INCLUDE NAME FIELD

□ SHOW ID

SHOW OWNER □ ON RECORD □ ON LIST □ ALLOW CHANGE

ACCESS TYPE Require Custom Rec...Entries Permission

☑ ALLOW UI ACCESS

□ ALLOW MOBILE ACCESS

☑ ALLOW ATTACHMENTS

☑ SHOW NOTES

□ ENABLE MAIL MERGE

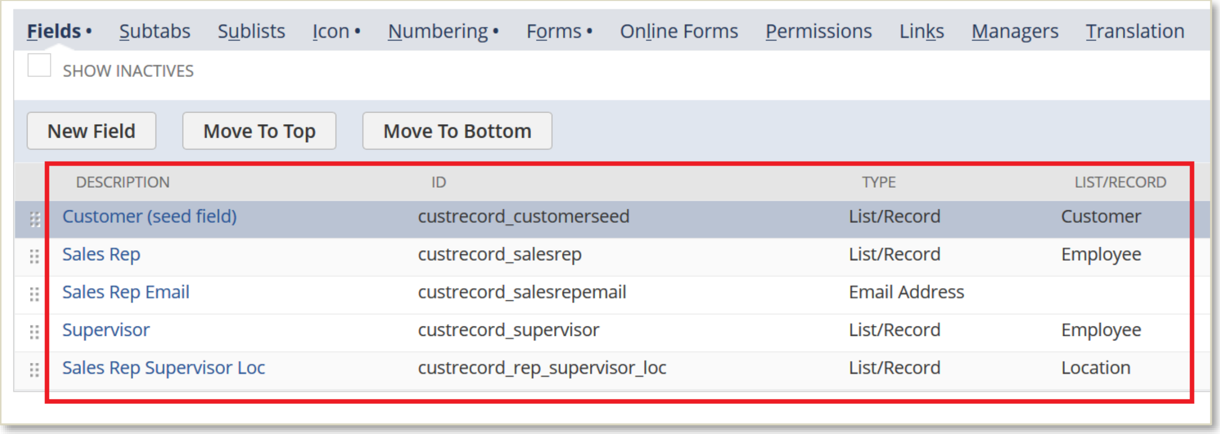
□ RECORDS ARE ORDERED

☑ SHOW REMOVE LINK □ ALLOW CHILD RECORD EDITING

□ ALLOW DELETE

□ ALLOW QUICK SEARCH

Each record and field in the record chain needs to be present in the Controller record. In use case #1 and #2, the customer, its sales rep, the sales rep’s email, and the sales rep’s supervisor are represented as custom fields in the Controller custom record. Lastly, in use case #3, the supervisor’s location is represented as another custom field. **It is important to faithfully represent the data type of those records and fields as they appear in the linked record chain schema**; this means they all must be List/Record fields that point to the correct lists in these use cases (Customer, Employee, and Location). Here is how they must be set up (note the meaningful IDs given to them):



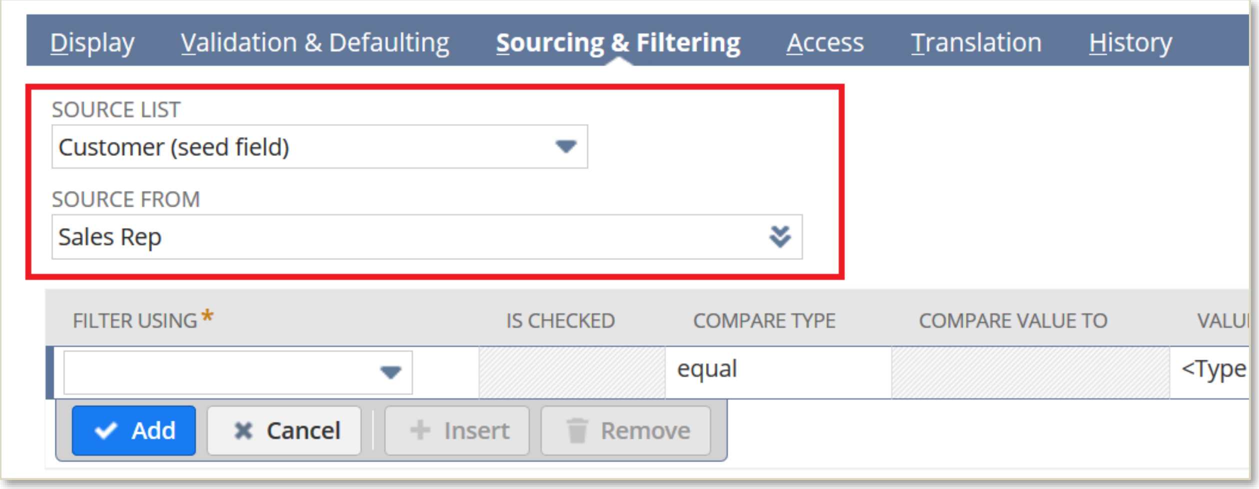
DESCRIPTION	ID	TYPE	LIST/RECORD
Customer (seed field)	custrecord_customerseed	List/Record	Customer
Sales Rep	custrecord_salesrep	List/Record	Employee
Sales Rep Email	custrecord_salesrepemail	Email Address	
Supervisor	custrecord_supervisor	List/Record	Employee
Sales Rep Supervisor Loc	custrecord_rep_supervisor_loc	List/Record	Location

The **Customer (seed field)** field represents a customer record, therefore, it is of type List/Record and points to the Customer list. The **Sales Rep** field represents the *customer.salesrep* field, which is an employee record. Therefore, it points to Employee list. The **Supervisor** field represents *customer.salesrep.supervisor*, which is an employee record. Therefore, it points to the Employee list. Note that the Customer field is called the "seed" field because it will be the first field to be set in server memory using a SuiteScript and it will drive the data propagation.

Once the custom fields are created, the next step is to set up their Sourcing and Filtering.

The seed field (**Customer field**) is always the first field to be populated either manually or via SuiteScript, therefore it does not need Sourcing to be set up.

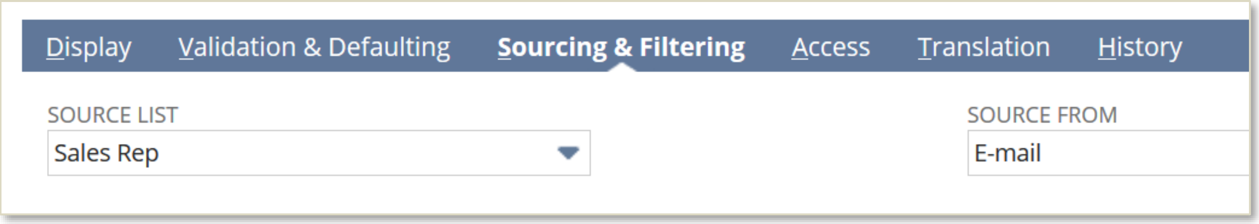
The next field is the **Sales Rep** field. Since it mimics *customer.salesrep*, its Source List will be **Customer (seed field)** and Source From will be **Sales Rep** as follows:



SOURCE LIST: Customer (seed field)

SOURCE FROM: Sales Rep

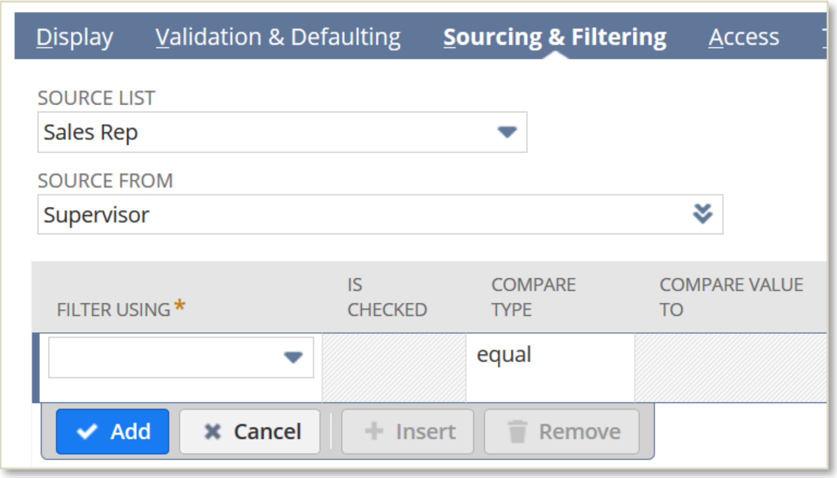
The next field is the Sales Rep Email field. Since it mimics *customer.salesrep.email*, its Source List will be **Sales Rep** and Source From will be **E-mail** as follows:



SOURCE LIST: Sales Rep

SOURCE FROM: E-mail

The next field is the **Supervisor** field. Since it mimics *customer.salesrep.supervisor*, its Source List will be **Sales Rep** and Source From will be **Supervisor** as follows:

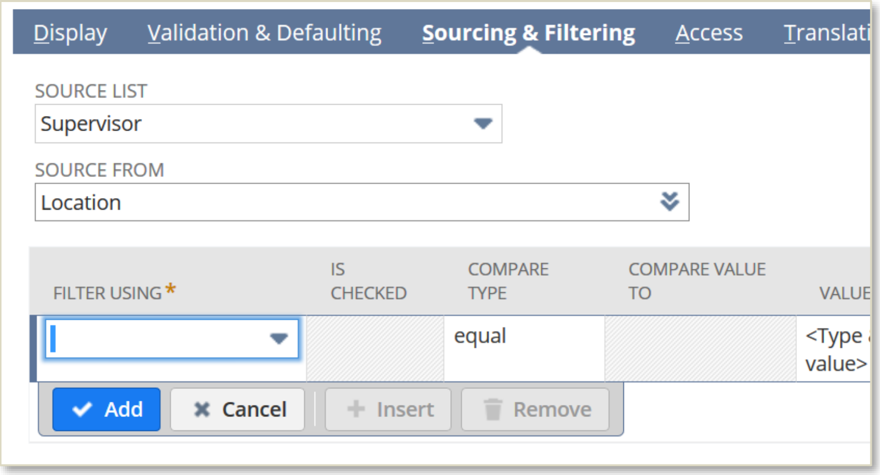


SOURCE LIST: Sales Rep

SOURCE FROM: Supervisor

FILTER USING *	IS CHECKED	COMPARE TYPE	COMPARE VALUE TO
		equal	

The last step is setting up the **Sales Rep Supervisor Loc** field. Since it mimics *customer.salesrep.supervisor.location*, its Source List will be **Supervisor** and Source From will be **Location** as follows:



SOURCE LIST: Supervisor

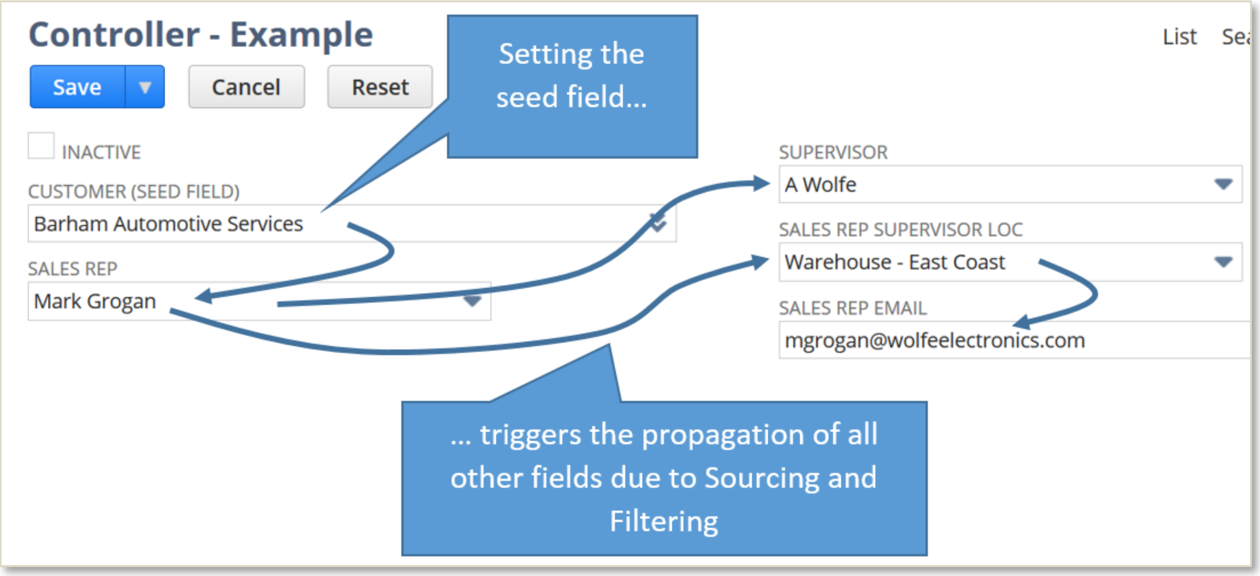
SOURCE FROM: Location

FILTER USING *	IS CHECKED	COMPARE TYPE	COMPARE VALUE TO	VALUE
		equal		<Type value>

Testing the Controller Custom Record

Since Controller record is meant to represent the linked record schema, it is important to test it fully before proceeding to use it in SuiteScript code.

Using the browser interface, first load the form to create a Controller record. Set the "Customer (Seed Field)" to point to one of the customer records. Since all the other fields in the Controller record's definition have Sourcing and Filtering set up, the action of setting the seed field will trigger the propagation of all their values.


Controller - Example
Save
Cancel
Reset
List
Se
Setting the seed field...
... triggers the propagation of all other fields due to Sourcing and Filtering

□ INACTIVE

CUSTOMER (SEED FIELD): Barham Automotive Services

SALES REP: Mark Grogan

SUPERVISOR: A Wolfe

SALES REP SUPERVISOR LOC: Warehouse - East Coast

SALES REP EMAIL: mgrogan@wolfeelectronics.com

Continue populating the seed field with different values to watch the propagation of other fields in real time. After you are satisfied that the Controller record faithfully mimics the linked record schema, **DO NOT** save the Controller record. It is important to remember that the Controller record is transient in nature. It should not be persisted. Its sole purpose is to quickly and easily transverse linked record chains, not to store data.

By viewing the Controller record in action via the browser a good indication if provided of how it will be processed in SuiteScript. The usage of NetSuite server side memory during execution is discussed next.

Using the Controller Record in SuiteScript

The second key part of the Transient Record Controller method is the SuiteScript dynamic mode. When working with records using SuiteScript, Dynamic mode can be used optionally. When used, the SuiteScript engine provides a virtual browser in memory that allows a script to manipulate a record as if it was in the browser with full field value sourcing.

In the previous section where the Controller record was tested, the seed field was set repeatedly in the browser to propagate the values in other fields, thereby effectively traversing multiple linked record chains and their fields. In the end, the Controller record was not saved.

The Controller record can be used in the exact same way by a SuiteScript in the server side memory to walk through multiple linked records and repeatedly setting the seed field and reading the sourced fields programmatically. The read values can be stored in memory (such as arrays) for later use. At the end the Controller record is discarded without being saved.

As reminder, here are the use cases for this example:

1. The supervisor for a given customer's sales rep

2. The email address for a given customer's sales rep

3. The location of the supervisor for a given customer's sales rep

The following is the SuiteScript code snippet that uses a Controller record in memory to obtain data for all three use cases listed above:

```
//Setting up the arrays to store the data in memory
var customerArray = ["469", "834", "1088"];
var customerSalesRep = new Array(customerArray.length);
var customerSalesRepEmail = new Array(customerArray.length);
var customerSalesRepSupervisor = new Array(customerArray.length);
var customerSalesRepSupervisorLoc = new Array(customerArray.length);

//instantiating a controller record
var controller = nlapiCreateRecord('customrecord_controller_example', {recordmode: 
'dynamic'});

//loop through reach linked record chain
for (var i=0; i < customerArray.length; i++)
{
     //setting the seed field
     controller.setFieldValue('custrecord_customerseed', customerArray[i]);

     //obtaining the propagated values from all the other fields
     customerSalesRep[i] = controller.getFieldValue('custrecord_salesrep');
     customerSalesRepEmail[i] = controller.getFieldValue('custrecord_salesrepemail');
     customerSalesRepSupervisor[i] = controller.getFieldValue('custrecord_supervisor');
     customerSalesRepSupervisorLoc[i] = 
controller.getFieldValue('custrecord_rep_supervisor_loc');
}
```

Since only one Controller record was created in memory to traverse multiple linked record chains and is discarded at the end, the script consumes a negligible amount of SuiteScript usage units and has almost no risk of exceeding the governance limits. In fact, only the **nlapiCreateRecord** API that created the Controller record in memory consumes SuiteScript usage. The Controller record created in memory was not saved because **nlapiSubmitRecord** was never called. The Controller record was discarded when the script finished execution. Therefore, no usage units were consumed.

The code example above traversed three linked record chains only. It may not offer much SuiteScript usage advantage over using the regular SuiteScript querying and lookup API calls. However, many more of these record chains can be traversed using the same code, and will consume the same amount of SuiteScript usage because **nlapiCreateRecord** was only invoked once. When many linked record chains are traversed using the regular querying and lookup APIs, the SuiteScript usage will increase linearly and eventually cause the script to exceed the governance limits. On the other hand, when traversing the same set of record chains using the Transient Record Controller pattern, it benefits from the economy of scale because it consumes the same amount of SuiteScript usage regardless of the number of record chains. Using the Transient Record Controller method is especially economical when the use case calls for a lot of linked data to be read.

If the linked record schema needs to be expanded to cover more fields and/or more linked records, then the Controller custom record type must be enhanced to include the new additions. The SuiteScript code also needs to be enhanced to take advantage of the extended schema and the bigger Controller record.

Sourcing and Filtering generally work very fast in memory. SuiteScript dynamic mode inherits this strength and benefits performance-wise. The **nlapiCreateRecord** call tends to be slower than other querying and lookup APIs – it takes approximately 100ms to create a custom record in memory. The rest of the **getFieldValue()** calls in the sample code are fast. When enough of the linked record chains need to be traversed, or when enough fields need to be read, then the elapsed time of all the **getFieldValue()** calls will offset the performance overhead of **nlapiCreateRecord** – the Transient Record Controller pattern also benefits from the economy of scale in relations to performance.

Pros

- Highly economical with SuiteScript usage for use cases that require a lot of linked records to be traversed
- Can support very long record chains and/or high number of record chains with relative ease
- Supports standard and custom records/fields
- Its reliance on Sourcing and Filtering provides very good data retrieval performance
- The code to utilize a Controller record is relatively simple

Cons

- Considerations for using this pattern needs to be factored into the **design stage** of the data schema because it is hard to retrofit it to use this method for data retrieval
- The large overhead of creating a Controller record in memory makes it unattractive for use cases with small amounts of linked records to be traversed
- The Controller custom record definition can be tricky to implement correctly

2.3.6 SuiteCloud IDE

Developers are encouraged to use the SuiteCloud IDE to write SuiteScripts. The SuiteCloud IDE's code auto-suggest capability eases development efforts and encourages the use of SuiteScript coding best practices.

For further information on the SuiteCloud IDE tool, please see *SuiteCloud IDE Overview* in the NetSuite Help Center.

You may also choose to search for answers to your specific questions on the SuiteCloud IDE in SuiteAnswers.

2.4 Further Reading

- *API Governance*
- *Script Usage Unit Limits*
- *nlobjGetContext.getRemainingUsage()*
- *Client Script Metering*
- *Workflow Governance*

;

3 Optimize Your SuiteApps to Conserve Shared Resources

Applications running on the multi-tenant SuiteCloud platform must be optimized to conserve shared resources such as CPU time. The coding and performance optimizations described in the sections below should be followed whenever possible.

- SuiteScript Performance Optimizations
- SuiteTalk Performance Optimizations
- Search Optimizations

3.1 SuiteScript Performance Optimizations

If your SuiteApp includes SuiteScript, be sure to follow the coding optimizations below. If you have already built a SuiteApp that does not follow these guidelines, re-work your code so that it adheres to the following:

- Do Not Re-Save Records in the afterSubmit Event
- Avoid Loading the Record for Each Search Result
- Do Not Put Heavy Lifting I/O Tasks into User Event Scripts
- Use Faster Search Operators
- Use Advanced Searches

3.1.1 Do Not Re-Save Records in the afterSubmit Event

When creating or editing a record, do not use nlapiSubmitRecord or nlapiSubmitField in the afterSubmit event to save the record again. Doing so writes the record twice (a CPU and I/O intensive task), and nearly doubles the time consumed for editing/creating the record.

Instead, set field values in the beforeSubmit event, or use client SuiteScripts where applicable. Either approach is more economical and will make the I/O changes only once.

The following sample code incorrectly resaves a record in an afterSubmit user event script:

```
var _afterSubmit = function(context)
{
   var record = _record.load({
      type: context.newRecord.type,
      id: context.newRecord.id
   });

   record.setValue({
      fieldId: "memo",
      value: "_afterSubmit"
   });

   record.save();

};
```

The following sample code shows a more efficient design that places the update logic in the beforeSubmit event:

```
var _beforeSubmit = function(context)
{
   var record = context.newRecord;

   record.setValue({
     fieldId: "memo",
     value: "_beforeSubmit"
   });
};
```

3.1.2 Avoid Loading the Record for Each Search Result

After performing a search, do not use nlapiLoadRecord or nlapiLookupField to load the record in memory to retrieve data. This approach is an inefficient way to retrieve data from search results because it requires unnecessary I/O API calls. This approach also has a negative impact on SuiteScript usage governance.

Instead, prior to running a search, add the desired columns to the search, to avoid loading the entire record for every matching result.

The following sample code shows the inefficient loading of a record from a search result to obtain data, in this case to obtain the entity ID of customers with a balance between 0 and 1000:

```
var myCustomerSearch = search.create({
   type : 'customer',
   filters : [ 'balance', 'between', 0, 1000 ]
});

myCustomerSearch.run().each(function(result)
{
   var recId = result.id;

   var customer = record.load({
     type: 'customer',
     id : recId
   });

   var customerId = customer.getValue('entityid');

   return true;
});
```

The following sample code shows a more efficient way to accomplish the same thing:

```
var myCustomerSearch = search.create({
   type : 'customer',
   columns : ['entityid'],
   filters : [ 'balance', 'between', 0, 1000 ]
});

myCustomerSearch.run().each(function(result)
{
   var entityId = result.getValue('entityid');

   return true;
});
```

3.1.3 Do Not Put Heavy Lifting I/O Tasks into User Event Scripts

User event scripts are not ideal for performing a large amount of I/O. The same is also true for the POST blocks in Suitelets. User event scripts and Suitelet POST blocks have a direct impact on user experience, as pages must wait for the scripts' logic to complete. Burdening these scripts with large amounts of I/O can make NetSuite pages or your SuiteApp's Suitelet pages slower, and may cause the script to run into SuiteScript usage governance limits.

Instead, delegate the heavy lifting I/O to scheduled scripts. (See Delegating I/O to Scheduled Scripts for more information. Also see User Event Scripts and Suitelets.)

3.1.4 Use Faster Search Operators

When possible, use search operators that define ranges for numeric columns (such as *between* and *within*), and are specific for text columns (such as *startswith*).

Even though you can access NetSuite data through SuiteScript and web services APIs, it is still an RDBMS that stores the data; therefore, NetSuite search APIs should be thought of as SQL. The *contains* operator, formulas, or existence search can be problematic for indexes, while keyword-based searches can be very fast. (See Search Optimizations for more information on designing more efficient searches.)

3.1.5 Use Advanced Searches

Web services advanced search APIs offer numerous performance and coding advantages. They are recommended for all use cases except the most trivial and lightweight ad-hoc searching.

Advanced searches support returning specific columns (instead of returning the entire record as basic searches do); therefore, advanced searches offer a significant performance improvement because the NetSuite server does not need to generate such a large result set.

Additionally, because advanced searches support referencing saved searches (presumably shipped in the SuiteApp), you may reference a saved search, rather than build a search from scratch. Additional filters and columns may also be added to an advanced search that references an existing saved search.

Note: For a more complete discussion of the benefits of saved searches, see Search Optimizations.

3.2 SuiteTalk Performance Optimizations

If your SuiteApp or solution uses web services, be sure to follow the coding optimizations below. If you have already built a SuiteApp that does not follow these guidelines, you should rework your code so that it adheres to the following:

- Avoid Unnecessary Authentication Operations

- Use Asynchronous Web Services for High Volume I/O
- Use Advanced Searches

Note: The section *Use Asynchronous Web Services for High Volume I/O* also includes a discussion about knowing when to use asynchronous over synchronous web services.

3.2.1 Avoid Unnecessary Authentication Operations

Successive non-login API calls that are within a session's expiration timeout limit should continue to use the existing session. Avoid extra authentication operations (the login and ssoLogin operations) if a session is still valid and usable. The authentication operations are some of the more expensive ones from a performance perspective, therefore, a web services application should use an existing session if one is available.

Rather than use the login operation, you should embed the Passport object in the I/O operation's SOAP header. This eliminates the need to manage a session, and is a faster alternative to the login operation. This authentication method is called Request Level Credentials authentication. When using Request Level Credentials authentication, the integration application must be ready to handle the concurrent operation exceptions that can be thrown. Simple "retry" logic is sufficient to handle these exceptions. Note that this is not applicable if you are using the token-based ssoLogin authentication API call.

Note: For more information, see *Authentication Using Request Level Credentials* in the NetSuite Help Center.

3.2.2 Use Asynchronous Web Services for High Volume I/O

Certain web services use cases are excellent candidates for using asynchronous web services. Two such candidates are one-time historical data imports and nightly data synchronizations.

There are two main advantages to using asynchronous web services. On the client side, after an application sends the SOAP request, the application is free to immediately sign off and does not need to wait for a response from NetSuite. On the NetSuite server side, the immediate load on the server is reduced, as there is no real-time demand to perform web services I/O. Overall, asynchronous web services (where applicable) is a better approach for multi-tenant cloud services such as NetSuite.

Note: Depending on the amount of data involved, some use cases such as historical data import can be done faster and easier using CSV import.

Synchronous and Asynchronous Web Services - A Closer Look

When developers experiment with SuiteTalk web services, they tend to start with synchronous operations because they are easy to use. However, it is important to know that the SuiteTalk interface also supports asynchronous mode in most of the CRUD (create, read, update, delete) operations. As a developer, you need to decide whether to use synchronous or asynchronous operations for your designs. This decision is generally based on your specific use case and the volume of data involved.

Why You Should Use Asynchronous Web Services

Depending on the time of the day and the volume of data a SuiteTalk application handles, synchronous web services can be more taxing on NetSuite servers than at other times. The primary reason for the implementation of SuiteTalk governance is to better manage the load on NetSuite servers placed by SuiteTalk applications during peak business hours.

The rationale for asynchronous web services is to provide integration applications with an interface that does not require immediate processing response from NetSuite servers. This approach frees up the servers from performing web services I/O in real time, and allows integrated applications the ability to postpone data processing until a time when there is less overall server CPU demand.

Depending on your data volume and use case, choosing to use asynchronous web services in your SuiteTalk application not only better utilizes NetSuite server resources, it may also allow your application to handle larger amounts of data permitted by SuiteTalk governance.

The following use cases are good examples of when to use asynchronous web services.

- Mass import – High data volume tasks such as mass import of historical data do not require quick response for improved user experience, and their data volume makes them demanding on NetSuite servers if done in a synchronous manner. Therefore, these tasks are best implemented with asynchronous web service operations.
- Nightly/batch data synchronization – External enterprise systems typically need to synchronize data with the system of record (NetSuite). These data sync tasks are best implemented with asynchronous web services because they are usually batch operations that do not require user input.

Note: For more information, see *SuiteTalk Governance* in the NetSuite Help Center.

When to Use Synchronous Web Services

The following use cases are good examples of when to use synchronous web services.

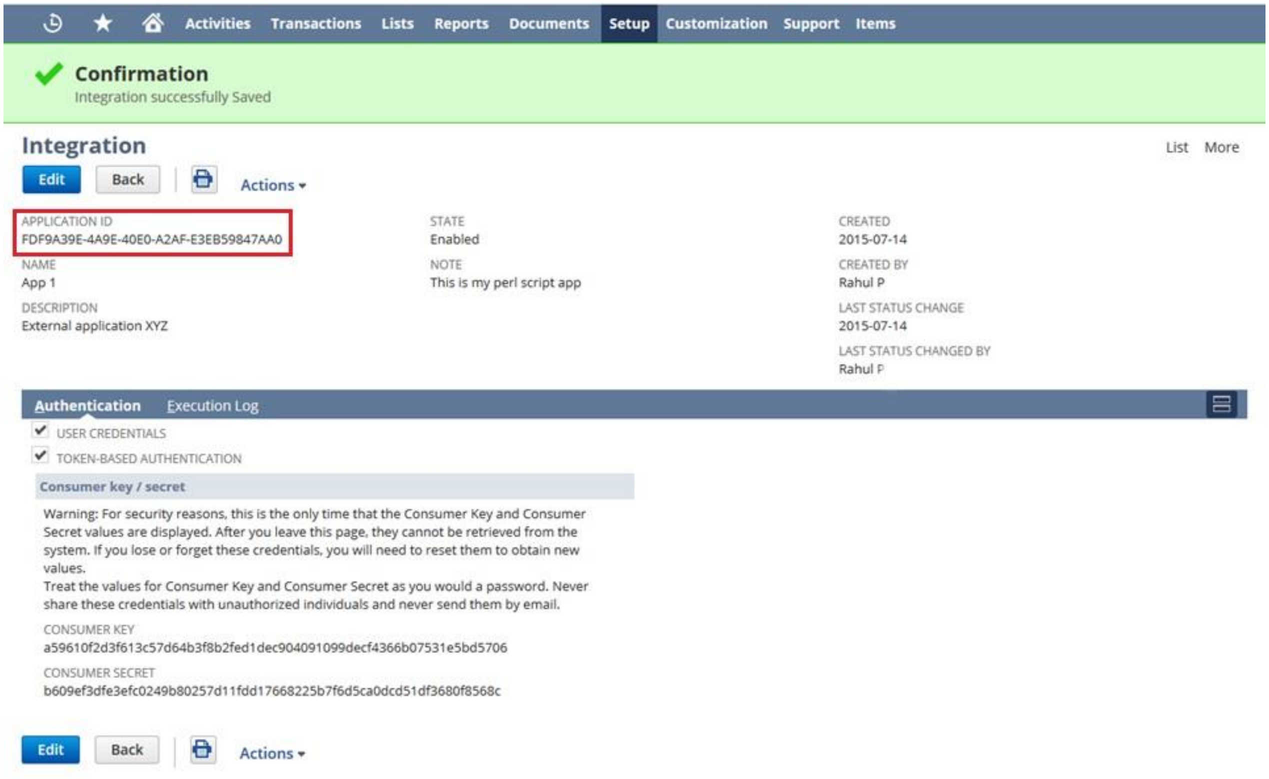
- Learning and experimenting – When learning the basic CRUD operations supported by SuiteTalk and the data schema of NetSuite records in SOAP form, it is best to use synchronous web services. The demand on NetSuite servers is generally low, and the real-time response promotes faster learning.
- Specific synchronous-only operations – There are some operations that are supported by synchronous mode only. Some examples are getSelectValue, attach, and getCustomizationId.
- Quick response – When users expect a quick response from a SuiteTalk application, synchronous web services should be used. From a usability perspective, it is likely impossible for these applications to use asynchronous operations, as users will have to wait for delayed responses from NetSuite servers.
- Low data volume – Applications that send and receive small amounts of data per web service API call, and/or those that make infrequent API calls, should also use synchronous web services. The low data volume has a lower impact on server resources, and the simpler code required for synchronous operations is easier to maintain and troubleshoot.

