SAFE Guide

SuiteApp Architectural Fundamentals & Examples

Version 2024.2

July 2024

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: Sections with titles highlighted in yellow have been updated or added for the 2024.2 version.

Contents

NOTE: Sections with titles highlighted in yellow have been updated or added for the 2024.2 version.

Introduction

The development principles described in this guide ("Principles") are for independent software vendors (ISVs) 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 publish 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, and may negatively affect the performance of other SuiteApps running in an account.

The Principles are:

- **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 Distributing Your SuiteApps in a Managed Fashion
- **Principle 8:** Maintain Your SuiteApps
- **Principle 9:** Agreements and Licensing
- **Principle 10:** Open Source and Third-party Software
- **Principle 11:** Industry Best Practice Security Principles

Important: This guide assumes you are familiar with the SuiteCloud platform. The above Principles are not intended to replace SuiteScript or SuiteTalk Web Services training. Note that while all SuiteCloud Developer Network (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 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

The SAFE Worksheet is provided for the benefit of any Select or Premier level SDN partners who are preparing to complete 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.

Starting with version 2024.1, the SAFE Worksheet is now contained in its own document. Refer to the SAFE Guide Worksheet (Questionnaire) PDF document for the worksheet.

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, refer to the Frequently Asked Questions section, particularly the "Where do I go to learn more about the concepts discussed in this guide?" topic, which provides information on using the NetSuite Help Center and SuiteAnswers to further research any topic.

Additionally, there is are "Further Reading" sections throughout this guide, that provide search terms you can use to find more information in the NetSuite Help Center or in SuiteAnswers. The topic titles listed under each Further Reading section are the exact titles you should use when searching in the Help Center or SuiteAnswers.

1. Principle 1: Understand NetSuite Features and Data Schema

As a SuiteApp developer or architect, 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. You must understand the features you want to extend before you start accessing and using NetSuite programming application programming interfaces (APIs).

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; you may develop a SuiteApp that extends this functionality in NetSuite by customizing the Order Fulfillment object, but you must not build an equivalent module that duplicates existing, built-in NetSuite order fulfillment functionality.

You must also be aware that enabling certain features (at Set Up > Company > Enable Features) will activate some workflows or data schema elements. Some examples are the Pick, Pack, and Ship feature, and the Multiple Shipping Routes feature. The Pick, Pack, and Ship feature adds additional steps to the sales order fulfillment process; the Multiple Shipping Routes feature 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. Your learning must include hands-on exercises with the NetSuite modules that are relevant to your product.

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 how to set up shipping times for Pick, Pack, and 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-added capabilities 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 to represent specific business objects in a warehouse environment.

While building the WMS, 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.

Clicking the Receive button on the standard NetSuite PO form does not invoke the custom WMS logic, so 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 should be disabled for items under the control of the WMS. And the ability to directly receive a PO in NetSuite may also need to be disabled.

However, the WMS must also 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 exempt 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, only. 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 available with standard objects.

In the rental car example in Solutions Must Work With Existing Processes, 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 custom record type. Calendar events and car rental agreements are fundamentally 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. Integration SuiteApps that use SuiteTalk Web Services must also use a Web Services Description Language (WSDL) that is supported by NetSuite (within a 3-year window). For example, for the version 2020.1 release, the oldest supported WSDL is version 2017.1. Any Integration SuiteApps that use version 2016.2 or older WSDLs will not meet the requirements set by the SDN quality program. This requirement ensures that Integrated SuiteApps will be fully functional and supported for production usage.

SuiteScript scripts, especially those with inline HTML UI components, are developed on supported combinations of operating systems 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 Integration 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 the field size, field type, format, etc. in 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.

1.2.5 JavaScript Best Practices

Since 2015, with the release of ES6, a new version of ECMAScript specification has been released each year. Each iteration adds new features, new syntax, and new ways to improve the quality of JavaScript code.

SuiteScript 2.1 uses a different runtime engine than SuiteScript 2.0 and also supports ECMAScript language features that are not supported in SuiteScript 2.0. This causes some capabilities in SuiteScript 2.0 scripts to function differently when the script is executed using SuiteScript 2.1.

Please refer to the topic *Differences Between SuiteScript 2.0 and SuiteScript 2.1* in the NetSuite Help Center for a thorough analysis of these differences.

The purpose of this section of this SAFE Guide is to encourage developers to adhere to new functional programming trends that have also been adopted by JavaScript.

In order to make use of ES6, it is mandatory to specify 2.1 as the API Version to be used in the script.

```
 /**
 * @NApiVersion 2.1
 * @NScriptType ...
*/
```

1.2.5.1 Variables Scope and Type

Since the beginning, JavaScript developers have been using the `var` keyword to declare variables. The main problem with this keyword is the scope of the variables created by using it.

```
var x = 10;
if (true) {
    var x = 15;     // the original value is override here
    console.log(x); // prints 15
}
console.log(x);     // prints 15
```

The `var` keyword does not narrow its value to a scope, therefore it is recommended to use `let` instead.

```
let x = 10;
if (true) {
    let x = 15;
    console.log(x); // prints 15
}
console.log(x);    // prints 10
```

One important aspect about `let` vs `var`, is that `let` can be updated within its scope; but, unlike `var`, it cannot be redeclared. The same variable can be declared twice only when it belongs to different scopes.

```
let greeting = "hello";
let greeting = "say hello"; //error: identifier 'greeting' has already been
declared
```

Even when `let` should be used instead of `var`, it is also important to be fully aware of value assignment along the code. Even though JavaScript does not support immutable variables like other languages, it strongly encourages the use of `const` whenever possible.

Variables declared with the `const` keyword, maintain constant values. `const` variables are also block scoped and cannot be updated or re-declared.

```
function saveRec(context) {
    const rec = context.currentRecord;
    const currentDate = new Date()
    const oneWeekAgo = new Date(currentDate - 1000 * 60 * 60 * 24 * 7);

    // Validate transaction date is not older than current time by one week
    if (rec.getValue({ fieldId: 'trandate'}) < oneWeekAgo) {
      return false;
    }
        return true;
}
```

1.2.5.2 Arrow Functions

Arrow functions are a very interesting feature that enhance functional aspects of JavaScript development and are the preferred format for emulating lambda expressions. This is important for a programming language that supports first-class functions, which means it is able to pass functions as arguments to other functions or assign them to variables.

```
var anon = function (a,b) {return a + b}  //Old JavaScript
const anon = (a,b) => {return a + b}      //ES6
```

One important caveat is that arrow functions do not have their own binding to `this` or `super`. Therefore, they should not be used as methods. Also, arrow functions are not suitable for `call`, `apply` and `bind` methods, which usually rely on establishing a scope.

Finally, arrow functions cannot be used as constructors and you cannot use `yield` within the function body. However, arrow functions can be used as lambda expressions and there is where they shine.

```
const numbers = [1,4,9,16];
const res = numbers.map(x => x * 2);
console.log(res);
```

1.2.5.3 Functional Programming

JavaScript is following the trend of most languages where functional programming prevails. In functional programming, functions are treated as first-class citizens, meaning they can be bound to names, passed as arguments and returned from other functions. This allows a more composable style of writing programs in a modular manner.

Choose Declarative over Imperative

Declarative programming is a programming paradigm where we specify the program logic without describing the flow control. Declarative programming is all about what to do to achieve a certain result.

```
const numbers = [1, 2, 3, 4, 5];
const sumNumbers = (n) => n.reduce((acc, current) => acc + current);
```

On the other hand, *imperative programming* is a programming paradigm where we explicitly specify the program logic describing the flow control. Imperative programming is all about how to achieve a certain result.

```
var numbers = [1, 2, 3, 4, 5];
var finalResult = 0;
for (let i = 0; i < numbers.length; i++) {
  result += numbers[i];
  }
console.log(result);
```

It is always advisable to use declarative programming techniques over imperative programming techniques, and to always think about solutions that involve concatenating functions in a piping style, instead of using single or multi-level loops.

Immutability

An immutable object cannot be modified after its creation. So, if we need to add a new property to an existing object, it is always recommended to create a new object, copy the old content and add it to the new object, instead of "mutating" the existing one. By applying Immutability concepts, you can avoid unexpected side effects in your codebase.

Since JavaScript is not a functional language *per se* (it does not force immutability), but you can ensure immutable behavior by avoiding using certain methods, in particular, array methods such as `fill`, `pop`, `sort`, `splice`, `unshift`, `reverse`, `push`, etc.

`You should use the object.assign` method with immutable data. Let's say you have an object and would like to create a new object with only one of the existing object properties modified. Instead of doing this:

```
const car = {
  model: 'AAAA',
  year: 2020
  }
const newCar = car;
newCar.model = 'BBBB';
```

You should do this:

```
const car = {
  model: 'AAAA',
  year: 2020
  }
const newCar = Object.assign({}, car, {
  model: 'BBBB';
});
```

Spread Operator

The spread operator allows arrays or strings to be expanded in places where zero or more arguments (or elements) are expected. When you want to implement immutability, the spread operator becomes handy for operations that return a new copy of the original object, array, or string.

```
const arr = [1,2,3];
const arr2 = [4,5];
arr = [...arr,...arr2];
console.log(arr); // [ 1, 2, 3,4,5]
```

1.2.5.4 String Interpolation

In the past, string concatenation with variables was done using the plus sign (+). Now it is recommended to use a new concept called *String Interpolation* where back ticks are used to enclose strings and variables are enclosed between curly braces prefixed by a dollar sign.

```
//Classic string concatenation
const name = "John";
console.log("Hello " + name);

//String interpolation
const name = "John";
console.log(`Hello ${name}`);
```

1.3 SuiteTalk Web Services: Choosing the right technology

NetSuite offers a wide range of options to externally access services and data. This section provides clarity on each SuiteTalk interface and helps developers choose the right interfaces to use.

Selecting the right SuiteTalk interface depends on many factors. It is worthwhile to note that there may not always be a unique approach for an optimal solution and a combination of multiple technologies could be a valid way. For example, it is important to be fully aware of the current general availability status of NetSuite records in the REST API, since it is possible to combine SuiteQL with REST Web Services to enhance query capabilities.

This section explains the available interface options, as well as the advantages and drawbacks of each one.

1.3.1 SOAP Web Services

The SOAP web services platform provides programmatic create, read, update, and delete (CRUD) access to your NetSuite data through an XML-based API.

Note: The use of SOAP will not be permitted for new SuiteApps, starting in 2024.2.

Pros

- Most mature platform (most NetSuite standard records and custom records are supported)
- Compatible with many commercially available connector software applications
- Human readable XML envelope
- Built-in logging reports in the UI
- Better suited for large ERP-related tasks

Cons

- OAuth 2.0 is not supported
- SuiteAnalytics Workbooks/SuiteQL is not supported
- Stateful, parallel threads must be manually managed on the client-side
- Higher latency than REST
- Endpoints expire over time

1.3.2 REST Web Services

The NetSuite REST web services provide an integration channel that extends the capabilities of SuiteTalk. REST web services provide a REST-based interface for interacting with NetSuite.

Pros

- Lightweight, more efficient than SOAP
- Stateless, easier to parallelize threads managed server-side
- Smaller payload than SOAP; better suited for mobile platforms and real-time requests
- OAuth 2.0 supported
- Simple access to records metadata (user and company-specific metadata)
- Easy-to-use API
- SuiteAnalytics Workbooks/SuiteQL is supported
- Built-in logging reports in the UI

Cons

- Legacy tax features are not supported. In order to work taxation through REST Web Services, the SuiteTax feature must be enabled

1.3.3 REST API Record Data Retrieval

The REST API enables web developers to retrieve data through different methods allowing seamless integration of their applications with NetSuite. Among its many features, the REST API provides two primary endpoints that can be used for data retrieval, each with its own advantages and use cases: (1) the REST record endpoint, and (2) the query endpoint. The following sections will explore some of these features, drawbacks, and optimal use cases for each data retrieval method, as well as the limitations and considerations of the REST API.

Depending on your data retrieval requirements and complexity, you may use the record service or the query endpoint. However, regardless of the method, there are system-level account-wide, and web services limits that may affect efficiency in either method of retrieval, which could result in some requests being bounced back due to server overload (HTTP 503) or concurrency violations (HTTP 400). See the REST API Limitations and Considerations section below for more information.

REST API Record Endpoint

With the REST API record endpoint, you can run a GET request to retrieve a list or an instance of records of the same type. This endpoint is suited for scenarios that require simple data fetching, such as retrieving a customer or transaction record by Id or ExternalId, making it an excellent choice for developers unfamiliar with SQL or more complex querying techniques. Nevertheless, due to its limited filtering capabilities, it can only retrieve records of the same type based on specified IDs, names, dates, or keys, rather than executing more complex queries.

REST API Query Endpoint – SuiteQL

For more sophisticated and complex data retrieval requirements, the query endpoint offers enhanced capabilities through SuiteQL queries, providing greater flexibility to execute a wide range of queries using joins to return multiple record types. While SuiteQL leverages SQL fundamentals, developers familiar only with basic SQL may require training to use SuiteQL effectively. It is important for queries to be written correctly as the performance of a query depends on the choice of clauses, join types, functions, data size, etc. For more information, see SuiteQL Best Practices.

REST API Query Endpoint – Datasets

SuiteAnalytics Workbooks features dataset and workbook objects that allow you to analyze and visualize record data in a NetSuite account through the UI. Datasets are the basis of workbooks and allow the combination of data from different record types, similar to SuiteQL queries. The REST API query endpoint allows you to execute requests to fetch data from both standard and custom datasets in a given NetSuite account. However, to query a dataset, the record must exist in the UI as it cannot be created through REST web services.

REST API Limitations and Considerations

Account Tier and SuiteCloud Plus Licenses

NetSuite has a limit on the number of concurrent web services requests that can be processed depending on the customer's NetSuite Account Service Tier and the number of SuiteCloud Plus licenses. For example, the Standard Tier has a base limit of 5 and is increased by 10 for each SuiteCloud Plus license up to the maximum concurrency of 15. The limit can also be increased by upgrading to a higher tier. It is recommended to build your application to be flexible enough to handle requests for Standard Tier accounts and accounts with higher web services bandwidth and data volume needs.

Accounts have a limit on the number of requests that can be made via web services and RESTlets, depending on their service tier. The Standard Tier is the most common, with a base limit of 5 requests. If they purchase an SuiteCloud Plus license, the limit increases to 15 (5 based + 10 additional).

Web Services Governance and Concurrency Limits

All external application requests are included in concurrency calculations, regardless of their integration method. Concurrency is defined as the number of combined web service requests that an account can process simultaneously. For example, if an account has a base limit of 5 and 6 integrations send a request at

the same time, only 5 will be processed. It is important to note that customers may allocate part of their account's current request limits to specific integrations, which may or may not include your own application.

Maximum Number of Records That can be Fetched per Request

The results of a SuiteQL query, a GET request to obtain a list of record instances, and a saved analytics dataset are returned in one or more pages. The default setting displays 1,000 results per page up to a maximum of 1,000 pages. Without SuiteAnalytics Connect, you can display a maximum of 100,000 results; with SuiteAnalytics Connect, the maximum number of results you can display is 1 million. *It is important to note that SuiteAnalytics Connect is a paid feature.*

Given these limitations, in some cases, it might be better to run a GET request to query an existing dataset, or a POST request to the SuiteQL endpoint, as both can return all available information of a given record type with fewer requests. For example, your application needs to fetch ~10,000 purchase orders (body and line item data) from a NetSuite account once every day. Here are three approaches you might consider to retrieve the data you need:

A. Using a GET request to the record service, your application could send a request to fetch the list of internal IDs and another request for each ID to retrieve record field and sublist data:

```
GET:
https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/purchase order
where
  • the total number of records to be retrieved is 10,000
  • the maximum number of results returned per request is 1000 for a maximum of 1000 pages
  • the total number of GET requests required is 10 (10,000 records/1000 max results per page)

GET * N:
https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/purchase?limit=1000& offset=1000
where
  • N is the number of requests to be executed to obtain all 10,000 purchase orders (N = 9)

GET * N:
https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/purchaseorder/{id}
where
  • N is the number of purchase orders obtained in the response of the first request (N = 10,000)
```

B. A SuiteQL query can achieve the same results without requiring the third GET statement from above:

```
POST:
https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/suiteql

BODY:
{"q":"SELECT * FROM Transaction WHERE Type ='PurchOrd' "}

Note: For simplicity, SELECT * is used to return all columns, but it is
recommended to query only the necessary fields in your request

GET * N:
https://demo123.suitetalkapi.netsuite.com/services/rest/query/V1/suiteql?limit=
1000&offset=1000
where
  • N is the number of requests to be executed to obtain all 10,000 invoices
    (N = 9)
```

C. A dataset query can achieve the same results as a SuiteQL query:

```
Dataset ID: custdataset_my_purchaseorders

GET:
https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/dataset/custdataset_
my_purchaseorders/result

GET * N:
https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/dataset/custdataset_
my _purchaseorders/results?limit=1000&offset+1000
where
  • N is the number of requests to be executed to obtain all 10,000 purchase orders
    (N = 9)
```

The main difference between Approach A (GET requests to the record service) and the two other approaches (B and C) is that Approach A does not require individual requests to fetch both header and line body fields for each transaction. This reduces the number of requests made to the NetSuite account to retrieve all the desired records' data, thereby reducing the possibility of your application encountering bounced-back requests due to concurrency limits being reached, as well as reducing the need to write logic in your application to retry failed web service requests for individual record data. Nevertheless, logic still needs to be incorporated to handle failed requests due to concurrency limits, server errors, etc., to ensure seamless integration between your application and NetSuite.

Other Limitations of the REST API

- The SuiteQL endpoint does not support asynchronous tasks, but you can use a RESTlet and run the query asynchronously using the N/task module.
- Batch processing is not supported.
- A GET request only returns a single record instance not the full list of records like SOAP – for that you need to use the SuiteQL endpoint.

Review the SuiteTalk REST Web Services API Guide in the NetSuite Help Center for additional information on benefits and limitations of using the REST API.

1.3.4 SuiteQL with Connect Service

SuiteQL is a query language based on the SQL-92 revision of the SQL database query language. It provides advanced query capabilities that you can use to access your NetSuite records and data, and it supports querying the analytics data source.

SuiteQL is currently available using SuiteAnalytics Connect, the N/query module in SuiteScript, and SuiteTalk REST web services.

To execute SuiteQL queries through REST web services, a POST request must be sent to the SuiteQL resource, and the query must be specified in the request body after the query parameter *q*. For example,

```
> POST https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/suiteql
> Prefer: transient
. {
.   "q": "SELECT email, COUNT(*) as count FROM transaction GROUP BY email"
. }
```

Pros

- SuiteQL queries can be created using the syntax for either SQL-92 or Oracle SQL, but cannot be mixed
- Creation of more complex query requests via POST HTTP verb by using SQL is allowed

Cons

- Read-only access
- The casing of record type names and field names in SuiteQL query results may not always be consistent and can change
- Not every SQL function is supported (see NetSuite help for full list)
- Limited built-in functions are available (see NetSuite help for full list)

- You need to create, deploy, and maintain scripts in your NetSuite account (comparing to REST Web Services where no customizations are placed in target accounts)

1.3.5 RESTlet (SuiteScript script type)

A RESTlet is a server-side SuiteScript script type that is made available for external applications to invoke. RESTlets are built using the SuiteCloud Development Framework (SDF) and can be deployed as SDF SuiteApps in the SuiteApp Control Center.

RESTlets can be useful when you want to bring data from another system into NetSuite, or if you want to extract data from NetSuite.

Since a RESTlet allows developers to create RESTful interfaces by accessing them through a proper authentication method (TBA or OAuth), it is possible to create a custom REST API.

Pros

- Built-in HTTPS listener, no need for additional server
- Can be packaged and pushed to accounts via the SuiteApp Control Center
- Built-in debugging tools (debugger, execution log)
- General availability, backed up SuiteScript engine
- Allows developers to create their own functions/routines
- Allows developers to create their own logging reports

Cons

- Limited thread execution time
- Limited by script governance for total I/O execution per thread
- Limited HTTP verbs supported (only GET, READ, POST, DELETE and PUT)

1.3.6 ODBC/SuiteAnalytics Connect PRO

The SuiteAnalytics Connect Service allows users to archive, analyze, and report on NetSuite data. If a company enables the Connect Service, NetSuite offers ODBC, JDBC, and ADO.NET drivers that can be downloaded and installed to use the Connect Service.

Note: SuiteAnalytics Connect is an add-on module for customers, therefore SuiteApps that use it will incur additional costs for their customers if they do not already have the module.

Pros

- Open standard SQL
- No coding needed to access data

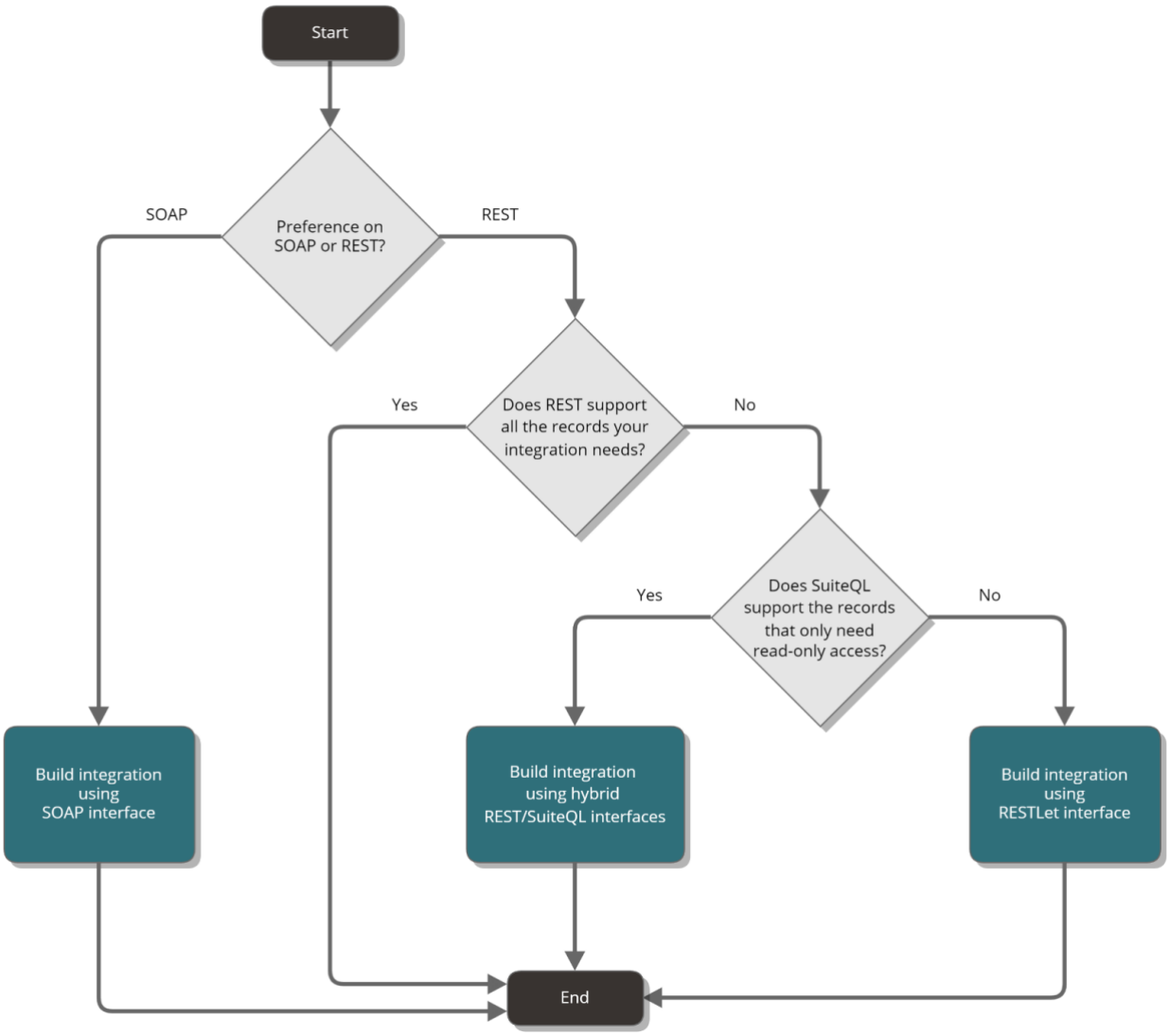
- Any standard third-party app that supports ODBC/JDBC can be used to execute SQL against the NetSuite schema
- Best option for data dumps
- No session handling is required
- No need for asynchronous requests
- Less vulnerable to timeouts

Cons

- Read only access
- Additional license charge to customers
- Not a true API
- Data is exposed as views, not database tables
- Single threaded
- No parallelism supported
- OAuth/SSO not supported (same password policy used in the NetSuite UI)
- Integrated SuiteApps built on it are not eligible for Built for NetSuite validation

1.3.7 Using the Right Interface

The following is a simple decision diagram to help developers new to SuiteTalk decide which API interface to use.



1.4 Developing for a Distributed Environment [Deprecated]

Although NetSuite customer accounts have been hosted in multiple data centers since 2012, in the current release SuiteApp developers do not need to know in which data center the account resides in. With the introduction of account-specific domains, the only important parameter is the account id.