Note: The RESTlet interface, a form of server-side SuiteScript available since NetSuite Version 2011 Release 2, may also be used for use cases 3 and 4 above. RESTlets are especially useful when you want to ship an integration application that is small in memory footprint.

3.2.3 Identification of Web Services Applications in NetSuite

Every SuiteApp that utilizes SuiteTalk to communicate from external servers to a NetSuite account must preface each transmission with a standard SOAP header that includes an Application ID. SuiteApps connecting to endpoints 2015.1 and older must insert Application ID in the ApplicationInfo tag to identify the SuiteApp in order to obtain Built for NetSuite status. The purpose of the Application ID is to identify the SuiteApp and to enable control and monitoring of the connections of specific SuiteApps on the NetSuite server side.

Prior to SuiteTalk version 2015.2, these IDs were assigned by the NetSuite Operations team at the time of the introduction of the SuiteApp to NetSuite SDN team. Version 2015.2 introduced a new Integration record (Set Up → Integration → Manage Integrations). This new feature provides many new capabilities, including system-generated Application IDs without the involvement from the NetSuite Operations team, therefore eliminating the turnaround time needed by the process. The following is a screenshot of an Integration record with its system-generated Application ID:



✓ **Confirmation**
Integration successfully Saved

Integration

APPLICATION ID
FDF9A39E-4A9E-40E0-A2AF-E3EB59847AA0

NAME
App 1

DESCRIPTION
External application XYZ

STATE
Enabled

NOTE
This is my perl script app

CREATED
2015-07-14

CREATED BY
Rahul P

LAST STATUS CHANGE
2015-07-14

LAST STATUS CHANGED BY
Rahul P

☑ USER CREDENTIALS
☑ TOKEN-BASED AUTHENTICATION

Consumer key / secret

Warning: For security reasons, this is the only time that the Consumer Key and Consumer Secret values are displayed. After you leave this page, they cannot be retrieved from the system. If you lose or forget these credentials, you will need to reset them to obtain new values.
Treat the values for Consumer Key and Consumer Secret as you would a password. Never share these credentials with unauthorized individuals and never send them by email.

CONSUMER KEY
a59610f2d3f613c57d64b3f8b2fed1dec904091099decf4366b07531e5bd5706

CONSUMER SECRET
b609ef3dfe3efc0249b80257d11fdd17668225b7f6d5ca0dcd51df3680f8568c

The following presents sample SOAP coding that contains an Application ID in its header (SOAP body omitted for brevity):

```
<?xml version="1.0" encoding="UTF-8"?>

   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <soapenv:Header>

         <ns1:passport soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0" xmlns:ns1="urn:messages_2015_2.platform.webservices.netsuite.com">

            <ns2:email
xmlns:ns2="urn:core_2015_2.platform.webservices.netsuite.com">nlbuild@netsuite.com</ns2:email>

            <ns3:password
xmlns:ns3="urn:core_2015_2.platform.webservices.netsuite.com">********</ns3:password>

            <ns4:account
xmlns:ns4="urn:core_2015_2.platform.webservices.netsuite.com">3604360</ns4:account>

            <ns5:role internalId="3" xmlns:ns5="urn:core_2015_2.platform.webservices.netsuite.com"/>

         </ns1:passport>

         <ns6:applicationInfo soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
soapenv:mustUnderstand="0" xmlns:ns6="urn:messages_2015_2.platform.webservices.netsuite.com">

            <ns6:applicationId>28E8EDD2-1A19-4061-9EC4-CE435035E73B</ns6:applicationId>

         </ns6:applicationInfo>

      </soapenv:Header>
```

Note: These system-generated Application IDs can be used on all existing supported SuiteTalk web services endpoints. The backward-compatibility nature of system-generated Application IDs allows developers to immediately adopt them in their integration apps without the need to upgrade to a newer WSDL and endpoint. Therefore, the NetSuite Operations team will no longer generate Application IDs on partners' behalf.

Important: The use of an Application ID and Token-Based Authentication (TBA) are mutually exclusive. When TBA is used, the NetSuite server maps the consumer key back to the Application ID from the originating Integration record, therefore the Application ID must not be explicitly specified in the SOAP request otherwise an "Invalid Application ID" error is thrown. For more details on TBA, please refer to section 5.11 of this guide.

3.3 Search Optimizations

NetSuite provides numerous approaches for searching for data. In SuiteApp development, one of the most useful types of searches is the saved search. See Saved Searches - the Benefits for complete details.

Note, however, certain solutions may require a more comprehensive search or a search that must be built in real-time, based on filtering criteria that are known only at the time of the search. See When Saved Searches Are Not Enough for details.

For information on how to use search operators to further optimize your search, see Search Operators and Performance.

3.3.1 Saved Searches - the Benefits

Saved searches provide a reusable search definition that can include advanced search filters and result set display options. You can create saved searches using the UI, or you can create them programmatically using nlapiCreateSearch and nlobjSearch.saveSearch(). Saved searches can be executed from the UI, or they can be invoked from an external application using SuiteTalk advanced search APIs.

Setting the search criteria in a saved search is not limited to the UI. As a developer, you can add additional filters to the saved search before invoking it, thereby limiting the number of saved searches you must create.

One of the advantages of a saved search is that it is reusable. You can go back to the Saved Searches list page in the NetSuite UI and see the results without having to redefine criteria. A saved search can also be used in other areas of NetSuite such as the dashboard. Adding a saved search to the dashboard makes accessing and viewing search results more convenient for end users.

Important: Saved searches are considered to be customization objects, and as such can be included in SuiteApps.

Another approach to searching in NetSuite is to programmatically define a search in scripts or external applications. One way of doing this is by referencing an existing saved search from your SuiteScripts. This approach cuts down on coding efforts when there are only minor changes in the searches during runtime.

3.3.2 When Saved Searches Are Not Enough

Certain use cases require a more comprehensive type of search. Simply referencing an existing saved search is not sufficient. You may know generally what you are looking for, but some fine-tuning may be necessary. By combining an existing saved search with a set of additional filters, you can further refine your search, giving you results that are closer to what you expected and saving you a lot of time.

Other use cases may require you to build searches at runtime, when filters and result columns are uncertain and should be built from scratch. This process can be tedious and complex compared to referencing a saved search, but can be accomplished using SuiteScript or web services search APIs.

Note: See *Searching with SuiteScript* in the NetSuite Help Center for more information on SuiteScript searches. Also see the documentation for the *search* operation (in the *Web Services Operations* section in Help) for information on searching by an external application (Java/C#).

3.3.3 Search Operators and Performance

One of the most important considerations when executing searches is performance. With the amount of data stored in a database, combined with the complex queries to retrieve the data, it is possible that extra time will be required to complete the search.

One way to prevent a time-consuming search is to minimize the volume of data returned to the application performing the search. Rather than retrieving all the fields of a record or set of records, specify the search result columns in your SuiteScripts or external applications. Specifying search result columns can be crucial, especially for web services, as the session may time out due to the huge result set being returned to the external application.

The use of search operators can have a positive impact on search performance, especially when the amount of data in a NetSuite account is large. In general, search operators that refer to specific keywords/values or “ranges” are faster. The more precise and confined the criteria, the more efficient the search becomes.

A use case may require you to get a list of items with purchase prices greater than X. The *greaterthan* search operator may seem the most logical operator to use. However, since this operator does not define a range to limit the scope of the search, the search could be slowed down by the size of the result set.

In this case, use a specific range as your criteria. Defining a range confines your search, and makes it faster because it returns less data. Search operators that define ranges are:

- *between*, *notbetween* – for currency, decimal number, or time of day
- *within*, *notwithin* – for dates

The following examples demonstrate how to use the *between* operator to search for inventory items with purchase prices with amounts between 100 and 500 dollars.

SuiteScript

```
var myInventorySearch = search.create({
   type : 'inventoryitem',
   columns : [ 'itemid', search.createColumn({name: 'cost'}) ],
   filters : [ 'cost', 'between', 100, 500 ]
});

myInventorySearch.run().each(function(result)
{
   log.debug('data',  result.getValue('itemid') + '\n' + result.getValue('cost'));

   return true;
});
```

Java (Web Service Application)

```
ItemSearchBasic itemSBasic = new ItemSearchBasic(); ItemSearchAdvanced itemSAdv = new
ItemSearchAdvanced(); // set criteria itemSBasic.setType(new SearchEnumMultiSelectField());
itemSBasic.getType().setOperator(SearchEnumMultiSelectFieldOperator.anyOf);
itemSBasic.getType().setSearchValue(new String[] {RecordType._inventoryItem});

itemSBasic.setCost(new SearchDoubleField());
itemSBasic.getCost().setOperator(SearchDoubleFieldOperator.between);
itemSBasic.getCost().setSearchValue(100.0); itemSBasic.getCost().setSearchValue2(500.0);

itemSAdv.setCriteria(new ItemSearch()); itemSAdv.getCriteria().setBasic(itemSBasic);
```

```
// set result columns ItemSearchRow itemSRow = new ItemSearchRow(); ItemSearchRowBasic itemSRowBasic =
new ItemSearchRowBasic();

itemSRowBasic.setItemId(new SearchColumnStringField[1]); itemSRowBasic.setItemId(0, new
SearchColumnStringField()); itemSRowBasic.setCost(new SearchColumnDoubleField[1]);
itemSRowBasic.setCost(0, new SearchColumnDoubleField());

itemSRow.setBasic(itemSRowBasic); itemSAdv.setColumns(itemSRow); SearchResult res =
_port.search(itemSAdv);
```

Another use case is searching for customers having the Name field as one of the filters. You may be lured into using the *contains* search operator, thinking that by doing so, your searches will return records containing the searched for keyword. However, using *contains* is not the most optimized way to execute this search from a performance perspective. Searches should be thought of as an SQL query; consequently, the use of the *contains* operator and formulas can be problematic for indexes.

As an example, when searching for a sales order with 'A Wolfe' as the customer name, the algorithm starts its search from the first set of characters matching the keyword's length (in this case, 7 characters). If the keyword is not found, it will search the keyword starting from the 2nd character until it reaches the last set of 7 characters.

Search Pass	Current Record	Keyword Found?
1st	Days Creek Electric Services	No
2nd	Days Creek Electric Services	No
3rd	Days Creek Electric Services	No
4th	Days Creek Electric Services	No

…	…	…
nth	Days Creek Electric Services	No

As demonstrated in the preceding table, the *contains* search operator could take several passes before the search can move to the next record to be evaluated, as opposed to the *is* and *startsWith* search operators. When using *is* and *startsWith*, the evaluation of the string is straightforward.

The *is* search operator is best to use if you know exactly what you are looking for. For cases in which only the beginning part of the keyword can be defined, the *startsWith* search operator is the most efficient, as it returns only the records that start with that keyword.

Performance Benefits of Web Services Advanced Searches

Whenever possible, web services applications should use advanced searches. One of the key features of advanced search is the ability to specify the fields to be returned. From a performance perspective, this feature provides a significant advantage over basic searches, which return entire records by default. When a web services application uses advanced searches that are precise in requesting only the desired fields be returned, the NetSuite server does not need to spend resources generating result sets that include unneeded fields. This key difference, combined with the efficient use of search operators discussed in this document, can provide vast performance improvements in searches.

The use of basic searches should be reserved primarily for development or troubleshooting purposes. When a web services application is deployed in a customer's production environment, its searches should be implemented using advanced searches whenever possible, in order to reap the associated performance benefits.

3.3.4 When to Avoid Using the 'noneof' Search Operator

In some use cases, a search excludes some records by using the *noneof* operator. This practice is typically done by providing a list of internal IDs in an array and setting it as a parameter in the search filters array.

SuiteScript

```
var excludeIDs = [3, 7, 9, 10, 11];

var myNoneOfSearch = search.create({
   type : 'salesorder',
   filters : [ ['internalid', 'noneof', excludeIDs], 'and', ['mainline', 'is', 'T'] ]
});
var searchResults = myNoneOfSearch.run();
```

The sample code above will execute efficiently provided that the number of internal IDs in the array are kept to a minimum. Once the cardinality gets up above 50 or more, the code becomes ineffective or impractical. The more records that you need to exclude from the search, the greater the decrease in speed and performance. Exclusion of many records may lead to longer response times or cause the search to timeout. To avoid these issues, it is a good idea to process the records relationally – an example is creating a custom record to store the processed records' internal IDs and defining additional logic to check them. Alternatively, you could also create a hash table instead of a custom record if hundreds of records are expected to be processed.

3.3.5 Limit the Use of Joined Searches

When searching through the SuiteScript or from the NetSuite UI, we give consideration to minimizing the size of the result by providing more search criteria. Logically, fewer search results equate to faster response times. However, developers should also be mindful of the filters and columns added to searches. Retrieving additional information by joining native or custom records to incorporated in a single search. May provide the needed search result but may not always equate to optimal performance. The more joined records in a search, the more time it takes for the query to complete because multiple tables could be involved.

To avoid sub-optimal performance, always remember to streamline your searches. When designing the application, spend enough time in identifying the search filters and columns that you really need and use them accordingly. For example, if you want to add search columns merely because they are nice-to-have information but are not necessarily requirements, then consider not adding them.

3.3.6 Handling Large Datasets

Some applications include daily tasks of performing backups, importing/exporting or reconciliation of data. These tasks normally involve extracting extremely large numbers of standard or custom records. The processing of these records can take hours to complete, and is very CPU-intensive. The completion time is not always directly attributed to the number of processed records. A longer than expected completion time could be caused by long response time of the searches due to improper use of search filters.

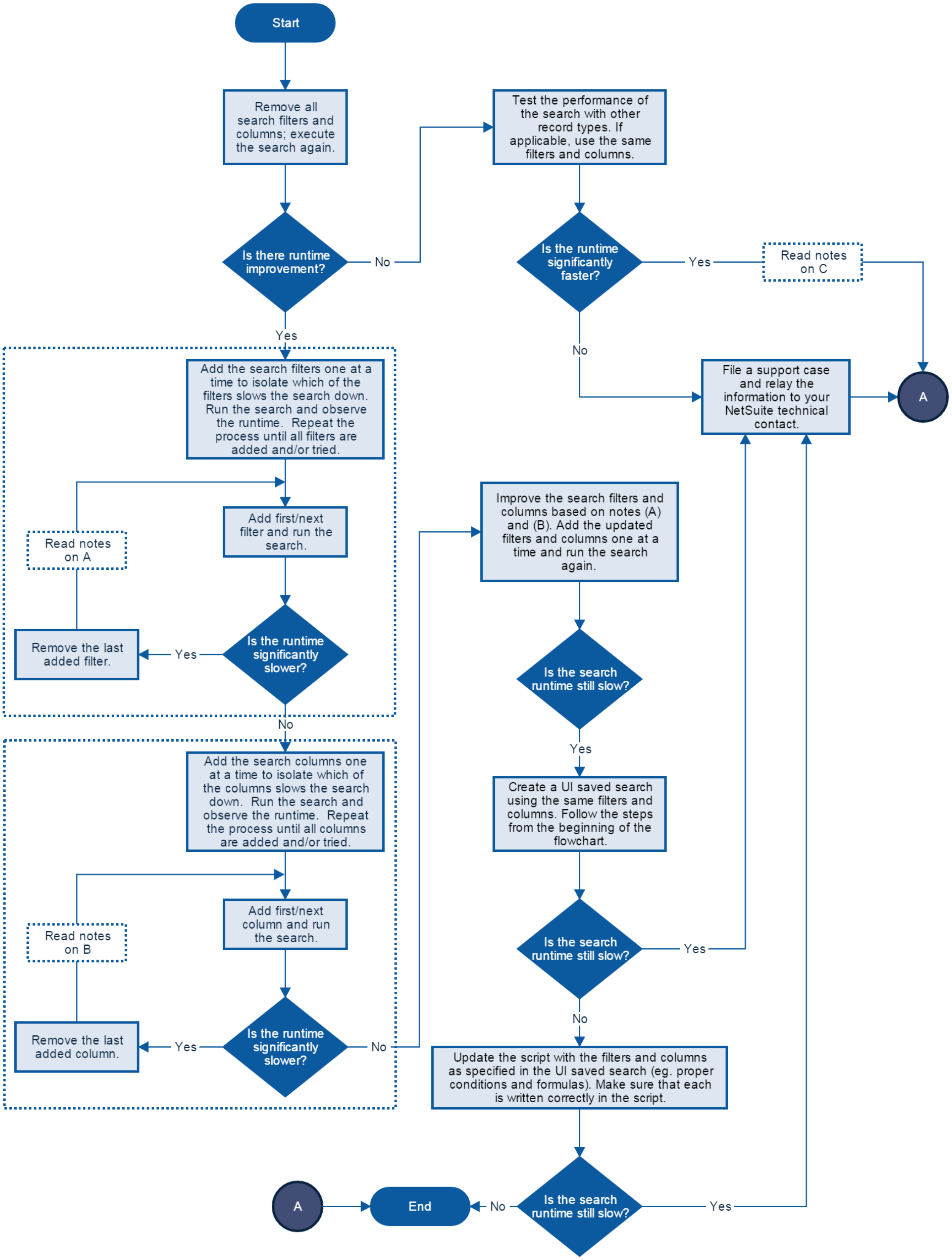
When searches are found to be poor performers, the root cause could be the improper use of compound filters. Since there are many filters available for searches and many possible permutations to form compound filters, indexing all possible permutations is impossible. Since compound filters are not indexed, they do not always present a real performance advantage in searches.

Be sure to understand what the driving condition of the search is. If possible, one search filter should at least be able to do the work of extracting and reducing the number of results. This filter could be a transaction date, an internal ID range or a custom field that can easily identify a set of records. If you have filter combinations such as "internalid = 5 and trandate = '11/20/2014'" it is best to use either the *internalid* or the *trandate*. Try experimenting with fewer search filters and see if there is an improvement in performance.

For saved searches to be efficient, it is highly recommended that the search criteria and result columns are kept to a minimum which was discussed in the section (refer to search optimization topic and page). However, there will be circumstances where an integration will have a search based on a *superset* such as the Transaction record -- wherein all transaction types and their respective columns are combined in the result set. This would end up in all the Transaction record columns being returned which could affect performance and the request timing out particularly on high volume queries.

There is a known bug where in some cases, the search would time out on the NetSuite server due to the sheer volume of data being retrieved. However, the SOAP response would still return a 'Success' status but will have 0 records in the result set which misleads the user. Since there is no specific number of records or result columns involved, it will be near impossible to isolate the problem and would be very hard to quantify the volume of records returned by these types of searches. What you can do is try to determine whether it is indeed a timeout or if records are actually returned. Execute the search with the same criteria but with very minimal number of result columns -- or better yet, just the internal ID. This will ease up the load on the NetSuite server and will give you a better chance of successfully completing the query. Once you have determined that there are records being returned, it is recommended to further split the results by adding more criteria to avoid search timeouts.

3.4 Search Optimization Flowchart


Start
Remove all search filters and columns; execute the search again.
Is there runtime improvement?
No
Test the performance of the search with other record types. If applicable, use the same filters and columns.
Is the runtime significantly faster?
Yes
Read notes on C
No
File a support case and relay the information to your NetSuite technical contact.
A
Yes
Add the search filters one at a time to isolate which of the filters slows the search down. Run the search and observe the runtime. Repeat the process until all filters are added and/or tried.
Read notes on A
Add first/next filter and run the search.
Remove the last added filter.
Yes
Is the runtime significantly slower?
No
Add the search columns one at a time to isolate which of the columns slows the search down. Run the search and observe the runtime. Repeat the process until all columns are added and/or tried.
Read notes on B
Add first/next column and run the search.
Remove the last added column.
Yes
Is the runtime significantly slower?
No
Improve the search filters and columns based on notes (A) and (B). Add the updated filters and columns one at a time and run the search again.
Is the search runtime still slow?
Yes
Create a UI saved search using the same filters and columns. Follow the steps from the beginning of the flowchart.
Is the search runtime still slow?
Yes
No
Update the script with the filters and columns as specified in the UI saved search (eg. proper conditions and formulas). Make sure that each is written correctly in the script.
Is the search runtime still slow?
Yes
No
End
A

3.5 Search Optimization Flowchart Notes

3.5.1 Filters that cause slowness in searches may be due to the following:

Performing a joined search - you should limit the use of joining fields as filters.

Using 'contains' condition on a text field - the 'contains' condition is one of the most resource-expensive mechanisms. Try replacing the condition with 'starts with' or 'has keywords.'

Using a formula to simplify filter criteria - when a search involves large data sets, using multiple filters instead of a single formula filter will yield faster results.

On large data sets, the best way to filter them is by using ranged criteria. For instance, using ranged dates for transactions (date is between 07/01/2015 and 07/30/2015) or looking for internal ids greater than 100.

3.5.2 Columns that cause slowness in searches may be due to the following:

Adding search columns from joined records - you should limit the use of columns that are from joined records.

Excessive number of result columns - the best practice is to select only the fields that are really needed in the result set. Remove less needed result columns.

3.5.3 If loading search results take longer than usual:

Check the settings of the search.

Determine the number of records being returned – displaying many thousands of rows of search results or report data can put considerable load on a NetSuite server. Consequently, you may experience a delay in the display of your search results

Check the criteria and results of the search to see if they can be modified for better performance (refer to A and B).

Consider scheduling the saved search to be run in the background, if you do not need real-time information.

Consider using the high volume data export capability for saved searches.

3.6 Concurrency and Data Bandwidth Considerations

Depending on the size of their business, some customers might have the needs for higher data I/O throughput. For example: a customer in the warehouse distribution vertical with larger warehouse(s) and/or larger amount of customers to serve might need higher data I/O throughput for the integrated SuiteApps that integrate to their NetSuite inventory and fulfillment data. Although these I/O throughput requirements are not unique to enterprise customers, they tend to be key considerations when they make their application purchasing decisions. Therefore, it is important for integrated SuiteApp developers to consider the throughput they can support now and in the future by choosing the correct integration architecture.

The most popular interfaces in the SuiteCloud platform for integrated SuiteApps are SuiteTalk web services and RESTlet. The former is a SOAP-based interface and the latter is a RESTful interface implemented using server side SuiteScripts. These two integration interfaces are equally strong in terms of NetSuite record exposure (standard and custom), security, as well as search-ability. Integrated SuiteApps will be well served by either interfaces. For a more detailed comparison of all the available SuiteCloud integration interfaces and methods, please refer to the SDN self-help video titled “Integration Deep Dive”.

Starting in version 2017.2, a unified concurrency governance model was adopted for both SuiteTalk and RESTlet, thereby enabling them to share a common pool of concurrent threads for integrations. The basic concept is the data I/O throughput of the integrated SuiteApp can be expanded by spawning multiple threads to get data in and out of NetSuite; thereby improving the overall performance and responsiveness. The default number of concurrency threads on a NetSuite account is based on its service tier; additional concurrent threads may also be added with the purchases of the SuiteCloud Plus add-on module.

3.6.1 Concurrency Support in SuiteTalk

Each integrated SuiteApp that uses SuiteTalk requires a NetSuite license. This license, otherwise known as an “integration user”, provides the integrated SuiteApp access to the NetSuite account. The integrated SuiteApp may spawn multiple threads concurrently making SuiteTalk API calls to NetSuite that are taken from the shared pool of concurrent threads. This pool’s limit is service tier based, and shared with RESTlets. If additional threads are needed for expanded I/O throughput, the customer may purchase the SuiteCloud Plus add-on module, which allows the integration user to have up to 10 additional shared concurrent threads.

Note: Even though SuiteTalk supports concurrency with a SuiteCloud Plus license (which is available to SDN partners on their SDN accounts free of charge), it incurs extra costs on the customer. Since not every customer has purchased this add-on module, integrated SuiteApps that rely on SuiteCloud Plus for expanded I/O bandwidth may see limited exposure in the market.

3.6.2 Concurrency Support in RESTlet

RESTlets may also service concurrent incoming threads, and the limit of the threads are also service tier based, and shared with SuiteTalk web services. The NetSuite system generates a unique URL for every RESTlet, which can be invoked by multiple threads spawned by a system outside of NetSuite. These threads can perform I/O in NetSuite data concurrently via the RESTlet.

Since a NetSuite account can have multiple RESTlets and/or SuiteTalk requests coming from different sources, it is the account administrator's responsibility to ensure the total number of threads do not exceed the governance limit of the service tier purchased.

Note**:** In order to make this administrator task possible, an integrated SuiteApp that utilizes concurrency support must provide a setting to configure the number of threads it will spawn.

3.6.3 Concurrency and Data Bandwidth Considerations

A unified concurrency governance model was adopted for SuiteTalk and RESTlet technology starting in version 2017.2. Beginning in version 2019.1 the model is adopted for REST Web Services (Beta), thereby enabling all to share a common pool of concurrent threads for integrations. The basic concept is the data I/O throughput of the integrated SuiteApp can be expanded by spawning multiple threads to get more data in and out of NetSuite faster. The model is intended to improve the overall performance and responsiveness of these services.

There are two types of limitations that apply simultaneously. The account level concurrency limit is derived from the service tier, the number of SuiteCloud Plus licenses and the account type. Additionally, the model applies user-level concurrency governance limits for specific authentication methods and APIs. User-level governance defines maximum limits but does not guarantee minimum resources due to per account limits. Review the Concurrency Governance Cheat sheet in the Appendices at the end of this document to see how to use these limits and calculate the number of threads for your account.

Concurrency Support in SuiteTalk

Each integrated SuiteApp that uses SuiteTalk requires a NetSuite license. This license, also known as an "integration user", provides the integrated SuiteApp access to the NetSuite account. The integrated SuiteApp may spawn multiple threads concurrently when making SuiteTalk API calls to NetSuite. These are taken from the shared pool of concurrent threads. To calculate the shared pool's limits, review the Concurrency Governance Cheat sheet in the Appendices at the end of this document. If additional threads are needed for expanded I/O throughput, the customer may purchase a SuiteCloud Plus add-on module. Each module allows the integration user to have up to 10 additional shared concurrent threads. For planning purposes, follow the decision tree in NS Help Center found here:

(https://system.netsuite.com/app/help/helpcenter.nl?fid=section_1500275531.html&whence)

This help center content will help you to understand how your SuiteApp would benefit from purchase SuiteCloud Plus licenses.

Concurrency Support in RESTlet

RESTlets may also service concurrent incoming threads and the rules for a limit of the threads These limits are also explained in the Concurrency Governance Cheat sheet in the Appendix at the end of this document. The NetSuite service generates a unique URL for every RESTlet, which can be invoked by multiple threads spawned by a system outside of NetSuite. These threads can perform NetSuite data I/O concurrently via the RESTlet. Since a NetSuite account can have multiple RESTlets and/or SuiteTalk requests coming from different sources, it is the account administrator's responsibility to ensure the total number of threads do not exceed the governance limit of the service tier including the number of SuiteCloud Plus licenses purchased.

3.7 Further Reading

SuiteScript Topics

- *Using Saved Searches*
- *Search Operators*
- *Searching with SuiteScript*
- *Search APIs*
- *Searching Overview*
- *Search Samples*

Web Services Topics

- *Advanced Searches in Web Services*
- *Search-Related Sample Code*
- *Web Services Asynchronous Operations*
- *Synchronous and Asynchronous Request Processing*

General Search-related Topics

- *Defining an Advanced Search*

4 Understand Your SuiteApp May Be One of Many in an Account

If your SuiteApps include user event scripts, you **must** design your scripts with the following considerations in mind:

- Your user event script may be one of many already deployed to a specific record type in a customer's account. For information on what this means to your SuiteApp, see Order of Script Execution.
- The user event script in your SuiteApp may be executed inadvertently, based on other SuiteApps running in a customer's account. For information on what this means to your SuiteApp, see SuiteApps Must be SuiteTalk-aware and SuiteApps Must be eCommerce-aware.

4.1 Order of Script Execution

An unlimited number of user event scripts can be deployed on NetSuite's three exposed record events (beforeLoad, beforeSubmit, afterSubmit). With an ecosystem of active developers and third-party solutions, customers can easily install multiple SuiteApps into their NetSuite accounts. Some of these SuiteApps may have user event scripts deployed on the same standard record, or even on the same events. You must consider the following when you include user event scripts in your SuiteApps.

- When developing user event scripts deployed on standard records, be aware that other user event scripts from other vendors may also be deployed in customer environments. This may cause data concurrency problems if multiple scripts, unaware of one another, update the same field(s).
- Understand that your script may not be executed in the desired order once other SuiteApps with user event scripts deployed on the same records are installed. If possible, design your script in a way so that it is agnostic of the order in which it is executed amongst a number of other scripts. If your script always needs to be the first or last one to execute, then the unexpected introduction of another script from another SuiteApp may cause problems.

Neither of the above problems can be easily resolved by using preventative coding. A more practical and economical approach is to document the fields your scripts update or change, and the order in which they must be executed (if applicable), and to educate your support team on how to aide customers if problems arise.

By clearly documenting how your user event scripts should perform, the customer, or your own support team, can effectively troubleshoot problems related to user event scripts resulting from different vendors' SuiteApps being deployed on the same record.

Important: To change the order in which user event scripts are executed in an account, go to the Scripted Record page at Set Up > Customization > Scripted Record , click Edit on the record, and drag and drop to arrange script execution order.

Also note that the existence of multiple user event scripts with the same trigger type, operating on the same record type, may negatively impact user experience. For example, if you include four beforeLoad user event scripts in your SuiteApp to be deployed to the Invoice record type, and the customer's account already has eight beforeLoad user event scripts deployed to the Invoice record type, when users access an Invoice record, 12 user event scripts will need to execute before the record even loads into the browser. The time it takes an invoice to load is increased, forcing users to wait before they can work with the record.

4.2 SuiteApps Must Be SuiteTalk-aware

Once your SuiteApp is deployed into a customer's account, you have little control over whether your user event scripts will be inadvertently executed by a web services solution that is integrated with the customer's account.

If the SuiteTalk feature is enabled in a customer's account, an external application can integrate with NetSuite by making an authenticated connection through a web services endpoint. By invoking NetSuite's web services classes and objects from an application, whether it is developed using Java, C#, or PHP, the application can perform record searches and CRUD operations.

Important: User access to records is role-based. Role-based authentication controls what users can do with a specific record.

Therefore, you must ensure that your scripts are compatible with potential web services requests, particularly imports, which can trigger the execution of user event scripts. Failure to recognize the effect web services solutions can have on the user event scripts in your SuiteApps can bring unexpected errors, performance issues, or data discrepancies. These issues are especially relevant for user event scripts deployed on standard records, particularly transaction records, because your scripts might perform business logic that has accounting and/or inventory implications.

Example

Consider a scenario in which Vendor A has developed a SuiteApp and Vendor B has developed a web services application. Both applications are completely unaware of one another.

Vendor A develops a user event script and deploys it to sales orders to perform processes that are unique to the business. These processes include inventory and/or custom record updates. Vendor A tests the script only in a browser, and everything runs as designed.

Vendor B has an existing web services application that imports sales orders. With Vendor A having its SuiteTalk feature enabled by its account administrator, Vendor B is able to develop an application that performs operations on Vendor A's records. Vendor B tests their application and it works as designed; the application is able to retrieve and process information successfully.

The problem with this scenario is that Vendor A's user event script is not designed to be triggered by web services requests. These requests might cause issues for Vendor A because its inventory and custom records may be manipulated without their knowledge and affect data integrity.

With the unexpected implications involved, Vendor A should be responsible for making sure that their user event script works properly when invoked. Vendor A's scripts should be able to tell if an execution request is coming from web services and decide whether to execute the script. Additionally, Vendor A must anticipate that additional external applications may need to access their records and may also trigger their scripts in the future.

Note: See Determining Script Execution Context for information on evaluating the execution context of a script.

4.2.1 Performance Implications

When SuiteApps have user event scripts that run on standard records, there may be a significant impact on performance if the scripts are not written properly. It is possible that when creating or updating transaction records through the NetSuite UI, additional validations may be performed in the beforeSubmit event. A script may also add a number of fields to a form in the beforeLoad event.

When a web services request is made on that particular record, the request may trigger the scripts deployed to the record. For the beforeSubmit event, some of the validation scripts may or may not apply to the web services call. And for the beforeLoad event, it does not make sense to execute scripts because there is no user interface to begin with, hence overall performance will be impacted.

Another similar scenario is when the script in an afterSubmit event is triggered. Consider what might happen if the page gets redirected to another page after updates have been done on a record. Page redirection may or may not be harmful, but it is unnecessary to execute additional code that is not needed by the external application.

The same thing occurs when a search is requested by an external application. The number of records that match the search criteria will have a significant impact on how fast the results are sent back to the requesting application. If the search criteria match hundreds of records having a complicated script attached to them, the search will overuse or exhaust system resources unnecessarily. Additionally, the web services session might time out, depending on how long the execution takes.

4.2.2 Determining Script Execution Context

Depending on the nature of your customer's business, external applications may need to access your platform application. In some cases, you will want your scripts to execute only when user actions occur through the UI. In other cases you will want your scripts to execute based on web services requests. Some cases will require your scripts to execute in both contexts: as a result of actions occurring through the UI and web services requests.

With this in mind, even if your scripts were originally meant to be triggered from actions occurring within the NetSuite UI, your script should be able to determine the invocation context. Calling nlapiGetExecutionContext() will tell the script in which context it is being triggered, ensuring that your scripts are executed only within a specific context.

In the following example, SuiteScript UI objects are added only when the caseBeforeLoad script is triggered from actions occurring in the user interface. If the user event is not occurring in the user interface, the code to programmatically add UI objects will not trigger, resulting in faster execution of the script and proper consumption of resources.

```
define(['N/runtime'],
    function(runtime)
    {
        function caseBeforeLoad(context)
        {
            if((runtime.executionContext == 'USERINTERFACE') && (context.type == 'edit' ||
            context.type == 'view'))
            {
             // user interface handling here (e.g. adding tabs or fields)
             log.debug('Runtime Execution Context: ',  runtime.executionContext);
             log.debug('User Event Context: ',  'Type: ' + context.type);
            }
       }
       return {
             beforeLoad: caseBeforeLoad
       };
    });
```

Checking for execution context within your script has the following pros and cons:

Pro

- When you check execution context, you have more control over how and when your script's logic will execute.

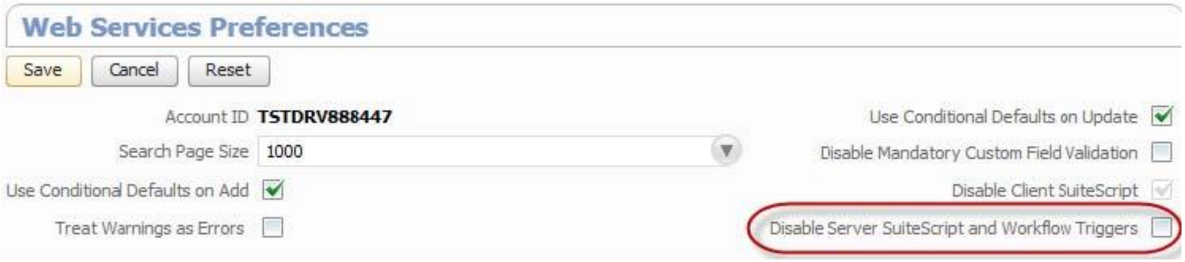
Con

- Checking execution context requires more thought, more lines of code, and therefore more development time.

Note: For more information on nlapiGetExecutionContext(), see *nlapiGetExecutionContext()* and *nlobjContext* in the NetSuite Help Center.

You can completely disable your scripts from being triggered by Web services calls by enabling the **Disable Server SuiteScript and Workflow Triggers** option on the web Services Preferences page.

You can enable this option at Setup > Integrations > Manage Integrations > Web Services Preferences, as shown in the following figure.



Account ID TSTDRV888447

Search Page Size 1000

Use Conditional Defaults on Add ☑

Treat Warnings as Errors ☐

Use Conditional Defaults on Update ☑

Disable Mandatory Custom Field Validation ☐

Disable Client SuiteScript ☑

Disable Server SuiteScript and Workflow Triggers ☐

Disabling server SuiteScript and workflow triggers has the following pros and cons:

Pro

- Enabling this option eliminates the need to assess on a per-script basis which scripts will execute. When this preference is selected, no scripts will execute.

Con

- All server scripts and workflow triggers are disabled. Therefore, you cannot control which scripts are triggered from specific events.

4.3 SuiteApps Must be eCommerce-aware

As a SuiteApp developer, you must be aware that the user event scripts in your SuiteApp can be triggered by SuiteCommerce Advanced or SiteBuilder solutions.

NetSuite customers can create one or more web sites or web stores. Their web sites and web stores can be hosted entirely by NetSuite. Or, customers can use another hosting service (outside of NetSuite) to organize and publish their web sites and web stores and integrate them with NetSuite's customer login, shopping cart, and checkout pages. When an online customer has finished shopping and checks out from the web store, the customer's data is processed and saved in a NetSuite sales order.

Another way of entering a sales order in NetSuite is through the user interface. This is the interface that allows sales order transaction entries from within the NetSuite application (for example, at Transactions > Sales > Enter Sales Orders). Whenever a sales order is accessed, scripts such as client SuiteScripts, Suitelets, and user event scripts get triggered. Similar to the behavior of a sales order within the UI, the creation of a new sales order that is triggered from the web store would trigger any user event script deployed to the sales order record.

If you have a SuiteApp that extends the Sales Order record, you must be mindful that the scripts deployed on the Sales Order might be inadvertently triggered by orders placed through the eCommerce site, even if the SuiteApp does not deal directly with eCommerce. Therefore, it is necessary to add code that allows your scripts to execute intelligently, depending on the execution context (for example, browser interface versus web store).

When the Web Store feature is enabled, depending on the use case, you should decide whether your scripts should be triggered from sales orders accessed through the web store, sales orders accessed through the UI, or both.

An example is an account service that specializes in fraud detection on eCommerce orders. When sales orders are created from the web store, the fraud detection script should be triggered to verify if the credit card is valid or stolen. However, the fraud detection script does not necessarily have to be triggered when the sales order is created by a Sales Rep through the UI. There will also be use cases where scripts should be triggered regardless of whether sales orders are created from the UI sales order or the web store. These cases could involve additional inventory handling or writing to a custom record log for transaction history purposes.

When necessary, script developers should isolate the UI scripts from the web store scripts deployed on the Sales Order record. Script developers must also take into account that the SuiteApp may be one of many implemented in a customer account. Failure to do so could trigger lines of code that are not meant to be executed and could also slow down the performance of your application and set off calculations specific only to either the UI sales orders or web store orders.

Additionally, SuiteApp scripts may contain special transaction/inventory handling or calculations for orders entered in the UI that are not applicable to orders submitted from the web store. To avoid this, scripts should be able to determine where the sales order was triggered by checking the execution context through calling the nlapiGetExecutionContext() SuiteScript API. This check ensures that UI custom scripts are executed only when new sales orders are created from the UI sales order, and web store custom scripts are executed only when orders are submitted from the web store.

In the following example, the SuiteScript UI objects are added only when the beforeLoad script is triggered from actions taken by the user through the user interface. In the beforeSubmit function, the validations and calculations for the orders made through the UI are segregated from the discount calculations of the web store orders, ensuring that unnecessary code is not executed.

```
define(['N/runtime'],
   function(runtime)
   {
      function caseBeforeLoad(context)
      {
         if ((runtime.executionContext == 'USERINTERFACE') && (context.type == 'create'))
         {
            // user interface handling here (e.g. adding tabs or fields)
            log.debug('Runtime Execution Context: ',  runtime.executionContext);
            log.debug('User Event Context: ',  'Type: ' + context.type);
         }
      }

      function caseBeforeSubmit(context)
      {
         if (context.type == 'create')
         {
            if (runtime.executionContext == 'USERINTERFACE')
            {
               //perform additional validates and calculations
               //based from UI entries
            }
            else if (runtime.executionContext == 'WEBSTORE')
            {
               // calculate special item discounts ordered by customer
               // from the web store
            }
         }
      }
      return {
         beforeLoad: caseBeforeLoad,
         beforeSubmit: caseBeforeSubmit
      };
   });
```

4.4 Namespace Conflicts between JavaScript Libraries

When a SuiteScript relies on a specific version of an open source JavaScript library, it is possible that it can conflict with a different version of the same library that NetSuite itself uses, for examples: ExtJS and jQuery. These conflicts occur when a SuiteScript references methods in libraries that are in fact NetSuite's own libraries.

These conflicts may not be apparent during the initial development and QA phases because the script continues to function normally, masking the fact that incorrect versions of the library methods are used. Once the script is deployed, the conflicts become dormant problems that could resurface during new NetSuite releases. When NetSuite upgrades its libraries, the upgraded library can become incompatible with the pre-existing SuiteScript, causing errors. In order for a SuiteScript to behave consistently and predictably, it is important for it to reference methods in its own library instead of those in NetSuite. Therefore, unique namespaces need to be defined in the SuiteScript code to avoid these conflicts.

Every library has its own mechanism to isolate its code definition. Developers should refer to the documentation accompanying each library being used for guidance.

As an example, if a specific version of the jQuery library must be included as part of a SuiteScript, the use of jQuery.noConflict() function is highly advisable. The following piece of code shows how two different jQuery library versions could work together.

```
<!DOCTYPE html>

<html lang="en">

<head>

     <!-- load jQuery 1.7.1 -->

     <script src="js/jquery-1.7.1.min.js"></script>

     <script type="text/javascript">

          var jQuery_1_7_1 = $.noConflict(true);

     </script>

     <!-- load jQuery 1.11.1 -->
```

The dollar sign in jQuery ($) is an alias, so this function allows the script to return control of $ back to the other library.

As seen in the previous source code, when two jQuery libraries need to be loaded in the same application, the $.noConflict(true) function (notice the True parameter) has to be invoked in order to return the globally scoped jQuery variables to the other version.

If any selector needs to be accessed using the 1.7.1 version of jQuery, here is how it has to be called:

```
jQuery_1_11_1('h1');
```

Note that the above is an example and the variable names can be chosen arbitrarily.

4.5 Considerations in the absence of SuiteCloud Plus

In Section 2.3.2 of this document, it is advised that developers should move long-running processes and I/O intensive processes to scheduled scripts in order to provide better user experience and obtain higher I/O throughput. While it is a good practice to use scheduled scripts in this manner, developers should be aware that most NetSuite accounts have one scheduled script queue (or scheduling queue). When multiple users and/or processes are programmatically invoking the same scheduled script simultaneously on an account with only one script queue, some invocations can fail to take place if these concurrency conditions are not handled correctly. Accounts with the SuiteCloud Plus license can have multiple scheduled script queues, but developers should nevertheless write scripts defensively to ensure they work in the majority of customer accounts that do not have this add-on module.

Strictly speaking, when a scheduled script is invoked and placed into the script queue, one of its **script deployment records** is placed into the script queue, not the script record. This is distinction is important. A script record can have multiple script deployment records associated with it.

Once invoked, a scheduled script deployment record is placed into the script queue and waits to execute. Until it finishes execution, any further attempts to invoke it will not be successful. However, invoking another script deployment of the same scheduled script, assuming it isn’t already executing or waiting in the script queue, can still be successful. Therefore, in order to avoid these concurrency issues, it is recommended that you have multiple script deployment records created for those scheduled scripts that could be invoked simultaneously by multiple processes or users. These scheduled script deployment records can be shipped in the bundle as part of the SuiteApp, or can be created by the account administrator.

Scheduled scripts and their script deployments behave the same way when invoked programmatically using SuiteScript API nlapiScheduleScript. When nlapiScheduleScript returns “QUEUED”, it means the invocation was successful. However, if it returns “INQUEUE”, then the invocation was unsuccessful because the script deployment is already waiting in the queue prior to the API call.

In the event of an INQUEUE return code, the caller script needs to look for the next available deployment record to invoke, call nlapiScheduleScript with the new script deployment ID and check the return code again. Alternatively, setting the deployId parameter in the API call (the second parameter in nlapiScheduleScript) to null instructs the NetSuite platform to look for the next available script deployment to invoke.

The approach of explicitly specifying the deployment record ID gives developers better control over their scripts. The following is sample code that demonstrates how to look for the next available deployment record in the event of a nlapiScheduleScript return code that is not “QUEUED”. Note that the code is hard-coded to loop through ten script deployment records that have IDs from customdeploy_simpleschedscript1 to customdeploy_simpleschedscript10.

```
var status = "";

if (status != 'QUEUED')
{
        //attempt to find the next available script deployment (assume 10 finite Deployments to select from) to executed
the script
        for (var i=1; i=<10; i++)
        {
                var statusAgain = nlapiScheduleScript('customscript_simpleschedscript', 'customdeploy_simpleschedscript' +
i);


                //if a free Deployment is found exit the loop;
                if(statusAgain  == 'QUEUED')
                break;
        }

        //if no deployment records are available, log this for the admin to create more Deployments via the UI.
        nlapiLogExecution(ERROR, 'QUEUE LIMIT REACHED', 'Manually create more Deployments in the UI');
}
```

4.6 Design Considerations for Using the externalId Field

4.6.1 Design Considerations for Using the externalId Field

The external ID field is a field found in all records supported by SuiteTalk web services. It is an optional field meant to store the primary keys for a matching table in an external database or system, for synchronization purposes with NetSuite. The SuiteCloud platform enforces uniqueness on the external Id field across all rows in a given table. A SuiteTalk integration SuiteApp can use this field to uniquely reference records inside NetSuite (just like it would with the internal ID field), thereby eliminating an extra search API call to obtain the records’ internal ID values.

Since the external Id field is a standard field that is meant to store external primary keys, developers building integrated SuiteApps using SuiteTalk web services should carefully consider whether their data stored in this field can be overwritten or reserved by another vendor's integration. This issue can occur when the external system is not the system of record for the data type in question and/or there are multiple external sources of that data type. For example, lead/prospect/customer records can potentially come from multiple systems, such as sales force automation systems, marketing automation systems, lead generation systems and others. If multiple systems set or update the external Id field, it is possible that these systems can overwrite each other's values, or one of these systems can fail to create records because there are duplicate external Id values already set by another system. In these cases, consider adding extra characters (such as your company's namespace as prefix) in the external Id value to ensure uniqueness.

In external applications that are the system of record for non-transactional data, the external Id field can be very useful. An example of a good use of the external Id field is the integration for a Human Resource Management System (HRMS). Consider the case where an external HRMS is the system of record for employee data. Even though NetSuite is not the employee system of record, it still needs to store employee records in order for most ERP transactions to work correctly. Therefore, employee records from the HRMS need to be imported into NetSuite. The external Id field is very useful in this integration because the HRMS can use it to uniquely identify employee records in NetSuite. There should be minimal risk of other systems overwriting or conflicting with its values because generally employee data is entered in only one system.

Note: When building integrations with external systems, NetSuite must be the system of record for ERP data such as journal entries, sales orders, purchase orders and invoices.

4.6.2 Using the External ID Field When Importing Critical Business Data

The uniqueness of the external ID field should be used to safe guard your critical business data during NetSuite system errors and exceptions. An example of this critical business data is sales order. It is common for sales order records to be imported into NetSuite from external shopping carts or EDI providers using SuiteTalk web services. This data tends to be revenue-generating or may carry contractual penalties if not processed in time. Therefore, extra care must be given to ensure these types of integrated SuiteApps are robust.

When importing complex transactions such as sales orders using synchronous web service, there is always the remote possibility of NetSuite server issues that cause the web service client to time out or throw an error. These issues may be the longer than expected time to process the orders, or the web services layer in the backend server stack throwing exceptions. In the latter scenario, the ERP or database tiers are likely still functioning and creating the orders. Unfortunately, this creates the condition to cause duplicate records if the integrated SuiteApp is not idempotent in its API calls. These problems can be avoided by using the external ID field set (see below).

After timing out, the web service client assumes there was a server side error and may execute its retry logic to attempt the import again.   Therefore, the retry logic inadvertently created duplicate sales orders – the first set of correct orders were still being processed by the system when the client code timed out, the second set of duplicate orders were imported by the retry logic.

The problem of duplicate orders created during these exceptions can be easily resolved by populating the External ID field. This field must be set in all the records in the original import AND their values must be maintained when the retry logic is executed. In the scenario described above, the external ID acts as a safety net to prevent duplicate records because the system will enforce the field's uniqueness and reject the retry logic's import. If there was indeed a deeper system failure during the import that causes some of the orders to be truly "lost", then the retry logic will execute successfully because the external ID values are still unique.

Alternatively, an easier way to make an Integrated SuiteApp idempotent with its add and update API calls is to use the upsert and upsertList operations, both of which require the externalID field to be populated.

Important: The original use case that drive the design of the external ID field is data synchronization with external systems – it is meant to store the primary key from an external system and can be used to uniquely address a NetSuite record. This feature has been in production for a long time, and the observation made by the NetSuite team is those integrated SuiteApps that use this feature tend to experience far less data-related issues because of the field acting as a safety net against data duplication. Due to its ease of use and other benefits such as duplicate data avoidance listed above, the use of the external ID field has emerged as a highly recommended best practice for SuiteTalk client integrations even if they do not have the need to synchronize NetSuite data with external systems.

Important: Any integrated SuiteApp that uses the "add" or "addList" operations should utilize the externalID field because it provides great benefits with very little engineering effort required.

4.7 SuiteApp Designs and Concurrency Issues

SuiteApps can encounter concurrency issues similar to those faced by applications developed on any other cloud-based or on-premise platform. Concurrency issues are especially likely to occur for SuiteApps that are built on the platform using tools such as SuiteBuilder and SuiteScripts. However, not all concurrency issues commonly encountered on traditional operating systems are relevant or applicable to SuiteApps running on the SuiteCloud platform. This section discusses concurrency issues within the context of SuiteApp designs.

Concurrency issues relevant to SuiteApps can be classified into two categories:

- Resource Starvation
- Race Conditions

4.7.1 Resource Starvation

Resource starvation can occur when a process in a multi-tasking environment is perpetually deprived of a critical resource such as CPU time.

Generally speaking, due to NetSuite's robust governance model, there is very little a SuiteApp can do to cause resource starvation problems. On the rare occasion where a scheduled script or CSV task is "stuck" in the queue and blocked, the problem needs to be reported to the NetSuite technical support team for a quick resolution.

4.7.2 Race Conditions

Race conditions arise when two processes, or threads, operate on a common piece of data without synchronization between them. The end result is data corruption that can often be difficult to reproduce and troubleshoot.

Race conditions are the concurrency problem most commonly faced by SuiteApps, especially SuiteScripts. Typically, race conditions occur when two or more processes (multiple SuiteScripts and/or browser users) make changes to the same record(s) at approximately the same time without awareness of each other's actions. The lack of synchronization between the processes can cause data corruption. Consider two SuiteScripts that both read from a custom record and then increment one of its integer fields. The following sequence demonstrates how a race condition can occur when both SuiteScripts are executed at roughly the same time:

- A custom record holds a value of 1 in an integer field.
- SuiteScript A looks up the custom record (or loads it into memory) and reads the value of 1.
- SuiteScript B looks up the custom record (or loads it into memory) and reads the value of 1.
- SuiteScript A increments the value to 2 in memory.
- SuiteScript B increments the value to 2 in memory.
- SuiteScript A writes the value 2 to the custom record.
- SuiteScript B writes the value 2 to the custom record.

The custom record should have held a value of 3, not 2, at the end because two SuiteScripts incremented it. These SuiteScripts should not have been executed concurrently; they should have been executed consecutively. Unfortunately there was no mutual exclusion mechanism to prevent these scripts from overlapping.

In other words, the two scripts are not aware of each other's concurrent actions, and the data changes made by one of them were trampled by the changes made by the other.

Platform-based vertical SuiteApps can be vulnerable to these race conditions because they rely heavily on custom records to represent specific vertical business objects. Race conditions can occur frequently in environments where multiple users can simultaneously invoke SuiteScripts that operate on the same set of custom records, or can edit these records using browsers. Without synchronization between the processes, or locking mechanisms on the custom records, data corruption can occur.

To significantly (but not completely) reduce the chances of SuiteScripts getting into race conditions, developers should move their scripts' critical sections into individual scheduled scripts. These scheduled scripts are then all placed into the same scheduled script queue. Since tasks placed in the scheduled script queue execute serially, they will not overlap. This design approach can help eliminate most SuiteScript race conditions. However, there are still some scenarios that can cause race conditions. For example, a user can still use a browser to edit a custom record that is being processed by a scheduled script and as a result, cause data corruption. A critical section scheduled script is also in danger of getting into race conditions if it yields because it is put at the end of queue, allowing another schedule script to edit its record before it has a chance to complete execution. This approach can also cause significant performance issues in a high-volume, multi-user environment, because the forced serial execution of scheduled scripts makes parallel processing impossible.

Note: NetSuite standard records have built-in support for record locking, therefore SuiteScripts that only write to standard records do not suffer from race conditions.

4.7.3 Optimistic Locking for Custom Record Types

Since Version 2012 Release 2, NetSuite has supported optimistic locking for custom record types. This feature makes custom records' concurrency control consistent with that of standard records. When multiple processes or users attempt to make changes to the same custom record, the first process or user that is able to successfully save its changes prevails. All the other processes will fail with an error; they must reload the custom record (which will include the changes made by the successful process) and submit their changes again.

Optimistic locking also works in SuiteScripts by first loading a custom record into memory with nlapiLoadRecord, then updating it with nlapiSubmitRecord.

When two or more SuiteScripts load the same record, with optimistic locking enabled, into their respective memory space for editing, only one of them will be able to submit the changes. The other one will fail with a CUSTOM_RECORD_COLLISION error message. If we reconsider the race condition scenario as described in the previous section, the following sequence of events will occur when optimistic locking is enabled on the custom record in question.

- A custom record holds a value of 1 in an integer field.
- SuiteScript A loads the custom record into memory using nlapiLoadRecord and reads the value 1.
- SuiteScript B loads the custom record into memory using nlapiLoadRecord and reads the value 1.
- SuiteScript A increments the value to 2 in memory.
- SuiteScript B increments the value to 2 in memory.
- SuiteScript A submits the updated custom record using nlapiSubmitRecord; the operation succeeds.
- SuiteScript B submits the updated custom record using nlapiSubmitRecord; a CUSTOM_RECORD_COLLISION error is thrown and the update operation fails.

In the last step above, SuiteScript B failed to update the custom record because optimistic locking detected that it was trying to update an outdated version of the record. The platform threw a CUSTOM_RECORD_COLLISION error that prevented SuiteScript B from overwriting the changes made by SuiteScript A.

Optimistic locking works by embedding a timestamp on a custom record when it is loaded. This timestamp is invisible and inaccessible to users and SuiteScripts, therefore optimistic locking in SuiteScripts only works when a custom record is loaded into memory using nlapiLoadRecord, then updated using nlapiSubmitRecord.

Important: Optimistic locking does not work when the following are used to update custom records because these techniques do not load the custom records into memory:

- nlapiSubmitField
- Inline editable sublists in parent-child record relationships. See Using Parent-Child Relationships to Perform Mass Update/Create.

Note: nlapiSubmitField is listed as the recommended API to update records (where applicable) for performance benefits. Developers should be aware that this API is not compatible with optimistic locking when writing SuiteScripts and consider using alternative designs.

If your SuiteApp's scheduled scripts are using nlapiSubmitField to update custom records due to its superior performance, and you want to take advantage of the data integrity advantages offered by optimistic locking, then consider using the Multiple Scheduled Script Queue feature available in the SuiteCloud Plus add-on module (free to SDN partners, at a cost to customers). You may spread your scheduled script deployments across multiple queues. This feature allows multiple scheduled scripts to run concurrently in order to obtain higher SuiteApp execution throughput. The added throughput might be able to offset the performance penalty of using the slower tandem APIs of nlapiLoadRecord and nlapiSubmitRecord.

Another noteworthy design consideration for SuiteApps that use optimistic locking is that this feature does not provide any capabilities for transactional management for multi-record operations that need to be atomic. An atomic process has a succeed or fail definition, meaning all of its operations must succeed or they have no apparent effect on the state of the overall system. SuiteScripts with this requirement need to take into account that the SuiteCloud platform does not provide a commit-rollback capability. For example a SuiteScript that updates two custom records may need to be atomic. This requirement means either both update API calls must succeed or both must fail; an intermediate state of only one of the records updated successfully is not acceptable. In the event that the second record fails to update, the SuiteScript must either attempt to undo the update on the first record or log the sequence of I/O events that led up to the failure in order to make manual rollback possible.

An alternative design to help avoid the need to write rollback logic in SuiteScript is to implement a virtual semaphore for processes prone to encountering these scenarios.

4.7.4 Virtual Semaphores by External ID

A semaphore is an abstract data type, provided by an operating system that controls the access of a common resource by multiple processes in a multi-user environment. The SuiteCloud platform does not provide semaphores to SuiteApps. However, a similar structure can be devised by using a special custom record type and its external ID field.

The external ID field is a field found in all records supported by SuiteTalk web services. It is originally meant to store the primary keys for a matching table in an external database for synchronization purposes with NetSuite. This field is an optional field, so. it can be null. Because it stores primary keys for external tables, any non-null values stored in it must be unique. A record with an external ID value that conflicts with that of another record of the same type cannot be created. This platform-enforced uniqueness of external ID field values is used to build virtual semaphores for SuiteScripts.

Consider a SuiteScript that performs an atomic transaction that consists of the following sequence of I/O operations:

- Updating a custom record of record, type = customrecord_type_a, internal ID = 123
- Updating a custom record of record, type = customrecord_type_b, internal ID = 456
- Updating a custom record of record, type = customrecord_type_c, internal ID = 789

This SuiteScript is used in an environment where it can be invoked concurrently by multiple users; each invocation may operate on a distinct set of records or on the same set of records. Without optimistic locking enabled on the three custom record types, race conditions can occur when two or more users invoke the script that operates on the same records at the same time. With optimistic locking enabled, race conditions are mostly averted. However, the transaction still has no atomicity because an update operation that fails due to optimistic locking can still leave the whole process in an intermediate state if no rollback logic is provided.

A solution to this problem is a specialized, stand-alone custom record type that serves as a semaphore. Before a SuiteScript attempts to operate on a set of custom records, it must first successfully create a semaphore record with an external ID that consists of the record types and internal IDs of those records to be updated. Since the platform enforces uniqueness on external IDs, the semaphore custom record can be used as a mechanism to reserve a unique combination of records to ensure they are updated by only one script invocation at a time.

The format of the semaphore's external ID must be well defined, and honored by all SuiteScripts that implement this design pattern. For the sample transaction above, the semaphore record's external ID format can be the combined string of each record type, followed by the internal ID field, so the value is "**a**123**b**456**c**789", which encompasses the record types and internal IDs of the records to be updated. Any other invocation of the same script that tries to operate on the same set of records cannot proceed before the first invocation completes, because it will not be able to create the semaphore record with that unique external ID. A DUP_CSTM_RCRD_ENTRY error is thrown, thus avoiding race conditions.

Once the transaction is completed, the script must release the semaphore by deleting the semaphore record. This deletion allows other invocations of the same script operating on the same set of records to create their own semaphore records and proceed.

The following sample code incorporates a virtual semaphore into a SuiteScript for the example use case described above.

```
try
{
   var recordCombination = 'a123b456c789';
   var semaphore = _record.create({ type: 'customrecord_sdn_semaphore'  }); // creating
   semaphore
   semaphore.setValue({ fieldId: 'externalid', value: recordCombination }); // setting
   external ID
   var semaphoreid = semaphore.save(); // saving semaphore

   // Code here to perform core I/O Operations

   _record.delete({ type: 'customrecord_sdn_semaphore', id: semaphoreid });
}
catch(e)
{
   log.error(e.name, e.message);
}
```

If the atomic logic terminates ungracefully for any reason, the semaphore it creates may not get deleted, which will prevent any subsequent invocation of the script on that combination of records from proceeding. Therefore, it is important to have a scheduled script that sweeps for semaphore records that have been in the system for too long, which may be the remnants of ungracefully terminated SuiteScripts.

Note: The virtual semaphore design pattern only works within the confines of those SuiteScripts that implement and honor it. Any other SuiteScripts and processes (such as end users) unaware of the reservation of a combination of records for editing can still directly update them and cause data integrity issues.

4.8 Further Reading

- *User Event Scripts*
- *Web Site Setup Overview*

5 Design for Security and Privacy

SuiteApps must be designed with the security requirements of cloud computing and SuiteCloud platform security recommendations in mind.

NetSuite was designed with certain security features. On the transport layer, all pages within the application are delivered using the HTTPS protocol. The backbone of NetSuite security is built on a roles and permissions model, in which users are given roles that define their access level to records and reports.