1.4.1 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. When users access a native SuiteApp 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.

1.4.2 Integration SuiteApps That Use Web Services and/or RESTlets

When your Integration SuiteApp interacts with NetSuite by using SOAP Web Services or RESTlets, you need to connect to the Web Services endpoint or RESTlet URL.

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 the domain https://123456.restlets.api.netsuite.com. Existing integrations that rely on data center-specific domains are no longer supported starting with the 2020.1 release.

The easiest way to programmatically determine the correct domains is to use the SOAP getDataCenterUrls operation or the DataCenterUrls REST operation. These methods do not require authentication, and only have the account ID as a parameter. Its response includes the system domain, web services domain, and RESTlet domain.

The steps below should be followed for an Integration SuiteApp which interacts with SOAP web services or RESTlets to work on all types of accounts (please refer to Principle 6: Test Your SuiteApps for details on sandbox accounts):

1. Use the SOAP getDataCenterUrls or REST DataCenterURLs operations to obtain the correct service URLs for a given account ID.

2. Append the required service path to the URL to construct the complete URLs for API calls. For SOAP web services this means the endpoint URL including the version. For RESTlets this means the script ID and deployment ID.

3. Send the request using the URL constructed in step 2.

4. Optionally, store the service URL to avoid making frequent unnecessary getDataCenterUrls calls.

For reference, the following is a sample of a getDataCenterUrls request and response:

```
<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_2020_1.platform.webservices.netsuite.com'>
    <soapenv:Header/>
    <soapenv:Body>
        <getDataCenterUrls xsi:type='platformMsgs:GetDataCenterUrlsRequest'>
            <account xsi:type='xsd:string'>TD1268222</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_2020_1.platform.webservices.netsuite.com">
            <platformMsgs:nsId>
WEBSERVICES__05302019162513373826575559_a2f4d0dbe5b8a584b4992</platformMsgs:nsId>
        </platformMsgs:documentInfo>
    </soapenv:Header>
    <soapenv:Body>
        <getDataCenterUrlsResponse
            xmlns="">
            <platformCore:getDataCenterUrlsResult
                xmlns:platformCore="urn:core_2020_1.platform.webservices.netsuite.com">
                <platformCore:status isSuccess="true"></platformCore:status>
                <platformCore:dataCenterUrls>

<platformCore:restDomain>https://td1268222.restlets.api.netsuite.com</platformCore:rest
Domain>

<platformCore:webservicesDomain>https://td1268222.suitetalk.api.netsuite.com</platformC
ore:webservicesDomain>
```

```
<platformCore:systemDomain>https://td1268222.app.netsuite.com</platformCore:systemDomai
n>                </platformCore:dataCenterUrls>
            </platformCore:getDataCenterUrlsResult>
        </getDataCenterUrlsResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

The following is a sample invocation of the getDataCenterURLs API 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_2020_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_2020_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:webse
rvicesDomain>

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

1.4.3 Integration SuiteApps That Use REST Web Services

For the REST Web Services, only account-specific domains are supported since release 2019.1. Follow these steps for Integration SuiteApps which interact with REST web services:

1. The service URL can be found in Setup -> Company -> Company Information -> Company URLs under SUITETALK (SOAP AND REST WEB SERVICES) section (for example 123456.suitetalk.api.netsuite.com where 123456 is the account ID). The service URL should be configurable parameter.

2. Use the service URL together with the REST service endpoint path (/rest/platform) and version to construct the complete URL for the API call.

REST web services API is GA starting in release 2020.1 but partners can only rely on record types exposed to REST webservices which are NOT signed as Beta. They can find it either programmatically or within REST Web Services API browser available in Help Center.

1.4.4 Execution Environment Considerations

A SuiteScript script can be executed in different environments. Some scripts may need to programmatically determine their execution environment and run accordingly.

For example, a script that submits transaction data to a governmental or regulatory body may do so by making an outbound call to an external host with the transaction data as the payload. Such a script may not need to make the outbound call if it is running in an environment that is not of “PRODUCTION”. This means when transactions are created in a user acceptance test (UAT) setting in a test environment, such as “RELEASE PREVIEW” or “SANDBOX”, or "INTERNAL", the script will not make the outbound call to submit the transaction.

Another example is a script that makes an outbound call to an external service that returns a tax rate or shipping cost. These services may charge the customer on a per API call basis. Therefore, it is important that these outbound calls predicate on the environment in which the script executes – the script should either avoid making such a call, or make a call to the service’s test host if there is one.

The SuiteScript API to programmatically determine the execution environment is `runtime.getCurrentScript()`, `runtime.getCurrentSession()`, and `runtime.getCurrentUser()`. Please refer to the SuiteScript documentation in the NetSuite Help Center for the list of valid returned values.

The REST WS API is GA starting 20.1, but partners can only rely on record types exposed to REST web services which are NOT signed as Beta. That information can be found either programmatically or within REST WS API browser available in NetSuite Help Center.

1.4.5 Built For NetSuite (BFN) Verification Scope

The NetSuite BFN verification, listing, and badging process has purview over SuiteApps that integrate with:

- The Core ERP
- SuiteCommerce Advanced (SCA)
- SuitePayment
- SuitePeople (requires NetSuite Product Management approval)

If other NetSuite modules are required, the SuiteApp is not eligible to be BFN certified. Please contact your Strategic Alliance Manager for more information or to discuss ideas for a new SuiteApp.

You are free to use any of our active tools and services. The following are considered the most current tools and services and are therefore future proof:

- SuiteTalk REST or RESTlets (instead of SuiteTalk SOAP)
- SuiteScript 2.x (instead of SuiteScript 1.0)
- SuiteAnalytics/Workbook (instead of saved searches)
- SDF-developed and SACC-deployed (instead of SuiteBundler)

Please keep a list of modules, tools, and services that your SuiteApp requires so that we may properly profile your SuiteApp.

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

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 most cases, the use of a Suitelet without login is strictly prohibited and will not be BFN approved. This is because it presents a distributed denial of service (DDos) vulnerability, in which the attacker exploits seemingly legitimate requests.

The following are Strictly Not Allowed:

- Suitelets the present endpoints that accept public GET/POST requests
    - These are vulnerable to HTTP Flood attacks
- Suitelets that are used for Webhook Monitoring
    - These are vulnerable to HTTP Flood attacks
    - Best practices for this are to accept webhooks external to NetSuite, accumulate/summarize data, then pass a scheduled summary to NetSuite
- Suitelets that present an external form to non-licensed users

The following are Allowed:

- Suitelets that are solely invoked by a server side script
- Suitelets that use tokenized payment links, provided they are:
    - Account-specific URL’s, and
    - One-time use URL’s, and
    - Tokenized/hashed URL query parameters, and
    - Perform data validation, and
    - HTTPS only

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 sections list technical factors that should be considered when designing a common solution for NetSuite OneWorld and single instance NetSuite.

1.6.1 The Subsidiary Field

In NetSuite OneWorld, the Subsidiary field is a mandatory field for most standard records. Sometimes, the Subsidiary field acts as a filter to other fields. For example, the Subsidiary field in the Sales Order record might filter the available selections in the Item 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 some records that are unique to NetSuite OneWorld. One 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

SuiteCommerce Advanced provides developers 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.

Extensions that you develop should adhere to the best practices and standards established by this framework (https://developers.suitecommerce.com) and they should be designed 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.

1.7.1 Translation for SCA Extensions

With the expansion of NetSuite around the world, customizations face a growing requirement for localization. This section describes how translation for SCA extensions was previously performed and how new techniques can be used.

How Translation Used To Be

Whenever an extension needed to be translated, a set of JavaScript files named after the location had to be created under a folder called *Languages*, for every application. For example,

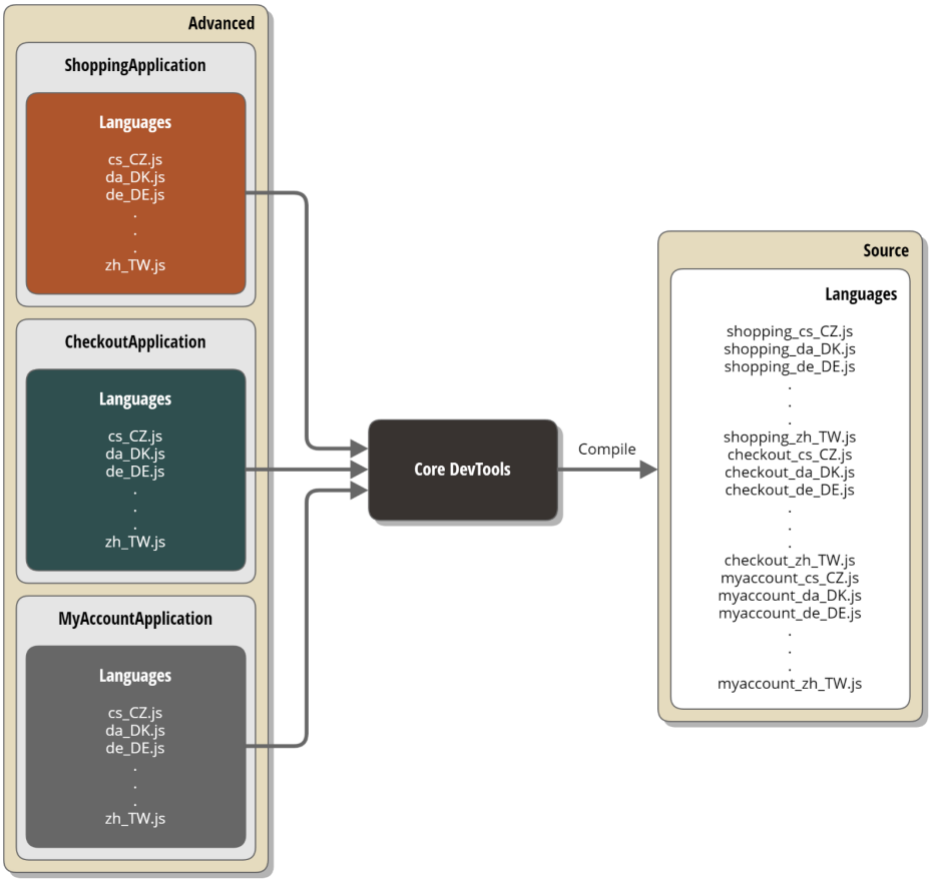
- ShoppingApplication/Languages
- CheckoutApplication/Languages
- MyAccountApplication/Languages

Localization files were then stored inside these folders. An example localization file is shown here:

shopping_es_ES.js

```
SC.Translations={
    "$(0) Product": "$(0) producto",
    "$(0) Products": "$(0) productos",
    "$(0) Products, $(1) Items": "$(0) productos, $(1) artículos",
    "$(0) Products, 1 Item": "$(0) productos, 1 artículo",
    ...
}
```

After the solution is bundled and uploaded to NetSuite, individual localization files were combined into one file and stored in a folder named *languages*, as shown in the following figure:



New Recommended Solution

The new recommended solution to add localization to an extension is to create JSON translation files and store them in the Translations folder. Then, references to these translation files must be added to the *manifest.json* file for the SuiteApp.

An example JSON translation file is shown here:

Translations/Languages JSON example file

```
{
 "$(0) Product": "$(0) producto",
 "$(0) Products": "$(0) productos",
 "$(0) Products, $(1) Items": "$(0) productos, $(1) artículos",
 "$(0) Products, 1 Item": "$(0) productos, 1 artículo",
 "$(0) Results for <span class=\"facets-facet-browse-title-alt\">$(1)</span>": "$(0)
resultados para <span class=\"facets-facet-browse-title-alt\">$(1)</span>",
 "$(0) Results for <span class=\"facets-faceted-navigation-title-alt\">$(1)</span>":
"$(0) resultados para <span class=\"facets-faceted-navigation-title-alt\">$(1)</span>",
```

```
 "$(0) items": "$(0) artículos"
}
```

An example manifest.json file is shown here:

manifest.json example file

```
{
    "name": "MyExtensionWithTranslations",
    "fantasyName": "My Extension With Translations",
    "vendor": "Acme",
    "type": "extension",
    "target": "SCA,SCS",
    "target_version": {
        "SCA": ">=18.1.0",
        "SCS": ">=18.1.0"
    },
    "version": "1.0.0",
    ...
    "translations": {
        "applications":{
            "shopping":{
                "es_ES": "Modules/MyModule/Translations/es_ES.json",
                "it_IT": "Modules/MyModule/Translations/it_IT.json"
            }
        }
    }
    ...
}
```

Notes

- Translations will be exposed in *SC.TranslationsExt* by the *Utils.Translate* method
- The new solution will only be available in SCA 20.2+ versions

1.8 Connector SuiteApps Considerations

Most Integration SuiteApps provide functional capabilities to end users while using the SuiteCloud integration interfaces 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 three 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 includes all synchronous and asynchronous operations, and all authentication methods. This is not to be confused with support for all 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 described in this guide. Failure to do so could result in the SuiteApp not successfully completing the SDN quality verification process.

1.9 SuiteTax Considerations

Note: As of release 2024.2, SuiteTax is the default framework for most new NetSuite accounts. Therefore, it is important to ensure your SuiteApp is compatible with both SuiteTax and legacy framework, if applicable.

SuiteTax is a new framework in NetSuite ERP that provides the calculation and storage of indirect tax charged in transactions (e.g., sales tax, GST/PST, VAT, etc.).

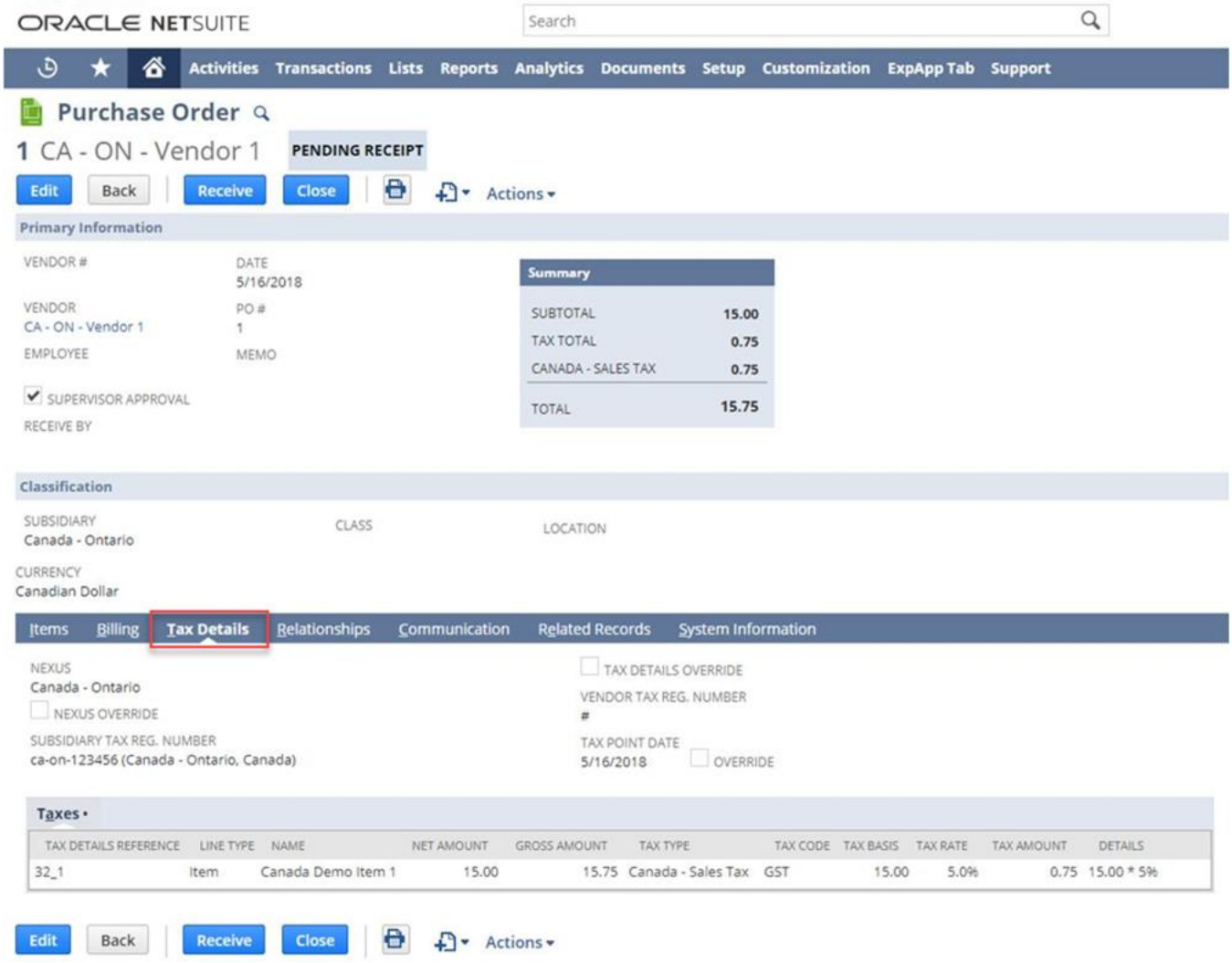
SuiteTax is tightly integrated with all pertinent ERP transactions and offers the following key benefits over the legacy taxation way of calculating and storing indirect tax in NetSuite:

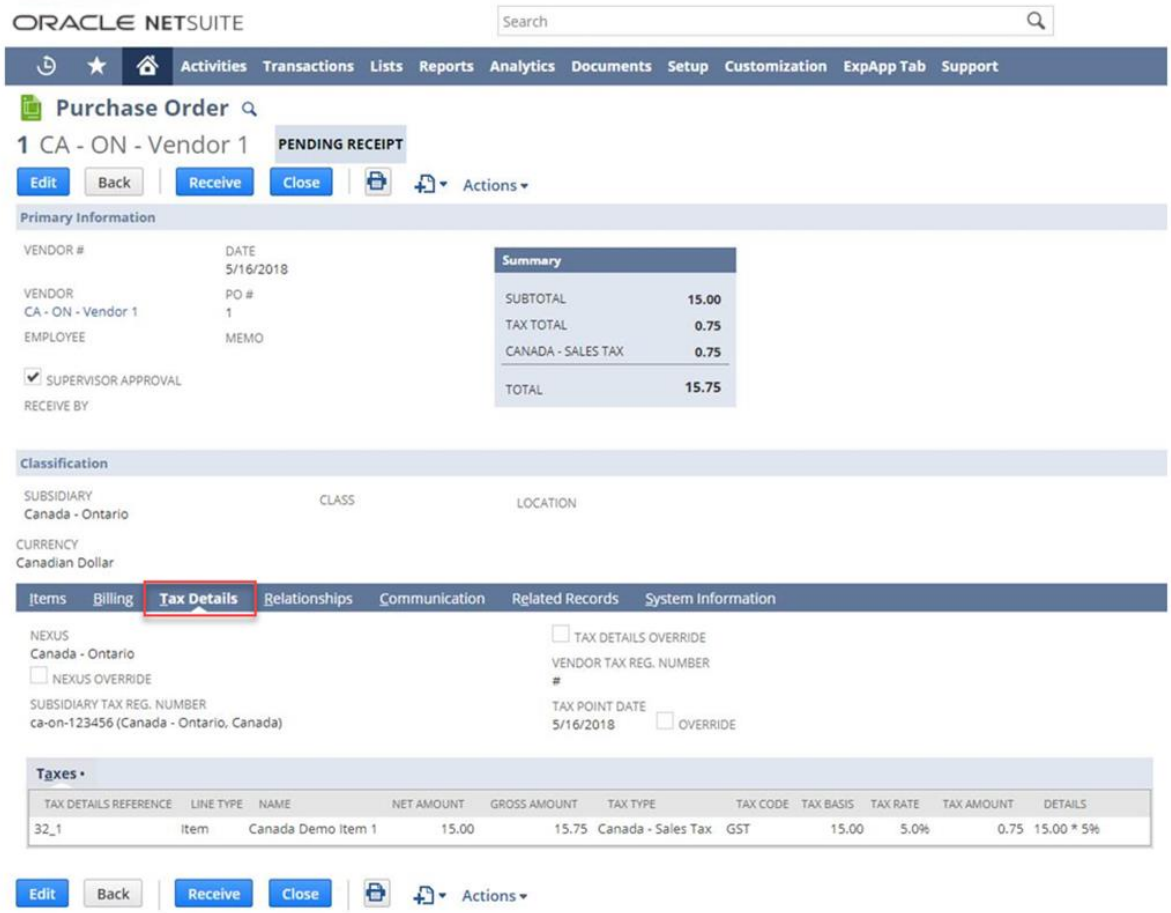
- Flexibility to support country-specific needs in sales tax calculation and reporting
- A third-party sales tax calculation engine can be used in the event a SuiteTax account's administrator chooses to forego the default sales tax engine. Such third-party engines must be available from a SuiteTax vendor who used the SuiteTax API.

A NetSuite account with legacy taxation can be migrated to SuiteTax; however, legacy taxation and SuiteTax cannot both be enabled at the same time. It is important to note that once SuiteTax is enabled, you cannot go back to using legacy tax calculation. In a future NetSuite release, SuiteTax will become the default sales tax calculation engine for newly provisioned accounts without any need for data migration. As of version 2020.1, the NetSuite SuiteTax engine is released and supports a growing list of countries (US, Canada, India, Brazil, VAT). Support for SuiteCommerce is available for the US market as of version 2020.1.

1.9.1 How Does SuiteTax Impact SuiteApps?

When SuiteTax is enabled on a NetSuite account, some of the sales tax related fields on transactions are replaced by a new Tax Details sublist. This new sublist contains additional information which contributes to the SuiteTax benefits listed above. An example of the Tax Details sublist is shown below on a Purchase Order transaction:





The new and restructured taxation information means the underlying data schema has also changed. Due to these changes, any SuiteApps that perform I/O on TaxCode or TaxRate as if they were still in the Items sublist may encounter compatibility issues when run on a SuiteTax account. The following tables lists three use cases where problems may occur:

Use Case	Solution
Reading the TaxCode or TaxRate field without performing a null check.	Perform null checks on these fields.
Programmatically searching for transactions in a NetSuite account running on Legacy tax account while using TaxCode and/or TaxRate as filters or columns returns an error. This error occurs because these fields are no longer in the Items sublist.	Reference the same fields (TaxCode, TaxRate) in the new Tax Details sublist.
Running a saved search that reference tax-related fields that have been removed will not work because tax-related field have been streamlined in SuiteTax.	Remove the reference to the fields in the saved search. Use the SuiteScript Records Browser (available in the NetSuite Help Center) to find information on the new fields in the Tax Details sublist.

1.9.2 How Do I Ensure My SuiteApp's Compatibility with SuiteTax?

You can find the exhaustive list of fields and sublist changes in the SuiteTax Migration Guide, available in the APC portal. In order to test for compatibility, you may request a SDN account with SuiteTax enabled by open a support case.

1.9.3 How Do I Programmatically Determine if the Runtime Environment is SuiteTax or Legacy Tax?

The following SuiteScript code can be used to programmatically determine whether the runtime environment is SuiteTax.