Always remember that security cannot be bolted on to a finished application.

The content in this section explains how SuiteApp developers can apply the NetSuite roles and permissions model to the customization objects they build.

As a SuiteApp developer, you are required to adopt designs and practices related to security and privacy. The following topics discuss these concepts and how to implement them:

- Roles and Permissions
- Secure SuiteScript Designs
- Validate Data Input
- External Suitelets and Their Security Risks
- Programmatic Access to NetSuite Passwords
- External System Passwords and User Credentials
- Credit Card Information
- SuiteSignOn
- Privacy Considerations
- Implementing Inbound Single Sign-On with Web Services

5.1 Roles and Permissions

A secure enterprise application should allow its users to accomplish their tasks using the least possible access to data and lowest possible privileges to perform system tasks. For SuiteApps, this goal is accomplished by relying on the NetSuite platform's roles and permissions model.

You will typically use the administrator role in your development account when writing a SuiteApp. The administrator role is convenient because it gives full access to all records and full permissions to perform all tasks – something that is vital during development. However, SuiteApp users are highly unlikely to be granted access to the administrator role. Therefore, it is important for you to rely on NetSuite's role-based permissions model in developing your SuiteApp security model.

NetSuite comes with a number of standard roles designed to be used by staff with specific roles within a company. These standard roles grant the users permissions to the records they need in order to perform their work, but restrict access to other records that are not required for their jobs. For example, the Accountant role has access permissions to journal entries, but does not have the permissions to receive items into the inventory. As a SuiteApp developer, you should reuse these standard roles where applicable. For example, access permissions to an “Automobile” custom record type shipped with a SuiteApp for auto dealers can be given to the appropriate sales-related roles such as Sales Manager and Sales Person.

When standard roles are too restrictive or too loose, you should create custom role(s) that are tailored to the specific access permission requirements for the SuiteApp. You can create custom roles by customizing an existing role (standard or custom). You can also create custom roles by starting with a blank role and gradually adding only those permissions required by your SuiteApp. This approach adheres to the security principle of enabling users to perform their tasks using the least possible level of access and privileges.

Note: An exception to the SuiteApp security best practice of using a non-administrator role is if your SuiteApp needs to perform tasks that can only be accomplished with administrator privileges. Examples include an HR SuiteApp that creates/grants employee records with the administrator role, and a SuiteScript that needs to dynamically invoke a scheduled script by calling nlapiScheduleScript.

Typically, custom roles are one of the first components you will create for a SuiteApp. The purpose of custom roles in SuiteApps is to limit access to SuiteApp components to only authorized users. These components will likely include custom data schema extensions (custom fields and custom record types) and custom logic (SuiteScript). All of these objects can have their own security applied.

Because NetSuite users view data from many angles, it is possible that data may be revealed to users in ways you have not anticipated. Therefore, when custom fields and custom record types contain sensitive information, it is important to rely on the platform’s role-based permissions model to control access to this data. The design principles for custom roles in SuiteApps should be driven by the following considerations:

- What tasks do the users need to do by using the SuiteApp?
- What is the lowest role level they can have to access data?
- What is the least amount of privileges needed to perform system tasks?

Note: The authentication model for integration applications (applications that use SuiteTalk wWeb services and/or RESTlets) are also role-based, therefore, the security design principles listed above also apply to them.

For information on choosing a permissions model for custom record types, see *Creating Custom Record Types* in the NetSuite Help Center.

5.2 Secure SuiteScript Designs

In general, SuiteScript-based CRUD operations should be implemented to execute business logic and to maintain data integrity, NOT to enforce data access security. For example, an equipment rental solution SuiteApp may ship a Rental Agreement custom record type and a SuiteScript that controls the CRUD operations on these records. Since there is business logic in setting the fields in Rental Agreement records, the rental use case will likely require write access to be done only by the bundled SuiteScripts under normal circumstances.

The SuiteApp should include custom roles that are created based on the requirements of different user types that need access to these components. A Rental Agent custom role may be included to access the SuiteScript that reads, creates, and updates the Rental Agreement custom records. An Agent Supervisor may have additional permissions to delete the Rental Agreement custom records, and/or have permissions to execute SuiteScripts designed for supervisor tasks.

In the rental agreement use case, SuiteScript designs that force users to enter data in a preset route, and validate the data entered, are good designs because they adhere to the rule of enforcing business logic and maintaining data integration. SuiteScript designs that have **Execute as Admin** set and/or check the user's role to perform data access are bad designs because they override the platform's role-based security model. As a SuiteApp developer you should avoid shipping SuiteScripts set to **Execute as Admin** whenever possible.

If direct access to the custom records is allowed in the use case (for example, the records can be accessed using forms instead of using SuiteScripts), you should consider shipping custom forms that are tailored for specific custom roles.

Important: SuiteScript deployments can be exposed to standard or custom roles. When performing unit testing, developers and QA engineers should test with the intended custom roles, NOT the administrator role. Role-based testing raises issues related to permissions. These issues can then be rectified early in the development cycle.

Custom roles and custom forms designed for SuiteApps should be made available as part of your SuiteApp.

Note: For more information on working with custom forms, see *Creating Custom Entry and Transaction Forms* in the NetSuite Help Center.

5.3 Validate Input Data

Most Suitelets are designed to accept data POST'd to them by their UI portion. Likewise, most RESTlets are designed to accept data POST'd to them by a specific client such as a mobile device.

You may want to write your scripts to accept input of expected data type, length, or list of selections coming from expected clients. However, you should be wary of other unintended clients that POST to Suitelets or RESTlets. Some of these clients may be freely downloadable debugging tools that can POST to URLs with arbitrary input you may not have anticipated. Therefore, it is important to sanitize all data input in Suitelets and RESTlets to avoid data integrity issues or security breaches.

SuiteScript developers should adopt the practice of zero-tolerance for allowing the input of unsanitized data. A request must have all of its data sanitized before it is permitted to be processed by the script. This practice ensures only data that can be safely consumed is processed by the core logic of the script. Data sanitation includes checking data type, length, ranges, and expected choices.

Example

A select field that can be set to 1, 2, and 3, with 3 being a reserved value to be used in the future, must ensure the input is either 1, 2, or null (if applicable). Any values outside this range must cause the entire request to be rejected with an error, even if the rest of the input is valid.

5.4 Programmatic Access to NetSuite Passwords

NetSuite login passwords for existing users cannot be accessed through the browser interface, SuiteScript APIs, or SuiteTalk APIs. As a consequence, the password field is not viewable when searching (nlapiSearchRecord, nlapiCreateSearch, SuiteTalk search), loading (nlapiLoadRecord, SuiteTalk get), or looking up Employee records (nlapiLookupField). SuiteApps should not be designed based on the assumption that user passwords can be programmatically obtained.

On the Employee record, the password field is settable using APIs only during create and update.

For SuiteTalk web services applications, login must be done by either prompting the user for passwords when the application is invoked, or by using the ssoLogin operation for automation SuiteApps that do not require human intervention. **NetSuite user passwords must not be stored outside of NetSuite for the benefit of easier authentications.**

5.5 External System Passwords and User Credentials

Passwords and user credentials for external systems should not be stored in NetSuite in an unencrypted format.

Passwords stored in custom records or custom fields are considered insecure. Even when data is hidden by custom forms and/or SuiteScripts that restrict access, forms and scripts can still be overruled by an administrator and pose a potential for security breaches in the external system.

SuiteApps that require access to external systems should do so using SuiteSignOn. SuiteSignOn provides secure connection points that allow SuiteScript and subtabs to sign in to external applications without needing to store passwords in NetSuite.

If using SuiteSignOn is not a feasible option, or if the external system user must be prompted to enter passwords inside the NetSuite interface, then the alternative is to use SuiteScript to add a credential field to a NetSuite form. This is accomplished using nlobjForm.addCredentialField(...).

The addCredentialField(...) method generates a password field on the form object, and ensures that entered data cannot be accessed programmatically. In order to provide end-to-end secure sign-in connections to external systems, nlapiRequestURLWithCredentials() should be used for the actual sign-in call. The vendor should provide an HTTPS URL for this API call.

Note: For information on adding a credential field, see *nlobjForm.addCredentialField() for SuiteScript 1.0 / Form.addCredentialField(…)* for SuiteScript 2.0 in the NetSuite Help Center.

5.6 Credit Card Information

The only secure place where credit card numbers may be stored in NetSuite is in the Credit Cards subtab, specifically in the Credit Card Number field of that subtab. The Credit Cards subtab is found on Customer records and on certain transaction records. The Credit Card Number field is encrypted in the NetSuite database, and is masked when displayed in the browser, unless the account owner explicitly enables the ability to view unmasked credit card numbers.

Credit card numbers should **never** be stored in NetSuite outside of the Credit Cards subtab, and NetSuite's customers are **contractually prohibited** from storing credit card numbers in fields other than those specifically designated by NetSuite. Additionally, the 3 or 4-digit card security code (found on the back of most credit cards) should never be stored in the system post-authorization and, ideally should not be stored at all.

NetSuite strongly recommends not enabling the View Unmasked Credit Cards feature unless the account has an overriding business need to do so. Most accounts do not. Users do not need to be able to view the full credit card number to process credit card transactions (such as authorization, sale, credit). Predominantly, the need to view unmasked credit cards numbers is associated with companies using third party logistics (3PL) vendors to complete order fulfillment and process the sale upon shipment. In this manner, full, unmasked, credit card numbers are settable and viewable only upon initial entry into the Credit Cards subtab.

It is important to note that the ability of SuiteScript and SuiteTalk to view or set credit card details is the same as that of the role as which they are running. Also, SuiteBuilder (point-and-click customization) does not support creating credit card objects.

An example of a situation where a SuiteApp might seek to store Credit Card numbers outside of the valid Credit Card subtabs could be in support of an external service that specializes in managing and paying company-issued credit cards. In this situation, because there is no Credit Cards subtab on the Employee record, there is no secure place to store company credit cards issued to employees inside NetSuite. Therefore, an external service must store the information for these credit cards. Each credit card may be issued a "nickname," which can be stored on the NetSuite Employee record.

Remember that whenever you design any credit card capabilities that store, transmit, or process credit card numbers in your SuiteApp, you are required to handle the credit card data and processing in compliance with the Payment Card Industry Data Security Standard (PCI-DSS) or Payment Application Data Security Standard (PA-DSS), as appropriate. The PCI compliance principles can currently be found in the documents kept at these sites:

https://www.pcisecuritystandards.org

https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2.pdf

https://www.pcisecuritystandards.org/documents/Prioritized-Approach-for-PCI_DSS-v3_2.pdf

5.7 SuitePayments API

SuitePayments, a new SuiteCloud platform feature, was made available in NetSuite version 2015.1. It is an API set based on the SuiteScript plug-in framework that allows authorized SDN partners and developers to build integration SuiteApps which utilize various payment method gateways.

Due to the sensitive nature of credit card and payment information typically handled by payment gateway solutions, customers/merchants have a very high expectation of the security of the design and implementation of SuiteApps that use SuitePayments. The stability, functional completeness, as well as the handling of credit card information, along with the accountability and traceability, of are of the utmost concern to customers. Therefore, NetSuite has chosen to make the APIs available to authorized SDN partners only, and have placed stringent design and implementation requirements onto these SuitePayments developers.

For information on how adhere to the compulsory key design and implementation principles, see the Built-for-NetSuite Verification section in the Payment Processing Plug-in documentation. Note that you must log in to NetSuite to access the documentation.

5.7.1 Accountability - Identification

The provider of a SuitePayments solution can be a payment gateway service or a third party developer/ISV. Since the latter is the most common use case, it is imperative for a SuitePayments solution developer to put in measures that establishes accountability to the users for all transactions that pass through it. It is required that end users of the merchants' NetSuite accounts (excluding eCommerce shoppers) can easily and correctly identify the provider of the SuitePayments solution by its legal name in the NetSuite user interface. The easiest and most intuitive way to accomplish this is to put the SuiteApp provider's legal name in the UI so it is visible during bundle installation and/or during use of the solution. See additional, bundling naming conventions found in Section 8.

5.7.2 Traceability

SuitePayments solutions provide the payment gateway URLs of which the merchant's credentials and unencrypted credit cards numbers are sent during normal operation. Prior to the general release of these SuiteApps, NetSuite whitelists the payment gateways URLs. This is the primary measure to protect sensitive data from being sent elsewhere.

During the Built-for-NetSuite verification process, the list of all error codes that can be returned by the payment gateway must be submitted to NetSuite for tracking purposes. The list of applicable error codes that the SuitePayments solutions handles must be identified.

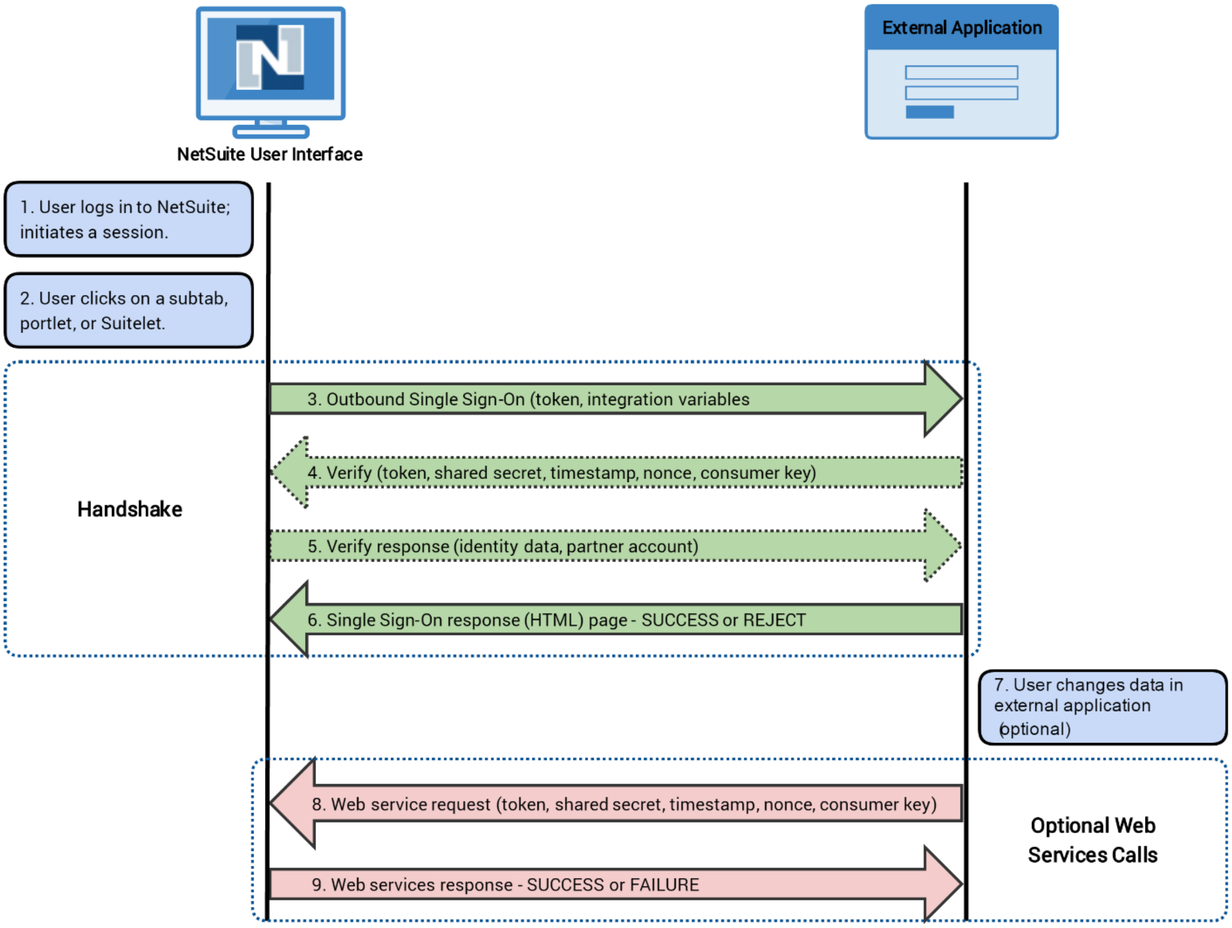
A SuitePayments solution must implement setReferenceCode(referenceCode) upon the completion of a successful payment operation. The method must write the reference code (P/N Ref) back to the originating transaction (usually a Sales Order or Cash Sale record) for traceability reasons.

5.8 SuiteSignOn

SuiteSignOn connection points allow NetSuite to be a trusted application and securely sign in to an external application, thereby embedding the external application’s UI into the NetSuite UI. Upon a successful connection, the external application may optionally make web services calls.

If an external application does not require secure authentication, SuiteSignOn should not be used. An application that needs to make web services API calls, but otherwise does not have UI integration requirements (for example, mash-ups), or does not require secure authentication for UI integrations, should not use SuiteSignOn solely as a means for obtaining a Web services session.

External applications that make web services API calls by using sessions obtained from SuiteSignOn should not manipulate data outside the scope of the UI integration. For example, a SuiteSignOn CRM application that integrates into Customer records should not use its web services sessions to touch any transactions or any unrelated standard or custom records.



NetSuite User Interface
External Application
1. User logs in to NetSuite; initiates a session.
2. User clicks on a subtab, portlet, or Suitelet.
Handshake
3. Outbound Single Sign-On (token, integration variables
4. Verify (token, shared secret, timestamp, nonce, consumer key)
5. Verify response (identity data, partner account)
6. Single Sign-On response (HTML) page - SUCCESS or REJECT
7. User changes data in external application (optional)
8. Web service request (token, shared secret, timestamp, nonce, consumer key)
9. Web services response - SUCCESS or FAILURE
Optional Web Services Calls

SuiteSignOn Connection Diagram

With regard to utilizing any external resources in your solutions, there are a few additional very important principles to bear in mind. First of all, it is best to simply avoid the risks. This risk avoidance is accomplished when your SuiteApp is designed to retain all of your customer's data and processes within the NetSuite environment. When it becomes necessary to include third party data stores or processes, you are obligated to communicate this design construct to your customer. Furthermore, you are obligated to validate that the third party components are also compliant. Just as you must adhere to the principles in this SAFE guide, you must ensure that your third party also adheres to these principles.

5.9 Privacy Considerations

As a SuiteApp developer, you and your SuiteApp will be interacting with sensitive customer data such as social security numbers, taxidentification numbers, bank a details, and credit card details. Customer data may be protected under privacy and data protection laws, ordinances, and regulations. In addition, NetSuite requires you to protect customer data (i) by designing your SuiteApp in accordance with these Principles, (ii) in your agreements with NetSuite, and (iii) in your agreements with your customers. In addition to designing your SuiteApp with features to securely transmit, process, and store customer data, you must make commitments and disclosures to your customers about such practices.

Your agreement with your customers must describe your privacy commitments to your customers and how you and your SuiteApp collect and use customer data. In addition, you must provide a link to any privacy policy which is applicable to your customers.

In connection with any data you receive or process from the European Union, you should consider becoming Safe Harbor Certified prior to commercializing any SuiteApps. The U.S. Department of Commerce manages Safe Harbor certification. Helpful information regarding Safe Harbor can be found here: http://export.gov/safeharbor/.

In addition, you should investigate European data protection requirements, including the standard contractual clauses for the transfer of personal data to processors. This is not limited to GDPR requirements. **GDPR** (General Data Protection Regulation) The General Data Protection Regulation (**GDPR**), is defined as an agreement by the European Parliament and Council in April 2016, which will replace the Data Protection Directive 95/46/ec in Spring 2018 as the primary law regulating how companies protect EU citizens' personal data.

NetSuite strongly recommends that you consult with data privacy counsel regarding your obligations and best practices with respect to data privacy and data security.

5.10 Token-Based Authentication for SuiteTalk web services

Starting with version 2015.2 of the SuiteTalk web services endpoint, integration applications can use the new Token-Based Authentication feature (TBA) as a secure authentication method that does not require user credentials.

Generally, there are these two types of integration applications that can make use of the TBA feature:

Automation integrations

These integration applications typically make API calls to NetSuite as a backend process in an automated fashion. They use a dedicated NetSuite license for integration purposes, and are not tied to a specific end user. An example is an application that performs nightly synchronization of NetSuite inventory item data. In this scenario, there will be one token for the dedicated user to complete the TBA.

User-centric integrations

These integration applications make API calls based on end user-driven actions. Every user must have a NetSuite license to make web services calls. An example is a sales automation integration that a sales rep uses to retrieve item information out of NetSuite in real time. In this scenario, there will be multiple tokens for the integration application – one for each sales rep.

Due to its security advantage, and industry standard implementation (RFC 2104-compliant signature), new integration applications built on the 2015.2 or newer endpoint should seriously consider using TBA as the authentication model. The endpoints are backward compatible. Therefore, existing integrations using Inbound SSO are still supported. Note that TBA does not have the UI-level, single sign-on capability that the Inbound SSO feature has.

This section provides an overview of how to develop and deploy a SuiteTalk web services integration that uses TBA as the authentication method. In order to demonstrate the lifecycles of these integrations, the material presented is divided into three sections: Development and QA Tasks, Deployment Tasks, and Customer Tasks. For the detailed “how-to” guidance of individual tasks, please refer to the NetSuite Help Center.

Important: It is mandatory that any consumer keys, secrets, and tokens generated by NetSuite and persisted by the integration applications in NetSuite be encrypted when stored at rest, and transmitted over encrypted channels.

5.10.1 Development and QA Tasks

There is no significant difference between building an integration application using TBA and building one using either Inbound SSO or credentials-based authentication. The only difference is on the authentication portion, which can be done independently from the core application logic development. The following are the steps and general guidance on development and QA. Note that step 1 and 2 are generic steps and not unique to TBA integration applications.

1) In the SDN development account, create an integration custom role that has the precise level of permissions and privileges the integration application needs (remember to include the “Web Services” permission); do not check the “Web Services Only Role” option until deployment time in order to ease QA efforts.

2) Assign the integration custom role in step 1 to a test user for core development and QA work.

3) Build the UI and backend data model to allow users to enter, update, and store the token ID and token secret pair. Alternatively, build the UI and data model to support invoking the token endpoint to programmatically create tokens. User-centric integrations need to have the ability to store the ID/secret pair for each user.

4) In the SDN development account, create an Integration record and enable the Token-Based Authentication option.

5) Incorporate the consumer key, consumer secret, and application ID* (optional) from step 4 in the core application. These properties must not be exposed to users, but will remain configurable internally should they need to be changed in the future.

6) Edit the integration custom role in step 1 and add either the "User Access Tokens" permission or "Log in Using Access Tokens" permission.

7) Create an internal custom role with the "Access Token Management" permission. Note that this role is meant for internal testing purposes and should not be part of the finished application.

8) Use the internal custom role in step 7 to assign an Access Token for the Integration to the test user. Specify the custom integration role in the Role field.

9) Login as the test user with the custom integration role, and create an Access Token for the integration. Encrypt and store the token ID and token secret pair in the UI built in step 3.

10) Build the code to use TBA. You may wish to download the Java or C# sample applications provided by NetSuite for use as sample coding.

Important: The use of an application ID and TBA are mutually exclusive. When TBA is used, the NetSuite server maps the consumer key back to the application ID from the originating Integration record. Therefore, the application ID must not be explicitly specified in the SOAP request otherwise an "Invalid application id" error is thrown. For details on Application ID, please refer to chapter 3 of this guide.

A web services request that uses TBA must use the `TokenPassport` complex type. By contrast, a request that authenticates in another way uses either the `Passport` or `SsoPassport` type.

The `TokenPassport` references the `TokenPassportSignature` complex type, which is another important element in the TBA process. Both complex types are defined in the Core XSD file, version 2015.2 and up. (https://webservices.netsuite.com/xsd/platform/v2015_2_0/core.xsd)

TokenPassport

The `TokenPassport` complex type uses the following fields. All are required:

- **account** – Your NetSuite account ID. You can find this number at Setup > Integration > Web Services Preferences, in the Account ID field.
- **consumerKey** – The consumer key for the integration record. This string was created when you checked the Token-based Authentication box on the integration record and saved it.
- **token** – This is the Token ID in the Access Token record. It represents a unique combination of an account, a user, a role, and an integration record. This string can be generated in multiple ways.
- **nonce** – This field should hold a string randomly generated using a cryptographically secure pseudo-random number generator (CSPRNG). Note: using timestamp as the nonce is not recommended

because it could cause rejected requests in highly concurrent environments

- **timestamp** – This field should hold a current timestamp in Unix format, +/- 5 minutes synchronization with the server
- **signature** – The signature is a hashed value. You create this value by using all of the other values in the TokenPassport, plus the appropriate token secret and consumer secret. Along with the actual signature, you must identify the algorithm used to create the signature.

TokenPassportSignature

You use the TokenPassportSignature complex type to identify the signature, which is a hashed value. The TokenPassportSignature also includes an attribute labeled algorithm, which you use to identify the algorithm used to create the signature.

At a high level, you create the signature by completing the following steps:

- Create a base string.
- Create a key.

Create a base string. The base string is from concatenating a series of values specific to the request. Use an ampersand as a delimiter between values. The values should be arranged in the following sequence:

- NetSuite account ID
- Consumer key
- Token
- Nonce
- Timestamp

For example, suppose you have the following variables:

- NetSuite account ID – 1234567
- Consumer key – 71cc02b731f05895561ef0862d71553a3ac99498a947c3b7beaf4a1e4a29f7c4
- Token – 89e08d9767c5ac85b374415725567d05b54ecf0960ad2470894a52f741020d82
- Nonce – 6obMKq0tmY8ylVOdEkA1
- Timestamp – 1439829974

In this case, the base string would be as follows.

```
1234567&71cc02b731f05895561ef0862d71553a3ac99498a947c3b7beaf4a1e4a29f7c4&89e08d9767c5ac85b
374415725567d05b54ecf0960ad2470894a52f741020d82&6obMKq0tmY8ylVOdEkA1&1439829974
```

Create a key. The key is a string variable created by concatenating the appropriate consumer secret and token secret. These two strings should be concatenated by using an ampersand.

For example, suppose you have the following variables:

- Consumer secret – 7278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd
- Token secret – 060cd9ab3ffbbe1e3d3918e90165ffd37ab12acc76b4691046e2d29c7d7674c2

In this case, the key would be as follows:

```
7278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd&060cd9ab3ffbbe1e3d3918e90
165ffd37ab12acc76b4691046e2d29c7d7674c2
```

Choose a supported hash algorithm to create an RFC 2104-compliant signature. The parameters of the algorithm are the base string and the key. The recommended algorithmis HMAC-SHA256.

Use the base string, the key, and the algorithm to create an RFC 2104-compliant signature. These are set in the TokenPassportSignature.value and TokenPassportSignature.algorithm properties. The signature must be encoded by Base64.

The `TokenPassport` object must be embedded in SOAP header to authenticate - similar to the way Request Level Credentials work. This is the only way to use `TokenPassport`, there is no dedicated web service operation to authenticate using TBA.

5.10.2 Deployment Tasks

The Integration record and the SuiteBundler are central to the deployment of TBA integration applications. Therefore, both must be used for deployment and distribution of the application. The following are the deployment tasks jointly owned by the development team and release/deployment team.

- On the development account, edit the integration custom role, enable the “Web Services Only Role” and Save the record.

- On the development account, create a bundle that contains the Integration record, the integration custom role, and any other customization objects that the integration requires to operate successfully.

- Use the Bundle Copy function to copy the bundle from the development account to the deployment account. The Integration record and integration custom role are now on the deployment account, and are bundle components within the copied bundle.

- On the deployment account, set the availability of the bundle copy to either Public or Shared

- Instruct customers to install the bundle from the deployment account. The customer must provide their NetSuite account ID if the bundle is shared.

Note that if the Integration record needs to be replaced due to changes in consumer keys and secrets, the change needs to start in the development account, and propagated to deployment and customer accounts. If the integration application is mission critical, the Managed Bundle feature should be used to ensure timely updates. Please see chapter 7 of this guide for more details on Managed Bundle.

5.10.3 Customer Tasks

The following are tasks that need to be performed by the customer administrator and/or the end users after the bundle is successfully installed in the target account. It is recommended that this information be made available to the customers as part of the integration application's set up documentation.

- The customer administrator assigns the integration custom role to the end users authorized to use the integration.
- The customer administrator creates a custom role with the "Access Token Management" permission and assign it to a user. Typically, this user is someone in the IT department who oversees all the integration applications used by the organization.
- The IT department user in step 2 logs in and assigns an Access Token for the Integration to each authorized integration user and specifies the custom integration role in the Role field.
- The authorized integration user must log in with a role that has either the "User Access Tokens" permission or "Log in Using Access Tokens" permission and assigns Access Tokens for the integration. The authorized user then provides the Token ID and Token Secret to the integration application (this step is more applicable to user-centric integrations)
- Alternatively, for step 4, if the integration uses the token endpoint to programmatically create tokens, the authorized integration user would use the integration application to invoke this functionality. The credentials need to be provided one time only in order to perform this task. Once the tokens are created successfully, these credentials must not be stored by the integration application.

5.11 Token-Based Authentication for RESTlets

Generally, there are these two types of integration applications that can make use of the TBA feature:

Automation integrations

These integration applications typically make API calls to NetSuite as a backend process in an automated fashion. They use a dedicated NetSuite license for integration purposes, and are not tied to a specific end user. An example is an application that performs nightly synchronization of NetSuite inventory item data. In this scenario, there will be one token for the dedicated user to complete the TBA.

User-centric integrations

These integration applications make API calls based on end user-driven actions. Every user must have a NetSuite license to make RESTlet calls. An example is a sales automation integration that a sales rep uses to retrieve item information out of NetSuite in real time. In this scenario, there will be multiple tokens for the integration application – one for each sales rep.

This section provides an overview of how to develop and deploy a RESTlet integration that uses TBA as the authentication method. In order to demonstrate the lifecycles of these integrations, the material presented is divided into three sections: Development and QA Tasks, Deployment Tasks, and Customer Tasks. For the detailed "how-to" guidance of individual tasks, please refer to the NetSuite Help Center.

Important: It is mandatory that any consumer keys, secrets, and tokens generated by NetSuite and persisted by the integration applications in NetSuite be encrypted when stored at rest, and transmitted over encrypted channels.

5.11.1 Development and QA Tasks

There is no significant difference between building an integration application using TBA and building one using either Inbound SSO or credentials-based authentication. The only difference is on the authentication portion, which can be done independently from the core application logic development. The following are the steps and general guidance on development and QA. Note that step 1 and 2 are generic steps and not unique to TBA integration applications.

1) In the SDN development account, create an integration custom role that has the precise level of permissions and privileges the integration application needs.

2) Assign the integration custom role in step 1 to a test user for core development and QA work.