```
define(['N/runtime'], function(runtime) {
    const self = {
        isSuiteTaxEnabled: function() {
            return runtime.isFeatureInEffect({
                feature: 'tax_overhauling'
            });
        },
    };

    return {
        isSuiteTaxEnabled: self.isSuiteTaxEnabled,
    };
});
```

Currently, there are no APIs to achieve the same using SuiteTalk (REST and SOAP). The recommendation is to store a configurable SuiteTax/Legacy setting in the external system that is invoking the SuiteTalk web service endpoints to help make the correct I/O operations on tax related fields.

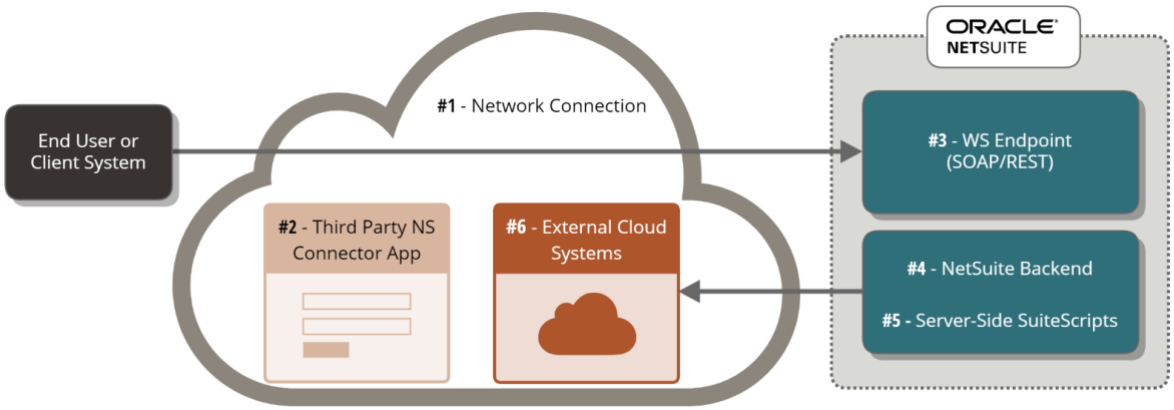
1.10 Building Fault Tolerant SuiteApps

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 components. In these exceptional situations, the enterprise application may suffer degradation in quality or performance; however, 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 guide. However, this section describes potential failure points and design considerations that are specific to SuiteApps.

1.10.1 Potential Failure Points

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



The most common potential failure points are:

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, Integration SuiteApps should log the failure and execute re-try logic. Users may also be notified, if applicable.
2. **Third-party NetSuite Connector SuiteApps** – Connector SuiteApps provide an abstract layer above the underlying SuiteCloud integration interfaces (see Creating Extensions for SuiteCommerce Advanced for details). These SuiteApps can be used as part of the infrastructure for Integration SuiteApps. Only those Connector SuiteApps that have obtained the Built 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 Integration 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 Integration 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 4.6.2 Using the ExternalId field When Importing Critical Business Data for details.
5. **Server Side SuiteScript** – Events such as NetSuite upgrades and unplanned system outages will impact all server side SuiteScript scripts. 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 actions you can take to minimize data loss and data integrity issues caused by these system disruptions are:

a. **Examine the type argument** – Upon entry of a scheduled script, the system-generated type argument must be examined to determine the circumstances that started the script execution. The specific type values that are relevant to fault tolerance are “aborted” and skipped”. The aborted type value indicates the script was re-executed after execution was aborted due to a system failure, and a recovery point was not set (see Set recovery points). 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. The skipped type value indicates execution of the script was postponed due to system downtime, but was 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. **Set recovery points** – To minimize issues caused by unplanned system failures, a scheduled script’s state of execution should be preserved 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 units 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 N/http API (or the credentials-focused API, N/https). 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, the SuiteApp will appear to have 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 within the SuiteApp.

1.11 Considerations when Building with SuiteSuccess

1.11.1 The SuiteSuccess Initiative

SuiteSuccess is the go-to-market model for NetSuite. During the second half of 2019, nearly 100% of customers were sold and implemented using SuiteSuccess. SuiteSuccess includes four major components:

- An End-to-End customer engagement model which focuses on business results, processes, and a stairway of growth.
- A verticalized solution with pre-configured leading practices delivered as SuiteApps with SKUs.
- A more prescriptive approach to selling – leading customers with experience rather than asking what they want the system to do.
- A revamped delivery model which accelerates time to value and go-live.

For partners, the impacts are many, but for the purposes of SAFE and BFN 2020.1, we will focus on the first technical effects of SuiteSuccess upon the SAFE guidelines. These can be found in Principle 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 preinstalled, in the account. The method of customization is via the SuiteCloud platform, and is ultimately delivered using SuiteBundler (migration to SDF and SuiteApp Control Center is currently underway). 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 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 a way that fits 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 initially sold to the new customer.

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

You must also 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 preexisting bundle residing in a SuiteSuccess account.

The initial set of principles documented in this SAFE guide address some of these best practice avoidances. For this reason, this SAFE guide now mandates that SuiteApps be developed in a non-SuiteSuccess account. This will ensure that no SuiteSuccess objects are added, changed, deleted or 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 provided in Principle 6: Test Your SuiteApps.

1.12 Working with NetSuite Technical Teams

As a Select level ISV partner, you will be working with various teams in NetSuite in the lifecycle of your SuiteApps. Some of these teams are technical; they can 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.

1.12.1 The SDN Solutions Engineering Team

The SDN 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 adherence to SAFE principles and with 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.

1.12.2 The SDN Quality Assurance Team

The SDN Quality Assurance (QA) team fulfills two primary roles. The first role is the administration of the BFN, or Built for NetSuite, program. BFN is the SuiteApp verification process. SDN QA team members can also help guide SDN partners throughout the BFN review process. 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 providing 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.

1.12.3 The NetSuite Product Management Teams

The Product Management (PM) teams 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 a 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.

1.12.4 The NetSuite Technical Support Team

The NetSuite Technical Support team is skilled in all aspects of NetSuite, and its members service both customers and partners. SDN Select level 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.

1.13 NetSuite Technologies and Functional Modules in SuiteApps

The NetSuite Product Team is responsible for releasing (and eventually deprecating) any NetSuite features/technologies such as SuiteTalk REST, SuiteTalk SOAP, RESTlets, SuiteScript, SuiteQL, SuiteAnalytics Workbooks, Saved Searches, etc. The Product Team also releases NetSuite functional modules such as SuiteAnalytics, SuitePayment, SuitePeople, SuiteCommerce Advanced, SuiteTax, SuiteGL, etc.

The NetSuite SDN Engineering team via the BFN Verification Program permits and regulates *the use by Partners* of both the technologies and functional modules in SuiteApps. In general, actions by both the Product Team and the Engineering team s are aligned; however, the BFN Verification Program needs to consider the fact that new SuiteApps may be in development for possibly months prior to BFN Verification– the true start of a SuiteApp's useful life as a product–a lifetime which could ultimately be years. The BFN Verification Program is also concerned with SuiteApp best practices, security, inter-operability, deployment, stability, and trackability.

When building SuiteApps, it is usually wise to use the latest NetSuite technologies and functional modules. In addition to an increase in performance, the use of our latest optimized technologies allows Partners to best future proof their SuiteApp offerings. We realize that the choice of technologies is sometimes complicated and a function of many factors such as development team capabilities, migration effort, installed base preferences, and to some extent, habit.

After a technology is replaced or otherwise dated, a Partner's migration effort typically continues to increase. Jumping to an optimized technology sooner rather than later is usually smart in terms of overall ROI.

The following table shows an example of current and changes in support for the SuiteTalk SAOP, SuiteBundler, and SuiteTalk (1.0, 2.0, 2.x) technologies relative to our bi-annual release cycles. When building and maintaining SuiteApps, the changes must be taken into account.

	New SuiteApps		Existing SuiteApps	
	24.1 Cycle Starting January '24	24.2 Cycle Starting July '24	24.1 Cycle Starting January '24	24.2 Cycle Starting July '24
SuiteTalk SOAP	• Do not use - Strong Warning • Replaced by SuiteTalk REST	• Do not use - Mandated • Replaced by SuiteTalk REST	Not Restricted	----
SuiteBundler	• Do not use - Mandated • Replaced by SDF/SACC	----	Not Restricted	----
SuiteTalk 1.0, 2.0, 2.X	• Do not use - Strong Warning • Replaced by SuiteScript 2.1	• Do not use - Mandated • Replaced by SuiteScript 2.1	• Migration to 2.1 is suggested	----

1.14 Multiple Administrators for DEV/QA/Deployment Accounts

When SDN partners are onboarded, they are provisioned with three separate SDN accounts: one for development, one for QA, and one for deployment purposes (see Setting Up the SDF SuiteApp Publisher Environment). 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 that employee leaves the company, this will lead to problems including the inability to create another user with administrative permission and the inability to develop and maintain SuiteApps. If this occurred in a deployment account, it 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.

1.15 Further Reading

You can find more information in the NetSuite Help Center or in SuiteAnswers, using the following search terms:

- *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. Principle 2: Manage SuiteScript Usage Unit Consumption

Managing the consumption of SuiteScript usage units (i.e., governance) must be a critical part of your SuiteApp design if high I/O volume is expected.

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

By following the SuiteScript usage governance suggestions described below, you can minimize 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 SuiteScript scripts, it is important to use only supported and documented APIs. Supported APIs can be found in the NetSuite Help Center. Additionally, SuiteScript does not support document object model (DOM) references.

2.1 Governance-related Issues

Many developers new to SuiteCloud development have a background of working on platforms that have little or no limits on I/O and/or CPU resources. A common hurdle that developers face when developing on the NetSuite platform is developing designs that are compatible with SuiteScript usage governance. This is especially problematic for applications that are I/O intensive, such as those that 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 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. Execution of the script cannot be resumed at the point of failure. As a result, applications that rely on the completion of a series of I/O instructions to be completed can potentially be terminated mid-execution, thus threatening data integrity upon which all downstream tasks and transactions rely.

Example

Consider a third-party shipping application built on the item fulfillment transaction. During the beforeSubmit event 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, it can reach usage limits 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

It can be difficult to resolve governance-related design flaws when they are discovered late in development or QA cycles. Such design flaws often require a significant redesign of the scripts. The redesign 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, redesigning can include 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 `search.lookupFields(options)` and `record.submitFields(options)` 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

NetSuite recommends the following script types and designs for handling higher volume I/O.

- User Event Scripts and Suitelets
- Delegating I/O to Scheduled Scripts
- Map/Reduce vs 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 approach is to use user event scripts and Suitelets to manage higher I/O volume. 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.

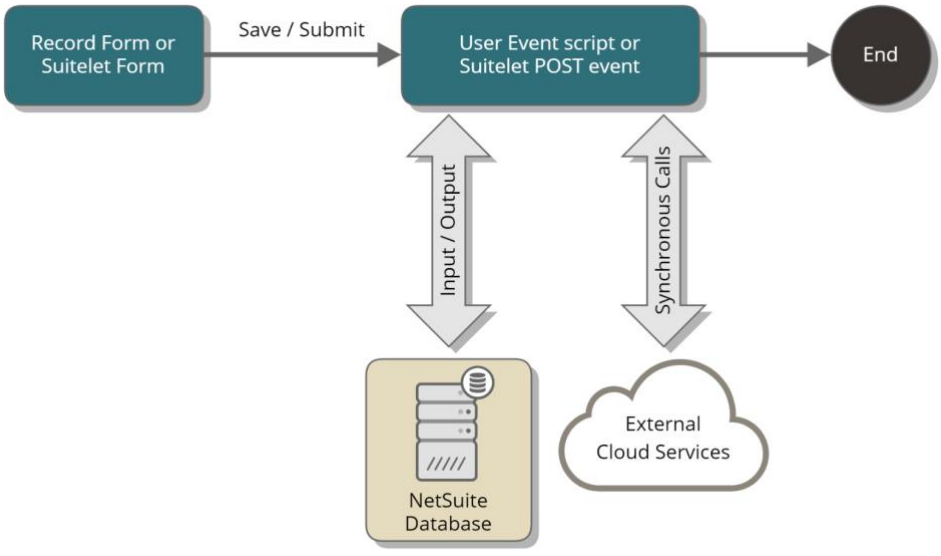
When writing user event scripts and Suitelets, you should respect the original design intent for both script types. NetSuite intends for user event scripts and Suitelets to be short-lived and lightweight provide the best user experience. Logic placed in user event scripts and Suitelet POST events is executed between the time a page is submitted and when 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 using the SuiteApp** may suffer. Therefore, you 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 using a scheduled script (as described in Delegating I/O to Scheduled Scripts). 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 a user event script or a Suitelet, the design goal should be to provide good user experience by writing code that executes the simplest logic possible, with a minimal 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) among all script types, units can be easily exhausted resulting in usage limit exceeded errors. When these types of errors are thrown, NetSuite terminates the execution of the script. (See Governance-related Issues for additional information.)

Also note that the APIs used to invoke external cloud-based services (N/http and N/https) execute synchronously. Consequently, invoking cloud-based external services from within user event scripts and Suitelets can also threaten the goal of keeping these scripts short-lived.

The following figure shows how user event scripts and Suitelets can be used to perform simple tasks programmatically:



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.
- User event scripts and Suitelets are invoked synchronously, thus the logic is triggered immediately.

Cons

- User event scripts and Suitelets execute synchronously, which means the user experience may be compromised if an excessive number of API calls is made.
- User event scripts and Suitelets have a smaller amount of allotted governance units, which can lead to more usage limit exceeded errors.
- 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 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.

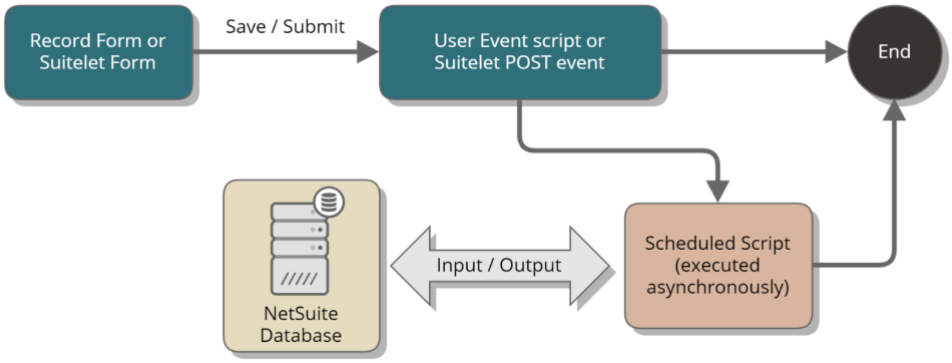
As a SuiteScript script developer, you can write user event scripts or Suitelets that delegate the "heavy lifting" of I/O API calls to a scheduled script. The caller script invokes a scheduled script designed to handle the bulk of I/O API calls by using the `task.create(options)` method. As soon as the API call is made, the control returns back to the caller script, thereby improving user experience. The user does not have to wait for API logic to finish executing. The invoked schedule script is picked by a scheduler which will decide the order of all jobs that are going to be sent to the processor pool and executed asynchronously. The scheduler decides by default based on priority and then on a timestamp of a given script deployment. Higher priority goes first, and if jobs have the same priority, the older has precedence. In 95% of scenarios, you don't need to play with priority elevation or processor reservation. When the scheduled script is close to reaching the usage limit, the script may call `task.create(options)` 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. Note that yields and recovery points are not manually scripted in SuiteScript 2.x. Robust scripts should use idempotent operations so the script can repeat itself with the same result. Alternatively, the script can begin cleanup or recovery operations if an abort event is detected.

To safely manage recovery states in a scheduled script it is more secure to use N/cache module rather than script parameters or .json file saved values.

Note: If your SuiteApp contains a high number of scheduled scripts and/or the scheduled scripts are invoked often, the standard two processors pool may not be able to process these scripts fast enough. You may consider using additional processors, which are included in the SuiteCloud Plus add-on module. The SuiteCloud Plus add-on module can be added to SDN partner test accounts free of charge, and is available to customers for a fee. Some customers may have additional processors available while others may not. In order to support all customers, your SuiteApp can query for the number of processors available in a customer account, and optimize scheduled script management according to whether the customer has more than two processors.

It's highly recommended to install the Application Performance Management (APM) SuiteApp in your account to analyze Web Services and SuiteCloud Processor's performance. The Concurrency and SuiteCloud Processor Monitor pages will allow you to obtain detailed information on the concurrency and requests being made to troubleshoot the error in reference and identify the culprit script(s). APM is also useful if you still have jobs running on scheduled queues.

The following figure shows how incorporating scheduled scripts into your script design can help manage high I/O volume:



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

- Scheduled scripts have a large number of allotted units making them well suited for high volume I/O.
- A scheduled script can call itself again (or call another scheduled script) to continue execution of very high volume I/O until the task is completed.
- Scheduled scripts provide a better user experience because time consuming I/O logic is executed asynchronously.
- Load balancing in ensured by the scheduler so that maximum throughput across all processors is the main preference by default.

Cons

- Designing scheduled scripts is more complex than designing user event scripts and Suitelets (see User Event Scripts and Suitelets).
- Asynchronous execution provided with scheduled scripts may not be feasible for some use cases, such as user-centric applications that require immediate feedback to be given to users.
- If there are dependent scheduled scripts and execution needs to be serialized, this needs to be scripted by adding logic to call dependent task(s) since order of execution for all jobs is handled automatically by the **scheduler**.

2.3.3 Map/Reduce vs Scheduled Scripts

As described in Delegating I/O to Scheduled Scripts, scheduled scripts can be a good solution when delegating the "heavy lifting" I/O API calls. However, NetSuite also offers the map/reduce script type which can speed up the processing of large amounts of data. Note that the map/reduce script is only available in SuiteScript 2.x).

A map/reduce script is best suited for situations where the data can be divided into small, independent parts. When a map/reduce script is executed, a structured framework automatically creates enough jobs to process all of these parts **unbeknownst to the user. And, these jobs can work in parallel with the level of parallelism 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 Scripts

As a rule of thumb, map/reduce scripts can be used for any scenario that requires multiple records to be processed because the logic can be separated into relatively lightweight segments. In contrast, a map/reduce script is not as well suited to scenarios where you want to enact a long, complex function for each part of your data set, such as loading and saving multiple records.

Other examples of when to use a map/reduce script include:

- Searching for invoices that meet certain criteria and applying a discount on each one
- Identifying a set of files in the File Cabinet, modifying them, and sending them to an external server
- Searching for duplicate customer records and applying certain business rule to the duplicates

When to Avoid Using Map/Reduce Scripts

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 when multithreading is needed.

Therefore, in scenarios where a parallel process depends on the update of another process, a map/reduce script would not be the best choice because the order in which each process will be triggered cannot be determined beforehand. Please refer to SuiteApp Designs and Concurrency Issues for concurrency related issues such as race conditions, and the methods used to address them.

Comparison of Scheduled Scripts and Map/Reduce Scripts

Scheduled Script	Map/Reduce Script
Processes run in a single thread	Processes run in multiple threads
Governance: • The memory limit for a scheduled script is 50 MB • The nlapiYieldScript() function must be called before this limit is reached	Governance: • The hard limit on total persisted data is 200MB • Hard limits on function invocations are: ○ getInputData function: 10,000 usage units ○ map function: 1,000 usage units (same as mass update scripts) ○ reduce function: 5,000 usage units ○ summarize function: 10,000 usage units • The soft limit on map/reduce jobs is 10,000 units
Manual yield	Automatic yield

Sample Map/Reduce Script (DANA is this in HC?)

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

The following script searches for transactions modified the previous day. The map function updates the memo field. The reduce function 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() {
        const today = new Date();
        const dd = today.getDate()-1;
        const mm = today.getMonth()+1;
        const yyyy = today.getFullYear();
        const yesterday = mm.toString() + '/' + dd.toString() + '/' + yyyy.toString();

        const myColumns = [{name:'internalid'},{name:'trandate'},{name:'recordtype'}];
        const myFilters =
[{name:'trandate',operator:'on',values:[yesterday]},{name:'mainline',operator:'is',valu
es:['true']}];
        return search.create({
```

```
            title: 'Sample Search',
            type: search.Type.TRANSACTION,
            columns: myColumns,
            filters: myFilters
        });
    }
    function map(context) {
        const searchResult = JSON.parse(context.value);
        const tranId = searchResult.id;
        const tranType = searchResult.values.recordtype;

        executeMapLogic(tranId,tranType);

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

    function executeMapLogic(tranId, tranType) {
        const 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) {
        const tranType = context.key;
        const tranId = context.values;
        const today = new Date();
        const dd = today.getDate()-1;
        const mm = today.getMonth()+1;
```

```
        const yyyy = today.getFullYear();
        const yesterday = mm.toString() + '/' + dd.toString() + '/' + yyyy.toString();
        const eBody = tranType + " transactions for " + yesterday + " : " +
JSON.stringify(tranId);
        const eTitle = 'Map Reduce result ' + tranType + " transactions for " +
yesterday

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

        const mySearch = search.create({
            type: search.Type.MAP_REDUCE_SCRIPT,
            columns: myColumns,
            filters: myFilters
        });
        mySearch.run().each(function(result) {
            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 because of the APIs these scripts used to perform mass creates and mass updates. When invoked repeatedly, the APIs can accumulate a sum that exceeds the amount of usage units allowed. The user experience of these scripts may 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. Scheduled scripts are executed asynchronously, but they 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 created en masse and updated programmatically, with a very low SuiteScript usage consumption and little adverse performance impact. This 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, and only the API calls spent on submitting the parent record consume SuiteScript usage units. All the child records are created or edited, essentially without consuming any SuiteScript usage units.

Parent-child relationships can also help create/update multiple 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 or updated are added to the parent record. When the parent record is saved, all changes made to the child records are also saved. Once 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, deleting them does not impact data integrity. And, because a child record can have multiple parent records, 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.

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

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

for(let 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.

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

const parentid = newparent.save();

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

const sublist = 'recmachcustrecord_sdn_child_parent';
parent.selectNewLine({
    sublistId: sublist
});
for(let 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.
- Child records are created and updated quickly, which 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 `search.lookupFields(options)` and the other searching APIs. Since you 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 cause a script to exceed its governance limits.

The searching APIs have established usage governance best practices and built-in capabilities that allow traversing to linked records. Please refer to the searching and governance topics in the NetSuite Help Center for more information.

An example of `search.lookupFields(options)` 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:

The following code sample shows how you can use the `search.lookupFields(options)` method to provide values in a linked record to obtain the supervisor for a given customer's sales rep.

```
const customerSalesRepSupervisor = search.lookupFields({
    type: 'customer',
    id: customer_id,
    columns: 'salesrep.supervisor'
});
```

However, if there is a need to traverse further down the chain of records, then more APIs need to be invoked. The following code sample adds to the sample above to include the sales rep's email and the supervisor's location :

```
const customerSalesRepSupervisor = search.lookupFields({
    type: 'customer',
    id: customer_id,
    columns: 'salesrep.supervisor'
});
```

```
const customerSalesRepEmail = search.lookupFields({
    type: 'customer',
    id: customer_id,
    columns: 'salesrep.email'
});