3) Build the UI and backend data model to allow users to enter, update, and store the token ID and token secret pair. Alternatively, build the UI and data model to support invoking the token endpoint to programmatically create tokens. User-centric integrations need to have the ability to store the ID/secret pair for each user.

4) In the SDN development account, create an Integration record and enable the Token-Based Authentication option.

5) Incorporate the consumer key, consumer secret, and application ID* (optional) from step 4 in the core application. These properties must not be exposed to users, but will remain configurable internally should they need to be changed in the future.

6) Edit the integration custom role in step 1 and add either the "User Access Tokens" permission or "Log in Using Access Tokens" permission.

7) Create an internal custom role with the "Access Token Management" permission. Note that this role is meant for internal testing purposes and should not be part of the finished application.

8) Use the internal custom role in step 7 to assign an Access Token for the Integration to the test user. Specify the custom integration role in the Role field.

9) Login as the test user with the custom integration role, and create an Access Token for the integration. Encrypt and store the token ID and token secret pair in the UI built in step 3.

10) Build the code to use TBA.

Important: The use of an application ID and TBA are mutually exclusive. When TBA is used, the NetSuite server maps the consumer key back to the application ID from the originating Integration record. Therefore, the application ID must not be explicitly specified in the SOAP request otherwise an "Invalid application id" error is thrown. For details on Application ID, please refer to chapter 3 of this guide.

OAuth 1.0 RESTlet Authorization Header

With TBA, you use the OAuth 1.0 specification to construct an authorization header. Some of these values can be obtained from the NetSuite UI. Other values must be calculated. Typically, your integration should include logic to identify these values and generate the finished header. Follow the OAuth 1.0 protocol to create the authorization header.

- **realm** – The ID of the NetSuite account where the RESTlet is deployed. You can find this number at Setup > Integration > Web Services Preferences, in the Account ID field.
- **oauth_consumer_key** – The consumer key for the integration record being used to track the calling application. This string was created when you checked the Token-based Authentication box on the integration record and saved it. To create an OAuth header, you also need the consumer secret that goes with the key, although you do not use the secret explicitly. If you no longer have these values, you can regenerate them. For details, see "Regenerating a Consumer Key and Secret" in SuiteAnswers.
- **oauth_token** – A token that represents a unique combination of a user and an integration record. This string can be generated in multiple ways. For details, see "Managing TBA Tokens" in SuiteAnswers. To create an OAuth header, you also need the token secret that goes with the token, although you do not use the secret explicitly.
- **oauth_nonce** – This field should hold a string randomly generated using a cryptographically secure pseudo-random number generator (CSPRNG). This shoul be a unique, randomly generated alphanumeric string of 6-64 characters. Note: using timestamp as the nonce is not recommended because it could cause rejected requests in highly concurrent environments
- **oauth_timestamp** – This field should hold a current timestamp in Unix format, +/- 5 minutes synchronization with the server.
- **oauth_signature_method** – A hash algorithm that can be used to create an RFC 2104-compliant signature. Supported choices are: HMAC-SHA1 or HMAC-SHA256
- **oauth_version** – The version of OAuth being used. Only one value is supported: 1.0.
- **oauth_signature** – The signature is a hashed value. You create this value by using all of the other values above, plus the appropriate token secret and consumer secret. Along with the actual signature, you must identify the HTTP method being used to make the call.
- Note: With many languages, an OAuth library is available to help you create the signature. For details about some of the third-party open source libraries that are available, see SuiteAnswers 42171. If you are working in a language that does not have a library, you may want to refer to SuiteAnswers 42019 for an overview of the signature-creation process.

OAuth Signature

To create an OAuth header, you must have a **consumer key** and **secret** that represents the application that will call the RESTlet. In general, you create these values by creating an integration record. When you create the record and enable the record's Token-based Authentication option, the system generates and displays the consumer key and secret.

You use the header to identify the **signature**, which is a hashed value. The **signature method** also includes an attribute labeled algorithm, which you use to identify the algorithm used to create the signature.

Creating the Signature

At a high level, you create the signature by completing the following steps:

- Create a base string
- Create a key

Create a base string. The base string is from concatenating a series of values specific to the request. Use an ampersand as a delimiter between values. The values should be arranged in the following sequence:

- NetSuite account ID
- Consumer key
- Token
- Nonce
- Timestamp

For example, suppose you have the following variables:

- NetSuite account ID – 1234567
- Consumer key – 71cc02b731f05895561ef0862d71553a3ac99498a947c3b7beaf4a1e4a29f7c4
- Token –89e08d9767c5ac85b374415725567d05b54ecf0960ad2470894a52f741020d82
- Nonce – 6obMKq0tmY8ylVOdEkA1
- Timestamp – 1439829974

In this case, the base string would be as follows.

```
1234567&71cc02b731f05895561ef0862d71553a3ac99498a947c3b7beaf4a1e4a29f7c4&89e08d9767c5ac
```

Create a key. The key is a string variable created by concatenating the appropriate consumer secret and token secret. These two strings should be concatenated by using an ampersand.

For example, suppose you have the following variables:

- Consumer secret – 278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd
- Token secret - 060cd9ab3ffbbe1e3d3918e90165ffd37ab12acc76b4691046e2d29c7d7674c2

In this case, the key would be as follows:

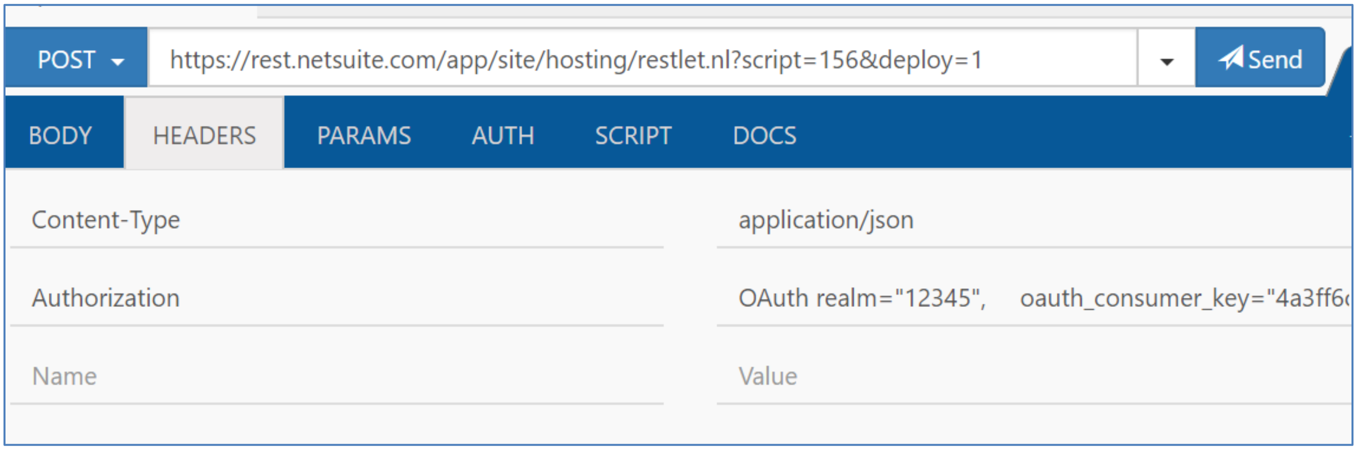
```
7278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd&060cd9ab3ffbbe1e3d3918
```

Choose a supported hash algorithm to create an RFC 2104-compliant signature. The parameters of the algorithm are the base string and the key. The recommended algorithm is HMAC-SHA256.

Use the base string, the key, and the algorithm to create an RFC 2104-compliant signature. The signature must be encoded by Base64.

The Authorization header field must be included in your request header to authenticate - similar to the way Request Level Credentials work.

e.g.



https://rest.netsuite.com/app/site/hosting/restlet.nl?script=156&deploy=1

Content-Type	application/json
Authorization	OAuth realm="12345", oauth_consumer_key="4a3ff6

The following snippet shows a correctly formatted OAuth authorization header:

Authorization:
OAuth realm="**12345**",
oauth_consumer_key="**4a3ff6c251a55057bb1e62d8dc8998a0366e88f3a8fe735265fc425368b0f154**",
oauth_token="**52cfe88fecf2e2b74e833e7dfc4cae79ff44c3ca9f696d61e2a7eac6c8357c3c**",
oauth_nonce="**qUwlmPvtGCS4sHJe8F7x**",
oauth_timestamp="**1462453273**",
oauth_signature_method="**HMAC-SHA256**", oauth_version="**1.0**",
oauth_signature="**8PI9lIYxUmUONjxEFJUSMD9oOmc%3D**"

5.11.2 Deployment Tasks

The Integration record and the SuiteBundler are central to the deployment of TBA integration applications. Therefore, both must be used for deployment and distribution of the application. The following are the deployment tasks jointly owned by the development team and release/deployment team.

- On the development account, create a bundle that contains the Integration record, the integration custom role, and any other customization objects that the integration requires to operate successfully.
- Use the Bundle Copy function to copy the bundle from the development account to the deployment account. The Integration record and integration custom role are now on the deployment account, and are bundle components within the copied bundle.
- On the deployment account, set the availability of the bundle copy to either Public or Shared
- Instruct customers to install the bundle from the deployment account. The customer must provide their NetSuite account ID if the bundle is shared.

Note that if the Integration record needs to be replaced due to changes in consumer keys and secrets, the change needs to start in the development account, and propagated to deployment and customer accounts. If the integration application is mission critical, the Managed Bundle feature should be used to ensure timely updates. Please see chapter 7 of this guide for more details on Managed Bundle.

5.11.3 Customer Tasks

The following are tasks that need to be performed by the customer administrator and/or the end users after the bundle is successfully installed in the target account. It is recommended that this information be made available to the customers as part of the integration application's set up documentation.

- The customer administrator assigns the integration custom role to the end users authorized to use the integration.
- The customer administrator creates a custom role with the "Access Token Management" permission and assign it to a user. Typically, this user is someone in the IT department who oversees all the integration applications used by the organization.
- The IT department user in step 2 logs in and assigns an Access Token for the Integration to each authorized integration user and specifies the custom integration role in the Role field.
- The authorized integration user must log in with a role that has either the "User Access Tokens" permission or "Log in Using Access Tokens" permission and assigns Access Tokens for the integration. The authorized user then provides the Token ID and Token Secret to the integration application (this step is more applicable to user-centric integrations)
- Alternatively, for step 4, if the integration uses the token endpoint to programmatically create tokens, the authorized integration user would use the integration application to invoke this functionality. The credentials need to be provided one time only in order to perform this task. Once the tokens are created successfully, these credentials must not be stored by the integration application.

5.12 Cryptographic Functions

SuiteScript 2.0 provides additional cryptographic functions that are not available in SuiteScript 1.0, such as Hashing and HMAC (Hash-based Message Authentication Code). Users are encouraged to use such features, when applicable.

When there is a need to implement additional information protection for data shared between NetSuite and external system, it is recommended to encrypt content using the n/crypto module (for confidentiality), then calculate the HMAC digest from the generated CipherPayload object (for integrity, authentication).

The following algorithms are considered deprecated for Hashing and HMAC operations, and are available for legacy purposes only:

- MD5
- SHA-1

New integrations are encouraged to use the following algorithms:

- SHA-256
- SHA-512

6 Test Your SuiteApps

It is your responsibility to ensure that the SuiteApps you deploy into your customers' accounts run as intended from one NetSuite release to the next.

To ensure your SuiteApps run as intended, you should use the QA tools provided by NetSuite, if possible and appropriate for your SuiteApp. After deploying a SuiteApp into an account, you must retest, and, if necessary, update your SuiteApps during the Release Preview phase that accompanies each new NetSuite release.

6.1 SuiteScript Testing Framework – now obsolete

Up until the 2016.1 cycle, NetSuite recommended the use of a tool called the SuiteScript Testing Framework. The SuiteScript Testing Framework was delivered as a bundled SuiteApp. It included jsUnity, a JavaScript-based unit testing framework. This bundle is no longer available, however, the jsUnity tools are available as open source. You are encouraged to seek out other automated testing tools from other sources.

To obtain more current information about automated testing in a NetSuite environment, contact SDNQA@netsuite.com.

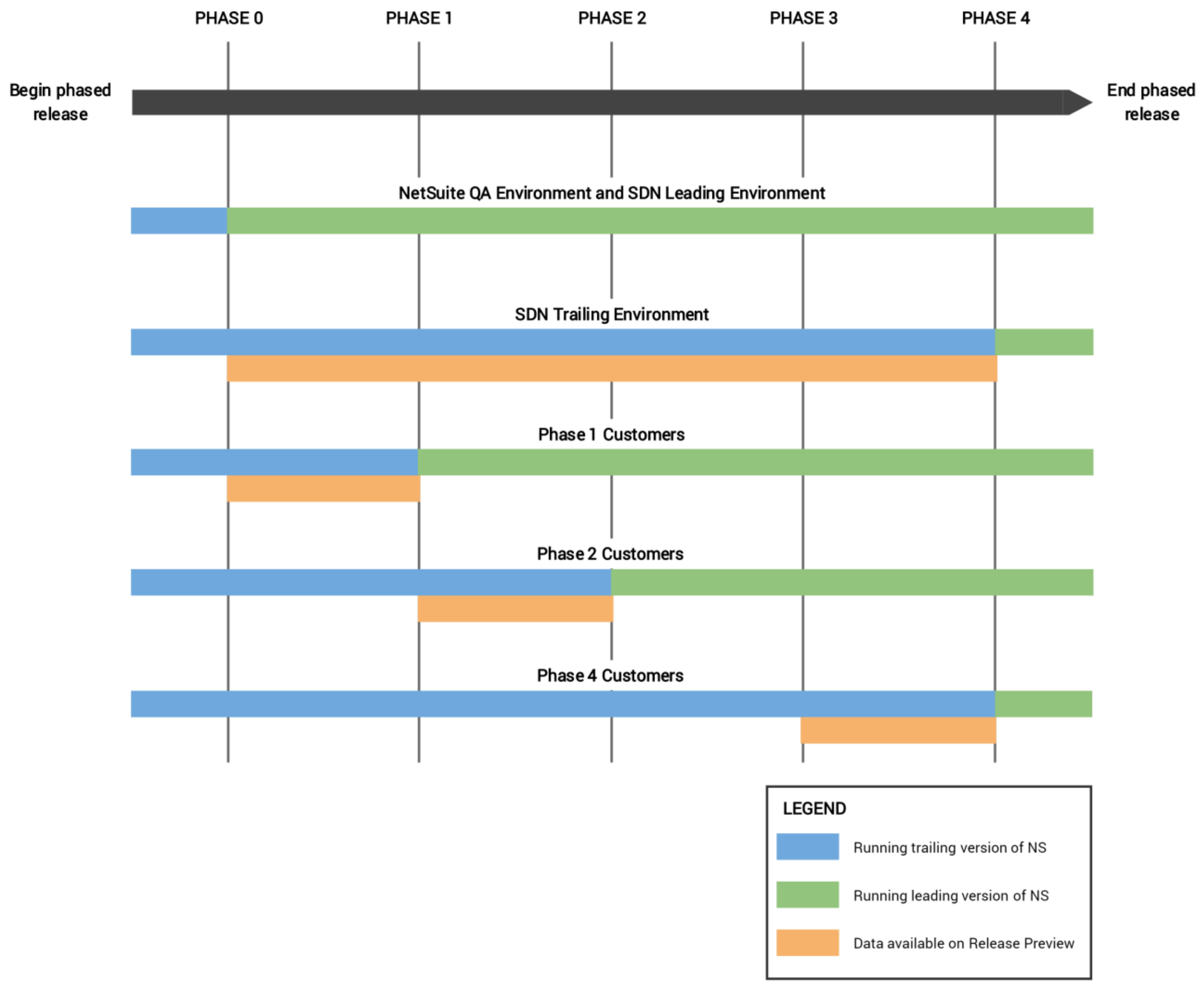
Important: The testing schedule for all SuiteApps should take into consideration NetSuite's model of phased releases. See Understand NetSuite Phased Releases for information below.

6.2 Understand NetSuite Phased Releases

As a cloud-based application, NetSuite's regular maintenance and upgrades are carried out and managed by the NetSuite Release team. NetSuite has two major releases every year, versioned 20xx.1 and 20xx.2, where xx is the year number (for example 2015.1 and 2015.2).

Customers are notified of their upgrade schedules and are asked to test their data in the Release Preview environment (an isolated set of infrastructure with new/leading NetSuite code and the customer's data). After every upgrade, all data are kept intact, and all customization objects are kept and continue to function as before.

Upgrades are implemented in a phased manner from phase 0 to phase 4, with two to three week intervals usually separating the phases. As NetSuite enters phased release, phase 0 is the first batch of NetSuite accounts to be upgraded, and consists of internal QA accounts and SDN leading accounts for testing purposes. Phase 1 is the first batch of live customer accounts to be upgraded; the percentage of customer accounts is small (~1% of total accounts). As the phased release period progresses, the total number of accounts to be upgraded increases. Phase 4 is the final phase, at which point 100% of the NetSuite accounts are upgraded, and NetSuite exits phased release until the next upgrade cycle begins.



NetSuite Release Phases

6.2.1 Phased Release Challenges Brought by SuiteApps

The SuiteCloud platform enables you to build applications based on the NetSuite platform. These SuiteApps are applications unto themselves. They are either external applications that integrate with NetSuite using web services and/or RESTlets, or applications built natively on the platform, or a hybrid of both. Their complexity and unique requirements bring the following challenges to the existing phased release process and testing infrastructure:

- As a SuiteApp developer, you must have early access to the new version of NetSuite in order to test your existing code to ensure it works on the new version of NetSuite.
- You may need early access to new features and APIs for prototyping purposes.
- During phased release, you will have a mix of customers that are spread across different phases.

Some customers have been upgraded while others have not. Therefore, you need to simultaneously support those customers on leading versions of NetSuite and those on trailing versions. SDN partners are contractually obligated to support all customers, regardless of the version of NetSuite the customers are currently using.

- SuiteApps need to reside in a stable deployment platform, and must not be impacted by early bugs on the leading version of NetSuite.

6.3 Leveraging SDN Testing Infrastructure

To tackle the challenges posed by the unique phased testing needs of SuiteApps, SDN provides tools and methodologies for SDN partners. These are collectively referred to as the "SDN Testing Infrastructure." The SDN Testing Infrastructure consists of the SDN Leading Environment, the SDN Trailing Environment, and Extended Access to Release Preview.

6.3.1 SDN Leading Environment

The SDN leading environment consists of servers available to SDN partners at the Select or Premier level upon their request. These leading environments are always upgraded in phase 0 of all NetSuite phased releases. This means SDN partners will always be among the first to have access to the leading version of NetSuite.

During phase 0, only NetSuite internal QA accounts and SDN leading environment accounts are running the leading version, before any live customers are upgraded. This is the first opportunity, and an ideal time, for SDN partners to test their SuiteApps against the leading version of NetSuite to find any potential platform bugs and report them to NetSuite Support.

The phase 0 upgrade phase of the SDN leading environment addresses the need for SDN partners to gain early access to the leading version of NetSuite. Therefore, SDN partners must have at least one SDN leading environment account to use for testing their SuiteApps on the leading version of NetSuite during phased releases. It is a best practice to set up the SDN leading accounts with sufficient test data in order to carry out these SuiteApp tests during phased releases.

Important: Due to potential instability problems in early phased releases, SDN leading environment accounts must not be used for production SuiteApp deployment purposes.

Accounts on the SDN leading environment are permanent and can be used repeatedly. Once NetSuite exits phased release, these accounts do not get purged, they simply run the common version of NetSuite until the next phased release begins again, at which point they are upgraded again in phase 0.

Important: To avoid incompatibility issues, SDN partners should not release SuiteApps that rely on new features and APIs during phased releases.

SDN partners are encouraged to request as many SDN leading environment accounts as they need for testing purposes. They can do this by contacting NetSuite Customer Support and opening support cases.

6.3.2 SDN Trailing Environment

The SDN trailing environment consists of servers dedicated to SDN partners that will always be upgraded in the last phase of all NetSuite phased releases. Accounts in the SDN trailing environment will always be among the last accounts to be upgraded to the leading version of NetSuite.

The trailing version of NetSuite is usually the more stable version during a phased release. This makes the trailing version ideal as a SuiteApp deployment environment. SDN partners are encouraged to use accounts on the SDN trailing environment to build their production SuiteApp deployment chains.

Accounts on the SDN trailing environment are permanent and can be used repeatedly. Once NetSuite exits phased release, these accounts do not get purged. The accounts simply run the common version of NetSuite until the next phased release begins again, at which point the accounts are upgraded again in the last phase.

SDN partners are encouraged to request as many SDN trailing environment accounts as they need for different purposes. They can do this by contacting NetSuite Customer Support and opening support cases.

6.3.3 Extended Access to Release Preview

Since SDN trailing environment accounts are the first set of accounts that are provisioned to SDN partners, and are frequently used as QA accounts, they tend to have more data that can be used for SuiteApp testing purposes. This data may not be available on SDN leading environment accounts, or may take some time to build up. Hence there is a need to test SuiteApps on the leading version of NetSuite against the data on an SDN trailing environment account.

The SDN trailing environment accounts are available on Release Preview during phased releases. At the beginning of a phased release, a snapshot of all the SDN trailing environment accounts are taken (including all data and customization objects), and placed on the Release Preview infrastructure. SDN partners may access the Release Preview environment by logging into https://system.na1.beta.netsuite.comThese accounts remain on Release Preview for the duration of the phased release.

When NetSuite exits phased release, the Release Preview domain is taken offline until the next phased release starts, at which point new snapshots of SDN trailing environment accounts will be placed there at the beginning.

Note that the snapshots of SDN trailing environment accounts are taken only once at the beginning of the phased release. The Release Preview infrastructure is separated from the production infrastructure; hence, the data on Release Preview will remain “stale” for the duration of the phased release.

Important: Even though SDN leading environment accounts are not available on Release Preview, it is recommended they be used as the primary QA environment **during** phased releases because they are more representative of a customer production environment.

6.3.4 Customer Access to Release Preview

Note that customers also have access to Release Preview. Just like SDN trailing environment accounts, snapshots of customers’ production accounts (including the SuiteApps installed) are placed on Release Preview.

However, unlike SDN trailing accounts, customers do not have *extended* access to Release Preview. Their access to Release Preview is approximately two weeks prior to their upgrade date.

Customers should use their access to Release Preview to test their own data and any installed SuiteApps against the leading Release Preview version of NetSuite. It is up to SDN partners to remind and encourage their customers to perform these tests and report platform bugs to both NetSuite Customer Support and to the SDN partner's own support team.

As an SDN partner, you are strongly encouraged to engage with a few of your customers to jointly test your SuiteApps in your customers' Release Preview accounts. Customers' accounts will always be different from your QA account due to configuration, other SuiteApps present, and real-world data from the customer.

6.4 Summary of SDN Testing Infrastructure

The following table summarizes the tools provided by the SDN Testing Infrastructure.

	Release Preview	Leading Environment	Trailing Environment
Upgrade Date	N/A	Phase 0	Phase 4 (last phase)
Primary Use	Testing SuiteApps on your data and customer data	Testing SuiteApps, prototyping new APIs	SuiteApp deployment platform; general QA
Available on Release Preview	N/A	No	Phase 0
Limitations	Data not refreshed; not replicated to production; purged after phase release	None	None
Can I request more?	No	Yes	Yes

SDN Testing Infrastructure Tools

6.5 SuiteApp Testing Methodologies During Phased Releases

As a SuiteApp developer and SDN partner, you are encouraged to follow these best practices and testing methodologies during NetSuite phased releases.

1 Attend the SDN new release webinars.

   a. These webinars take place prior to Phase 0.

   b. The webinars are intended for developers, QA engineers, and release engineers.

2 If necessary, request access to SDN leading environment accounts by opening support cases with NetSuite.

3 Use the Customer Lookup tool provided to SDN partners to look up the exact upgrade date and Release Preview start and end dates for your customers.

   a. The information given by this tool can help you formulate testing schedules.

   b. Your customers will receive emails from NetSuite regarding the start and end dates of their own Release Preview. Remind them to test their installed SuiteApps during that time.

   c. This tool is found in the APC portal.

4 During phase 0, install your SuiteApps in an SDN leading account (if not already done) and test them.

   a. The two to three week window between phase 0 and phase 1 is the ideal time to test because customers are not yet using the new release in production. Therefore, bugs found and fixed during this window of time never get exposed to customers.

   b. Report platform bugs to NetSuite Customer Support.

5 Starting from phase 0, access Release Review and test your SuiteApps against the snapshots of your data on the SDN trailing environment accounts.

   a. You may do this anytime during the phased release, but the sooner, the better.

   b. Optionally, test any new features and/or APIs that you might be interested in using in the future.

6 Throughout the phased release, continue to support existing customers. Install SuiteApps for new customers from your SDN trailing environment accounts.

6.6 QA Checkpoints for SuiteApps

Cloud-based application require some unique QA elements that must be considered by QA engineers and application designers.

The following are QA checkpoints that must be followed for your SuiteApp to be successful in your customers' live environments.

1 Perform role-based testing – Module-based testing and system testing of SuiteApps should be done using the intended role(s).

   Rationale – Using custom roles (where applicable) for data access control adheres to the guidelines that NetSuite recommends to application developers. It is also more representative of how customers will use the applications when the applications are deployed. (Also see Principle 5: Adhere to NetSuite Security

Requirements.)

2 Test during peak hours – Some portions of stress testing of SuiteApps should be done during peak hours defined by NetSuite.

Rationale – Peak hours are a good representative of the amount of server resource and response time available to a SuiteApp. Therefore, it is important to stress test SuiteApps during peak use hours when more users are online and server load is higher.

3 Stress testing - You must test your SuiteApp in an environment and scenarios where I/O throughput, network bandwidth, and data volume exceed your SuiteApp's capability.

4 Rationale - Testing your SuiteApp in the "worst case" scenario helps you determine what its utmost limits are, its expected behavior under these circumstances, and the impact it can have on user experience and data integrity.

5 Set up multiple QA accounts in your SuiteApp deployment environment – Request multiple SDN accounts to use as your QA accounts.

Rationale – NetSuite is a highly customizable application used by thousands of customers. Use multiple QA accounts to ensure your SuiteApp works on different editions of NetSuite and/or with different configurations or modules. For example, a standard NetSuite QA account and a OneWorld QA account may be needed for your SuiteApp.

6 Test the SuiteApp deployment process **–** Before installing/pushing SuiteApp updates to customers, ensure installation/push processes work in your own QA accounts.

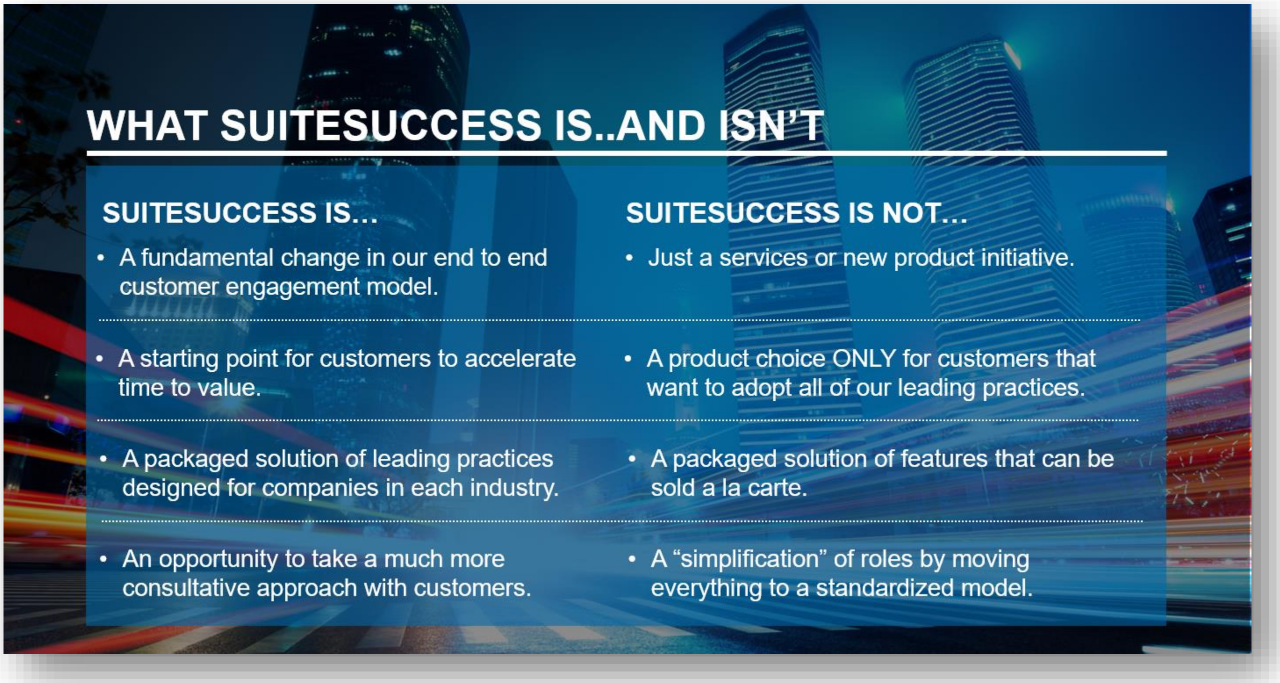
Rationale – Bundle installation scripts are server scripts that execute when a SuiteApp is installed or updated. Sometimes these scripts handle data conversion and cleanup tasks. Therefore, it is imperative that they are tested on QA accounts prior to being executed on customer accounts. This testing is done by installing and updating the SuiteApp in QA accounts.

7 Often the only way to thoroughly test the SuiteApp install/update process, including all bundle installation scripts, is to have multiple QA accounts (see QA checkpoint #4). Also note that SuiteApps may be impacted by bug fixes or feature upgrades associated with the rest of the platform. Therefore, it is important to test the existing update process on QA accounts before pushing updates to live customer sandbox or production accounts.

6.7 SuiteApp Considerations with SuiteSuccess

6.7.1 The SuiteSuccess Initiative

SuiteSuccess is a NetSuite strategic initiative to deliver a verticalized cloud solution by combining the NetSuite unified suite, leading industry practices derived from thousands of earlier customer engagements, and an innovative customer engagement model. SuiteSuccess encompasses sales and service and support staff guiding customers on how they should be using NetSuite for their business, to the out-of-the-box customizations optimized for specific vertical industries.



WHAT SUITESUCCESS IS..AND ISN'T

SUITESUCCESS IS…	SUITESUCCESS IS NOT…
• A fundamental change in our end to end customer engagement model.	• Just a services or new product initiative.
• A starting point for customers to accelerate time to value.	• A product choice ONLY for customers that want to adopt all of our leading practices.
• A packaged solution of leading practices designed for companies in each industry.	• A packaged solution of features that can be sold a la carte.
• An opportunity to take a much more consultative approach with customers.	• A "simplification" of roles by moving everything to a standardized model.