const supervisorLocation = search.lookupFields({
    type: 'employee',
    id: customerSalesRepSupervisor,
    columns: 'location
});
```

Note that each call to `search.lookupFields(options)` will consume SuiteScript usage units. While the above 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 to use a design pattern called the Transient Record Controller. This design pattern relies on two key elements to drastically reduce SuiteScript usage and provide fast data-read capabilities. These two key elements are: 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.

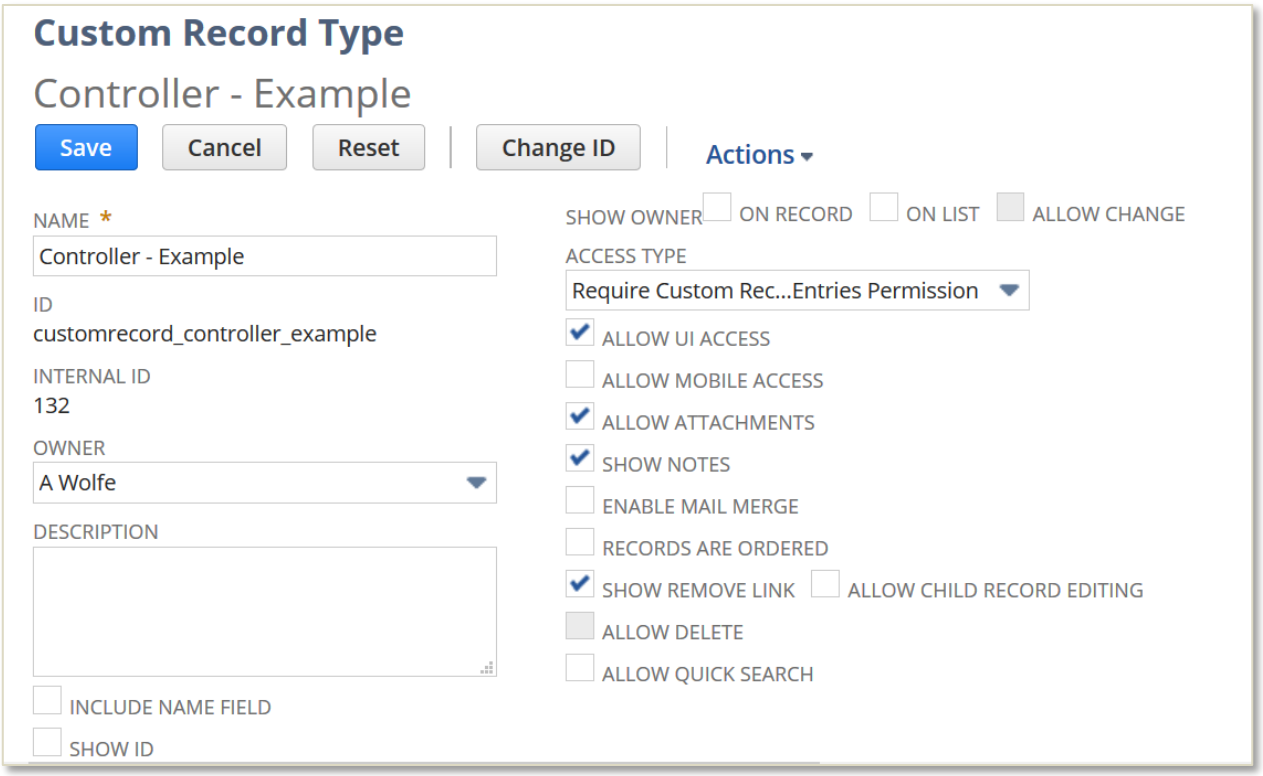
Constructing the Controller Custom Record Type

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

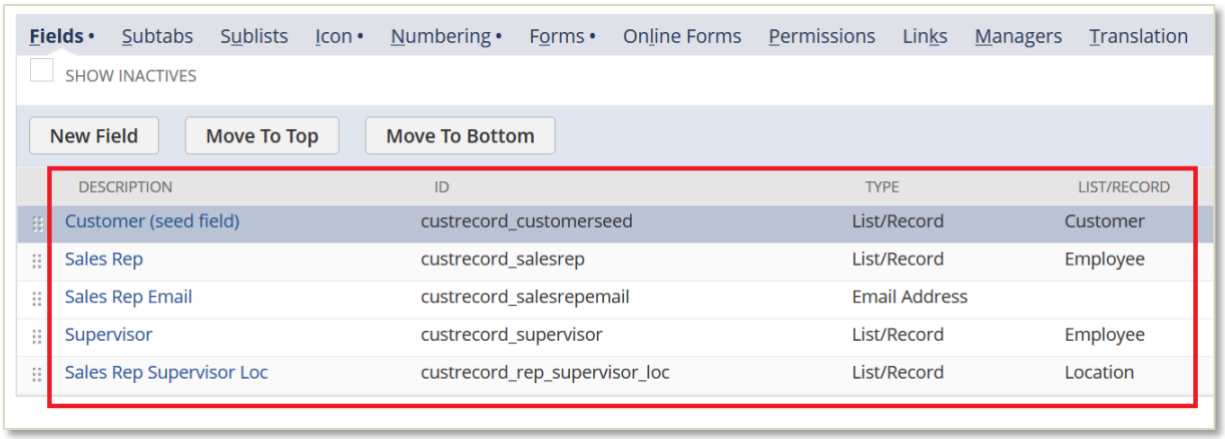
First, create the custom record type using a meaningful name and ID as shown below. Uncheck the Include Name Field setting (it is not necessary for this example).



Create the Fields

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. 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).

The following image shows how each field must be set up (note their IDs are meaningful):



- 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 the Employee list.
- The **Supervisor** field represents *customer.salesrep.supervisor*, which is an employee record. Therefore, it points to the Employee list.

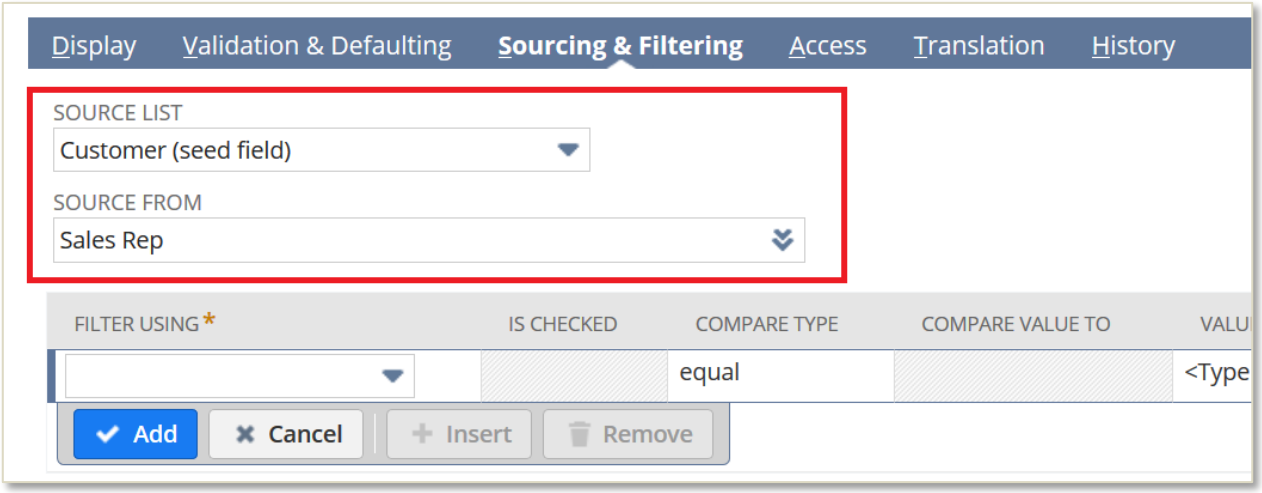
Note that the Customer field is called the "seed" field because it will be the first field to be set in server side memory using a SuiteScript and it will drive the data propagation.

Set Up Sourcing and Filtering

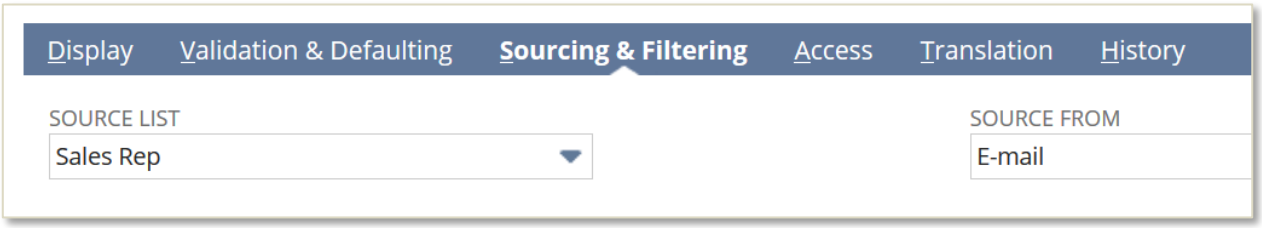
Once the custom fields are created, the next step is to set up their Sourcing and Filtering.

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

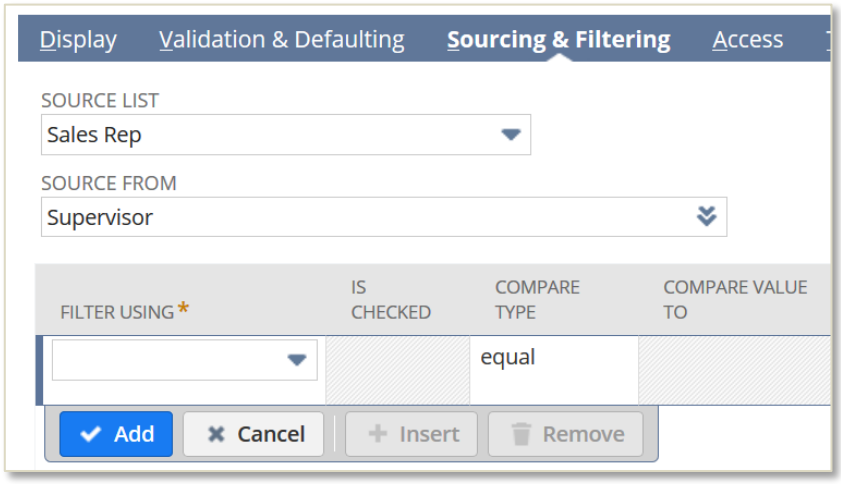
The next field is the **Sales Rep** field. Since it mimics *customer.salesrep*, its Source List will be **Customer (seed field)** and its Source From will be **Sales Rep** :



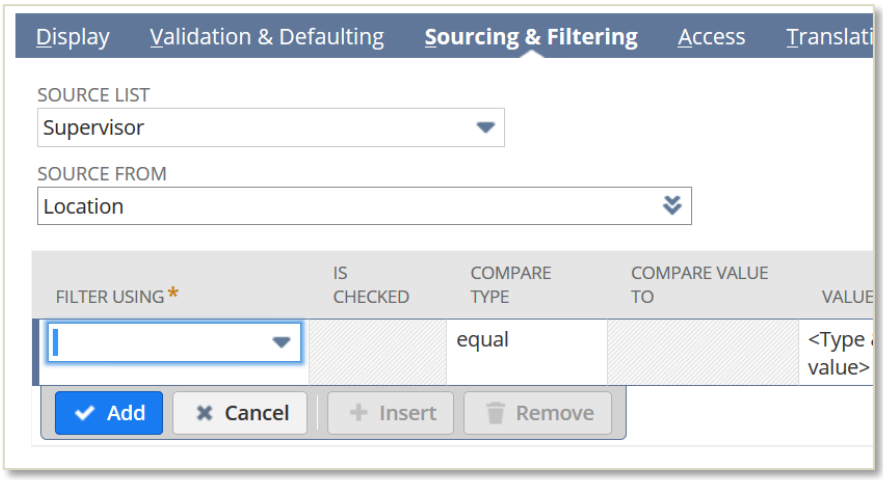
The next field is the Sales Rep Email field. Since it mimics *customer.salesrep.email*, its Source List will be **Sales Rep** and its Source From will be **E-mail**:



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



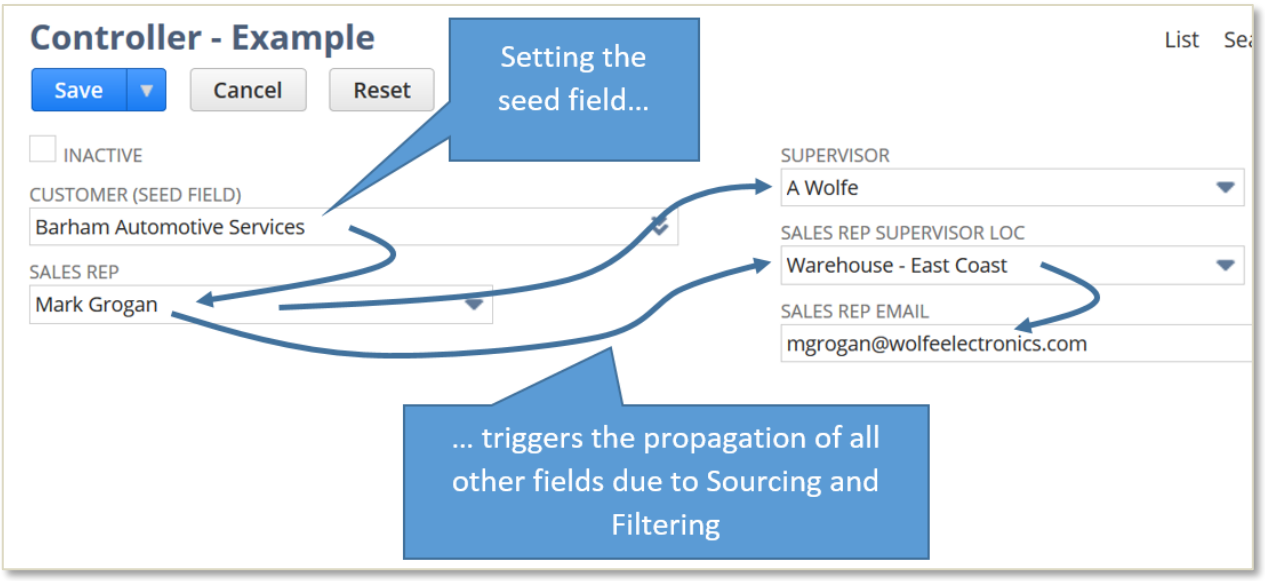
The last field is the **Sales Rep Supervisor Loc** field. Since it mimics *customer.salesrep.supervisor.location*, its Source List will be **Supervisor** and its Source From will be **Location**:



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 a SuiteScript script.

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 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.



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.

When the Controller record was tested above, 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. But remember, the Controller record was not saved.

The Controller record can be used in the exact same way by a SuiteScript script 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 that uses a Controller record in memory to obtain data for all three use cases listed above:

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

// Instantiate a Controller record
const controller = record.create({
    type: 'customrecord_controller_example',
    isDynamic: true
});

// Loop through each linked record chain
for (let i = 0; i < customerArray.length; i++) {
    // Set the seed field
    controller.setValue({
        fieldId: 'custrecord_customerseed',
        value: customerArray[i]
    });

    // Obtain the propagated values from all other fields
    customerSalesRep[i] = controller.getValue({
        fieldId: 'custrecord_salesrep'
    });
    customerSalesRepEmail[i] = controller.getValue({
        fieldId: 'custrecord_salesrepemail'
    });
    customerSalesRepSupervisor[i] = controller.getValue({
        fieldId: 'custrecord_supervisor'
    });
    customerSalesRepSupervisorLoc[i] = controller.getValue({
        fieldId: '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 governance usage units and has almost no risk of exceeding the governance limits. In fact, only the `record.create(options)` method that created the Controller record in memory consumes usage units. The Controller record created in memory was not saved because `record.save(options)` 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 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 `record.create(options)` was only called once. When many linked record chains are traversed using the regular querying and lookup APIs, the unit usage will increase linearly and eventually cause the script to exceed the governance limits. On the other hand, traversing the same set of record chains using the Transient Record Controller pattern 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 script will also need 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 `record.create(options` 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 `record.getValue(options)` 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 `record.getValue(options)` calls will offset the performance overhead of `record.create(options)`– the Transient Record Controller pattern also benefits from the economy of scale in relations to performance.

Using the Transient Record Controller pattern has the following pros and cons:

Pros

- It is highly economical use cases that require a lot of linked records to be traversed
- It can support very long record chains and/or high number of record chains with relative ease
- It 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 need 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

As a SuiteApp developer, you are encouraged to use the SuiteCloud Integrated Development Environment (IDE) to write SuiteScript scripts. 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 Plug-in* 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

You can find more information in the NetSuite Help Center or in SuiteAnswers, using the following search terms:

- *API Governance*
- *Script Type Usage Unit Limits*
- *runtime.getCurrentScript().getRemainingUsage()*
- *Client Script Metering*
- *Workflow Governance*

3. Principle 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 optimizations:

- 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
- Use a Cache

3.1.1 Do Not Re-Save Records in the afterSubmit Event

When creating or editing a record, do not use record.save(options) or record.submitFields(options) 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 scripts where applicable. Both approaches are more economical and will read/write the data only once.

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

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

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

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

```
const _beforeSubmit = function (context) {
    const 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 `record.load(options)` or `search.lookupFields(options)` to load the record in memory to retrieve data. Using those methods is an inefficient way to retrieve data from search results because it requires unnecessary I/O API calls. Those methods also use more SuiteScript governance usage units which can have negative impacts.

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

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:

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

myCustomerSearch.run().each(function(result) {
    const recId = result.id;
    const customer = record.load({
        type: 'customer',
        id: recId
    });
```

```
    const customerId = customer.getValue({
        fieldId: 'entityid'
    });
    return true;
};
```

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

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

myCustomerSearch.run().each(function(result) {
    const entityId = result.getValue('entityid');
    return true;
};
```

3.1.3 Do Not Put Heavy Lifting I/O Tasks into User Event Scripts or Suitelet POST 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.

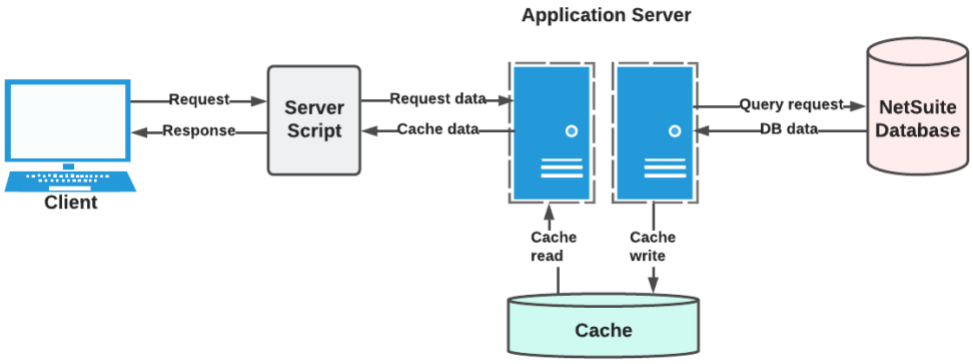
Additionally, because advanced searches support referencing saved searches (presumably shipped in the SuiteApp), you may reference a saved search, rather than build a search from scratch. Additional filters and columns may also be added to an advanced search that references an existing saved search.

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

3.1.6 Use a Cache

Cache is a special storage space for temporary data that can allow a server script to run faster and more efficiently. A cache significantly reduces the access time for frequently requested data and a shorter response time results in improved script performance.

As shown in the following diagram, the server script does not need to query the NetSuite database whenever data is requested that has already been placed in the cache. The data present in the cache results in faster processing because there is less data and information to be exchanged.



The N/cache module allows you to store a key/value pair in a segment of memory for a minimum of 300 seconds (5 minutes); there is no maximum. However, it is intended for short term usage as there is no guarantee that a cached value will remain for the full duration of the time to live specified.

The maximum size of a value that can be placed is limited to 500 KB. It can be removed by a server script using the key used to place it in the cache. The cache scope availability can be limited using one of the following options:

- **PRIVATE – Accessible to** the current server script only

- **PROTECTED** – Accessible to all server side scripts in the current SuiteApp
- **PUBLIC** – Accessible to all server side scripts in the NetSuite account

A cache can be used for a variety of purposes, functions, and applications. SuiteApps can implement a protected cache as a method for exchanging data between its server scripts that require interaction with each other during their respective scripted executions and benefit from the performance gain to improve overall user experience.

The following examples show using a cache in a Map/Reduce script and in a Suitelet. In the Map/Reduce script example, the performance gain of using a cache is demonstrated with sample calculations. The Suitelet example allows users to bulk approve journal entries using the N/cache module to prevent users that access the interface at the same time from seeing the same list of transactions thus preventing duplicate record processing.

3.1.6.1 Map/Reduce Using N/Cache Example

A map/reduce script is expected to fulfill approximately 10,000 sales orders per execution during a given day. There is a limit on the number of sales orders that can be fulfilled each day for a given customer based on their credit score rating, which is maintained outside of NetSuite and re-calculated once at midnight. A request to an external system end point must be made to obtain the maximum number of orders that can be fulfilled for a given customer. When the maximum number of orders for a customer per day has been reached, a request needs to be made to a different external system end point for further processing outside of NetSuite.

Sales orders are fulfilled during the reduce stage where the total units of API usage available is 5,000. The total units consumed during each reduce stage invocation required to fulfill an order is 30 units and the maximum number of orders that can be fulfilled during one script execution is 166.

where

Reduce Stage Available Units = 5,000

Total API Units Required = `record.transform` + `record.save` = 20 + 10 = 30

Total Orders = Reduce Stage Units / Total Units Required = 5,000 / 30 = 166

Without N/cache

The reduce stage would require additional units to be factored in to obtain the maximum number of orders from the external system using the `https.request` method. Moreover, logic to store and retrieve the number of orders fulfilled for every customer during each given day would need to be introduced. For example, a custom field in the customer record can be used to serve as a temporary placeholder along with `search.lookupFields` and `record.submitFields` to retrieve and update the value respectively.

Since a request needs to be made to the second end point using the `https.request` method to flag the maximum number of orders fulfilled for a customer when the threshold has been reached for the day, the units required for it must be accounted for as well in the calculations. Thus, the approximate number of units required per order in the reduce stage becomes 56, which brings the total number of orders that can be fulfilled down to 89.

Total API Units Required = `https.request` (10 units) * 2
+ `record.transform` (20 units)
+ `record.save` (10 units)
+ `search.lookupFields` (1 unit)
+ `record.submitFields` (5 units)
= 20 + 20 + 10 + 1 + 5 = ~56

Total Orders = Reduce Stage Units / Total Units Required = 5,000 / 56 = 89

With N/cache

A private cache is created during the input stage. During the reduce stage, the script checks if a cache object for the customer in reference exists using the `cache.get` method. If the cache object returned does not have the `maxOrders` property assigned, a request to the external system is sent using the `https.request` method to obtain the maximum number of orders and then stored in memory using `cache.put`. Whenever an order is fulfilled, a cache object property that is added during the first instantiation (`orderCounter`) is incremented by one.

When the fulfillment of an order causes the `orderCounter` to be equal to the `maxOrders` value found in the cache object for that customer, a request is made to the second end point using `https.request` method to flag the maximum number of fulfillments for that customer as completed for the day.

Total API Units Required = `https.request` (10 units) * 2
+ `record.transform` (20 units)
+ `record.save` (10 units)
+ `cache.get` (1 unit)
+ `cache.put` (10 unit)
= 20 + 20 + 10 + 1 + 1 = 52

Total Orders = Reduce Stage Units / Total Units Required = 5,000 / ~52 = 96

At first glance, there isn't a significant difference between the two implementations. Using N/cache simply provided room to fulfill ~7 additional sales orders. Nonetheless, the added benefit does not come just from the additional number of records that can be processed, but from the performance gain that is obtained from it which can only be understood by adding some variables into the calculations.

For example, a total of 10 sales orders for four different customers are found during the input data stage. The sales orders and maximum number of orders breakdown per customer is:

Customer	Input Data – SO Count	Fulfillment Threshold	Fulfillments Created
A	3	2	2
B	3	10	3
C	2	1	1
D	2	5	2
Total	10	-	8

The governance usage units required to process each order during the reduce stage would then be computed as follows:

Usage Units Without Using N/cache

Customer	SO#1	SO#2	SO#3	Units Consumed
A	46	56	16	118
B	46	46	56	148
C	56	16	-	72
D	46	56	-	102
Reduce Usage				440

Usage Units Using N/cache

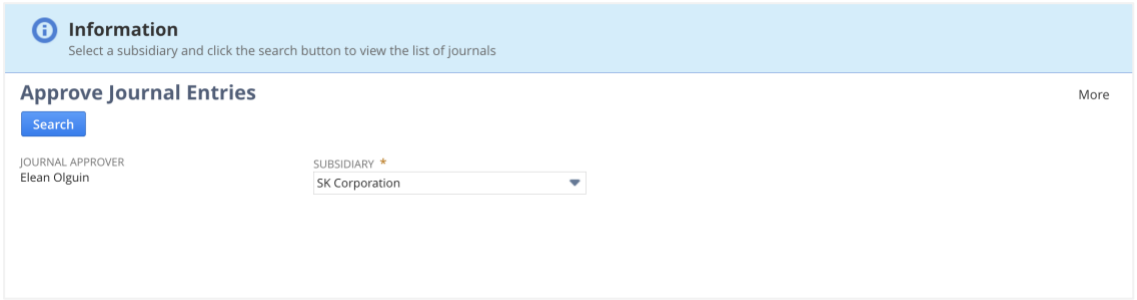
Customer	SO#1	SO#2	SO#3	Units Consumed
A	42	42	2	86
B	42	32	42	116
C	42	2	-	44
D	42	42	-	84
Reduce Usage				**330**

The values obtained from the calculations demonstrate that using N/cache lowers the overall consumption of governance units, which decreases the total execution time required to process each order as a result. This provides a performance gain in the script that is obtained from a faster execution of the logic during runtime.

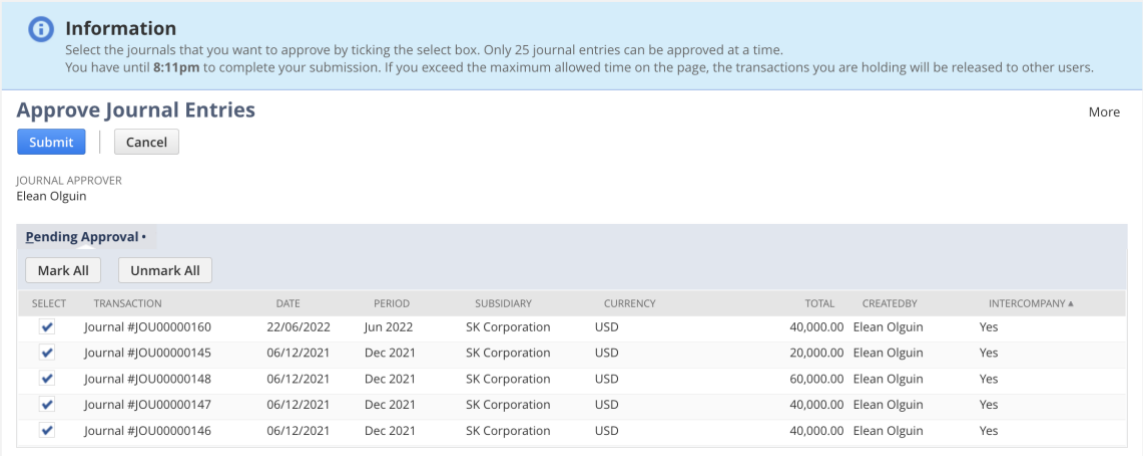
3.1.6.1 Suitelet Using N/Cache Example

Refer to Appendix 4 for the complete Suitelet sample code.

Users select a subsidiary to see the list of journal entries that are pending approval. When the form is submitted, a private cache is created. The cache key corresponds to the subsidiary internalid and the values are the internalds of the journal entries that are pending approval for that subsidiary.



Once the form is submitted, users are given 5 minutes to complete their submission, which corresponds to the time to live set on the cache. After that time, the cache is automatically cleared.



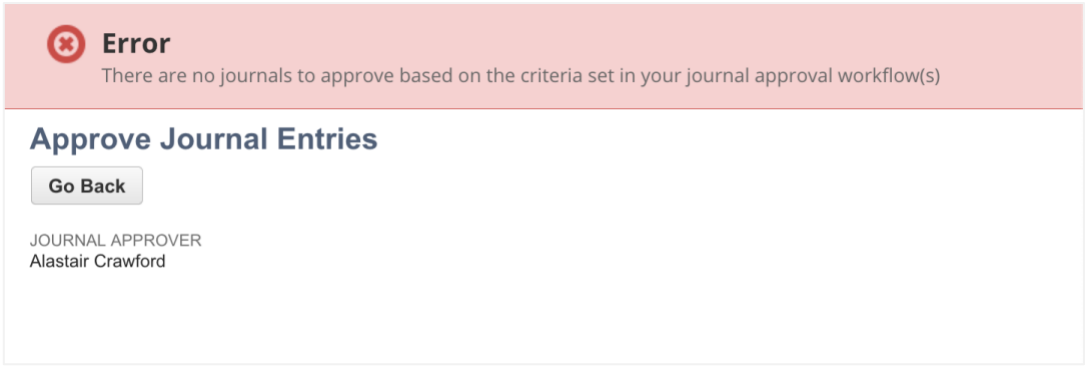
The code snippet below demonstrates how to perform this logic using a Suitelet.