From the customers' perspective, their SuiteSuccess experience will be one of prescriptive selling, rapid deployment, improved ROI enabled by well-honed vertical best practices, and harmonious operations in their third party SuiteApps. When the program is completely rolled out, all net-new customers will be engaged, implemented and supported using the SuiteSuccess model.

Partner SuiteApps have been integral to customers' success in using NetSuite. These SuiteApps will be equally important for SuiteSuccess customers. Therefore, it is imperative that SDN partner SuiteApps are tested within SuiteSuccess environments. Partners will be required to verify their SuiteApps' compatibility with SuiteSuccess during their Built for NetSuite badging and renewal process in every BFN review cycle.

6.7.2 Requesting SuiteSuccess Accounts

On the product and technology side, SuiteSuccess is delivered as a set of SuiteBundles that are preinstalled on customer accounts. These bundles are vertical specific, which means there are distinct SuiteSuccess SuiteBundles for every key vertical that NetSuite supports.

SDN test accounts with specific SuiteSuccess vertical SuiteBundles pre-installed will be made available once SuiteSuccess is rolled out to all SDN partners (ETA to be determined). The process to request them is the same as when requesting any SDN accounts – via opening support cases.

Note that these SuiteSuccess accounts are strictly meant to be used for testing your SuiteApps and for sales demo purposes. They must not be used as development or deployment accounts. Since the availability of these accounts may be limited, and partner access to them may only be temporary, it is advisable to record your demos on them for future sales activities.

6.7.3 Compatibility Testing with SuiteSuccess

SuiteSuccess will be ramped up gradually to almost all new NetSuite customers. Retrofitting existing customer accounts to SuiteSuccess is also possible. Therefore, it is imperative that SDN partner SuiteApps be compatibility tested with SuiteSuccess due to its prevalence in the very near future.

6.7.4 Rules on Integrating with SuiteSuccess Objects

Unless specifically requested by NetSuite, a SDN partner SuiteApp must not integrate directly with objects contained in any SuiteSuccess SuiteBundles.

Important: This means the following actions must not be done by a SuiteApp author when developing and deploying a SuiteApp:

1. Write to any custom fields contained in the SuiteSuccess test account
2. Create any "rows" of custom records that are defined in the SuiteSuccess test account
3. Create any customization objects that reference object(s) contained in a SuiteSuccess test account
4. Intentionally, or unintentionally, include any objects in a SuiteBundle that originate from the SuiteSuccess test account

These requirements are driven by the following two reasons:

1. At the time of this document's writing, there are no supported APIs for SuiteSuccess customizations in any of the vertical versions
2. When a customization object that references a SuiteSuccess object is included in a SuiteApp bundle, the SuiteBundler will include the referenced object to maintain referential integrity; this will cause data issues when the SuiteBundler's conflict resolution logic runs on the customer account at the time that the partner SuiteApp is installed.

Important: **ONLY** upon Request from NetSuite, some partners will be asked to evaluate integration directly with SuiteSuccess. The SuiteCloud Development Framework (SDF) will be the deployment tool of choice for those partners. More details will be shared on the deployment methodology separately.

6.7.5 Testing on SDN Accounts

When ready to begin testing within SuiteSuccess test accounts, partners must choose to test their SuiteApps on all the relevant SuiteSuccess vertical(s). Since these SuiteApps are prohibited from direct integration with any SuiteSuccess objects, the primary goal of the testing in the SuiteSuccess account is to ensure there are no conflicts between any objects in the new partner SuiteApp and the objects in combined NetSuite and SuiteSuccess system.

When a new SDN account with is provisioned to you with a specific set of SuiteSuccess customizations, your SuiteApp bundle must be installed on it from your deployment account. Any object conflicts between SuiteSuccess and your SuiteApp should be treated as defects in your SuiteApp. If the SuiteBundler detects any conflicts on the objects during installation, you should halt the installation and resolve those conflicts on your development account. Use the established SDN SuiteApp QA and deployment infrastructure and methodology to push these fixes out.

If there are no object conflicts between your SuiteApp and the SuiteSuccess vertical in use at the time, then you may perform a basic set of tests of your liking to ensure your SuiteApp performs correctly.

If you have an Integrated SuiteApp that does not contain a SuiteBundle, then there is no risk of object conflicts introduced by the SuiteBundler. However, the "no data-write" rules listed in section 3.1 are still applicable.

6.8 What are NetSuite Sandbox Environments?

NetSuite sandbox accounts contain a replica of the configuration, customization and data from a live production account as of a specific date, but does not process external transactions such as payments or email campaigns. Their isolated from production use make them ideal environments for developing customizations for a company's own internal use, testing SDN partner SuiteApps against their business data, and for user acceptance testing. They are NOT meant for SDN partners to develop their SuiteApps or for general QA of these SuiteApps because they contain real customer business data. Therefore, **sandbox environments are not provisioned to SDN partners.**

Sandbox accounts are not a standard feature. Customers should contact their NetSuite sales representative if they want to purchase sandbox accounts.

SDN partners are not provisioned sandbox accounts. However, there are two scenarios where a SDN partner may use a sandbox:

1) A customer is testing a SDN partner SuiteApp on their sandbox account prior to rolling it out to production; and the SDN partner was given access to it
2) The SDN partner is a NetSuite user itself, and has its own sandbox account(s)

6.8.1 Key New Sandbox Changes

In the past, sandbox accounts were hosted in an isolated environment with its own domain (system.sandbox.netsuite.com), and with a separate login page (https://system.sandbox.netsuite.com/pages/customerlogin.jsp). All the integration endpoints (RESTlets and SuiteTalk web services) were also hosted in the sandbox domain, separate from their production counterpart.

The new sandbox is hosted in the same environment as production accounts, using the same login page and same domain (system.netsuite.com). All the integration endpoints are also in the production domain.

Since a production account and its sandbox account(s) are hosted in the same environment, and can be accessed in the same Change Role page, they have different account IDs as the differentiating attribute. Sandbox accounts have a "_SBx" suffice, where x is the sandbox number. For example, if a production account with account ID 123456 has two sandbox accounts, then their account IDs will be 123456_SB1 and 123456_SB2.

6.8.2 Building your Integrated SuiteApp to Support Sandbox Accounts

The old dedicated sandbox domain will be retired in the future (including the associated login page, web services endpoint URLs, RESTlet endpoints). To support new sandbox accounts, integrated SuiteApps need to point to the production domain. An integration SuiteApp needs to specify the sandbox account ID when making SuiteTalk API calls or when invoking RESTlets.

In essence, accessing a new sandbox account using SuiteTalk or RESTlets is very similar to accessing a production account using these integration interfaces. This means all the best practices and guidelines, such as the support for multi-datacenter hosting (see chapter 1), also apply to sandbox accounts. There are no changes to the actual APIs. The SuiteScript API to return the runtime environment also works correctly in the new sandbox with no changes needed.

7 Consider Deploying Your SuiteApps as Managed Bundles

Wide-scale ISV SuiteApp deployments are best done with managed bundles. The Managed Bundles feature allows you to turn your SuiteApps into true cloud-based applications, where updates are pushed to the install base without actions from users or administrators.

Note: For more information on managed bundles, see *Understanding Managed Bundles* in the NetSuite Help Center.

The Managed Bundles feature is available only to Select and Premier SDN partners with active statuses in the SDN program. To ensure the Managed Bundles feature is activated on deployment environments that are robust and easily supportable, SDN partners must deploy their SuiteApps using the Deployment Account Method, which consists of a development account and a deployment account. Additionally, a separate QA account must also be part of the bundle deployment environment. To ensure SuiteApps that can be deployed in a managed fashion are of sound quality, they must have achieved the latest Built for NetSuite validation. Note that development accounts and deployment accounts must be SDN trailing accounts.

This capability to actively push updates is particularly important for those SuiteApps that provide compliance and regulatory features to NetSuite. Some examples include SuiteApps that provide localization to international customers and SuiteApps that support time-specific functionality such as tax calculations.

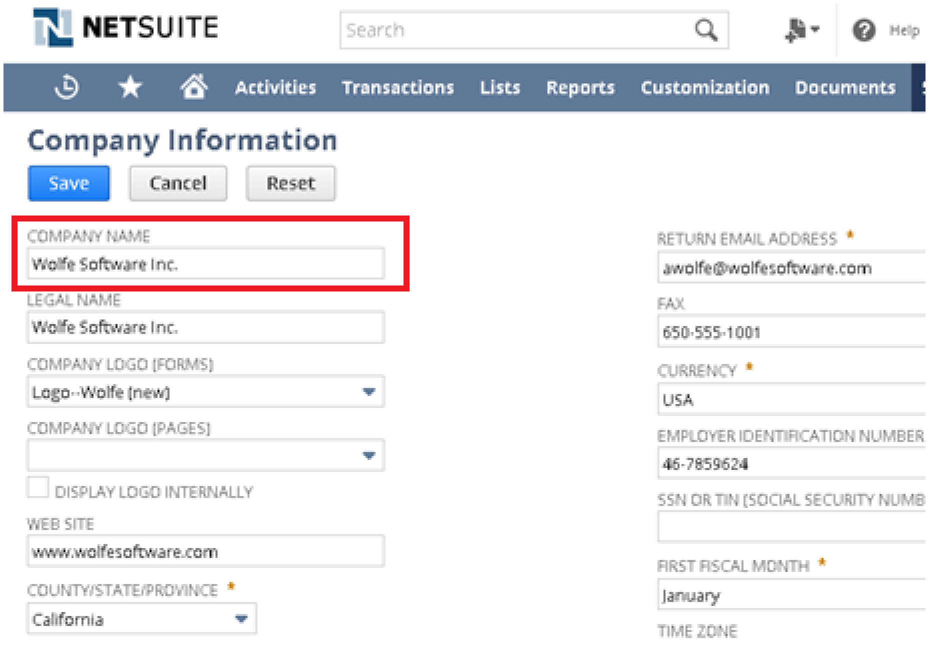
Note: For more information on the Deployment Account Method, see *Deployment Account Method* in the NetSuite Help Center. In this guide, see *Leveraging SDN Testing Infrastructure* for information on testing in deployment accounts.

NetSuite enables the Managed Bundles feature only on deployment accounts for those SuiteApps that have achieved the latest Built for NetSuite validation. To request the Managed Bundles feature, or to request more SDN test accounts, please open a support case.

8 Maintain Your SuiteApps

SuiteBundler allows SuiteApp developers to quickly and easily deploy applications to multiple NetSuite accounts.

When a bundle is created, there are a number of attribute fields that can be populated including Name, Version, Abstract and Description. SuiteBundler only enforces the Name field to be mandatory. When a bundle is positioned for general ly available (GA) release, it must also have the Version, Abstract, and Description fields populated. The deployment accounts of these GA bundles must also be named with the bundle provider's company name. An account's Company Name setting can be changed at Set Up → Company → Company Information as shown below:



Company Information

COMPANY NAME
Wolfe Software Inc.

LEGAL NAME
Wolfe Software Inc.

COMPANY LOGO (FORMS)
Logo--Wolfe (new)

COMPANY LOGO (PAGES)

□ DISPLAY LOGO INTERNALLY

WEB SITE
www.wolfesoftware.com

COUNTY/STATE/PROVINCE *
California

RETURN EMAIL ADDRESS *
awolfe@wolfesoftware.com

FAX
650-555-1001

CURRENCY *
USA

EMPLOYER IDENTIFICATION NUMBER
46-7859624

SSN OR TIN (SOCIAL SECURITY NUMB

FIRST FISCAL MONTH *
January

TIME ZONE

The following are additional bundle naming convention requirements:

- The name of the bundle should describe the feature or functionality delivered
- Do not include the words "Bundle" or "SuiteApp" in the bundle's Name field
- Use descriptive and consistent names, for example, Japan Tax Reports
- Indicate non-GA bundles accordingly, for example, add "BETA" to the end of the name

Using SuiteBundler, developers can build application deployment environments with the following attributes, which are unique to the cloud paradigm:

1 Multi-tenant structures where a deployment environment can serve multiple customers.

2 Managed Bundles feature that provides automated application upgrades without customer actions. (See *Understanding Managed Bundles* in the NetSuite Help Center for more information about this feature.)

3 Deployment account architecture that allows customer support teams to easily provide technical support to customers on the current code base. (See *Deployment Account Method* in the NetSuite Help Center for more information about this feature.)

4 Copy-Deprecate capability that allows development teams to maintain separate code streams for developing future versions of SuiteApps. (See *Bundle Deployment Using Copy and Deprecate* in the NetSuite Help Center for more information about this feature.)

Due to its complexity and its need for robustness, a SuiteApp deployment environment that enables ISVs to deploy applications to multiple customers is one that requires maintenance.

Therefore, it is important for ISVs to recognize that a release engineer must have knowledge of SuiteBundler, the SuiteApps themselves, and cloud application distribution practices. Knowledge in each of these areas is required for the ongoing operation and maintenance of a SuiteApp deployment environment.

SuiteApp development teams should review the following sections to learn more about SuiteApp deployment environments, SuiteApp object mappings, deployment “chains,” how these chains can be broken, and how they can be repaired.

- Setting Up the SuiteApp Deployment Environment
- SuiteBundler Object Mapping
- Object Mapping Breakages
- Repairing Broken Object Mappings

8.1 Setting Up the SuiteApp Deployment Environment

This section provides an overview of the SuiteApp deployment environment model.

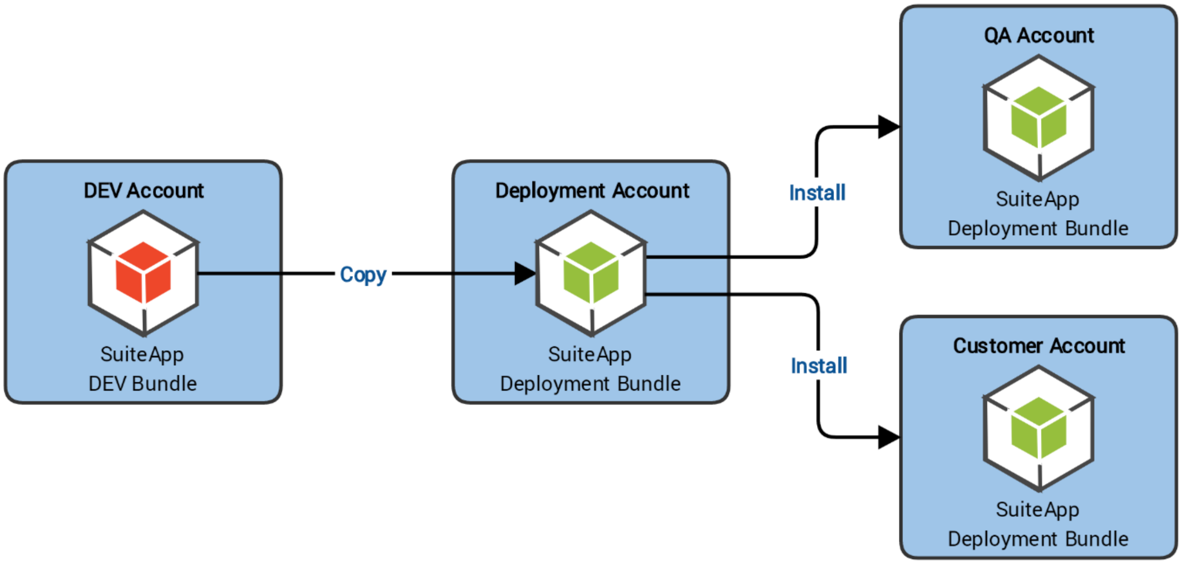
Note: For additional details on SuiteApp deployment environments, see *Bundle Deployment Using Copy and Deprecate* in the NetSuite Help Center.

In a typical ISV SuiteApp deployment environment, at least four accounts are involved:

- the development account
- the deployment account
- the QA account
- the customer account(s)

The SuiteApp author (the ISV) controls the development, deployment, and QA accounts. The NetSuite account administrators control their respective customer accounts.

A SuiteApp is copied from the development account to the deployment account. This copy action creates a SuiteApp deployment bundle in the deployment account. The SuiteApp deployment bundle is then installed in the QA account for testing purposes. Finally, after the SuiteApp is fully tested, the SuiteApp deployment bundle is installed into customer accounts from the deployment account. The following diagram illustrates this model.



SuiteApp Deployment Environment Model

8.2 SuiteBundler Object Mapping

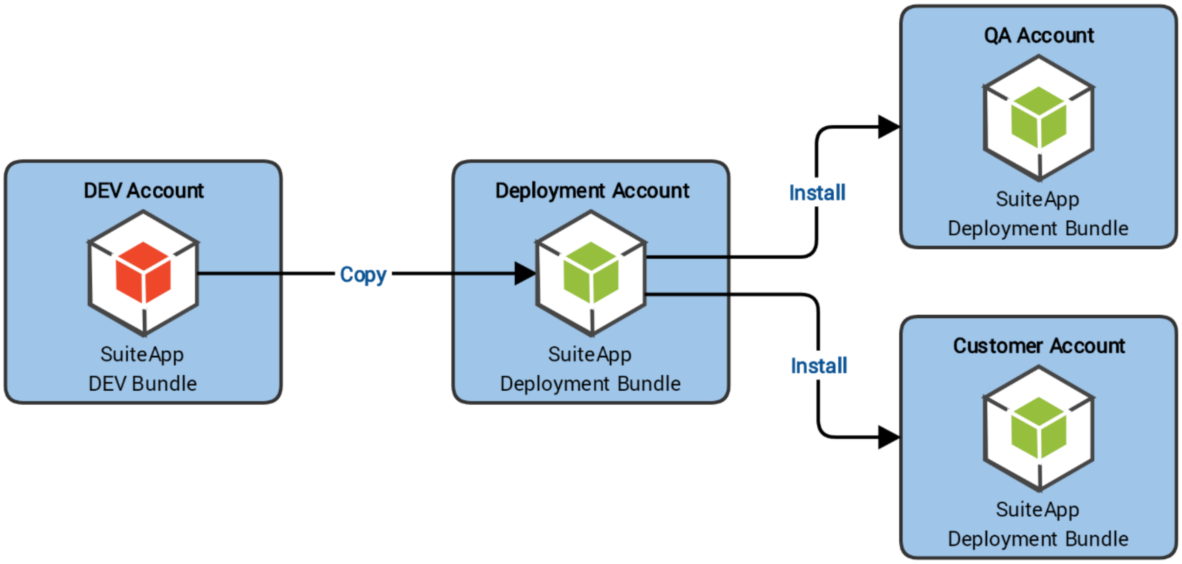
To understand how a functioning SuiteApp deployment chain can be broken, it is important to understand how SuiteBundler object mapping works.

When a SuiteApp is created, customization objects from the account where it is created are put into the SuiteApp. When this bundle is copied or installed in another account, these customizations from the original account to a target account are copied into the new account.

In order for SuiteBundler to update the installed/copied bundle with the latest changes made in the original source account, SuiteBundler needs to keep track of the mappings between the objects in the two accounts.

For every bundled object installed or copied in the target account, SuiteBundler creates a mapping between it and the object in the original source account. Internal IDs are used as keys for these object mappings. These object mappings are maintained “behind the scenes” by SuiteBundler and are not accessible to users and developers.

The following diagram shows the relationships between a development account customization object, deployment account customization object, SuiteBundler, and object mappings. For simplicity, the QA account and customer accounts are not shown.



SuiteApp Object Mappings

With these object mappings in place, the following SuiteBundler tasks can be accomplished:

1 When bundle recopying (or installed bundle updates) is performed, SuiteBundler relies on the object mappings to correctly update the deployment account objects. In the diagram above, custom record A with internal ID 456 on the deployment account gets its updates from custom record A with internal ID 123 from the development account.

2 When custom record A on the deployment account is deleted (perhaps inadvertently), SuiteBundler uses the object mapping to look for its definition on the development account, then rebuilds it on the deployment account when the bundle is recopied.

3 When custom record A on the development account is removed from the bundle, SuiteBundler uses the object mapping to remove the record from the deployment account when the bundle is recopied or updated.

8.3 Object Mapping Breakages

As stated in SuiteBundler Object Mapping, SuiteBundler can remove objects from a SuiteApp when objects become obsolete or are no longer needed. Objects are removed by removing them from the SuiteApp, or by deleting the objects from the source account.

Upon the next update on the installed/copied bundle on the target account, the object mappings are updated; mappings for objects deleted on the source account will be removed. Objects are then subsequently deleted from the target account.

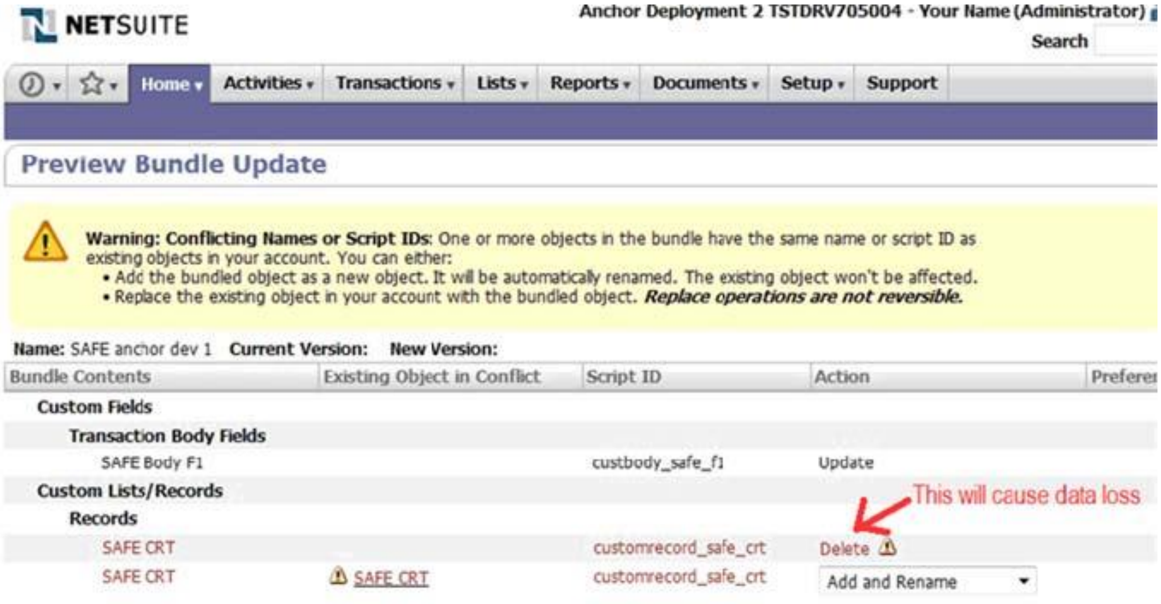
Problems arise when objects that store data (custom record types and custom fields of all types) are inadvertently deleted from the source account. This deletion can happen when the source account is a development account that is actively used by SuiteScript developers and/or developers creating custom records and fields. When an object is inadvertently deleted from the source account, the person who deleted the object usually realizes the mistake quickly and adds the object back by creating a new, identical one. The person also adds the new object back into the SuiteApp. However, this practice creates an underlying problem, which only surfaces when target accounts update their SuiteApp or when updates to a copy of the SuiteApp occur.

Note: If the object inadvertently deleted is not a data-storing object (for example, SuiteScript, saved searches, dashboards, and so on), then replacing it with a new identical one will not cause problems.

When a developer creates a new custom field or custom record type to replace one that was accidentally deleted, the developer is essentially creating a brand new object. Even though the object appears to be identical to the one deleted, it is still a different object with a different internal ID. Since SuiteBundler object mappings rely on internal IDs to link objects between accounts, once a customization object is deleted from the source account, SuiteBundler will sever the mapping the next time the installed SuiteApp is updated or recopied.

When the target account updates the installed/copied bundle, SuiteBundler will see that the original object was removed, therefore it will delete its counterpart on the target account. If the object is a custom field or custom record type, it means the data stored will be lost. SuiteBundler will then attempt to add the new replacement object back. However, SuiteBundler will not populate the replacement object with the lost data because it is viewed as a different object.

The following screenshot shows the Preview Bundle Update page when SuiteBundler is about to delete a custom record and then add it back. The “Delete” action is a cue that data loss is about to occur. At this point, the user must stop the bundle update to avoid losing data.



Warning: Conflicting Names or Script IDs: One or more objects in the bundle have the same name or script ID as existing objects in your account. You can either:

- Add the bundled object as a new object. It will be automatically renamed. The existing object won't be affected.
- Replace the existing object in your account with the bundled object. ***Replace operations are not reversible.***

Name: SAFE anchor dev 1 **Current Version:** **New Version:**

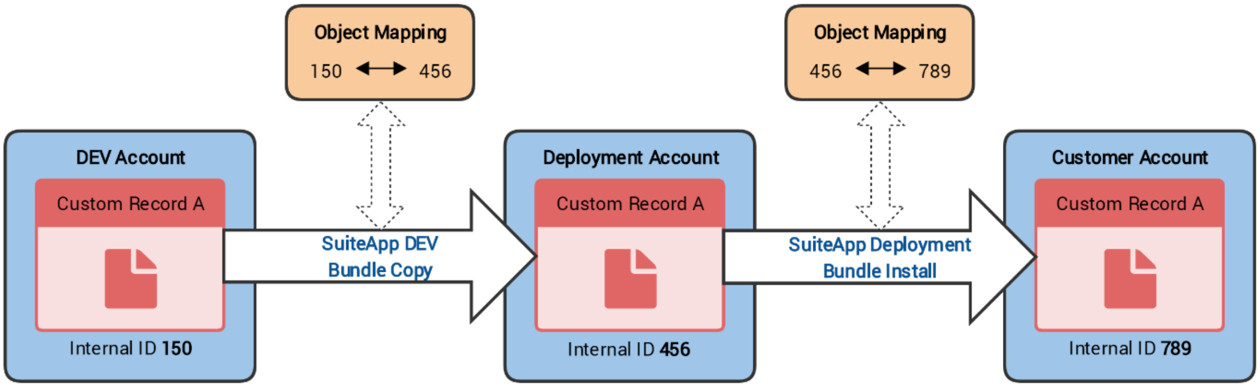
Bundle Contents	Existing Object in Conflict	Script ID	Action	Preferer
Custom Fields				
Transaction Body Fields				
SAFE Body F1		custbody_safe_f1	Update	
Custom Lists/Records				
Records				
SAFE CRT		customrecord_safe_crt	Delete ⚠	
SAFE CRT	⚠ SAFE CRT	customrecord_safe_crt	Add and Rename ▾	

8.4 Repairing Broken Object Mappings

The remedy for repairing broken object mappings is a combination of both process and SuiteBundler techniques. Generally speaking, development teams should **not** delete custom fields and custom record types from SuiteApp source accounts (usually development or deployment accounts) unless these teams truly intend to remove these custom objects from a SuiteApp.

In the event that these objects are inadvertently deleted, the objects should be replaced by an identical copy (with the same name and same script ID). Additionally, SuiteApp engineers and release engineers should be notified immediately if any custom objects are inadvertently deleted. SuiteApp/release engineers must add new (replacement) objects back into the SuiteApp and formulate a plan for repairing broken object mappings.

Note: See *Deployment Account Method* in the NetSuite Help Center for more information on the SuiteApp deployment chain.



Typical SuiteApp Deployment Chain

As the diagram shows:

- Development work is done in the development account.
- The completed SuiteApp is created in the development account and copied to the deployment account. (For simplicity, the QA account is not shown in the diagram.)
- A SuiteApp is created in the deployment account and can be installed from there into customer account(s). Object mappings are created behind the scenes to link the objects between the accounts.

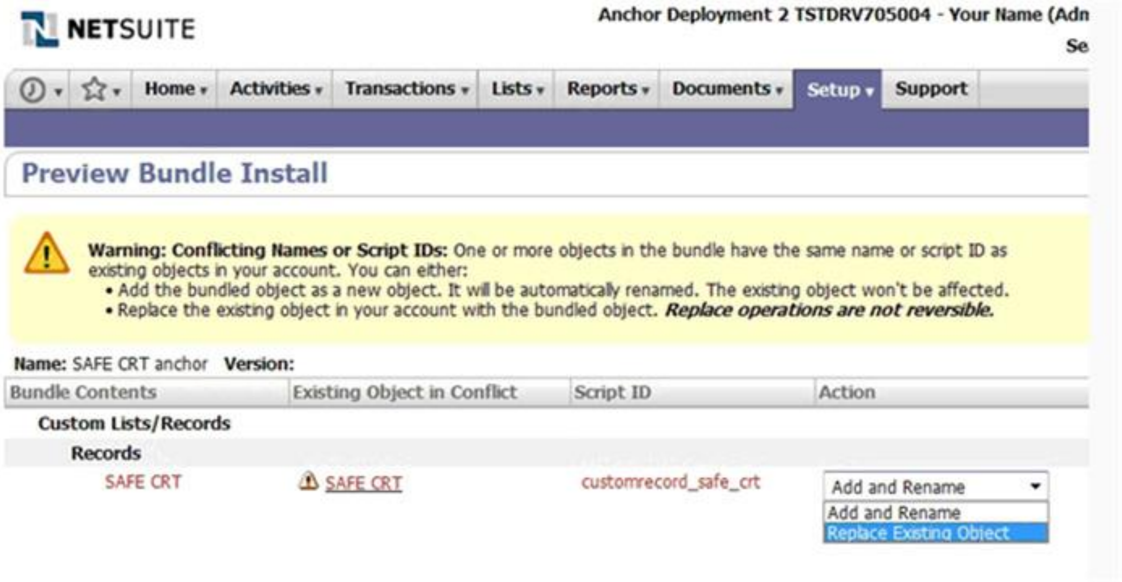
If custom record A on the development account is deleted and a replacement added, custom record A on the deployment account will be removed (and data lost) once the bundle is recopied and a replacement object is brought in. Since the custom record A replacement in the deployment account is also a new object (and will not have internal ID 456), the downstream effect occurs when a customer updates an installed SuiteApp. In this case, custom record A of that SuiteApp will be deleted and replaced with a new one. As a consequence, data is lost along the way.

To prevent data loss, object mapping needs to be kept intact and must be repaired to point to the new replacement custom record A in the development account.

Development teams must take the following steps to repair object mappings:

- Add any new replacement objects into the SuiteApp in the development account.
- Create a new SuiteApp in the development account that contains only the replacement object(s). This SuiteApp is called the "anchor SuiteApp" because it will anchor the impacted objects throughout the deployment chain and prevent them from getting deleted.
- Install the anchor SuiteApp into the deployment account. During the installation process, SuiteBundler will detect if there is a conflict between custom record A in the deployment account and the incoming custom record A from the anchor SuiteApp.

On the Preview Bundle Install page (see screenshot below), choose the Replace Existing Object option for those objects you want to protect.