N/cache – Suitelet Code Sample #1

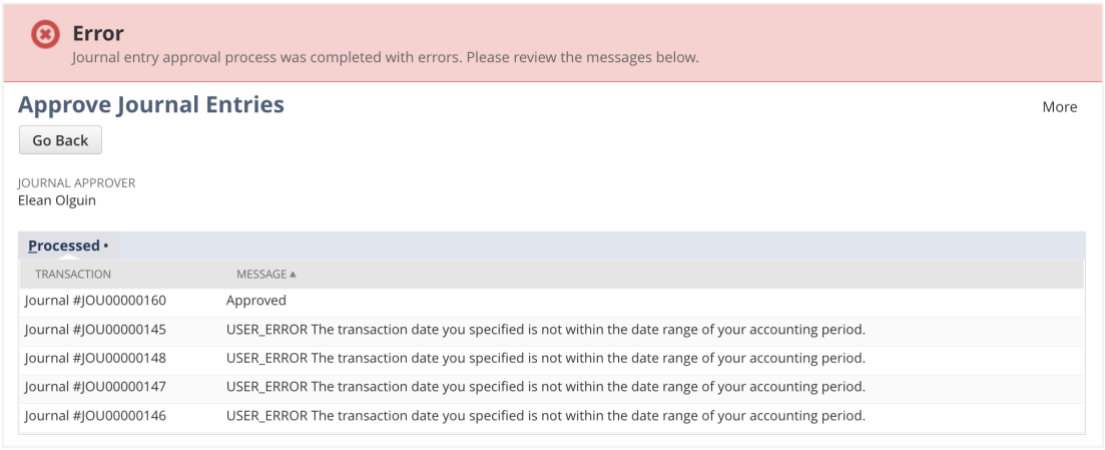
```
//Get the end time to be displayed for the user, which corresponds to the maximum
//allowed time to process the submission

const endDateTime = getEndDateTime();
const subsidiaryId = params.custpage_subsidiary;

//Get the cache, if it does not exist it will be automatically created
const appCache = cache.getCache({
    name: 'JournalApprovalCache',
    scope: cache.Scope.PRIVATE,
});

//Get journals for the subsidiary selected that are currently in the cache
const cacheValues = getCacheValues(appCache, subsidiaryId);

//Get pending approval journals excluding the ones in the cache
const journals = getPendingApprovalJournals(subsidiaryId, cacheValues);

//Add current user journal entries into cache
updateCacheValues(appCache, subsidiaryId,
cacheValues.concat(journals.map((a) => a.id)));

/** HELPER FUNCTIONS **/

//Retrieve values stored in the cache
const getCacheValues = (appCache, cacheKey) => {
   const cacheValues = appCache.get({
         key: cacheKey
    });
   return cacheValues ? cacheValues.split(',') : [];
 };
```

```
//Add new values into the cache
const updateCacheValues = (appCache, cacheKey, cacheValues) => {
    const uniqueValues = cacheValues.map((d) =>
    parseFloat(d)).reduce((u, b) => u.includes(b) ? u : [...u, b], []).join(',');
    if (uniqueValues.length > 0) {
          appCache.put({
             key: cacheKey,
             value: uniqueValues,
              ttl: 300
           });
     } else {
       //Remove if there are no values stored in the cache
         purgeCache(appCache.name, cacheKey);
      }
};



//Remove cache
const purgeCache = (cacheName, cacheKey) => {
  cache.getCache({
      name: cacheName,
      scope: cache.Scope.PRIVATE
   }).remove({
      key: cacheKey
    });
};

//Additional logic here…
```

If a different user attempts to view the list of journal entries that are pending approval for that same subsidiary during the time that the previous user is processing their submission, they will receive an error if there are less than 25 journal entries that are pending approval for the subsidiary in reference.



If there are more than 25 journal entries, they will not be able to see in the results the list of journal entries that are being displayed for the previous user unless the journal entries failed to be approved by the previous user.



Failed transactions are removed from the cache to allow these to become available to other users who might be able to process their approval. Processed transactions are also removed to keep the cache clean and up to date; however, these will no longer be captured by the query that fetches the journal entries because they will no longer be in a pending approval status. Refer to the code sample below for the SuiteScript cache update logic.

N/cache – Suitelet Code Sample #2

```
//Journal approval logic here..

//Get cache, if it does not exist it will be automatically created
const appCache = cache.getCache({
   name: 'JournalApprovalCache',
   scope: cache.Scope.PRIVATE,
});

//Get journals for the subsidiary selected by the user that are currently in the cache
const cacheValues = getCacheValues(appCache, subsidiaryId);

//Remove journal entries that failed to be processed from the cache
if (notProcessed.length > 0) {
   updateCacheValues(appCache, subsidiaryId, cacheValues.filter((a) =>
!notProcessed.includes(a)));
 }
//Additional logic here ...
```

3.1.6.1 Considerations when using the N/cache Module

The following are considerations when using the N/cache module:

- The N/cache module is only available for server-side script types.

- Cache is meant to be used short term. SuiteApps should not heavily rely on cache for long term storage as there is no guarantee that a cached value will remain for the full duration of the time to live specified.
- Cache data is not persistent and there is a limit of 500KB per value. Consider using a loader function to make use of a more efficient design to set and retrieve data.
- If the value stored in a cache is not a string, it will be automatically converted to one with JSON.stringify(). When reading the value, if the latter is not a string, JSON.parse() must be used to convert the value back to its original format.
- SuiteApps should set their cache scope to Private or Protected to avoid other SuiteScript customizations in the NetSuite account to be able to interact with it.

For more information, refer to N/cache SuiteScript API documentation.

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:

- Use Asynchronous Web Services for High Volume I/O
- Identification of Web Services Applications in NetSuite

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

3.2.1 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, it is free to immediately sign off, and does not need to wait for a response from NetSuite. On the NetSuite server, the immediate load on the server is reduced, as there is no real-time demand to perform web services I/O. Overall, using 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. You will need to decide whether to use synchronous or asynchronous operations for your SuiteApp 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 use of large amounts of data puts a demand 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:

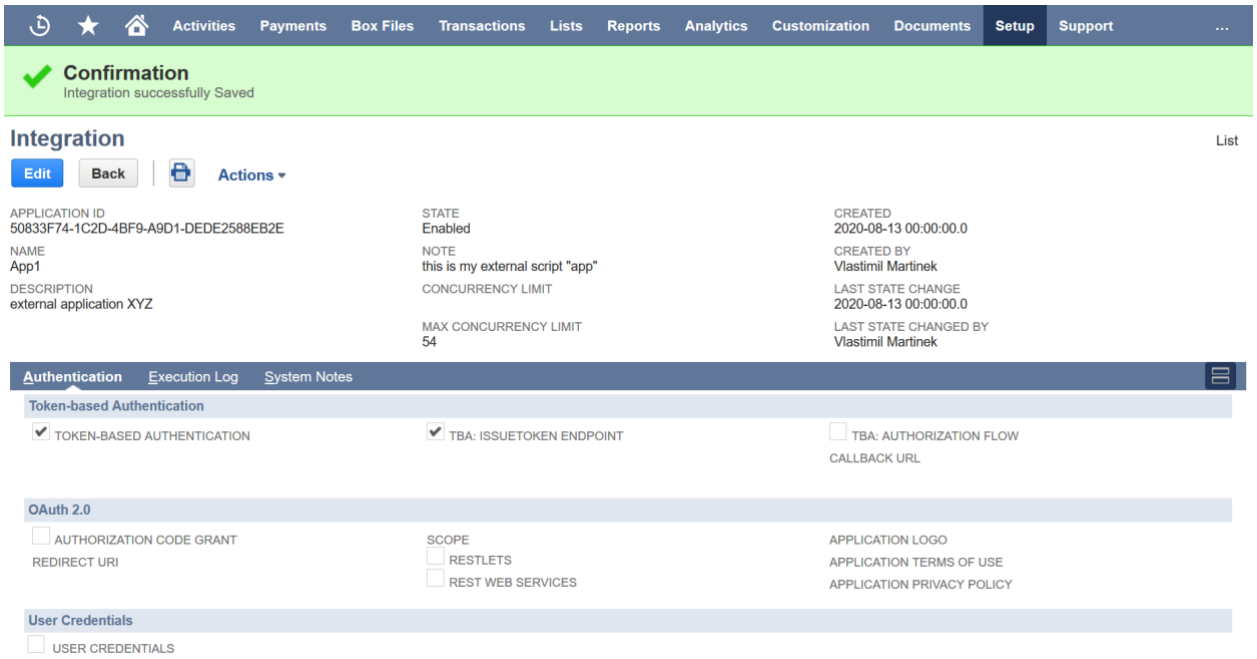
- **Learning and experimenting** – When learning the basic CRUD operations supported by SuiteTalk and the data schema of NetSuite records in SOAP form, it is best to use synchronous web services. The demand on NetSuite servers is generally low, and the real-time response promotes faster learning.
- **Specific 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 unwise 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.

3.2.2 Identification of Web Services Applications in NetSuite

NetSuite Release 2015.2 introduced a new Integration record (Set Up > Integration > Manage Integrations). An Integration record serves as a unique representation of an external application within NetSuite, that is used to identify the SuiteApp and enable control, authentication methods, and monitoring of the connections of specific SuiteApps on the NetSuite server.



Note: An Integration record must be unique for each external application.The record could be either distributed to customers accounts within SuiteApp bundle or SDF SuiteApps.

3.2.3 Distributing Integration Records using SDF

As of the 2020.2 release, SDF supports the distribution of Integration records, and they can be installed to customer accounts through the SuiteApp Control Center. An Integration record must be created from your NetSuite development account and imported to your SuiteApp project in the IDE (they cannot be created directly in the IDE). SDN has prepared a guide on how to manage your integration records starting from the development phase up to the distribution phase. This guide is available in the SDN APC Portal (Technical > Distributing Integration Records) as well as on the Technical Onboarding Site (https://sites.oracle.com/site/SDN_Site/).

Important: It is mandatory that you distribute your Integration record (along with role and possibly other objects) with your Integrated or Hybrid SuiteApp. This will help to control and monitor connections with your solution and will also eliminate unnecessary manual setup on the part of the customer. Asking customers to create new Integration records on their production accounts to enable Token-Based Authentication (TBA) for your SuiteApp is not a practice that is compliant with the BFN validation program.

3.2.4 OAuth 2.0 Integration Record Creation

Even though the OAuth 2.0 Authorization Code Grant Flow allows for the creation of the Integration Record directly into the customer's account without the need for distributing it via a SuiteApp, BFN requires partners to package it in the corresponding SuiteApp for distribution. It's important to note that the access provided by the OAuth 2.0 Authorization Code Grant has an expiration of seven days and needs to be manually renewed. For a machine-to-machine OAuth 2.0 process, please check the OAuth 2.0 Client Credentials Flow at:

https://system.netsuite.com/app/help/helpcenter.nl?fid=section_162730264820.html

Important: Older integrations using OAuth2.0 Client Credentials flow may need to be updated to use the RSA-PSS scheme. Integrations using the old RSA PKCSv1.5 scheme will stop working October 1, 2024. Alternatively, you can use EC key when generating your new certificate, which is considered to be even more secure. The length of the EC key must be 256 bits, 384 bits, or 521 bits. Please check the link above to see the valid examples on how to create a valid certificate with OpenSSL.

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 The Benefits of Saved Searches 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 The Benefits of Saved Searches

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 `search.create(options)` and `Search.save(options)`. 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. You can also 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 a SuiteScript script or in external applications. One way of doing this is by referencing an existing saved search from your script. This approach reduces 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, where simply referencing an existing saved search is not sufficient. You may generally know 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 expect 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 it 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 SuiteScript scripts 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 a range of values are faster. The more precise and confined the criteria, the more efficient the search becomes.

For example, you may be required 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. When searching for strings, avoid using the `contains` operator as it is inefficient from a performance perspective. Searches should be thought of as SQL queries; consequently, the use of the `contains` operator and formulas can be problematic for indexes. See SearchOptimizationFlowchartNotes for more information.

The following examples demonstrate how to use the `between` operator to search for active inventory items with a purchase price between 100 and 500 dollars sorted from highest to lowest price.

SuiteScript (using N/search)

```
const myInventorySearch = search.create({
    type: search.Type.INVENTORY_ITEM,
    columns: [‘itemid’, search.createColumn({
        name: ‘cost’,
        sort: search.Sort.DESC
    })],
    filters: [[“isinactive”, “is”, “F”], “AND”, [‘cost’, ‘between’, 100, 500]]
});
myInventorySearch.run().each((result) => {
    log.debug({
        title: ‘Item Data’,
        details: result.getValue(‘itemid’)
            + ‘\n’ + result.getValue(‘cost’)
    });
    return true;
});
```

SuiteScript (using N/query – SuiteAnalytics Workbook)

```
const myInventoryQuery = query.create({
    type: query.Type.ITEM
});
myInventoryQuery.columns = ['itemid', 'cost'].map((name) => {
    return myInventoryQuery.createColumn({
        fieldId: name,
        alias: name
```

```
    });
});
myInventoryQuery.sort = [myInventoryQuery.createSort({
    column: myInventoryQuery.columns[1],
    ascending: false
})];
myInventoryQuery.condition = myInventoryQuery.and([{
    fieldId: 'itemtype',
    operator: query.Operator.ANY_OF,
    values: ['InvtPart']
}, {
    fieldId: 'isinactive',
    operator: query.Operator.IS,
    values: [false]
}, {
    fieldId: 'cost',
    operator: query.Operator.BETWEEN,
    values: [100, 500]
}].map((options) => {
    return myInventoryQuery.createCondition(options);
}));
myInventoryQuery.run().asMappedResults().forEach((result) => {
    log.debug({
        title: 'Item Data',
        details: result.itemid + '\n' + result.cost
    });
});
```

SuiteScript (using N/query - SuiteQL)

```
const myInventorySuiteQLQuery = query.runSuiteQL({
    query: `
        SELECT
           ItemId AS itemid,
           Cost AS cost
        FROM
           Item
        WHERE
```

```
            ( ItemType = 'InvtPart' )
            AND ( Cost BETWEEN 100 AND 500 )
            AND ( NVL(isinactive, 'F') = 'F' )
        ORDER BY
          cost DESC
          `
});
myInventorySuiteQLQuery.asMappedResults().forEach((result) => {
    log.debug({
        title: 'Item Data',
        details: result.itemid + '\n' + result.cost
    });
});
```

Performance Benefits of Indexed Saved Searches

When a Saved Search is created or updated through the NetSuite UI rather than being generated programmatically using SuiteScript, it is indexed and will perform better. If a Saved Search has been created programmatically, updating it though the UI will cause it to be indexed. An indexed Saved Search will retain performance benefits when accessed via SuiteScript, even if additional columns and criteria are added to the Saved Search programmatically.

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 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 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, as shown in the following script:

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

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

const 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, for example, 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 Joined Searches Limits and Advanced Query Joins

When searching in a script or from the NetSuite UI, you can minimize the size of the result by providing more search criteria. Logically, fewer search results equate to faster response times. However, you should also be mindful of the filters and columns added to searches. Retrieving additional information by joining native or custom records to incorporate 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 to identify 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 provide nice-to-have information, but are not necessarily requirements, then consider not adding them.

When working with multiple record data or large datasets, consider using N/query instead which runs on the new and faster SuiteAnalytics Workbook engine. A non-paged query returns up to 5,000 results and has a more powerful API for performing advanced joins. Multilevel joins can be used to retrieve and aggregate data from multiple record types in a single query call whereas saved searches can only join within one level per search call.

SuiteQL supports the following SQL join types:

- **Cross Joins** — A cross join is used to produce all row combinations between two tables. The result is known as the Cartesian product of the rows in both tables.
- **Inner Joins** — An inner join is used to produce row combinations between two tables based on a common value. The results include only those rows that share the common value.
- **Outer Joins** — An outer join is also used to produce row combinations between two tables based on a common value. The results include all rows from one or both tables, depending on the type of outer join.

You can use these join types to customize the results you receive from your SuiteQL queries. By default, when you join record types in SuiteAnalytics Workbook, the join performed is a left outer join. The left outer join type is appropriate for many use cases, but in some situations, you may want to use a different join type to obtain a more customized result set. Use the Records Catalog to see the fields, joins available for each record type cardinality, etc.

When you work with query results, make sure that you do not depend on name-casing. You can also use built-in functions to perform certain operations in SuiteQL queries. These functions extend the capabilities that are provided by the SQL-92 specification. For a list of supported built-in functions and SuiteQL name-casing limitations, see *SuiteQL Supported Built-in Functions* in the NetSuite Help Center.

The following sample code shows how to retrieve the total quantity received and billed (from approved bills) for a given purchase order grouped by item using columns aggregation and different joins.

N/query - SuiteQL

```
SELECT
   MAX(PO.TranID) AS PONumber,
   Item.ItemId AS Item,
   MAX(NVL(POLine.Quantity,0)) AS TotalOrdered,
   SUM(CASE WHEN PTLL.NextType = 'ItemRcpt'
            THEN NVL(TransactionLine.Quantity,0)
            ELSE 0 END) AS TotalReceived,
   SUM(CASE WHEN PTLL.NextType = 'VendBill'
            THEN NVL(TransactionLine.Quantity,0)
            ELSE 0 END) AS TotalBilled
FROM
  Transaction
   INNER JOIN TransactionLine ON
    ( TransactionLine.Transaction = Transaction.ID )
   INNER JOIN Transaction AS PO ON
    ( PO.ID = TransactionLine.CreatedFrom )
   INNER JOIN Item ON
    ( Item.ID = TransactionLine.Item )
  LEFT OUTER JOIN PreviousTransactionLineLink AS PTLL ON
     ( PTLL.NextType = 'ItemRcpt' OR PTLL.NextType = 'VendBill'  )
     AND ( PTLL.NextDoc = TransactionLine.Transaction )
     AND ( PTLL.NextLine = TransactionLine.ID )
  LEFT OUTER JOIN TransactionLine AS POLine ON
    ( POLine.Transaction = PO.ID )
     AND ( POLine.id = PTLL.PreviousLine )
WHERE
```

```
   (PO.ID = ?)
    AND ( PO.Voided = 'F' )
    AND ( Transaction.Type = 'ItemRcpt' OR Transaction.Type = 'VendBill'  )
    AND ( TransactionLine.Quantity <> 0 )
    AND (( CASE WHEN Transaction.Type = 'ItemRcpt' THEN 1
                WHEN BUILTIN.DF(Transaction.ApprovalStatus) = 'Approved' THEN 1
                END ) = 1 )
 GROUP BY
   Item.ItemId
```

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 = ‘11/20/2021’” 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.

When expecting less than 1,000 results, use runPaged because it is faster and it consumes less governance units per call. The following sample code retrieves all GL posting Vendor Bills and Invoices in the system that are found in closed accounting periods. A comparative analysis using each available method is summarized in the table below to demonstrate the differences in execution time.

N/search

```
const transactionSearch = search.create({
      type: search.Type.TRANSACTION,
      filters: [["mainline", "is", "T"], "AND", ["posting", "is", "T"],
               "AND", ["accountingperiod.closed", "is", "T"],
               "AND", ["type", "anyof", "CustInv", "VendBill"]],
      columns: ['internalid', 'postingperiod', 'trandate', 'type',
               'tranid', 'subsidiary', 'currency', 'fxamount']
});
```

N/query – SuiteAnalytics Workbook

```
const transactionQuery = query.create({
    type: query.Type.TRANSACTION
});

transactionQuery.columns = ['id', 'postingperiod', 'trandate', 'type', 'tranid',
'currency', 'transactionlines.subsidiary', 'foreigntotal'].map((name) => {
     return transactionQuery.createColumn({ fieldId: name });
});

transactionQuery.condition = transactionQuery.and([{
     fieldId: 'postingperiod.closed',
     operator: query.Operator.IS,
     values: true
}, {
     fieldId: 'posting’,
     operator: query.Operator.IS,
     values: true
}{
    fieldId: 'recordtype',
    operator: query.Operator.ANY_OF,
    values: ['vendorbill', 'invoice']
}, {
    fieldId: 'transactionlines.mainline',
    operator: query.Operator.IS,
   values: true
}].map((options) => {
   return transactionQuery.createCondition(options);
}));
```

N/query - SuiteQL

```
SELECT DISTINCT
   Transaction.ID, PostingPeriod, Trandate, Type,
   TransactionLine.Subsidiary, Tranid, Currency, ForeignTotal
```

```
FROM
   Transaction
INNER JOIN TransactionLine ON
   ( TransactionLine.Transaction = Transaction.ID )
INNER JOIN AccountingPeriod ON
   ( Transaction.PostingPeriod = AccountingPeriod.ID )
WHERE
   NVL(Transaction.Posting, 'F') = 'T'
   AND NVL(AccountingPeriod.Closed, 'F') = 'T'
   AND ( Transaction.Type = 'CustInvc' OR Transaction.Type = 'VendBill' )
```

Total Results: 801						
Module Method | Run	**1**	**2**	**3**	**4**	**5**	**Avg**
search.run	318	271	315	310	306	304
query.run	384	349	387	384	371	375
query.runSuiteQL	228	187	179	188	180	192
search.runPaged	64	62	63	91	60	68
query.runPaged	267	283	293	286	290	284
query.runSuiteQLPaged	153	178	184	180	179	175
Time (in ms)						

As depicted above, on average and during each test run, runPaged executed significantly faster; however, it is important to remember that execution time is largely influenced by the record type, number of joins, criteria/conditions, columns, among others. It is recommended that you carefully assess each available search/query method during the designing phases to identify the most efficient method for your application.

For searches and queries to be efficient, it is highly recommended that the search criteria/conditions and result columns are kept to a minimum. 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 result 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 limitation where in some cases, the search times 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 nearly impossible to isolate the problem and would be very hard to quantify the volume of records returned by these types of searches. In this case, you can try to determine whether it is indeed a timeout or if records are actually returned. To do this, execute the search with the same criteria but with very minimal number of result columns – or better yet, just the internal ID. This will

lighten 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.

3.3.7 SuiteQL Best Practices

The performance of a SuiteQL query depends on various factors, including the complexity of the query, the choice of clauses, join types, functions, data size, and the use of indexes and aggregates. It is essential to consider these factors, among others, and apply optimization techniques and follow recommended practices when writing a SuiteQL query to reduce the time it takes to retrieve data for optimal results.

*3.3.7.1 Using SELECT **

Using SELECT * in a query can significantly impact performance because it retrieves all columns from the record type being queried even if all columns are not needed. This means more data is retrieved from the database which increases the amount of data transferred and processed. With more data retrieved, there is an increase in CPU and memory usage because the database engine must process and store the data which directly impacts result retrieval time and overall query performance.

It is highly recommended to specify the specific columns you need in your query instead of using * (which retrieves all columns). Specifying the columns can reduce the amount of data retrieved, processed, and stored, improving your query's efficiency and enhancing performance.

3.3.7.2 Resource Intensive Functions

The use of regular expressions in a query can be detrimental to performance because they manipulate text data, which can be slow and resource intensive. Functions such as REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR can be particularly slow, especially for large strings, because they require a full scan of the input string to find matches. Similarly, the use of DECODE and INSTR functions can also be slow because they need to perform a full scan to find the position of the match, which can be even slower than using regular expressions for large strings. It is generally best to avoid using those functions with large strings, if possible.

3.3.7.3 BUILT.IN Functions

Built-in functions allow you to perform certain operations in SuiteQL queries. In some cases, these can be used to simplify calculations or to skip the need to add a join to a table to retrieve a value. For example, the following query retrieves a list of vendor bills along with the names of the associated vendors:

```
SELECT
  Transaction.ID AS id,
  Transaction.Entity AS vendor,
  Vendor.EntityId AS vendorName
FROM
  Transaction
 INNER JOIN Vendor ON
```

```
  ( Transaction.Entity = Vendor.Id )
WHERE
  ( Transaction.Type = 'VendorBill' )
```

The query can be simplified with the “BUILTIN.DF” function which returns the display value of a field from the target record type, eliminating the need for an explicit join.

```
SELECT
  Id As id,
  Entity AS vendor,
  BUILTIN.DF(Entity) AS vendorName
FROM
  Transaction
WHERE
  ( Transaction.Type = 'VendBill' )
```

3.3.7.4 SQL Joins and Join Types

Before writing a query, you need to think carefully about which type of join is most appropriate for your use case and define the correct join condition to ensure your query will return the expected results set in the most optimal manner. SuiteQL supports explicit and implicit joins and the following SQL join types: Inner Join, Outer Join, and Cross Join.

To ensure your SuiteQL queries are clear, readable, and return the expected output efficiently and effectively, you can follow these best practices for using joins in your queries.

Explicit vs Implicit Joins

Use explicit joins instead of implicit joins because they are easier to read, clarify how record types are joined, and allow you to specify the join type. For example, the following query retrieves the list of accounts that are restricted by department:

```
SELECT
  Account.Id AS account,Department.Id AS department
FROM
  Account, Department
WHERE
  ( Account.Department = Department.Id )
```

With an explicit query, you can use an outer join to filter the list of accounts by department and location, regardless of whether there are matching rows.

```
SELECT
  Account.Id AS account,
  Department.Id AS department, Location.Id AS location
FROM
  Account
JOIN Department ON
  ( Account.Department = Department.Id )
LEFT OUTER JOIN Location ON
  ( Account.Location = Location.Id )
```

Inner Join

An inner join is often the most performant join type because it only returns rows that match the specified condition, making it efficient for returning a small number of rows. However, to determine which join is more performant, factors such as the nature of the data, record types, and query complexity must be considered.

An inner join is best used in the following cases:

1. You want to return only the matching rows between two record types. For example, to return only the rows from the entity record type where the entity ID matches the entity ID in the transaction record type, you can use an inner join and the ORDER BY clause to return the rows in a specific transaction.

   ```
   SELECT DISTINCT
     Transaction.ID AS id,
     Transaction.TranDate AS trandate,
     Transaction.Type AS type
   FROM
     Transaction
   INNER JOIN Entity ON
     ( Entity.Id = Transaction.Entity )
   WHERE
     ( Transaction.Entity = ? )
   ORDER BY
     Transaction.TranDate DESC
   ```

2. You want to return only the matching rows between two record types and include the unmatched rows from one of the record types. For example, to return only the matching rows between the entity and the transaction record types and include the unmatched rows from the entity record type, you can use an inner join with the UNION clause.

```
SELECT
  Id, Type, Entity, ForeignTotal
FROM
  Transaction
WHERE
  NVL (ForeignTotal,0) < 1000
  AND ( Type = 'CashSale' )
UNION
  SELECT
    Transaction.Id, Transaction.Type,
    Transaction.Entity, Transaction.ForeignTotal
  FROM
    Transaction
  INNER JOIN Entity ON
    ( Entity.Id = Transaction.Entity )
  WHERE
    ( Entity.IsInactive = 'T' ) AND
    ( NVL (Transaction.ForeignTotal,0) > 1000 )
```

3. You want to return the matching rows between two record types and include the unmatched rows from both record types. For example, if you want to return the matching rows between the entity and the transaction record types and include the unmatched rows from both, you can use an inner join with the UNION clause and the EXCEPT clause.

```
SELECT
  Id, Type, Entity
FROM
  Transaction
EXCEPT UNION
  SELECT
    Transaction.Id, Transaction.Type, Transaction.Entity
  FROM
    Transaction
  INNER JOIN Entity ON
    ( Entity.Id = Transaction.Entity )
  WHERE
  ( Entity.IsInactive = 'T' ) AND ( Transaction.Posting = 'F')
```

These are just a few examples of when it would be best to use an inner join. There are many other situations where it would be appropriate, such as when you want to return only the matching rows between two record types and exclude the unmatched rows from both record types.

Outer Join

An outer join is less performant than the inner join because it returns all rows from one table and only the matching rows from the other, making it less efficient when returning a small number of rows. This type of join is commonly used to return all rows from one table along with the matching rows from another, even if there are no matching rows.

An outer join would be best used in the following cases:

1. You want to return all the rows from one record type, even if they don't match any rows in the other record type.

2. You want to return all the rows from one record type, but only the matching rows from the other record type.

3. You want to return the unmatched rows from one record type even if they don't match any rows in the other record type.

4. You want to return the unmatched rows from one record type but only the matching rows from the other record type.

For example, to return the rows between the entity group and the entity group member record types and include only the matching entity group member rows, use the following query:

```
SELECT
  EntityGroup.Id,
  EntityGroupMember.Member
FROM
  EntityGroup
LEFT OUTER JOIN EntityGroupMember ON
  ( EntityGroup.ID = EntityGroupMember.Group )
WHERE
  ( EntityGroup.GroupTypeName = 'Employee' )
```

For example, to return the list of scripts and script deployments, you can use either a LEFT or RIGHT join to retrieve all matching or unmatching client scripts and deployments, use the following query:

```
SELECT
  Script.ScriptType AS scriptType,
  Script.scriptId AS scripId,
  ScriptDeployment.ScriptId AS deploymentId,
```

```
  ScriptDeployment.RecordType AS deploymentRecord
FROM
  Script
LEFT OUTER JOIN ScriptDeployment ON
  ( Script = Script.Id ) AND (Script.ScriptType = 'CLIENT' )
```

Cross Join

A cross join is the least performant because it returns all possible combinations of rows from both tables, regardless of whether they match. This makes it the least efficient join when only a small number of rows are required. It is used to combine results from two or more queries that have many-to-many, one-to-many, many-to-one, or one-to-one relationships. It is called a cross join because it combines the results of the queries in a "cross" manner, meaning the results of one query are combined with the results of another.

For example, the following query combines the results of one query that retrieves the departments with the results of another query that retrieves a list of employees and their departments.

```
SELECT
  Department.Id AS department,
  Department.Name AS departmentName,
  Employee.EntityId AS employee,
  Employee.Department AS employeeDepartement
FROM
  Department
CROSS JOIN
  Employee
```

Note: The cross join is not supported in SuiteAnalytics Connect; however, you can simulate a cross join by using an implicit inner join or a full outer join.

3.3.7.5 Using Aliases

Use aliases, including column aliases, when joining multiple record types or when working with record types that have long names to make the query output more understandable. For example, the following query returns the internal ID of both the file and folder. Assigning a column alias, as in the second query, makes it easier to distinguish between them when fetching the results.

```
SELECT
  File.Id,
  File.FileType,
  MediaItemFolder.Id
FROM
```

```
  File
 INNER JOIN MediaItemFolder ON
  ( File.Folder = MediaItemFolder.Id )
 WHERE
  ( MediaItemFolder.AppFolder = 'SuiteScripts')
```

Adding an alias to the folder table shortens the overall length and makes it easier to identify by assigning a familiar name.

```
SELECT
  File.Id AS file,
  File.FileType AS type,
  MediaItemFolder.Id As folder
FROM
  File
 INNER JOIN MediaItemFolder AS Folder ON
  ( File.Folder = Folder.Id )
 WHERE
  ( Folder.AppFolder = 'SuiteScripts')
```

Note: When running SuiteQL queries using the N/query module, be sure to incorporate aliases for your columns when using the .asMappedResults() method.

3.3.7.7 Small vs Large Queries

Small queries are typically faster to execute than large queries because they require less processing and memory resources making them more efficient at retrieving specific data from the database. This is particularly useful when working with large datasets as it can reduce the amount of data that needs to be retrieved and processed. Additionally, small queries are easier to read and understand which facilitates modifications and subsequent updates to the query.

When working with large datasets, it is important to consider limitations such as the maximum execution time allowed per script type and the limit on the number of results that can be fetched. If the SuiteAnalytics Connect feature is enabled, there is no limit to the number of results. Otherwise, a query can return a maximum of 100,000 results per execution.

Example

You want your SuiteApp to generate a report for users that lists all transactions impacting the general ledger and are posted to the current accounting period. To do this, you can use a Suitelet with the following query:

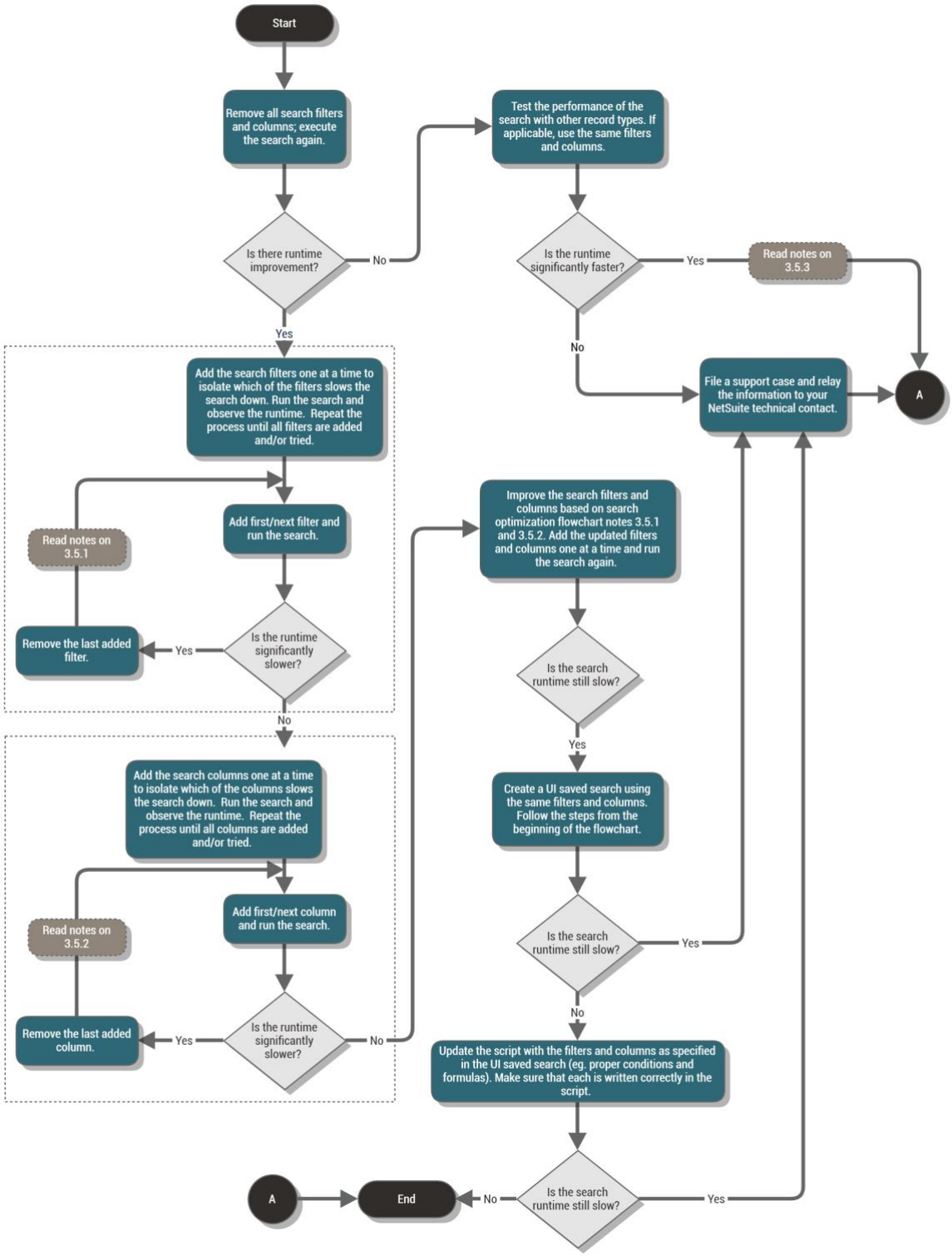
```
SELECT *
FROM
    Transaction
INNER JOIN AccountingPeriod ON
    ( AccountingPeriod.id = Transaction.PostingPeriod )
WHERE
    ( Transaction.Posting = 'T' ) AND
    ( AccountingPeriod.Closed = 'F' )
```

A Suitelet has a time limit of 300 seconds for a single execution. This could be problematic if the dataset is very large - it might result in an SSS_TIME_LIMIT_EXCEEDED error if the query does not return results within the time limit, thereby preventing the rest of the script logic from completing successfully. Moreover, using "SELECT *" will return all columns from the transaction table for each row, which can significantly impact performance in large datasets; the more data there is to process, the longer the query may take to execute. To mitigate this risk, consider the following:

- Avoid returning unnecessary information in your query. Specify only the relevant columns rather than using "SELECT *".
- Break down the query into smaller ones by adding filtering criteria. For example, you could return the list of transactions filtered by accounting period, a date range, subsidiary, or transaction type by prompting the user to select one or more of these to generate the report.
- If applicable, use RANK to sort data based on a specific column or a combination of columns. RANK can be a powerful tool for sorting and ranking data in a flexible and efficient manner.
- Offload the logic to a map/reduce or scheduled script and notify the user when the report generation is completed by the backend script.

Overall, there are many ways to enhance performance and mitigate potential risks when querying large datasets. It is important to experiment with different dataset sizes to determine what works best for your specific use case. Factoring in these considerations before writing the query will also allow you to determine the most appropriate script type or to break down your data fetching into subprocesses to find the most optimal and efficient design.

3.4 Search Optimization Flowchart



3.5 Search Optimization Flowchart Notes

3.5.1 Slow Searches Due to Filters

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 Slow Searches Due to Columns

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 Slow Search Result Loading

If loading search results takes 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 need higher data I/O throughput. For example, a customer in the warehouse distribution vertical with larger warehouse(s) and/or a larger number of customers might need higher data I/O throughput for the Integration 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 Integration 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 Integration 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 SuiteScript scripts. These two integration interfaces are equally strong in terms of NetSuite record exposure (standard and custom), security, as well as searchability. Integration 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”.

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 thereby enabling all to share a common pool of concurrent threads for integrations. The basic concept is the data I/O throughput of the Integration 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 Appendix 2 at the end of this guide to see how to use these limits and calculate the number of threads for your account.

3.6.1 Concurrency Support in SuiteTalk

Each Integration SuiteApp that uses SuiteTalk requires a NetSuite license. This license, otherwise known as an “integration user”, provides the Integration SuiteApp access to the NetSuite account. The Integration 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 is 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. To calculate the shared pool's limits, review the Concurrency Governance Cheat sheet in the Appendices at the end of this guide. 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 NetSuite Help Center found at:

https://nlcorp.app.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.

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, Integration 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 are 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 including the number of SuiteCloud Plus licenses purchased.

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

3.7 Further Reading

You can find more information in the NetSuite Help Center or in SuiteAnswers, using the following search terms:

3.7.1 SuiteScript Topics

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

3.7.2 Web Services Topics

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

3.7.3 General Search-related Topics

- *Defining an Advanced Search*

4. Principle 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 `runtime.executionContext` 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({
                title: 'Runtime Execution Context: ',
                details: runtime.executionContext
            });
            log.debug({
                title: 'User Event Context: ',
                details: 'Type: ' + context.type
            });
        }
        return {
            beforeLoad: caseBeforeLoad
        };
    }
}
```

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

Pros

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

Cons

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

Note: For more information on script execution contexts, see `runtime.executionContext` 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.

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

Pros

- 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.

Cons

- 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 scripts, 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, you should isolate the UI scripts from the web store scripts deployed on the Sales Order record. You 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 `runtime.executionContext`. 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 === 'edit'
|| context.type === 'view')) {
            // User interface handling here (e.g., adding tabs or fields)
            log.debug({
                title: 'Runtime Execution Context: ',
                details: runtime.executionContext
            });
            log.debug({
                title: 'User Event Context: ',
                details: 'Type: ' + context.type
            });
        }
    }
    function caseBeforeSubmit(context) {
        if (context.type === 'create') {
            if (runtime.executionContext === 'USERINTERFACE') {
```

```
                // Perform additional validation and calculations
                // based on UI entries
            }
            else if (runtime.executionContext === 'WEBSTORE') {
                // Calculate special item discounts order 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 example, 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. You 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">
         const 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, it has to be called using `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 Delegating I/O to Scheduled Scripts, it is advised that you move long-running processes and I/O intensive processes to scheduled scripts to provide better user experience and obtain higher I/O throughput. While it is a good practice to use scheduled scripts in this manner, you should be aware that most NetSuite accounts have only two processors Accounts with the SuiteCloud Plus license can have additional processors available, but you should nevertheless write scripts defensively to ensure they work in the majority of customer accounts that do not have additional processors.

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

Once invoked, a scheduled script deployment record is picked by the scheduler, placed into the processor pool, 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 processor pool, can still be successful. Therefore**,** 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 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 the SuiteScript API `task.create(options)`. When `task.create(options)` returns a task id, it means the invocation was successful. However, if it returns “null”, then the invocation was unsuccessful because the script deployment is already waiting in the processor pool prior to the API call.

4.6 Design Considerations for Using the externalId Field

4.6.1 Design Considerations for Using the externalId Field

The externalId 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 externalId 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 internalId field), thereby eliminating an extra search API call to obtain the records’ internal ID values.

Since the externalId field is a standard field that is meant to store external primary keys, if you are building Integration SuiteApps using SuiteTalk web services, you 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 externalId 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 externalId field can be very useful. An example of a good use of the externalId 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 externalId 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 externalId Field When Importing Critical Business Data

The externalId field should be used to safeguard your critical business data during NetSuite system errors and exceptions. An example of this critical business data is the 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 Integration 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 Integration SuiteApp is not idempotent in its API calls. These problems can be avoided by using the externalId 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 ExternalId 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 Integration 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 externalId 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 Integration 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 externalId 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 Integration 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 SuiteScript. 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 and Race Conditions, as described below.

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 SuiteScript scripts. Typically, race conditions occur when two or more processes (multiple SuiteScript scripts 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 SuiteScript scripts 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 scripts are executed at roughly the same time:

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

Note: NetSuite standard records have built-in support for record locking, therefore SuiteScript scripts 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 SuiteScript by first loading a custom record into memory with `record.load(options)`, then updating it with `record.save(options)`.

When two or more SuiteScript scripts 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.
- Script A loads the custom record into memory using `record.load(options)` and reads the value 1.
- Script B loads the custom record into memory using `record.load(options)` and reads the value 1.
- Script A increments the value to 2 in memory.
- Script B increments the value to 2 in memory.
- Script A submits the updated custom record using `record.save(options)` ; the operation succeeds.
- Script 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 scripts, therefore optimistic locking in scripts only works when a custom record is loaded into memory using `record.load(options)`, then updated using `record.save(options)`.

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:

- `record.submitFields(options)`

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

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

If your SuiteApp's scheduled scripts are using `record.submitFields(options)` to update custom records due to its superior performance, and you want to take advantage of the data integrity advantages offered by optimistic locking, consider using more processors 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 processors. 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 `record.load(options)` and `record.save(options)`.

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. SuiteScript scripts 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 externalId field.

The externalId 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 externalId field values is used to build virtual semaphores for SuiteScript scripts.

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 scripts 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 internalId 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
{
    const recordCombination = 'a123b456c789';

    // Create semaphore
    const semaphore = _record.create({
        type: 'customrecord_sdn_semaphore'
    });

    // Set external ID
    semaphore.setValue({
        fieldId: 'externalid',
```

```
        value: recordCombination
    });

    // Save semaphore
    const semaphoreid = semaphore.save();

    // Code 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 scripts.

Note: The virtual semaphore design pattern only works within the confines of those scripts that implement and honor it. Any other scripts 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 Localization SuiteApps in NetSuite OneWorld Environments

Localization SuiteApps are SuiteApps that help make NetSuite compliant with local governmental regulations and regional peculiarities for a target locale. Every target locale (or country) has unique regulations that necessitate businesses that operate there to ensure their IT systems are compliant, including ERP.

Due to the unique localization requirements for every country, there is no standard profile for a localization SuiteApp. However, it is common for localization SuiteApps to include customizations in standard NetSuite transaction records, such as the Invoice record.

For example, an ERP system may be required to submit all its invoices opened in a calendar year in a specific file format to a service hosted by a local regulatory body or taxation authority. This requirement may be addressed by a SuiteScript that is included in a localization SuiteApp.

4.8.1 Design Considerations for Localization SuiteApps

Naturally a NetSuite customer that has a localization SuiteApp installed is very likely to be a company with an international footprint. They are likely to be a NetSuite OneWorld user, and due to the global nature of their business, it is highly likely that there are multiple localization SuiteApps installed for different countries' needs. Therefore, the profile for these types of customers is an important consideration for an ISV who is developing a localization SuiteApp.

The typical localization SuiteApp customer would have multiple localization SuiteApps installed in their NetSuite OneWorld account; and most of these localization SuiteApps will have SuiteScript scripts (client scripts and user event scripts) and customizations on standard transaction records. This means that as the number of countries in which the company operates increase, so will the number of installed localization SuiteApps, and each localization SuiteApp will take up more SuiteScript slots and processing resource on transaction records.

Each SuiteScript on a transaction record not only takes up a script slot (10 for user event scripts and 10 for client scripts per transaction record), they also need to be loaded into server side memory for execution. Therefore, it is important that each localization SuiteApp adopts design best practices to optimize this limited resource. Failure to do so will result in performance degradation for the most important transaction records in the NetSuite ERP system.

To avoid excessive amount of SuiteScript scripts deployed on transaction records, SuiteApps (especially localization SuiteApps) must consolidate multiple scripts that address different requirements into a smaller number of scripts. This practice improves transaction performance and eases the administrator's management task by cutting down on "stacked" scripts from multiple vendors. This can be achieved by having mature product management oversight to ensure a smaller number of scripts are designed to adequately address different functional requirements.

In NetSuite version 2020.1, a new SuiteScript feature Localization Context was introduced. This feature allows you to define the localization context in which a user event script or client script on a record will execute and prevents the script from executed if it is not relevant to the record's subsidiary in question. User event scripts and client scripts that use the localization context feature do not get counted into the allocated SuiteScript slots for the record in which they are deployed, thus they are helpful in alleviating the SuiteScript stacking problem. For example, if a user event script or client script is written to localize NetSuite for businesses operating in Italy with an Italian subsidiary and is not relevant for other countries, then it must use the localization context feature to limit its invocation to be by that subsidiary.

Important: A localization SuiteApp must not have more than three user event scripts and three client scripts per standard transaction record that do not utilize the Localization Context feature.

Important: A SuiteScript script must be written in SuiteScript 2.0 or later in order to use the Localization Context feature. For localization SuiteApps that were developed prior to the advent of SuiteScript 2.0, additional time and resource may be required to upgrade in order to utilize functionality available in the Localization Context feature. Depending on the nature and size of the SuiteApp, this may be a significant investment. Therefore, an interim solution may be considered where only the user event scripts and client scripts deployed on the most commonly used standard transaction records are upgraded to SuiteScript 2.x and use the Localization Context feature. The purpose of this interim solution is to utilize the Localization Feature to minimize the performance impact on transaction records without incurring significant time and resource into upgrading an entire SuiteApp. These most commonly used transaction records are: Invoice, Vendor Bill, Credit Memo, Vendor Credit, Sales Order.

4.9 Further Reading

You can find more information in the NetSuite Help Center or in SuiteAnswers, using the following search terms:

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

5. Principle 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.

This section explains how you can apply the NetSuite roles and permissions model to the customization objects.

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

- Roles and Permissions
- Secure SuiteScript Designs
- Validate Data Input
- Programmatic Access to NetSuite Passwords
- External System Passwords and User Credentials
- Credit Card Information
- SuitePayments API
- NetSuite as OIDC Provider
- Privacy Considerations
- Token-Based Authentication for SuiteTalk Web Services
- Token-Based Authentication for RESTlets

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 roles 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 `task.create(options)`.

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 Web Services and/or RESTlets) are also role-based, therefore, the security design principles 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.

5.1.1 Using Bundle Installation Scripts

A bundle installation script is a specialized SuiteScript script that is executed upon installation, update or uninstall of a bundle/SuiteApp. It is a useful tool to initialize your SuiteApp. This script type can handle setup tasks, configuration tasks and data management tasks prior to or after bundle install/update to ensure smooth operation of your SuiteApp.

Bundle installation scripts are executed without an audience and with an administrative role/permission, even on accounts that don't have the SuiteScript feature enabled. Therefore, it is imperative that security play a big consideration when constructing the script. With administrator privilege, actions such as editing and deleting records/objects that are not part of the bundle should not be performed. Additionally, a bundle installation script should not alter an account's settings and configuration such as the enabling or disabling of a feature.

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 script 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 SuiteScript scripts 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. For example, a Rental Agent custom role may be included to access the script that reads, creates, and updates the Rental Agreement custom records. An Agent Supervisor custom role may have additional permissions to delete the Rental Agreement custom records, and/or have permissions to execute SuiteScript scripts designed for supervisor tasks.

In this 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 SuiteScript scripts set to Execute as Admin whenever possible.

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

Important: SuiteScript script 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 components. 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.

You 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 (`search.load(options)`, `search.create(options)`, SuiteTalk search), loading (`record.load(options)`, SuiteTalk get), or looking up Employee records (`search.lookupFields(options)`. SuiteApps should not be designed based on the assumption that user passwords can be programmatically obtained.

Note: 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 SuiteScript scripts 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 NetSuite as OIDC Provider. This feature is based on OpenID Connect (OIDC) Single Sign-on, with NetSuite acting as the OIDC provider (OP). The NetSuite as OIDC Provider feature uses OAuth 2.0 as an authorization framework.

If using NetSuite as OIDC Provider is not a feasible option, or if the external system user can't be prompted to enter passwords inside the NetSuite interface, then the alternative is to use Secrets Management.

The Secrets Management feature provides a way to store, manage, and reference API secrets securely in NetSuite. API secrets include hashes, passwords, keys, and other secret values for managing digital authentication credentials. Access Secrets Management at Setup > Company > Preferences > API Secrets.

By preventing SuiteApps from accessing secret values, partners can perform system-to-system integration because secret values are only referenced by script ID, and the password value cannot be displayed by target account administrators.

The supported SuiteScript modules for managing secrets are N/sftp, N/https, N/crypto, N/keyControl, and N/certificateControl. For further information on how to use secret management, refer to 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 in which they are running. Also note that SuiteBuilder (point-and-click customization) does not support creating credit card objects.

An example of a situation where a SuiteApp might need 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 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). 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

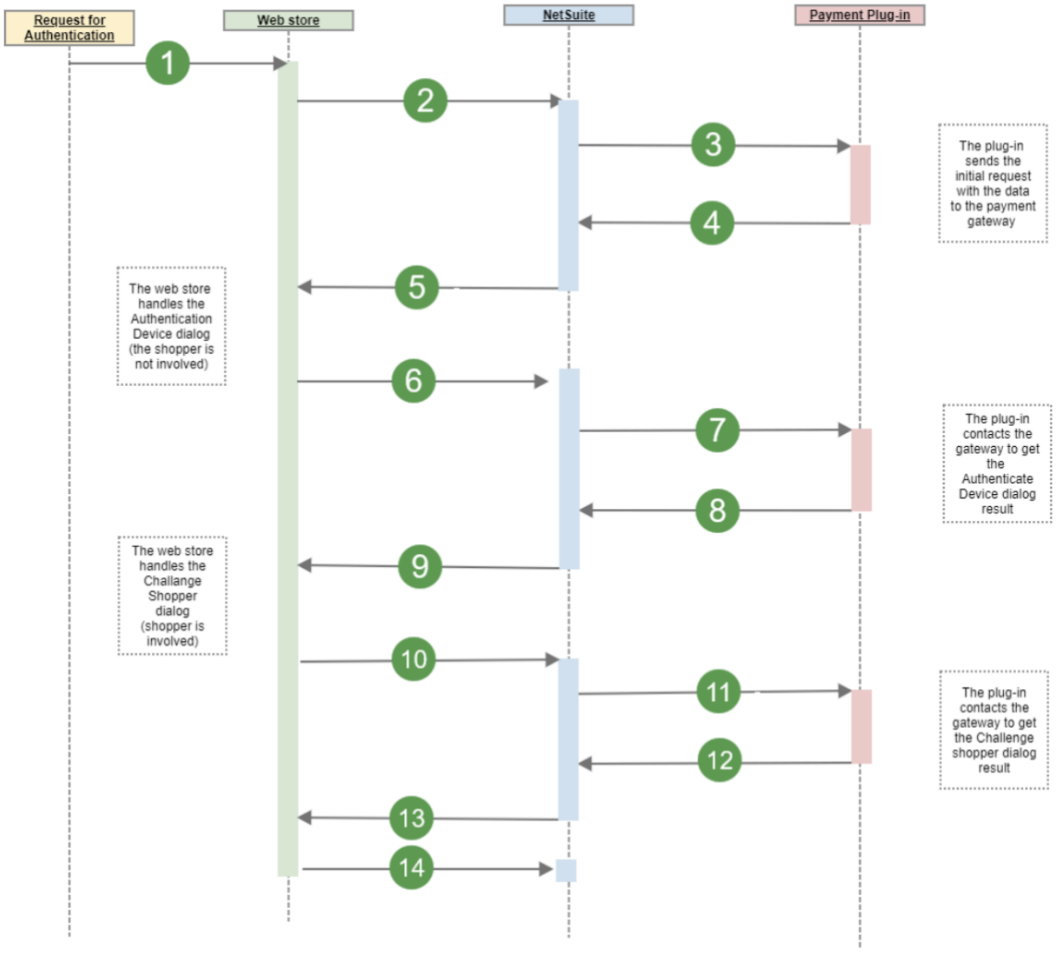
The provider of a SuitePayments solution can be a payment gateway service or a third-party developer/ISV. Since the latter is the most common use case, it is imperative for a SuitePayments solution developer to put in measures that establishes accountability to the users for all transactions that pass 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 SuiteApp installation and/or during use of the solution. See additional, bundling naming conventions found in Principle 8: Maintain Your SuiteApps.

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.



Full 3D Secure Workflow

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.

5.8 NetSuite as OIDC Provider

The NetSuite as OIDC Provider feature provides an alternative to NetSuite's Outbound Single Sign-on (SuiteSignOn) feature. This feature is based on OpenID Connect (OIDC) Single Sign-on, with NetSuite acting as the OIDC provider (OP). The NetSuite as OIDC Provider feature uses OAuth 2.0 as an authorization framework.

The Outbound Single Sign-on (SuiteSignOn) feature is scheduled for end of support in version 2024.2. You should update your integrations to use NetSuite as OIDC Provider as soon as possible.

For more information, follow the links below:

- SuiteAnswers: https://suiteanswers.custhelp.com/app/answers/detail/a_id/98241/loc/en_US
- Tutorial: https://nlcorp.app.netsuite.com/core/media/media.nl?id=372684933&c=NLCORP&h=ktTrGe1_o7GfU8KyQHSU7DjLkctl0tbMfj_sKCCZa0vA8OT3&_xt=.zip

5.9 Privacy Considerations

As a SuiteApp developer, you and your SuiteApp may be interacting with sensitive customer data, such as social security numbers, tax identification numbers, bank account 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: https://2016.export.gov/safeharbor/index.asp.

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, Publishing Tasks, and Customer Tasks. For guidance on how to perform 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

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 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.

A web services request that uses TBA must use the `TokenPassport` complex 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/v2022_1_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 NetSuite 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&060cd9ab3ffbbe1e3d3918
e90165ffd37ab12acc76b4691046e2d29c7d7674c2
```