Preview Bundle Install

Warning: Conflicting Names or Script IDs: One or more objects in the bundle have the same name or script ID as existing objects in your account. You can either:

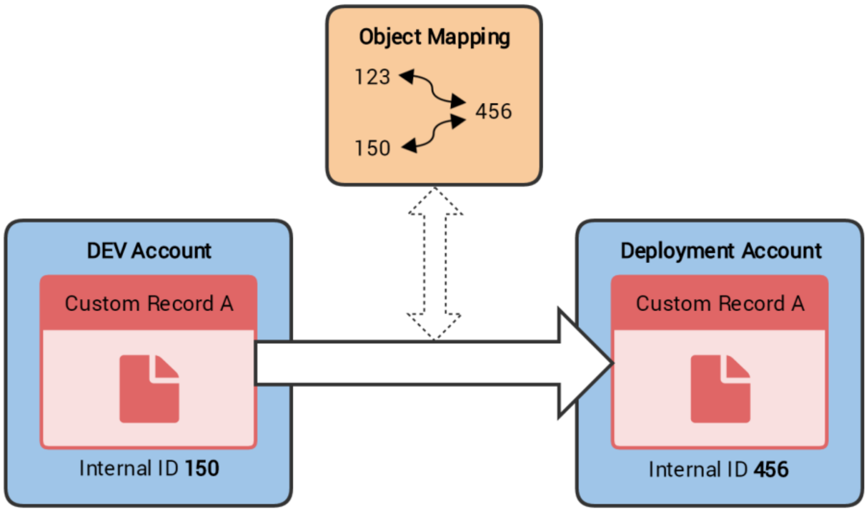
- Add the bundled object as a new object. It will be automatically renamed. The existing object won't be affected.
- Replace the existing object in your account with the bundled object. ***Replace operations are not reversible.***

Name: SAFE CRT anchor **Version:**

Bundle Contents	Existing Object in Conflict	Script ID	Action
Custom Lists/Records			
Records			
SAFE CRT	⚠ SAFE CRT	customrecord_safe_crt	Add and Rename Add and Rename Replace Existing Object

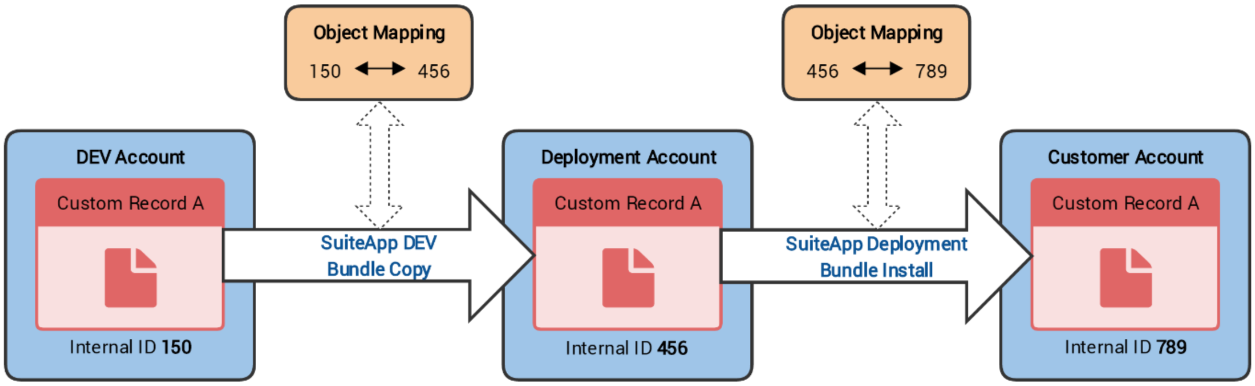
After the bundle is installed, the protected objects will be anchored and will not be deleted by SuiteBundler. At this point, the custom record type "SAFE CRT" (shown in the screenshot above) has two object mappings: one that links the custom record type to the original development SuiteApp, and another that links it to the anchor SuiteApp.

In the following diagram, the 123-to-456 linkage denotes the original object mapping. The 150-to-456 linkage denotes the new mapping between the deployment account and the anchor bundle. The two object mappings protect each other from being deleted by SuiteBundler and prevent data loss.



Original and New Object Mappings

- Once custom record A on the deployment account is protected by the double object mappings, the original Dev SuiteApp may be recopied from the development account to the deployment account. Two things will happen:
    - The mapping between 123 and 456 will be replaced with a new, functioning mapping between 150 and 456.
    - The 150 to 456 mapping introduced by the anchor SuiteApp installation will prevent SuiteBundler from deleting 456, which in turns keeps the data intact.
- The release engineer must then verify that everything is correct and no data loss has occurred.
- Finally, the anchor SuiteApp can be uninstalled from the deployment account. After this process, the deployment chain will look like the following.



Repaired Object Mappings

Because custom record A with internal ID 456 on the deployment account was never deleted, the object mapping between 456 and 789 does not need to be repaired. This object mapping repair process, if followed correctly, does not impact customers. Note that this process also works for managed bundles.

9 Agreements and Licensing

This section provides guidance on important concepts that are necessary to protect your ownership interest in your SuiteApps and to prevent compromise of the confidentiality of your products, services, and development efforts.

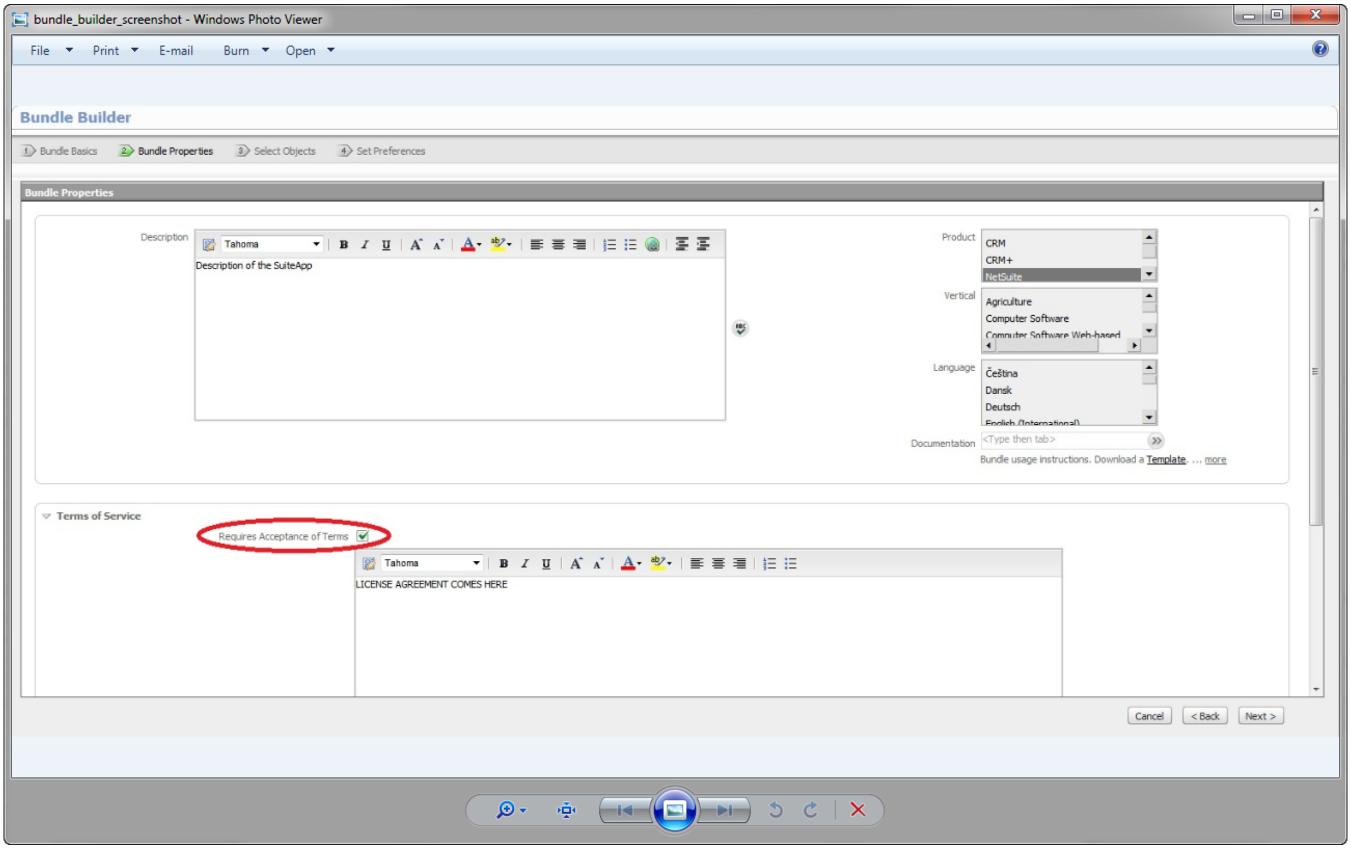
9.1 Agreements with Employees, Consultants, Customers, and Partners

You should carefully protect your intellectual property in your agreements with employees, consultants, customers, and partners. You must have the right to permit NetSuite to make the SuiteApp available to customers through the SuiteApp repository. You should have appropriate agreements with your employees and consultants which include proper (i) grants of ownership and assignment of intellectual property rights in developments, and (ii) confidentiality obligations. Failure to have the appropriate agreements in place with your employees, consultants, customers, or partners may jeopardize your ownership interest in and to the SuiteApp and compromise the confidentiality of your products, services, and development efforts. You should carefully draft and have in place appropriate agreements with your customers and your consultants. Your customer agreement for use of your SuiteApp must describe your privacy commitments to your customers and how you and your SuiteApp collect and use customer data. In addition, you must provide a link to any privacy policy which is applicable to your customers.

NetSuite has included sample customer agreements on the partner portal (you will need your partner account credentials to log in to the partner portal) for your convenience: a sample license agreement for the use of your SuiteApp and a sample professional services agreement for when you are providing implementation services for your SuiteApp to a customer. NetSuite strongly recommends that you solicit advice from experienced, independent counsel with expertise in intellectual property and licensing, including advice on how to maintain ownership of your developments. If you have employees and consultants in multiple jurisdictions, you may wish to obtain legal advice from counsel with expertise in the laws of the specific jurisdiction of your employee or consultant.

9.2 Bundling a Click-through Agreement in your SuiteApp

NetSuite’s SuiteCloud technologies enable you to include a click-through agreement setting forth the terms and conditions for use of your SuiteApp in your SuiteApp. You must include an agreement with your SuiteApp or default terms regarding bundles set forth in the SuiteCloud Terms of Service will govern a customer’s use of your SuiteApp. Once you have the click-through agreement content ready, you can add the agreement while packaging your SuiteApp using the Bundle Builder. You have the ability to configure the SuiteApp such that users can install the SuiteApp only after they agree to the license terms and conditions presented in the click-through agreement.



Description: Description of the SuiteApp

Product: CRM, CRM+, NetSuite

Vertical: Agriculture, Computer Software, Computer Software Web-based

Language: Čeština, Dansk, Deutsch, English (International)

Documentation: <Type then tab>

Bundle usage instructions. Download a Template. ... more

Terms of Service

Requires Acceptance of Terms ☑

LICENSE AGREEMENT COMES HERE

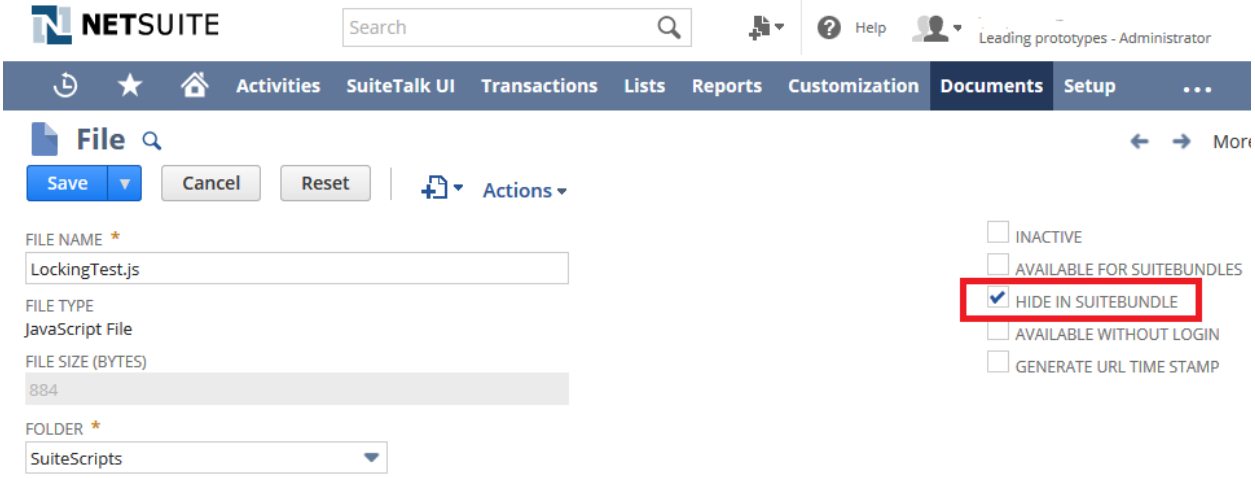
9.3 Protecting the Intellectual Property in your SuiteApps

The key focus of this chapter so far has been to help developers build SuiteApps that securely handle business data. Generally, these best practices apply to the business data and privacy of information for the companies and end users that use the SuiteApps. However, Developer Partners should also consider available options to help protect their intellectual property (IP) that might be contained within the SuiteApp. Developer Partners are solely responsible for controlling access to, and the accessible attributes of, their SuiteApps, as well as the terms governing access and use.

9.3.1 Securing SuiteBundle Content

Developer Partners should consider their IP that might be contained in a SuiteApp, including any SuiteScript source code, the definition of custom fields and custom record types, saved searches, etc.

To control access to your SuiteScript source code, enable the “Hide in SuiteBundle” setting. With this setting enabled, users of the target accounts for this SuiteBundle will not be able to view the source code. This setting can be found in the “edit” page of the source code file in the File Cabinet (see screenshot below).



FILE NAME *
LockingTest.js

FILE TYPE
JavaScript File

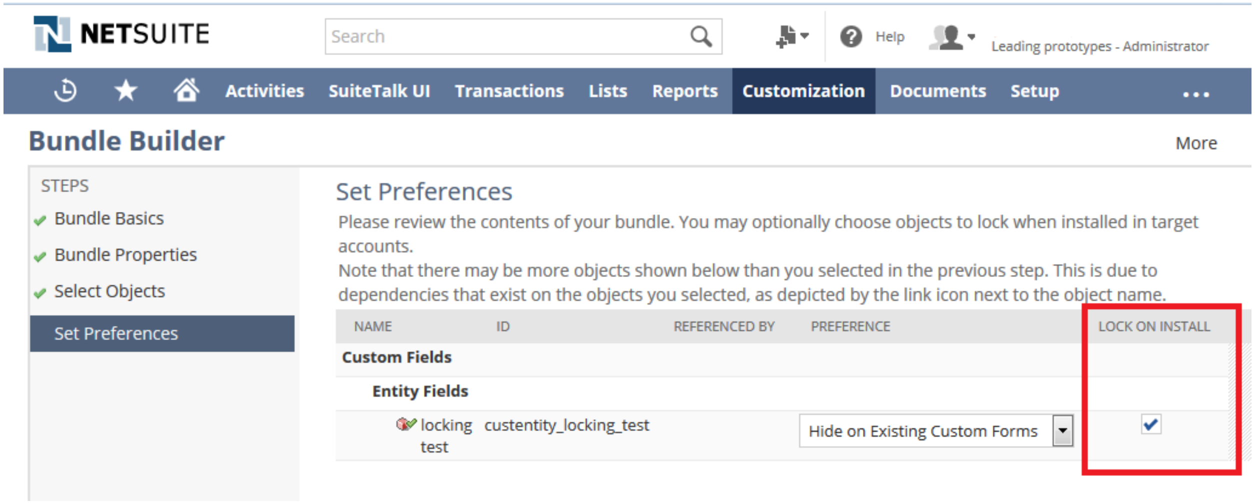
FILE SIZE (BYTES)
884

FOLDER *
SuiteScripts

□ INACTIVE
□ AVAILABLE FOR SUITEBUNDLES
☑ HIDE IN SUITEBUNDLE
□ AVAILABLE WITHOUT LOGIN
□ GENERATE URL TIME STAMP

Note that the "Hide in SuiteBundle" setting should only be enabled on server side SuiteScript source code files. Client-side SuiteScript files with source code files configured to be hidden in SuiteBundle will not operate because browsers will not be able to download them into their runtime virtual machines – this limitation is outside of NetSuite's control. As such, Partner [developers] should understand that client-side SuiteScript files will be accessible in the target account and viewable in the client-side browser.

The definition of non-SuiteScript customization objects in SuiteBundles such as custom fields cannot be hidden from target accounts. However, their definitions may be locked in order to prevent intentional or unintentional modification. Locking customization objects in bundles ensure they perform predictably, and future bundle upgrades are backward compatible. You may do so using the "Lock on Install" settings found in the Bundle Builder's Set Preferences page (see screenshot below).



Set Preferences

Please review the contents of your bundle. You may optionally choose objects to lock when installed in target accounts.
Note that there may be more objects shown below than you selected in the previous step. This is due to dependencies that exist on the objects you selected, as depicted by the link icon next to the object name.

NAME	ID	REFERENCED BY	PREFERENCE	LOCK ON INSTALL
Custom Fields				
Entity Fields				
locking test	custentity_locking_test		Hide on Existing Custom Forms	☑

If you are using a dedicated development account and a dedicated deployment account, and deploy your SuiteBundle using the bundle copy action, then these settings should be enabled on the deployment SuiteBundle on the deployment account. Please refer to section 8.1 for details on this deployment methodology.

Enabling the “Hide in SuiteBundle” setting in JavaScript source code files controls access to and helps protect your IP; setting the “Lock on Install” preference limits the unwanted modification of your SuiteApps by target account users. Therefore, NetSuite encourages the use of these features in deployment bundles.

9.3.2 Redistribution of SuiteBundle Components

In the enterprise, customer best practice is to test the interaction of all types of SuiteApps and internal customizations, in Sandbox before moving them production. This is increasingly important as customers grow and compliance concerns drive them to document processes, increase diligence and formalize testing for internal SuiteApps. In order to enable this, the SuiteBundler allows for the redistribution of a SuiteBundle or its components, which allows customers to move packages of functionality between production accounts or from sandbox to production.

Developer Partners need to be aware that this feature may also pose risks of their SuiteApps being distributed outside the control of the original developer, especially if best practices are not followed. This SuiteBundler behavior can present risks to your SuiteApp or its components to be distributed without your consent and authorization, but the following best practices can help minimized or neutralize these risks:

[[In order to enable this, SuiteBundler provides the ability for a user to create a SuiteBundle which includes components installed into their account in order to move packages of functionality between production accounts or from sandbox to production. All Developer Partners need to be aware that this feature does not limit what components can be included in a SuiteBundle, nor does it limit the accounts into which the SuiteBundle can be shared. As such, components of a Developer Partner’s SuiteBundle could be shared beyond the target accounts into which they have initially been shared by that Developer Partner. Therefore, it is incumbent upon the Developer Partner to take appropriate steps to manage the access, accessible attributes and terms of use of its SuiteBundles including:

a. Use of a click-through agreement (see section 9.2) with appropriate terms of use and restrictions.

b. Enable the “Hide in SuiteBundle” setting in your SuiteScript source code files. This action will hide your code-level IP as described in the previous section.

c. Name your SuiteApp’s customization objects with your company name, AND set their “Lock on Install” preferences in the SuiteBundler as described in the previous section. This action prevents your company name from getting removed from the customization objects in the event they are re-distributed without your consent.

d. Whenever possible, implement account authorization coding in your SuiteScripts. This action ensures only authorized customers in good standing can successfully execute your SuiteApp in their NetSuite accounts.

9.3.3 SuiteCloud Development Framework and Distribution of SuiteApps

The SuiteCloud Development Framework (SDF) is a framework that de-couples the SuiteApp from its development account. This is accomplished by representing a SuiteApp and all its components as XML files. Another key driver behind SDF is the trackability of SuiteApps by tagging them with publisher IDs.

SuiteApp Definitions as XML and SuiteScript Files

Historically, ISV developers have developed their SuiteApps using their SDN development accounts, building various customization objects defined using SuiteBuilder in the browser and SuiteScript source code written outside of NetSuite (often in the SuiteCloud IDE). Then, ultimately, the finished SuiteScript source code was uploaded to the SDN development account's File Cabinet. Under this model, while the SuiteApp might be designed to execute on many NetSuite accounts, its artifacts are intrinsically tied to the primary development account they reside in.

The advent of SDF enables developers to decouple the SuiteApp from development accounts. The SuiteCloud IDE allows all the artifacts of a SuiteApp to be defined and developed outside of any NetSuite account, and represented as standalone XML files and SuiteScript source code files which are not tied to any NetSuite accounts. NetSuite provides a full-featured integrated development environment (IDE) to develop SuiteApps called the SuiteCloud IDE. Developers may use the SuiteCloud IDE to develop SuiteApp projects and tag them with their publisher IDs. The resulting XML files and SuiteScript source code files can be uploaded into a NetSuite account using the SuiteCloud IDE or the SDF Comment Line Interface (CLI), and SDF will generate the artifacts that comprise the SuiteApp.

The XML files plus the SuiteScript source code files contain all the IP of a SuiteApp. Essentially, they are the blueprint of a SuiteApp and should be treated with care and caution.

The reverse of the SDF action described above is also possible. This means any user or developer with administrator privileges can use SDF to export customization objects as XML files. If these customization objects came from a SuiteBundle installation, in which the components are locked (see section named "Securing SuiteBundle Content"), then SDF will honor these restrictions when generating the XML files. If the SuiteScript source code files are configured to be hidden in SuiteBundles, then they cannot be downloaded.

When offline from NetSuite, the XML files and SuiteScript source code files can be serialized or saved into a change management system of the SuiteApp author's choice. Note that the legacy method of developing SuiteApps using development accounts is still fully supported.

Important: As of version 2018.1, SDN Partner SuiteApps must be deployed to customer production accounts using the SuiteBundler. SuiteApps must not be deployed directly on customer production accounts by using SDF, XML files and SuiteScript source code files (using either the SuiteCloud IDE or SDF CLI). SDF should only be used between the SuiteCloud IDE and the development accounts. The deployment methodology of using development account, QA account and deployment account must be used where applicable (see section 8.1 for details).

Tagging SDN Partner SuiteApps with SDF Publisher IDs

A key feature of SDF that is designed specifically for SDN partner is the ability to tag their SuiteApps with their unique Publisher IDs. When tagged with a Publisher ID, a SuiteApp and all of its objects can be tracked by the authoring SDN partner (the “publisher”) and NetSuite. This is an important feature because ISVs need to be able to track their install base and prevent unauthorized distribution of their SuiteApps.

Publisher IDs are registered by the SDN team, and are only available to SDN partners. When a new company joins the SDN program, their unique publisher ID is provided to them during the onboarding process. Publisher IDs can be retroactively registered and provided to existing SDN partners.

When a SDN partner develops a SuiteApp using SDF and the SuiteCloud IDE, the project must be set as “SuiteApp” instead of “customization”. When set as SuiteApp, the publisher ID will be required.

As of version 2018.1, the full deployment, tracking and publishing capabilities of SDF are not available yet. These features will be released in subsequent releases, and SDN partners will be notified and assistance will be provided to help the adoption. Please stay tuned in this key area

10 Open Source and Third Party Software

You are solely responsible for the content of your SuiteApp, including any third party code contained in your SuiteApp. Your SuiteApp may not include any software that is subject to any license or other terms that may require you or NetSuite to do the following with your SuiteApp or any code integrated with your SuiteApp: a) disclose or distribute code in source form, b) redistribute code free of charge, or c) make code available to enable others to make derivative works. For example, some open source licenses, including the GNU Affero General Public License, GNU General Public License, GNU Lesser/Library GPL, and other licenses, may include such requirements. Additionally, in connection with your use of third party software that is not prohibited by NetSuite, you should keep good records related to any third party software included in your SuiteApp, including permitted open source. NetSuite recommends keeping the following information updated and available at all times:

A list of any third party and open source software included in any of your products developed for the NetSuite platform.
Copies of all third party licenses.
List of links where any open source software was obtained.
The license agreement for all open source software.

11 Industry Best Practice Security Principles

NetSuite has implemented technical and organizational security measures across its own organization. These measures are in line with NetSuite business objectives and with various industry standards and best practices.

NetSuite recognizes that the Open Web Application Security Project (OWASP) has created an ecosystem of continuously improving guidance for web developers. As a SuiteApp developer you should advance the maturity of your secure development process over time. OWASP is one of many resources that can provide helpful information in the furtherance of that effort.

As a starting point, the OWASP Security Principles help developers make security decisions in new situations with the same basic ideas. By considering each of the principles, developers can derive security requirements, make architectural and implementation decisions, and identify possible weaknesses in their systems prior to any deployments.

The industry best practice questions found in the Questionnaire are based upon the principles defined by OWASP. These principles can currently be found at this site:

https://www.owasp.org/index.php/Category:Principle

There is a great deal of additional and valuable reference material to be found on the OWASP site. You are strongly encouraged to surf and research the OWASP references to further your education and understanding on these important security topics.

For example, if you haven't already done so, you should familiarize yourself with the OWASP Top Ten Project regarding web application vulnerabilities. A document containing the details of this can currently be found at the following link:

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Another very important and useful document for developers can be found here:

https://www.owasp.org/index.php/Category:OWASP_Guide_Project

12 FAQ

General

Where do I go to learn more about the concepts discussed in this guide?

You can search the NetSuite Help Center for any of the concepts discussed in this guide. Access the Help Center by clicking the Help link in the upper-right corner of your NetSuite account.

You can also search for terms in SuiteAnswers. SuiteAnswers combines all the material in the NetSuite Help Center with training videos (provided by the Training Department) and Knowledge Base articles (provided by NetSuite Customer Support). Access SuiteAnswers by clicking the Support tab in your NetSuite account, and then click *Visit the SuiteAnswers Site*.

If I am developing a new SuiteApp, am I required to follow the design principles outlined in this guide?

If you are developing new SuiteApps for mutual customers, you must follow the design, development, and testing principles described in this SAFE Guide. The engineering principles described in this guide are meant to help you design better, more optimized SuiteApps. In many cases, if the principles are not followed, your SuiteApps will not run. In other cases, they may run, but may negatively affect the performance of other SuiteApps running in an account.

What do I do if I have existing SuiteApps that do not follow the design principles outlined in this guide?

If you have already developed SuiteApps that do not follow the principles outlined in this guide, you must reevaluate each SuiteApp you have deployed. You should then recode your SuiteApps, where necessary, so that they adhere to the principles outlined in this guide.

13 Appendix 1 – Sample Code

Generating TBA Headers using Javascript to Test RESTlets

HTML code for generating POST headers to test your TBA setup for RESTlets. Headers can be tested with REST clients such as “Postman”:

```
<html>

<title>NetSuite JavaScript TBA Header Example</title>



<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>



<!-- CryptoJS functions available from https://cdnjs.com/libraries/crypto-js -->



<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/crypto-js.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/core.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/hmac.min.js"></script>



<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/hmac-sha256.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/sha256.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/hmac-sha1.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/sha1.min.js"></script>

<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/enc-base64-min.js"></script>
```

```
</head>

<body>

<script>
    function myFunction() {
      var rest_url = document.getElementById("rest_url").value.trim();
      var consumer_key = document.getElementById("consumer_key").value.trim();
      var consumer_secret = document.getElementById("consumer_secret").value.trim();
      var token = document.getElementById("token").value.trim();
      var token_secret = document.getElementById("token_secret").value.trim();
      var ns_account_number = document.getElementById("ns_account_number").value.trim();
      var method = document.getElementById("method").value.trim();
      var contenttype = document.getElementById("contenttype").value.trim();
      var signature_method = document.getElementById("signature_method").value.trim();

                 var oauthValues = {
                 rest_url: rest_url,
                 consumer_key: consumer_key,
                 consumer_secret: consumer_secret,
                 token: token,
                 token_secret: token_secret,
                 ns_account_number: ns_account_number,
                 method: method,
                 contenttype: contenttype,
                 signature_method: signature_method

      };

//      Generate oauth signature - function is inside script_tba.js library file

      var oauth_headers = generateSignature(oauthValues);

      var oauth_header_string = "";

      var x = 0;

      for(keys in oauth_headers){

              oauth_header_string += (x++==0?"":", ") + keys + '="' + oauth_headers[keys] + '"';

      }


              //headers to test posting to RESTlet

              var headers = {"Authorization" : "OAuth " + oauth_header_string, "Content-Type":
oauthValues.contenttype};


              //HTML output representation of headers

              var headers_string = "<H1>RESTlet TBA Headers</H1><BR>  <B>Content-Type:</B> " +
oauthValues.contenttype + "\n<BR> " + "<B>Authorization:</B> OAuth " + oauth_header_string;



      document.getElementById("demo").innerHTML = headers_string;

    }



              //generateSignature

              var generateSignature = function(oauthValues){

              var timestamp = Math.round((new Date()).getTime() / 1000.0);
              var oauth_headers = {
                 oauth_version: "1.0",
                 oauth_nonce: btoa(encodeURIComponent(timestamp)).replace(/=/g, ""),

                 oauth_signature_method: oauthValues.signature_method,
                 oauth_consumer_key: oauthValues.consumer_key,
                 oauth_token: oauthValues.token,
                 oauth_timestamp: timestamp
                 };

              var url_params = oauthValues.rest_url.split("?")[1].split("&");
              var signature_params = {};

              for(key in oauth_headers){
                      signature_params[key] = key + "=" + oauth_headers[key];
                 }
              for(key in url_params){
                 var temp = url_params[key].split("=");
                 signature_params[temp[0]] = url_params[key];
              }

              var signature_string = "";
              var sortedkeys = Object.keys(signature_params).sort();

              for(var i = 0;i < sortedkeys.length;i++){
                 signature_string += (i==0?"":"&") + signature_params[sortedkeys[i]];
```

```
                    }

                    var base_string = oauthValues.method + "&" +
encodeURIComponent(oauthValues.rest_url.split("?")[0]) + "&" + encodeURIComponent(signature_string);

                    var composite_key = encodeURIComponent(oauthValues.consumer_secret) + "&" +
encodeURIComponent(oauthValues.token_secret);


                    var oauth_signature = encodeURIComponent(CryptoJS.HmacSHA256(base_string,
composite_key).toString(CryptoJS.enc.Base64));


                    oauth_headers.oauth_signature = oauth_signature;

                    oauth_headers.realm = oauthValues.ns_account_number;


                    return oauth_headers;

          }


    </script>








<p id="demo"><B>Click to generate TBA Headers</B></p>




    <form action="#" method="post" name="form_name" id="form_id" class="form_class">


        <label>rest_url:</label>
        <input type="text" size="100" name="rest_url" id="rest_url"
value="https://rest.netsuite.com/app/site/hosting/restlet.nl?script=156&deploy=1" />

        <br>

        <label>consumer_key:</label>

        <input type="text" size="100" name="consumer_key" id="consumer_key"
value="2b582098c08395f93045d31bc2433cb0e33a6fd85fc64251f7ccf3eb2e65d206" />

        <br>

        <label>consumer_secret:</label>

        <input type="text" size="100" name="consumer_secret" id="consumer_secret"
value="d520ffd306616daff7b93f017c8df05d4aa726fc5f8f44b15afbe8684967459b" />

        <br>

        <label>token:</label>

        <input type="text" size="100" name="token" id="token"
value="d16cbb3b254aa7948c8b056d7c82e233f7c3cd9404e52a6fff6efc040ba8caae" />

        <br>

        <label>token_secret:</label>

        <input type="text" size="100" name="token_secret" id="token_secret"
value="34e4b51a3a587cf51749d41d2df44edbcc5e19f84502ea4d724c15186932791f" />

        <br>

        <label>ns_account_number:</label>

        <input type="text" size="100" name="ns_account_number" id="ns_account_number"
value="TSTDRV993249" />

        <br>

        <label>method:</label>

        <input type="text" size="100" name="method" id="method" value="POST" />

        <br>

        <label>contenttype:</label>

        <input type="text" size="100" name="contenttype" id="contenttype" value="application/json" />
        <br>

        <label>signature_method:</label>

        <input type="text" size="100" name="signature_method" id="signature_method" value="HMAC-SHA256" />


    </form>


    <button onclick="myFunction()">Click me</button>


</body>


</html>
```

14 Appendix 2: SAFE Worksheet – 2019.1

Included in this Appendix is the SAFE Worksheet. The SAFE Worksheet is provided for the benefit of Select-level or Premier-level SDN partners who are preparing the submission of their responses to the questions in the verification process.