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. 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. This is the only way to use `TokenPassport`, there is no dedicated web service operation to authenticate using TBA.

5.10.2 Publishing Tasks

The Integration record and the SuiteApp Control Center/ SuiteBundler are central to the distribution of TBA integration applications. Therefore, both must be used for publishing your SuiteApp. The following are the 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.
- Create a SuiteApp project and import all necessary custom objects from your development account so that it contains the Integration record, the integration custom role, and any other customization objects that the integration requires to operate successfully.
- Generate a Project ZIP file from your SuiteApp project and upload it to the SuiteApp Control Center. When creating the SuiteApp definition, first decide if your SuiteApp will be managed. If the Managed checkbox is not available, then first create request before proceeding.

- Set the availability of your SuiteApp as public or shared. If shared, add customer account IDs so the customers have access to the Install button in the SuiteApp Marketplace.
- Instruct customers to install your SuiteApp.

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 be propagated to your SDF SuiteApp and customer accounts.

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, Publishing Tasks, and Customer Tasks. For guidance on how to perform 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

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 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.

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 should 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 NetSuite server.
- **oauth_signature_method** – A hash algorithm that can be used to create an RFC 2104-compliant signature. Supported choices are: 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 article 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:

```
1234567&71cc02b731f05895561ef0862d71553a3ac99498a947c3b7beaf4a1e4a29f7c4&89e08d9767c5ac
85b374415725567d05b54ecf0960ad2470894a52f741020d82&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 – 278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd

- Token secret – 060cd9ab3ffbbe1e3d3918e90165ffd37ab12acc76b4691046e2d29c7d7674c2

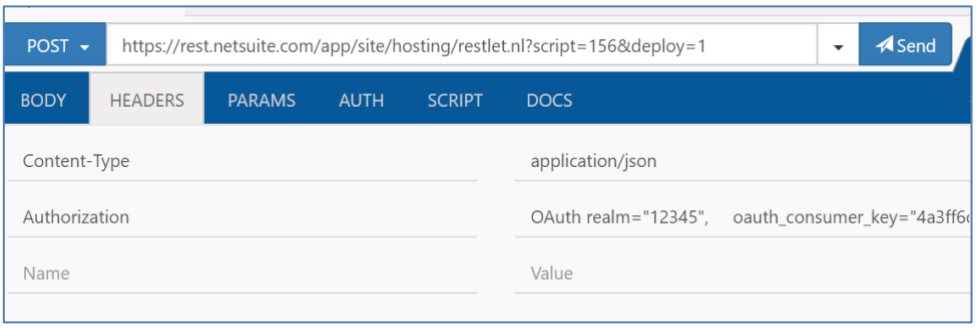
In this case, the key would be:

```
7278da58caf07f5c336301a601203d10a58e948efa280f0618e25fcee1ef2abd&060cd9ab3ffbbe1e3d3918
e90165ffd37ab12acc76b4691046e2d29c7d7674c2
```

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.



The following snippet shows a correctly formatted OAuth authorization header:

```
Authorization:
  OAuth realm="12345",

oauth_consumer_key="4a3ff6c251a55057bb1e62d8dc8998a0366e88f3a8fe735265jc425368b0f154",
  oauth_token="52cfe88fecf2e2b74e833e7dfc4cae79ff44c3ca9f696d61e2a7eac6c8357c3c",
  oauth_nonce="qUwlmPvtGCS4sHJe8F7x",
  oauth_timestamp="1462453273",
  oauth_signature_method="HMAC-SHA256", oauth_version="1.0",
  oauth_signature="8PIP9lIYxUmUONjxEFJUSMD9oOmc%3D",
```

5.11.2 Publishing Tasks

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

- Create a SuiteApp project and import all necessary custom objects from your development account so that it contains the Integration record, the integration custom role, and any other customization

objects that the integration requires to operate successfully.

- Generate a Project ZIP file from your SuiteApp project and upload it to the SuiteApp Control Center. When creating the SuiteApp definition, first choose whether your SuiteApp will be managed. If the Managed checkbox is not available, then first create request before proceeding.
- Set the availability of your SuiteApp as public or shared. If shared, add customer account IDs so the customers have access to the Install button in the SuiteApp Marketplace.
- Instruct customers to install your SuiteApp.

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 your SDF SuiteApp and customer accounts.

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.x provides additional cryptographic functions that are not available in SuiteScript 1.0, such as hashing and Hash-based Message Authentication Code (HMAC). 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 an external system, it is recommended to first encrypt content using the SuiteScript N/crypto module (for confidentiality), and then calculate the HMAC digest from the generated CipherPayload object (for integrity and authentication).

Note: The MD5 and SHA-1 algorithms are considered deprecated for hashing and HMAC operations, and are available for legacy purposes only. New integrations are encouraged to use the SHA-256 and SHA-512 algorithms.

5.12.1 Using the N/crypto/random module from Math.random()

The Math.random function is not generally considered cryptographically secure due to its algorithmic design, which is typically a pseudo-random number generator (PRNG) that produces a sequence of numbers that appear random; however, they are generated from an initial seed value. In theory, if you know the initial seed value, you can predict the entire sequence of numbers.

For use in cryptographic applications (i.e., generating keys, tokens, or passwords), SDN recommends using the N/crypto/random module instead of the Math.random function. For example, you can use the N/crypto/random module along with the N/encrypt module to generate nonce – an arbitrary string/number that is to be used only once in cryptographic communication.

The following sample code demonstrates this operation:

```
let randomByte = random.generateBytes({size: 16});
let _out = '';
for (let i = 0; i < randomByte.length; i++){
    _out += String.fromCharCode(randomByte[i].toString());
};
let _nonce = encode.convert({
    string: _out,
    inputEncoding: encode.Encoding.UTF_8,
    outputEncoding: encode.Encoding.BASE_64_URL_SAFE
});
return _nonce;
```

The N/crypto/random module is used to create random bytes while the N/encode module is used to convert the bytes to a Base64 string for the use in nonce. This provides a reliable and secure way to generate cryptographically secure pseudorandom numbers to ensure that your application can use leverage strong randomness thereby enhancing overall security and robustness.

6. Principle 6: Test Your SuiteApps

It is your responsibility to ensure that the SuiteApps distributed 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. You must retest, and, if necessary, update your published SuiteApps during the Release Preview phase that accompanies each new NetSuite release.

6.1 SuiteCloud Unit Testing

The SuiteCloud Software Development Kit (SuiteCloud SDK) is a suite of tools that powers the development of SuiteApps enabling users to interface with NetSuite's development environment. It notably includes a Jest module for testing SuiteScript 2.x applications. To learn more about Jest, go to https:/jestjs.io.

The SDK includes pre-created stubs, which are essentially class definitions for NetSuite modules. These stubs expose the same methods as their real counterparts but can be used as mockups for testing purposes.

Each test should be defined within a file named with the ".test.js" extension and the file should be located in the "**tests**" directory at the root of the project folder. This file must import the SuiteScript modules under test, along with stubs for any modules to be mocked (for example, "N/record"). Additionally, the SDK provides stubs that simulate the behavior of common object instances used in NetSuite development.

Here are a few examples of import statements:

```
import Suitelet from "SuiteScripts/Suitelet";
import record from "N/record";
import Record from "N/record/instance";
```

The jest.mock('function') command instructs Jest to use the mocked version of a function or object in place of the actual one during testing. To apply this in practice, add the following lines to the test file (for the N/record module, as an example):

```
jest.mock ("N/record");
jest.mock ("N/record/instance");
```

After setting up the test, define instances using the conventional syntax provided by Jest. For example:

```
describe("Suitelet Test", () => {
    it("Sales Order memo field has been updated", () => {
        // given
        const context = {
            request: {
                method: 'GET',
```