Some of the questions in the SAFE Worksheet include educational notes to aid partners with their responses to some of the verification questions. The questions in the SAFE Worksheet are provided only to aid partners with their preparations for the verification process. When engaging in the verification process, partners must submit their responses through the online tools provided by SDN.

Important: The questions in the SAFE Worksheet are very similar to those used in the verification process. However, the questions in the SAFE Worksheet may not be an exact, comprehensive copy of all of the questions, both the online and the follow-up questions, which must be answered during the verification process.

Section 1 – Understand NetSuite Features and Data Schema

Q1.01 – In which version of NetSuite did you last test your SuiteApp? (must be new release only)

1.02 – Please list the custom record types provided by your SuiteApp.

If the list is too large to fit in this text area, please send the list via email to SDNQA@netsuite.com.

Q1.02.1 - For any custom record type that REPLACES an existing NetSuite standard record, please explain why your design must replace the standard NetSuite object.

Q1.03 – Please list the custom fields provided by your SuiteApp including the following: name, description or purpose, applies to, data type, sensitive or personal data - Y|N. If the list is too large to fit in this text area, please send the list via email to SDNQA@netsuite.com.

Q1.04 – Please list the features of your SuiteApp that enhance or replace standard NetSuite features.

Q1.05 – Please list the standard NetSuite records that any of your scripts operate with.

Q1.06 - Does your SuiteApp utilize DOM references? (No)

Q1.07 - Does your SuiteApp utilize any unsupported NetSuite functions or APIs not documented in NetSuite Help? (No)

Q1.08 - If your SuiteApp includes integration, is it coded to perform in any NetSuite account hosted by any NetSuite data center? (Yes)

Q1.09 - Is your SuiteApp designed to work with both NetSuite and NetSuite OneWorld? (yes)

Q1.10 - Does your SuiteApp operate on any record(s) that are specific to OneWorld?

Q1.11 - Can your SuiteApp handle scenarios where a NetSuite account has had a temporary or permanent change in the data center from which it is host? (Yes)

Q1.12 - Is your SuiteApp a generic connector solution that uses SuiteTalk web services to communicate with NetSuite? (No)

Q1.12.1 - If yes, please enter each version of the SuiteTalk WSDL that is supported. (e.g., WSDL V2015.1) Otherwise, enter NA.

Q1.12.2 - Are all the operations exposed by each supported SuiteTalk WSDL are also supported by your connector - both synchronously and asynchronously? Otherwise, enter NA. (Yes)

Q1.13 - Do your scheduled scripts examine the type argument as part of their fault tolerance logic? (Yes)

Q1.14 - Does your integrated SuiteApp function correctly when it is connected to accounts with a “TSTDRV” prefix and to those without this prefix? (Yes)

Q1.15 - Is your integrated SuiteApp compatible with the new mandatory 2FA compliance rules? (Yes)

Q1.15.01 - If your SuiteApp uses a non-UI access of NetSuite (RESTlets, SuiteTalk integrations, mapSso API calls) has it been upgraded to either, a) Use a less privileged role to authenticate instead of using Administrator or a highly privileged role, or b) Use the Token-based Authentication (TBA) as the login method?

Q1.16 - If your SuiteApp is an integrated SuiteApp, do the pertinent elements in the external application’s data structure match those with NetSuite? (Yes)

Q1.16.01 - If No, is the information in the affected fields truncated/reformatted to ensure data integrity? (Yes)

Q1.17 - Does any version of your integrated SuiteApp use a version of SOAP Web Services endpoint (WSDL) more than 3 years old?

Section 2 - Manage SuiteScript Usage Unit Consumption

Q2.01 – Does this SuiteApp contain SuiteScripts that may perform a high amount of I/O that can exceed the NetSuite governance units?

Q2.01.1 – If there are high volume I/O processes in this SuiteScript, have they been moved to scheduled SuiteScripts to be executed asynchronously? (Yes)

Q2.02 – Do the SuiteApp's server side SuiteScripts (user event scripts, Suitelets) specifically address use cases in which they are close to exceeding the usage governance limits? (Yes)

Q2.03 – Do the SuiteApp’s server side SuiteScripts invoke external hosts or external web applications? (No)

Educational Note - Making http/https calls to external hosts consumes usage governance, and may also impact user experience if a host is responding slowly to your script’s requests.

Q2.04 – Does your SuiteApp utilize the Multiple Scheduled Script Queue feature in the SuiteCloud Plus add-on module?

Educational Note - The Multiple Scheduled Script Queue feature offers the ability to concurrently execute multiple scheduled scripts, therefore increasing the throughput of your SuiteApp where applicable.

Section 3 – Optimize Your SuiteApps to Conserve Shared Resources

SuiteScript

Q3.01 – Do your after-submit user event scripts use nlapiSubmitRecord to resave the same record? (No)

Q3.02 – Do you load non-transactional records into memory (using nlapiLoadRecord) in order to read data from them?

Q3.03 – Do you submit existing non-transactional records (using nlapiSubmitRecord) in order to update them?

Q3.04 – Do your user event scripts and Suitelets delegate heavy I/O to scheduled scripts whenever possible?

Q3.05 – Does your SuiteScript contain exception handling when an external host accessed by nlapiRequestURL takes too long to respond to the SuiteScripts' requests? (Yes)

SuiteTalk

Q3.06 – Does your SuiteTalk web services integration application use the explicit login or the SSOlogin API operation? (No)

Educational Note - Web services login can be accomplished using the explicit login operation. Login can also be accomplished by embedding the sign-in credentials into the request level header. The latter approach is easier to code and manage at runtime because it allows developers to delegate session management to NetSuite servers, thereby eliminating excessive login calls. Note that as of Version 2012 Release 2, SSO login is supported only through the explicit login operation.

Q3.06.1 – Does your SuiteTalk web services integration application obtain a new session for every operation it performs (add, update, search etc.)? (No)

Educational Note - The explicit login operation is a resource-intensive operation on NetSuite servers. Therefore, excessive login invocations should be avoided. Instead, you should reuse existing valid Web services sessions obtained from previous Web services API calls. You can also use request-level header credentials.

Q3.07 - Do your SuiteTalk web services utilize advanced search capability for searches?

Educational Note - Among other benefits, Web services advanced search allows you to specify the desired columns to be returned. This results in vastly better searching performance, as the NetSuite server can generate much smaller result sets to be returned to the Web services client application.

Q3.08 – Does your SuiteTalk (or REST-based) web services integration application perform mass data import from an external data store in to NetSuite?

Q3.09 – Does your SuiteTalk (or REST-based) web services integration application perform mass data export out of NetSuite in to an external data store?

Q3.10 – If your SuiteTalk web services integration application performs any mass data I/O, does it use asynchronous API's to do so? (Yes)

Educational Note - Asynchronous Web services put less demand on NetSuite servers. Therefore, tasks that involve mass data I/O should be performed using asynchronous Web services.

Q3.11 - If your SuiteApp uses SuiteTalk Web services, does it embed the NetSuite-assigned application ID in the SOAP header for every request? (Yes) (This becomes mandatory for 2014.2)

Q3.12 - If your integrated SuiteApp is multi-threaded (via SOAP or REST), does it provide a setting for the administrator to configure the maximum number of threads it can spawn? (Yes)

Section 4 – Tolerance with Other SuiteApps

Q4.01 – Have you consolidated all user event SuiteScripts that are deployed on the same record into a single script?

Q4.02 – If your user event SuiteScripts have sequence dependencies, eg, must be first, or must be last, to be executed on its designated event, do you provide user/administrator documentation regarding these dependencies? (Yes)

Educational Note - Be aware that another SuiteApp might introduce user event scripts deployed to the same record, and may disrupt the order in which you want your scripts to execute. Document the order in which your scripts must be executed so that account administrators can adjust their multi-SuiteApp environment accordingly.

Q4.03 – If your user event scripts read from or write to any standard NetSuite fields, do you provide user/administrator documentation regarding these fields?

Educational Note - You are encouraged to document the fields your user event scripts read from and write to. Your documentation will help account administrators troubleshoot data concurrency problems when multiple SuiteApps introduce user event scripts into the same records.

Q4.04 – For user event SuiteScripts in your SuiteApp that are execution context sensitive (user interface, SuiteTalk (or REST-based) web services, CSV etc.), do you exclude other execution contexts from invoking the logic in that SuiteScript?

Q4.05 – Does your SuiteApp include any user event scripts deployed on the NetSuite sales order record?

Q4.06 - If your SuiteApp operates on NetSuite sales orders, does your QA process include testing sales orders created through the browser interface (with a NetSuite user who is logged in) as well as through NetSuite eCommerce?(Yes)

Educational Note - NetSuite eCommerce shopping carts create sales orders, which can trigger user event scripts deployed to the Sales Order record type. SuiteApps that include user event scripts on sales orders should ensure that the scripts are eCommerce compatible.

Q4.07 - If your SuiteApp resides on the NetSuite platform, are there any outgoing 3rd party web services requests transferred over http rather than https? (No)

Q4.08 - If your SuiteScript uses any open source JavaScript libraries, are their referenced methods isolated in order to avoid potential conflicts with the same libraries that NetSuite might use? (yes)

Q4.09 - If your SuiteScript programmatically invokes scheduled scripts, does it gracefully handle conditions where the scheduled scripts are not successfully invoked and placed into the queue? (yes)

Q4.10 - If your integrated SuiteApp is one that uses the add or addList operations, does it utilize the externalID field. (Yes) (will be mandatory in 2018.1)

Section 5 – Designing for Security and Privacy

Secure SuiteScript Designs

Q5.01 – Please list the custom roles included in your SuiteApp; include the purpose of each and the permissions associated with it. Note that NetSuite may request screenshots of these custom roles. If the list is too large to fit in this text area, please send the list via email to SDNQA@netsuite.com.

Q5.02 – Do any of your server side SuiteScripts (excluding scheduled scripts) have the “Execute as Admin” setting enabled? (No)

Q5.03 – Does any Suitelet included in your SuiteApp have the “Available without Login” setting enabled? (No)

Q5.04 - Do your SuiteApp suitelets contain comprehensive data sanitation logic on all custom fields including checking for data type, length, ranges, and expected choices?(Yes)?

Educational Note - Please be prepared to discuss the details of your data sanitation practices with SDN.

Passwords

Q5.05 – Does your SuiteApp capture NetSuite login credentials and subsequently store them inside or outside of NetSuite?

Educational Note - SuiteApps must not store NetSuite user passwords in any manner inside NetSuite.

Q5.06 – Does your SuiteApp store passwords or user credentials for external systems in NetSuite in an unencrypted format?

Q5.07 - If your SuiteApp logs in to external systems via SuiteScript, does it use the following SuiteScript APIs to do so: nlobjForm.addCredentialField() and nlapiRequestURLWithCredentials()? (Yes)

Educational Note - The APIs nlobjForm.addCredentialField and nlapiRequestURLWithCredentials are designed to use a Suitelet page to securely log in to an external system. The user-entered passwords are handled securely and are not stored in NetSuite after the login.

Q5.07.2 - If your SuiteApp logs in to external systems, is the password sent using SSL to do so? (Yes)

Credit Card Information

Q5.08 - Does your SuiteApp store credit card information anywhere in NetSuite other than in the Credit Cards sub-tab of the NetSuite customer record? (No)

SuiteSignOn

Q5.09 - If your SuiteApp makes SuiteTalk web services calls, does it do so using authenticated sessions generated from a successful SuiteSignOn connection handshake? (Select NA if “No Access” is specified in “Web Services Access” field)?

Q5.10 - If your SuiteApp makes outbound SuiteTalk (or 3rd party-based) web services calls via your SuiteSignOn Connection Point, does it use “Same as UI Role” in the “Web Services Access” field? (No)

Q5.11 - If your SuiteApp makes outbound SuiteTalk (or 3rd party-based) web services calls via your SuiteApp’s SuiteSignOn Connection Point, does it specify a custom role in “Web Services Access” field that is configured to only have the permissions required to accomplish the SuiteTalk web services API calls’ tasks? (Yes)

Privacy

Q5.12 – Have you evaluated which privacy and data protection laws, ordinances, and regulations apply to your SuiteApp? (Yes)

Q5.13 – Does your SuiteApp comply with all such privacy and data protection laws, ordinances, and regulations that apply to your SuiteApp? (Yes)

Q5.14 – Does your SuiteApp customer agreement protect the privacy and legal rights of your customer and their users? (Yes)

Q5.15 – Does your SuiteApp customer agreement, or any applicable privacy policy you provide, fully disclose to customers how any information that your SuiteApp collects is used and shared? (Yes)

Q5.16 – Does your SuiteApp customer agreement specify that the SuiteApp will transmit, process, and store customer data in a secure manner and only as disclosed to, and approved by, your customers? (Yes)

Q5.17 - Does your SuiteApp require copies of your private key to be stored outside of your organization? (No)

Q5.18 - Have you embedded your private key in your mobile app? (No)

Q5.19 - Do you have a plan in place if the key is compromised? (Yes)

Q5.20 - If your SuiteApp uses the SuitePayment API, have you implemented all payments operations that your plug-in declares? The payments operations are Authorization, Capture, Sale, Refund, Refresh, and Void (Yes)

Q5.21 - If your SuiteApp uses the SuitePayment API, have you implemented a testing flag that allows merchants to test their payment gateway integration? (Yes)

Q5.22 - If your SuiteApp uses the SuitePayment API, have you clearly indicated to all users (excluding ecommerce shoppers) of relevant transactions that the payment gateway integration is handled by your company? (Yes)

Q5.23 - If your SuiteApp uses the SuitePayment API, is the setRequest() object method declared before the requestPaymentGateway() object method? (Yes)

Q5.24 - If your SuiteApp uses the SuitePayment API, please submit the exhaustive list of all error codes to SDNQA@netsuite.com, including those that are applicable to your SuitePayment SuiteApp and handled by it.

Q5.25 - If your solution is an integration application, does it use Token-based Authentication, or an application ID generated by an Integration Record from your SDN trailing development account? (Yes)

Q5.25.1 - If yes, do you include the Integration record in your SuiteApp bundle? (yes)

Q5.26 - Does your integration application store consumer key, consumer secret, token ID, or token secret in an unencrypted manner? (No)

Q5.27 - If your SuiteApp uses the SuitePayment API, does your plug-in implementation always declare a merchant ID by setting the setMerchantId() function on the ProcessSetupOutput object at the processSetup() function entry point? (Yes)

Q5.28 - If your SuiteApp uses the SuitePayment API, have you clearly indicated to all users (excluding ecommerce shoppers) of relevant transactions that the payment gateway integration is handled by your company? (Yes)

Q5.29 - If your SuiteApp uses the SuitePayment API, is the setRequest() object method declared before the requestPaymentGateway() object method? (No)

Q5.30 - If your SuiteApp uses the SuitePayment API, does your plug-in implementation use any deprecated API functions? (No)

Q5.31 - Does your integration use MD5 or SHA-1 for Hashing operations? (No)

Section 6 - Testing your SuiteApps.

Testing Framework

Q6.01 - Have you installed the SuiteScript Testing Framework bundle (also known as the jsUnity bundle) into any of your testing environments? (Note that the SuiteScript Testing Framework bundle is sometimes referred to as the jsUnity bundle.)

Phased Release

Q6.02 - During release preview periods, do you utilize a leading SDN account, or accounts, for testing your SuiteApp on the leading version of NetSuite? (Yes)

Q6.03 - Do you deploy your SuiteBundle SuiteApp to your Netsuite Customers from your SDN trailing account(s) exclusively? (Yes)

Q6.04 - Do you continue to utilize your trailing accounts during the release preview period in order to support your existing customers who have not yet moved to the new release? (Yes)

Q6.05 - Did you attend the SDN new release webinar or view the latest recording after the webinar broadcast?

Q6.06 - Did you install your SuiteApp in an SDN leading account and test it within one month after obtaining access to the new release of NetSuite? (Yes)

Q6.07 - Do you perform collaborative testing with any of your customers during the phased release period? (Yes)

QA Checkpoints for SuiteApps

Q6.08 - Does your testing methodology include role-based testing? (Yes)

Q6.09 - Does your testing methodology include some peak hour performance testing (Monday to Friday, 9am to 2pm PST)?

Q6.10 - Does your testing methodology utilize QA account(s) that are separate from Development account(s)? (Yes)

Q6.11 - If your SuiteApp is NetSuite-resident, does your testing methodology include the testing of your SuiteApp deployment by installing the SuiteApp/bundle in to a separate test account? (Yes)

Q6.12 - If your SuiteApp is approved for SuiteSuccess, does it write to any custom fields contained in your SuiteSuccess test accounts that were not created by your solution? (No)

Q6.12.1 - If Yes, then please list the SuiteSuccess vertical(s) that your SuiteApp is integrating to, and the method or design pattern employed for this integration.

Q6.13 - If your SuiteApp is approved for SuiteSuccess, does it create any "rows" of custom records that are defined in the SuiteSuccess test accounts, but not created by your solution? (No)

Q6.13.1 - If Yes, then please list the SuiteSuccess vertical(s) that your SuiteApp is integrating to, and the method or design pattern employed for this integration.

Q6.14 - If you have access to SuiteSuccess accounts, does your SuiteApp create any customization objects that reference object(s) contained in a SuiteSuccess test account, but not created by your solution? (No)

Q6.15 - If you have access to SuiteSuccess accounts, does your SuiteApp intentionally, or unintentionally, include any objects in your SuiteBundle(s) that originated in your SuiteSuccess test accounts? (No)

Q6.16 - Can your SuiteApp be configured to perform with a customer's sandbox environment? (Yes)

Q6.17 - Have you defined multiple users with administrative permissions for your DEV/QA/Deployment accounts? (Yes)

Section 7 - Deploying Your SuiteApps as Managed Bundles

Q7.01 – If your SuiteApp is NetSuite-resident, is it deployed as a managed bundle?

Q7.02 - Does your bundle deployment account have your company name set in the Company Information page? (yes)

Q7.03 - Is your bundle named using the guidelines provided in chapter 8 of the SAFE document? (yes)

Q7.04 - If your SuiteApp contains any customization objects, are they created on the customers' production accounts by the SuiteBundler?

Section 8 - SuiteBundle Deployment Chain Maintenance

No questions related to section 8 of the SAFE guide at this time.

Section 9 - Licensing and Agreements

Q9.01 - Do your customer agreements for this SuiteApp include an appropriate license or subscription agreement? (Yes)

Q9.02 - Is the license agreement implemented as a click-through that is included within the SuiteApp's bundle?

Q9.03 - If your SuiteApp uses SuiteTalk web services, does it require a separate or additional NetSuite end-user licenses? (No)

Q9.04 - Do you have agreements with all of your relevant employees and consultants that grant you appropriate ownership rights in their development related to the SuiteApp and contain obligations of confidentiality? (Yes)

Q9.05 - In your deployment bundle, do you enable the “Hide in SuiteBundle” setting in your server side SuiteScript source code files in order to protect your intellectual property? (Yes)

Q9.06 - In your deployment bundle, do you enable the “Lock on Install” preferences in order to avoid intentional or unintentional tampering of your bundle’s components? (Yes)

Q9.07 - Are all the customization objects in your SuiteApp tagged with the SDF publisher ID that was assigned to you by NetSuite? (Yes)

Section 10 - Open Source and Third Party Software

Q10.01 - Do you review and keep a record of licensing terms for all third party code (including open source) included in your SuiteApp? (Yes)

Q10.02 - Does your SuiteApp include any prohibited open source code as described in the Open Source and Third Party Software Section of the SAFE Guide, including any code licensed under the GNU Affero General Public License, GNU General Public License or GNU Lesser/Library GPL? (Yes)

Section 11 - Additional SuiteApp-specific Security Questions

Q11.01 - Does your SuiteApp receive, produce, or process any data from external systems outside of the NetSuite environment?

Q11.02 - Are there any external systems that receive or process your SuiteApp’s data?

Q11.03 - If your SuiteApp supports externally stored data, have you implemented backup and retention capabilities to ensure the continuity of the data NOT stored in the NetSuite database? (Yes)

Q11.04 - What are your up-time service level agreements for the data and the processing outside of NetSuite? Please list the percentage up-time, or respond NA.

Q11.05 - Does your SuiteApp log or store any error-handling, audit log, or other messaging data outside of NetSuite? (No)

Q11.06 - If your SuiteApp resides primarily outside of NetSuite, does it have awareness and special behavior when the NetSuite account to which it is linked is in maintenance mode, disconnected, or out of service? (Yes)

Q11.07 - If your SuiteApp resides primarily inside NetSuite and links to external applications, does it have awareness and special behavior when the external application to which it is linked is in maintenance mode, disconnected, or out of service? (Yes)

Section 12 - Additional Company-specific Security Questions

Q12.01 - Are there any processes or mechanisms, such as code reviews, used to detect malicious or compromised code in your SuiteApp or in its integration with external applications? (Yes)

Q12.02 - Please explain briefly what processes you have implemented to handle security incidents in your organization.

Q12.03 - Are there processes or tools implemented for controlling changes to your SuiteApp's code, such as a version control system? (Yes)

Q12.04 - Are there processes implemented for the deployment of code updates to the external applications integrated with your SuiteApp? (Yes)

Q12.05 - Have you restricted access to your source code within your development environments to the authorized employees in your company or other, authorized parties under contract? (Yes)

Q12.06 - Do you have a process to immediately terminate the systems and source code access of an employee or an outside contractor whenever the circumstances warrant? (Yes)

Q12.07 - Do you currently maintain a separation of duties between the development, QA, and deployment resources within your organization? (Yes)

Section 13 – Miscellaneous - Future Use

Q13.01 - What RF devices or other mobile devices (other than smart phones, tablets, etc.) do you support (please include device make/model, OS, web browser name/version)? (Please list or enter NA)

Q13.02 - Do your product and its 3rd party components meet the same support levels as NetSuite for the browsers and the mobile devices (both HW and SW)? (Yes)

Q13.03 - Do you validate the support levels of any/all 3rd party components (HW & SW) that can be added to your solution? (Yes)

Q13.04 - Does your SuiteApp embed the NetSuite UI inside an external application's UI using technologies such as IFrame? (No)

15 Appendix 3: Concurrency governance cheat sheet

Included in this Appendix is the Concurrency Governance cheat sheet. This document is referenced by the section of the same name in section 3.6.3. (Note: next two pages are in landscape format.)

Concurrency governance cheat sheet

2 governance types are in place simultaneously

User-level limit

Defines limit per user and applies for specific authentication method and specific API. User-level limit defines maximum but does not guarantee minimum available due to account limit (10 requests for concurrent WS user (cWSu)* are not guaranteed).

API	Authentication Method			
	Request-level Credentials (RLC)	Login/Logout (L/L)	SSOLogin	Token-based (TBA)
SOAP WS	1/10**	1/10	1/10	No limit per user
RESTlet	No limit per user	Not applicable	Not applicable	No limit per user

* One concurrent WS user (cWSu) can be defined on employee record for each SC+ license you get

** 1/10 = 1 request/user OR max 10 requests/cWSu. You can have 1 concurrent request per user at given time OR maximum 10 concurrent requests if it is cWSu

Account-level limit

This limit applies to the combined total of SOAP Web Services (WS) and RESTlet requests per given account. Maximum request count is derived from the service tier, the number of SuiteCloud Plus (SC+) licenses and account type (developer accounts have base limit = 5).

Service Tier	Account Base Limit*	1 SC+ Licence	2 SC+ Licence	10 SC+ Licence
Shared, 3	5 concurrent requests for the entire account	5+1×10=15	5+2×10=25	5+10×10=105**
2	10	10+10	10+20	10+100**
1, 1+, 0	15	15+10	15+20	15+100

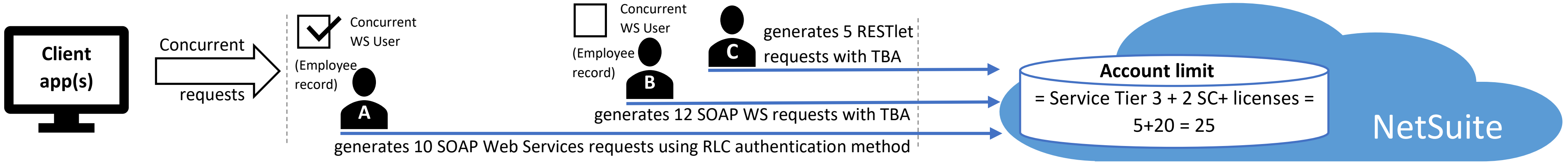
*The base limit is increased by 10 for each SC+ license. The number of SC+ licences may vary from 1 to many.

** Not a standard license count for this service tier

Sample Scenarios – how many concurrent requests can I have?

End users/Client application(s) are querying my company account with following amount of requests

	Snapshot of all incoming requests at one time									Account			Requests status	
	SOAP WS							RESTlet	Requests Total	Service Tier	SC+ License	Total Account Limit	Success	Fail*
	RLC		L/L		SSOLogin		TBA							
	user	cWSu	user	cWSu	user	cWSu								
	2	n/a	-	n/a	-	n/a	2	-	4	Shared	0	5	4	0
	1	4	2	-	1	-	7	1	16	Shared	1	15	15	1
	-	n/a	-	n/a	-	n/a	6	2	8	3	0	5	5	3
	-	9	-	-	-	-	6	3	18	3	1	15	15	3
A scenario explained in detail:	-	10	-	-	-	-	12	5	27	3	2	25	25	2



Client app(s)
Concurrent
requests
☑ Concurrent WS User
(Employee record)
A
□ Concurrent WS User
(Employee record)
B
C
generates 5 RESTlet requests with TBA
generates 12 SOAP WS requests with TBA
generates 10 SOAP Web Services requests using RLC authentication method
Account limit
= Service Tier 3 + 2 SC+ licenses =
5+20 = 25
NetSuite

Any 2 of incoming 27 requests can* fail

User-level limits apply here

User A is concurrent WS User (company can enable it for 2 users because it has 2 SC+ licenses). **User A** can send max 10 concurrent requests. Without this privilege, 9 requests of user A would be rejected and all others processed.

User B is using TBA and does not have to be a concurrent WS user to be able to send more requests. TBA is recommended authentication method with even more benefits.

Account-level limit applies here

External application(s) send concurrently 27 requests in total. Any 2 requests can be rejected due to account limit 25.

Concurrent vs sequential requests: If one of above requests is being processed for 1 minute (e.g. a long search), during that 1 minute can be sent and processed MORE than 24 other requests if those requests are not reaching the server in one moment, but sequentially after response to the other request.

* Please note: Governance framework can temporarily shortly allow more concurrent requests

See Help Center for other scenario

Concurrency governance cheat sheet

Recommended Actions

1. Analyse the frequency and level of concurrency peaks and consider rescheduling requests to be outside of regular peak times.
2. Consider if more SC+ licenses are needed and learn on SC+ settings.
3. Handle the error codes in client application.
4. Implement retry logic.
   a. Retry gradually increasing the delay if more attempts needed.
5. For non-concurrent users serialize your requests in client applications to not overlap.
6. Use TBA to take advantage of a more flexible concurrency.
7. Monitor trends in concurrency usage to prevent broken integrations (see Navigation table below).

Code example demonstrates basic handling of WS error codes

```
int i = 0;
int maxAttempts = 5; // try it 5 times, then fail for good

        while (i < maxAttempts) {
                response = doWSCall();
                isSuccess = response.getIsSuccess();
                errorMsg = response.getErrorMsg();

                if (isSuccess == false && (errorMsg == WS_CONCUR_SESSION_DISALLWD || errorMsg == WS_REQUEST_BLOCKED)) {
                        wait();
                        i++; // try again
                } else {
                        break; // end the cycle
                }
}
```

Method	Error codes	
	SOAP Fault	Error Message
Web Services + L/L or RLC	ExceededRequestLimitFault	WS_CONCUR_SESSION_DISALLWD
Web Services + TBA	ExceededConcurrentRequestLimitFault	WS_REQUEST_BLOCKED
RESTlet	HTTP error code: 400 Bad Request	
	SuiteScript error code: SSS_REQUEST_LIMIT_EXCEEDED	

Error can occur for any of the requests that exceed the limit at that moment

NetSuite navigation

What	Where
Account concurrency limit	Setup > Integration > Integration Management > Integration Governance
If account concurrency limit is enabled	
Total requests (number, ratio)	
Rejected requests	
Reports about rejected SOAP WS requests	Reports > New Search -> Web Services Operations
Reports about rejected RESTlet requests	RESTlet script record > Log
Details about SOAP WS requests that were rejected due to concurrency violation	Setup > Integration > Web Services Usage Log
Web Services performance dashboard	The Application Performance Management SuiteApp (link)
Concurrency Monitor dashboard, monthly/hourly overview (heatmap), charts showing concurrency usage with drill down possibility to seconds	
Scheduling of integrations	
Decision tree – considering additional license, what is appropriate account concurrency limit	Help Center article