```
                parameters: {
                    salesOrderId: 1352
                }
            }
        };
        record.load.mockReturnValue(Record);
        Record.save.mockReturnValue(1352);
        // when
        Suitelet.onRequest(context);
        // then
        expect(record.load).toHaveBeenCalledWith({id: 1352});
        expect(Record.setValue).toHaveBeenCalledWith({fieldId: 'memo', value:
'foobar'});
        expect(Record.save).toHaveBeenCalledWith({enableSourcing: false});
    });
});
```

In this example, the test is configured to update the memo field of an order with the ID 1352. The setup includes a context object that mimics NetSuite's structure returning the saleOrderId. When the record module's load method is invoked, it returns the instance of the record. Additionally, the save method is configured to return a specific ID, which matches the one defined in the context. Ultimately, Jest's `expect` API is utilized to confirm that the function's output matches the expected result.

6.2 Understand NetSuite Phased Releases

As a cloud-based application, NetSuite's regular maintenance and upgrades are performed 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 2024.1 and 2024.2). These typically start in January and July, respectively.

Customers are notified of their upgrade schedules and are asked to test their data in the Release Preview account (an isolated 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 starting with Phase 0 and then Upgrade Month 1 (UM1) to Upgrade Month 3 (UM3). 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. UM1 is the first batch of live customer accounts to be upgraded; the percentage of customer accounts is small (~20% of total accounts). As the phased release period progresses, the total number of accounts to be upgraded increases. After UM3, there are a few special upgrade periods when trailing and demo accounts are upgraded. Then, NetSuite exits phased release until the next upgrade cycle begins. The following figure illustrates the phased release approach:

	Phase 0	UM1	UM2	UM3	Special upgrades
XX.1	Januay	February	March	April	
XX.2	July	August	Septembre	October	
Schedule	Process Start	About one month after Phase 0	About one month after UM1	About one month after UM2	After UM3
Partner Accounts	Leading SDN accounts				Trailing SDN accounts
Customer Accounts	0%	20%	60%	100%	100%

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 release 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 NetSuite 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 NetSuite 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.beta.netsuite.com**.** These 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.

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 access to Release Preview is limited to the 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 3 (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

6.5 SuiteApp Best Practices and 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, which take place prior to Phase 0. These webinars are intended for developer, 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 (available in the APC portal) provided to SDN partners to look up the exact upgrade date and Release Preview start and end dates for your customers. The information returned can help you formulate testing schedules. Your customer 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.

4. During Phase 0, install your SuiteApps in an SDN leading account (if not already done) and test them. The two to three week window between Phase 0 and Phase 1 is the ideal time to test your SuiteApp because customers are not yet using the new release in production. Therefore, bugs found and fixed during this time frame will not be exposed to customers.

5. Report platform bugs to NetSuite Customer Support.

6. Starting from Phase 0, access Release Review and test your SuiteApps against the snapshots of your data on the SDN trailing environment accounts. You may do this anytime during the phased release, but the sooner, the better.

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

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 applications require some unique QA elements that must be considered by QA engineers and application designers.

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

- **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: Design for Security and Privacy.)

- **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 resources 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.

- **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.

  Rationale – Testing your SuiteApp in a 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.

- **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.

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

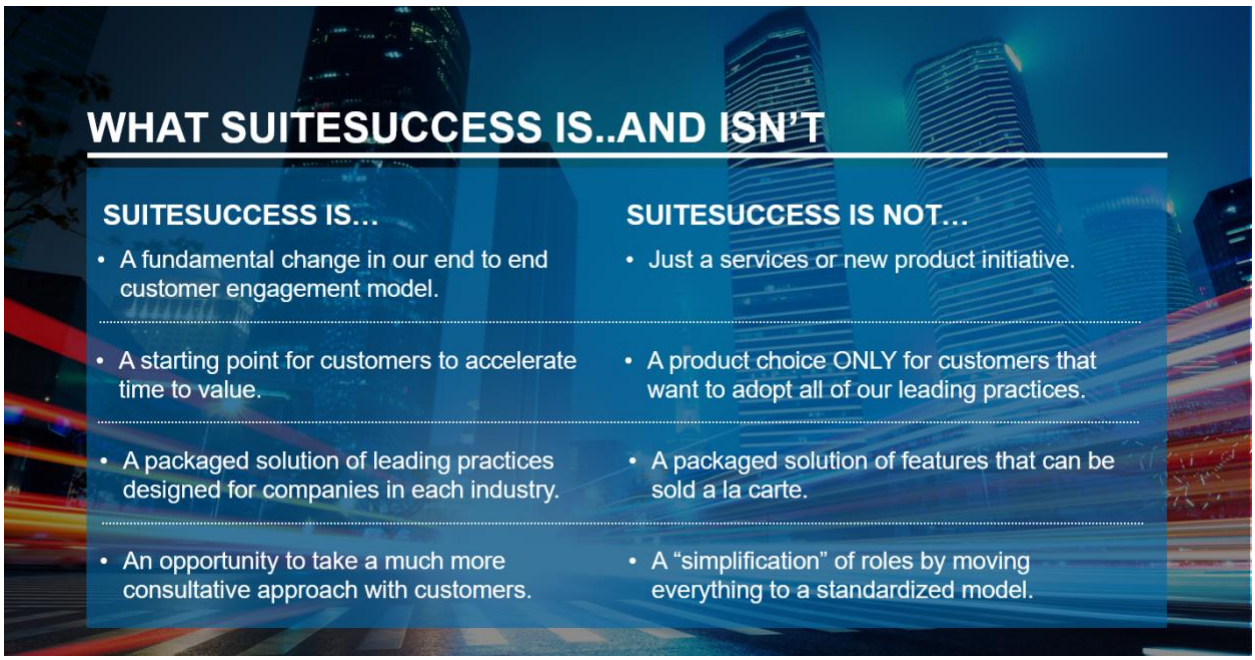
  Rationale –Installation scripts (Bundler or SDF) are server side scripts that execute when a SuiteApp is installed or updated. Sometimes these scripts handle data conversion and cleanup tasks. Therefore, it is imperative that they are tested on QA accounts prior to being executed on customer accounts.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

Often the only way to thoroughly test the SuiteApp install/update process, including all installation scripts, is to have multiple QA accounts (see the stress testing QA checkpoint). 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, 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.



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 SuiteSuccess 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:

- Write to any custom fields contained in the SuiteSuccess test account
- Create any “rows” of custom records that are defined in the SuiteSuccess test account
- Create any customization objects that reference object(s) contained in a SuiteSuccess test account
- Intentionally, or unintentionally, include any objects in a SuiteBundle that originate from the SuiteSuccess test account

These requirements are driven by two reasons:

- At the time of this guide’s writing, there are no supported APIs for SuiteSuccess customizations in any of the vertical versions.
- 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 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 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 Integration 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 SuiteScript Performance Optimizations are still applicable.

6.8 What are NetSuite Sandbox Accounts?

NetSuite sandbox accounts contain a replica of the configuration, customization and data from a live production account as of a specific date, but do not process external transactions, such as payments or email campaigns. Sandbox accounts 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:

- 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, and
- The SDN partner is a NetSuite user itself, and has its own sandbox account(s).

A production account and its sandbox account(s) are hosted in the same environment, and can be accessed from system.netsuite.com. The My Roles page or the Change Roles list can be used to select a role. They have different account IDs as the differentiating attribute. Sandbox account IDs include "_SBx", 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.

6.8.1 Building your Integration SuiteApp to Support Sandbox Accounts

In essence, accessing a new sandbox account using SuiteTalk or RESTlets is very similar to accessing a production account. This means all the best practices and guidelines also apply to sandbox accounts. There are no changes to the actual APIs. Instead, an Integration SuiteApp simply needs to specify the sandbox account ID when making SuiteTalk API calls or when invoking RESTlets.

7. Principle 7: Consider Distributing Your SuiteApps in a Managed Fashion

Wide-scale ISV SuiteApp deployments are best done with managed SuiteApps. The Managed SuiteApps 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.

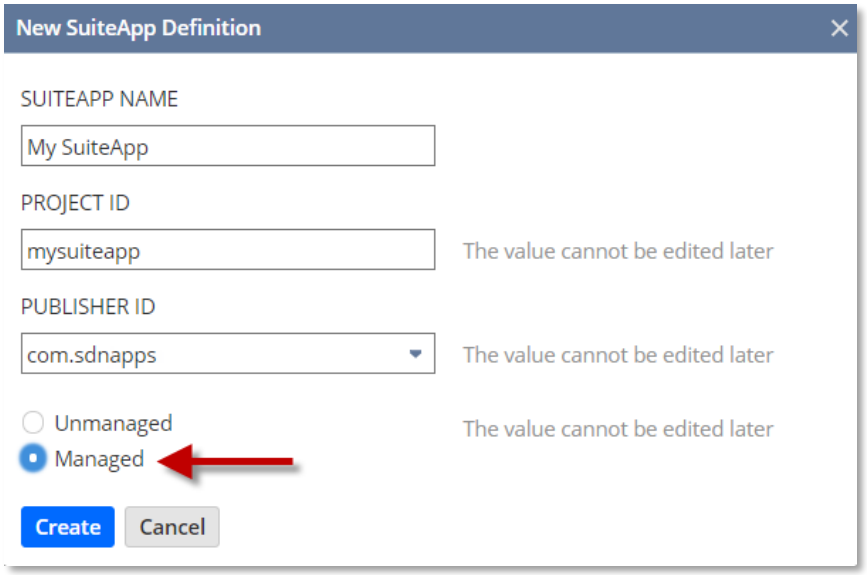
Note: For more information on managed bundles, see *Understanding Managed Bundles* in the NetSuite Help Center.

To ensure the Managed SuiteApps feature is activated on publisher environments that are robust and easily supportable, SDN partners must publish their SuiteApps using the Publisher Environment Model, which consists of a development account, a QA account (leading and lagging) and a publisher account. To ensure SuiteApps that can be deployed in a managed fashion are of sound quality, they must have achieved the latest BFN validation. Note that development, QA, and publisher accounts must be SDN trailing accounts.

The 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. Therefore, it is important for vendors to push managed SuiteApp updates to their install bases with the utmost of care.

Note: For more information about publishing your SDF SuiteApps, see Publishing of SDF SuiteApps to SuiteApp Marketplace.

NetSuite enables the Managed SuiteApps feature only on publisher accounts, which is the account linked to your Publisher ID. This allows the SuiteApp Release Manager to select the Managed option when creating a new SuiteApp Definition in the SuiteApp Control Center.



It is important to select the Managed option when you set up the SuiteApp Definition because it cannot be changed once the definition is created. Enabling the Managed option, however, does not affect how SuiteApps are published in the SuiteApp Marketplace. Each SuiteApp must still undergo BFN and listing verifications prior to publishing.

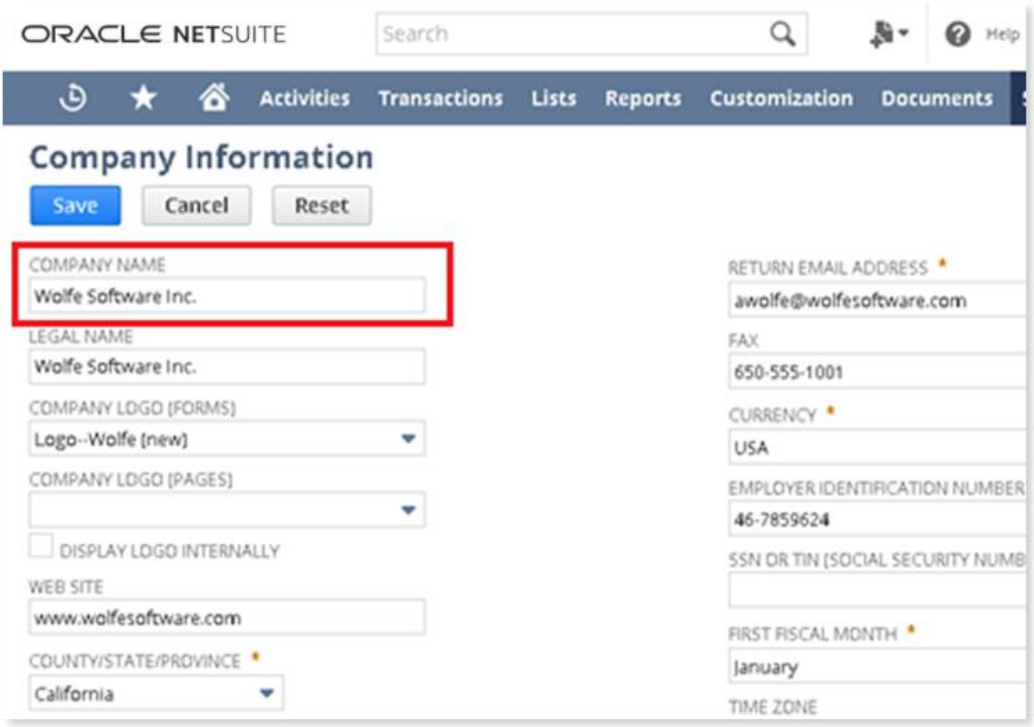
If the Unmanaged/Managed options do not appear in the New SuiteApp Definition window, it means that the Managed SuiteApp feature is not enabled in your publisher account. You need to request the feature to be enabled first before creating the SuiteApp Definition.

Important: (Warning) If a SuiteApp Definition is created as unmanaged, it cannot be changed to managed, even if SDN subsequently enables the Managed SuiteApp feature. To request the Managed SuiteApp feature, please open a NetSuite support case via your APC login role.

8. Principle 8: Maintain Your SuiteApps

SuiteApp Control Center allows SDF SuiteApp developers to quickly define and easily install/upgrade leading and lagging versions of your applications to your customer install base.

When creating an SDF SuiteApp definition, there are a number of mandatory attribute fields including SuiteApp Name, Project ID, and Publisher ID. When an SDF SuiteApp is positioned for generally available (GA) release, its latest version (leading) must be created in the SuiteApp Control Center (by uploading a new SuiteApp ZIP archive) and then released to deprecate its older version (lagging). The Publisher Account, which is the SDN account linked to the Publisher ID, must also be named with the SDF SuiteApp provider's company name. An account's Company Name setting can be changed at Set Up > Company > Company Information as shown below:



The following are additional SuiteApp naming convention requirements:

- The name of the SuiteApp must describe the feature or functionality delivered.
- Do not include the words "SDF" or "SuiteApp" in the SuiteApp's name.
- Use descriptive and consistent names, for example, Japan Tax Reports.
- Indicate non-GA SuiteApps accordingly, for example, add "BETA" to the end of the name.

Using SuiteApp Control Center, the definition and management of the SuiteApp distribution process can be done in a singular location and consists of the following attributes unique to the cloud paradigm:

- Multi-tenant structures where it can serve multiple customers from a single location.

- Managed SuiteApp feature that provides automated application upgrades without customer actions.
- The concept of leading and lagging versions to support version phasing and ensure smooth upgrading of your SuiteApp during phasing periods. See *SuiteApp Versions* in the NetSuite Help Center for more information about this feature.

Due to the complexity of the SuiteApp distribution process, and its need for robustness, a mechanism such as the SuiteApp Control Center that enables ISVs to publish applications to multiple customers, is one that requires maintenance. Therefore, it is important for ISVs to recognize that a SuiteApp Release Engineer must have knowledge of SuiteApp Control Center, the SuiteApps themselves, and cloud application distribution practices. This knowledge is required for the ongoing operation and maintenance of SuiteApp phased versions.

SuiteApp development teams should review the following sections to have an overview of SuiteApp publisher environments, development processes, and SuiteApp publishing.

- Setting Up the SDF SuiteApp Publisher Environment
- Development Process of your SDF SuiteApps
- Publishing SDF SuiteApps to SuiteApp Control Center

8.1 Setting Up the SDF SuiteApp Publisher Environment

This section provides an overview of the SDF SuiteApp publisher environment model. In a typical ISV SDF SuiteApp publisher environment, at least four accounts are involved:

- The development/publisher account
- The trailing QA account
- The leading QA account
- The sales demo account

The SDF SuiteApp author (the ISV) controls the dev/publisher, and QA accounts. The NetSuite account administrators control their respective customer accounts.

The following SuiteCloud features must be enabled in your development/publisher and QA SDN accounts. These features are required for SDF to work. To enable the features, go to Setup → Company → Enable Features and click on the SuiteCloud subtab. Select each of the following features and then click the Save button.

- SuiteScript – Client SuiteScript, Server SuiteScript, and SuiteScript Server Pages
- SuiteTalk (Web Servies) - SOAP Web Services
- Manage Authentication - Token-Based Authentication
- SuiteCloud Development Framework – SuiteCloud Development Framework and SuiteApp Control Center

It is also necessary to assign the Developer role to users who will develop, deploy, and test using SDF. The Developer role has an existing set of permissions for development purposes. If the default Developer role does not suffice, it can be customized as necessary.

In addition, account authentication via token credentials must also be set up in the IDE against your development and QA accounts in order to import and deploy customizations to your SDN accounts. This also allows the creation of new custom objects directly into your account using the IDE.

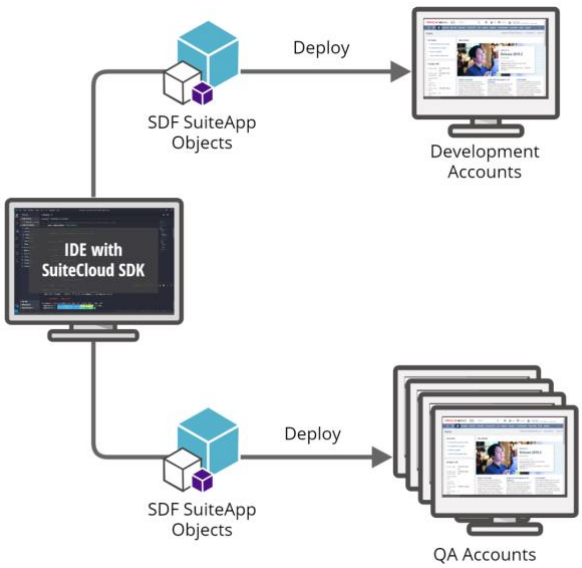
Note: For additional details on setting up your token credentials for development, see *SuiteCloud IDE Tutorial* on the SDN Technical Onboarding site.

Maintenance of your SDF SuiteApps through the SuiteApp Control Center relies on the dev/publisher account being linked to the Publisher ID. Hence, the SuiteApp publisher environment administrator(s) should ensure that this is always the case. If the ISV, for any reason, needs to be provisioned a new dev/publisher account or wants to have their Publisher ID changed, a support case must be created through the APC portal in order for the SDN Operations team to perform the necessary actions and link the publisher account and Publisher ID.

8.2 Development Process of Your SDF SuiteApps

An SDF SuiteApp is developed using an IDE, preferably Visual Studio Code (VS Code) with the SuiteCloud Extension or WebStorm with the SuiteCloud IDE Plug-in installed (which is a tool from the SuiteCloud SDK).

The SuiteApp project is deployed to the development account for unit testing. You can also deploy a SuiteApp project using the SuiteCloud Command-Line Interface (CLI) tool for Node.js or Java which provides a means to create batch shell scripts to automate validation and deployment of the SuiteApp project. Once development is complete, deploy the SuiteApp project to the leading or lagging QA account(s) to fully test and ensure quality. The following diagram illustrates this process.



Since SDF development is de-coupled from the NetSuite account, the ISV can use a third-party revision control system for code change management and versioning purposes.

Note: For additional details on referencing objects in an SDF SuiteApp, preparing your SuiteApps for deployment, and the SuiteCloud SDK, see the following topics in the NetSuite Help Center:

- *SDF Custom Object Dependencies in SuiteApps*
- *SuiteCloud Project Deployment Preparation*
- *Getting Started with SuiteCloud SDK*

8.3 Publishing SDF SuiteApps to SuiteApp Marketplace

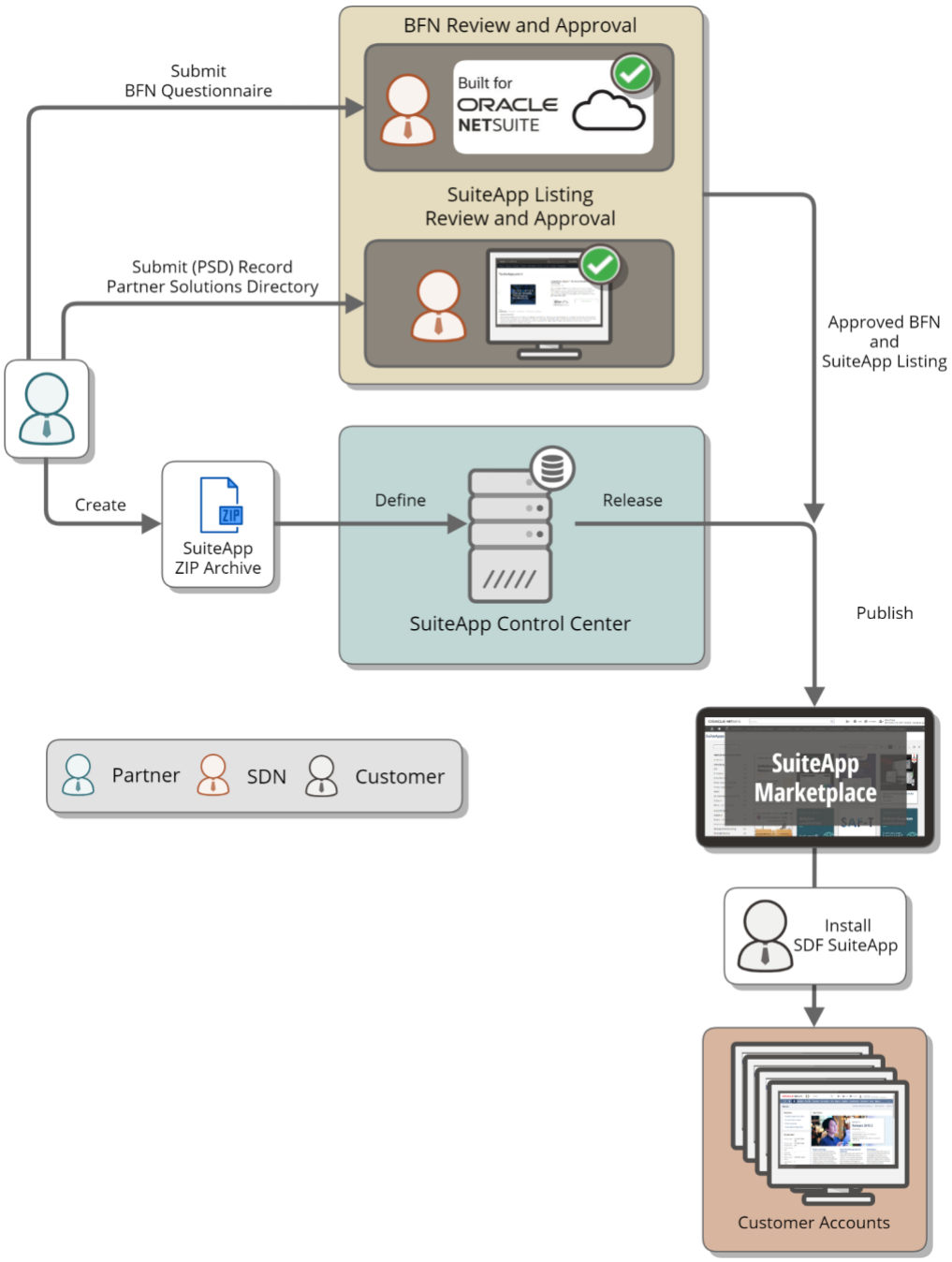
The following are the prerequisites for an ISV to publish an SDF SuiteApp to the SuiteApp Marketplace:

- It must be defined and released in the SuiteApp Control Center
- It must be BFN approved
- Its SuiteApp listing submission must be approved by SDN

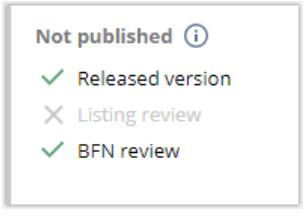
Note: The above prerequisites need not happen in a specific order. The review and approval process can be requested and can take place even before the SuiteApp is defined and released in the SuiteApp Control Center. Furthermore, the correct Application ID must be used in the SuiteApp listing, otherwise, it will not be linked to the SDF SuiteApp in the SuiteApp Control Center.

To define an SDF SuiteApp, the SuiteApp Release Manager must upload the SuiteApp ZIP archive (also referred to as the SuiteCloud Project ZIP file) to create a definition in the SuiteApp Control Center. Refer to Principle 7: Consider Distributing Your SuiteApps in a Managed Fashion, to learn more about the Managed option when creating a new SuiteApp Definition.

When the SDF SuiteApp is released, it means that it is ready for publishing (leading or lagging), however, it will remain hidden from customers in the SuiteApp Marketplace until the appropriate approvals are obtained. Once BFN and SuiteApp listing approvals are obtained, the listing is published by SDN and the SDF SuiteApp will be available for customers to install from the SuiteApp Marketplace. The following diagram illustrates this process.



If the publishing prerequisites are not met, required actions are displayed in the SDF SuiteApp definition in the SuiteApp Control Center as shown here:



In this example, the SuiteApp is released and the BFN review completed, but is pending SuiteApp listing review and approval.

Note: For additional details on releasing SuiteApp versions, listing access and requirements, see the following topics in the NetSuite Help Center:

- *SuiteApp Versions*
- *SDF SuiteApp Creation and Listing Access*
- *Requirements for SuiteApp Marketplace Availability*

9. Principle 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

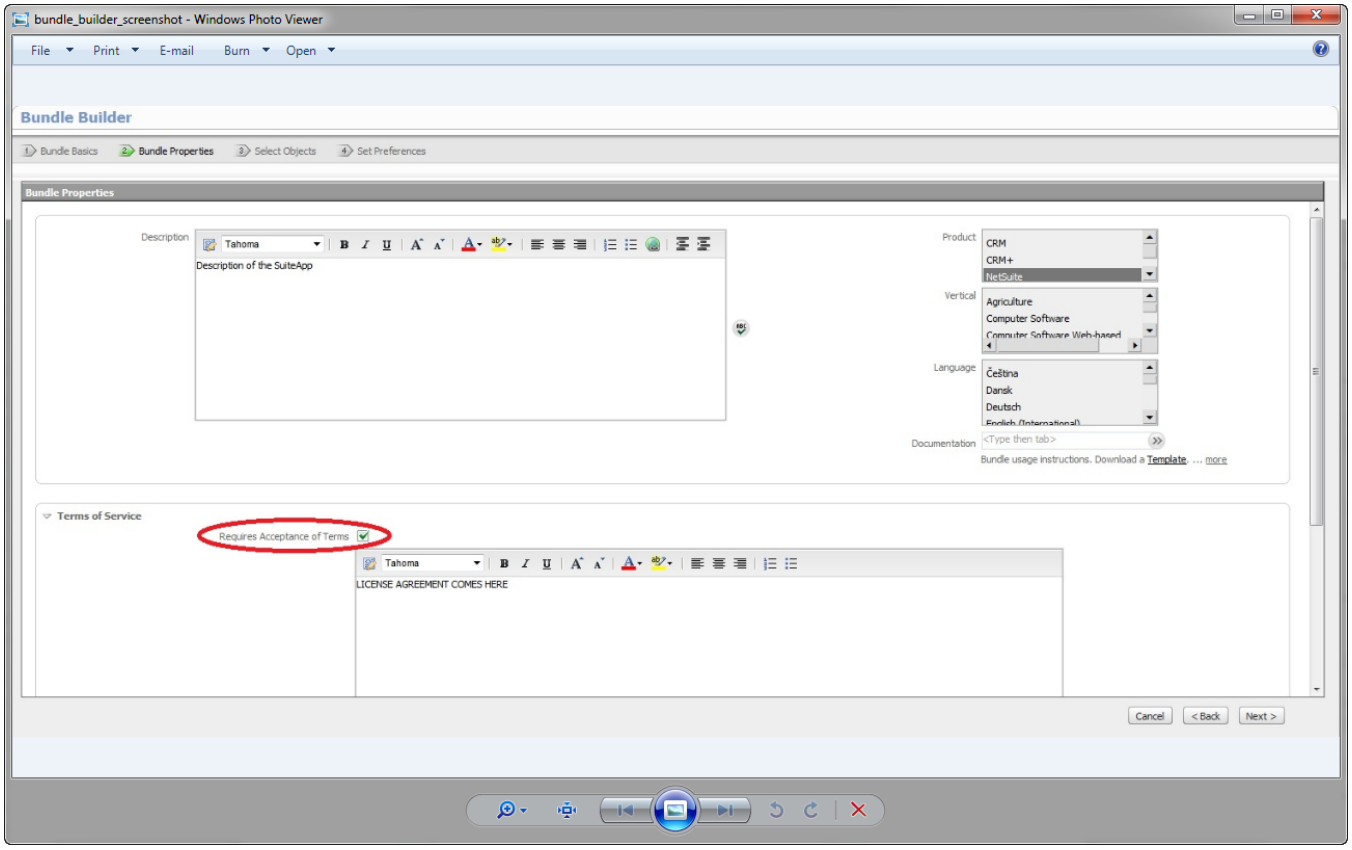
You should carefully protect your intellectual property in your agreements with employees, consultants, customers, and partners. You must have the right to permit NetSuite to make the SuiteApp available to customers through the SuiteApp repository. You should have appropriate agreements with your employees 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. These sample agreements include (1)a sample license agreement for the use of your SuiteApp, and (2) 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.

Note: For SuiteApp Control Center, the click-through agreement functionality is currently not GA and will be available in a future version.



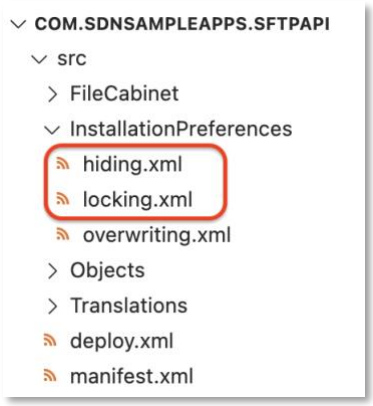
9.3 Protecting the Intellectual Property in Your SuiteApps

The key focus of this Principle so far has been to help you 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 SuiteApps. However, Developer Partners should also consider available options to help protect their intellectual property 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 SuiteApp Content

SDN Partners should consider securing their intellectual property contained in their SuiteApp, including SuiteScript source code, the definition of custom fields and custom record types, saved searches, etc.

To control access to your SuiteScript source code, an *InstallationPreferences* folder must be created in your SuiteApp project containing the *hiding.xml* and *locking.xml* files. These preferences files allow you to indicate which files and custom objects to restrict access to upon installation. Users of the target accounts will not be able to view, download, or overwrite the SDF SuiteApp's source code and modify its custom objects (see image below).



Note that the hiding installation preference should only be used on server side SuiteScript script files. Client-side script files with source code files configured to be hidden 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, the Developer Partner should understand that client-side script files will be accessible in the target accounts and viewable in the client-side browser.

The definition of non-SuiteScript customization objects in SDF SuiteApps, 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 SDF SuiteApps ensure they perform predictably, and future SDF SuiteApp upgrades are backward compatible.

Note: For additional details on supported lockable custom objects, see *Lockable Custom Objects Supported by SDF* in the NetSuite Help Center.

Setting the *hiding.xml* installation preference in your SuiteApp project's JavaScript source code files controls access to and helps protect your intellectual property; setting the *locking.xml* installation preference limits the unwanted modification of your SDF SuiteApps by target account users. Therefore, NetSuite encourages the use both of these preferences in released SDF SuiteApps.

9.3.2 Redistribution of SuiteBundle Components (Legacy Content)

Note: This section is only applicable to SuiteApps distributed using the SuiteBundler. The use of SuiteBundler deployment will not be permitted for new SuiteApps, starting in 24.1

In the enterprise, customer best practice is to test the interaction of all types of SuiteApps and internal customizations, in the sandbox account 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 minimize 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:

1. Use of a click-through agreement (see Bundling a Click-through Agreement in Your SuiteApp) with appropriate terms of use and restrictions.

2. Enable the "Hide in SuiteBundle" setting in your SuiteScript source code files. This will hide your code-level intellectual property as described in the previous section.

3. Name your SuiteApp's customization objects with your company name, AND set their "Lock on Install" preferences in the SuiteBundler as described in Securing SuiteBundle Content. This prevents your company name from getting removed from the customization objects in the event they are re-distributed without your consent.

4. Whenever possible, implement account authorization coding in your SuiteScript scripts. This 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 decouples the SuiteApp from its development account. This is accomplished by representing a SuiteApp and all its components as XML files. A key benefit of using SDF is that it provides 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 using SuiteBuilder in the browser and SuiteScript source code written outside of NetSuite (often in the SuiteCloud IDE). Then, they would upload the completed SuiteScript script to the File Cabinet in their SDN development account. 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.

SDF enables you 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 to be 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. You 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 to a NetSuite account using the SuiteCloud IDE or the SDF Comment Line Interface (CLI) for testing purposes, and SDF will generate the artifacts that comprise the SuiteApp.

The XML files plus the SuiteScript source code files contain all the intellectual property 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 also use SDF to import customization objects as XML files into a SuiteApp project from the SuiteCloud IDE or SDF CLI, or use NetSuite to export a customization object using the **Download as XML** action in the NetSuite UI. If these customization objects were a part of a SuiteBundle installation in which the components are locked (see 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.

The XML files and SuiteScript source code files can also be serialized or saved into a change management system outside of NetSuite. 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, or preferably, with SDF SuiteApp published via SuiteApp Marketplace. 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 Deploy command 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 Setting Up the SDF SuiteApp Publisher Environment 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.

Note: When an SDN partner develops a SuiteApp using SDF and the SuiteCloud IDE, the project must be set as “SuiteApp” instead of “Account Customization”. When set as SuiteApp, the publisher ID will be required.

10. Principle 10: Open Source and Third-Party Software

You are solely responsible for the content of your SuiteApp, including any third-party code contained within 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 any of 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.
- A list of links where any open source software was obtained.
- The license agreements for all open source software.

11. Principle 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 pursuit of that effort.

As a starting point, the OWASP Security Principles help you make security decisions in new situations with the same basic ideas. By considering each of the principles, you 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:

https://www.owasp.org/index.php/Category:Principle

There is a great deal of additional and valuable reference material available on the OWASP site. You are strongly encouraged to peruse the OWASP references to further your education and understanding on these important security principles.

For example, you should familiarize yourself with the OWASP Top Ten Project regarding web application vulnerabilities. Details of this project can currently be found at:

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Another very important and useful resource is the OWASP Guide Project which can be found at:

https://www.owasp.org/index.php/Category:OWASP_Guide_Project

12. Frequently Asked Questions

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 includes all the material in the NetSuite Help Center along 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 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 still run, but may result in data-related, performance, or user experience issues, and 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 redesign your SuiteApps, where necessary, so that they adhere to the Principles outlined in this guide.

Appendix 1: Sample Code

Generating TBA Headers using JavaScript to Test RESTlets

This sample provides HTML code to generate POST headers to test your token-based authentication (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() {
        const rest_url = document.getElementById("rest_url").value.trim();
        const consumer_key = document.getElementById("consumer_key").value.trim();
        const consumer_secret = document.getElementById("consumer_secret")
.value.trim();
```

```
        const token = document.getElementById("token").value.trim();
        const token_secret = document.getElementById("token_secret").value.trim();
        const ns_account_number = document.getElementById("ns_account_number").
value.trim();
        const method = document.getElementById("method").value.trim();
        const contenttype = document.getElementById("contenttype").value.trim();
        const signature_method = document.getElementById("signature_method").
value.trim();

        const 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
        const oauth_headers = generateSignature(oauthValues);
        let oauth_header_string = "";
        let x = 0;
        for(let keys in oauth_headers){
            oauth_header_string += (x++==0?"":", ") + keys + '="' + oauth_headers[keys]
+ '"';
        }

        // Headers to test posting to RESTlet
        const headers = {"Authorization" : "OAuth " + oauth_header_string, "Content-
Type": oauthValues.contenttype};

        // HTML output representation of headers
        const 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
        const generateSignature = (oauthValues) => {
            const timestamp = Math.round((new Date()).getTime() / 1000.0);
            let 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
            };

            let url_params = oauthValues.rest_url.split("?")[1].split("&");
            let signature_params = {};

            for(let key in oauth_headers){
                signature_params[key] = key + "=" + oauth_headers[key];
            }
            for(let key in url_params){
                let temp = url_params[key].split("=");
                signature_params[temp[0]] = url_params[key];
            }

            let signature_string = "";
            const sortedkeys = Object.keys(signature_params).sort();

            for(let i = 0;i < sortedkeys.length;i++){
                signature_string += (i==0?"":"&") + signature_params[sortedkeys[i]];
            }

            const base_string = oauthValues.method + "&" +
encodeURIComponent(oauthValues.rest_url.split("?")[0]) + "&" +
encodeURIComponent(signature_string);

            const composite_key = encodeURIComponent(oauthValues.consumer_secret) + "&"
+ encodeURIComponent(oauthValues.token_secret);

            const 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="TD993249" />
        <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>
```

Appendix 2: Concurrency Governance Cheat Sheet

This appendix includes the Concurrency Governance Cheat Sheet. This cheat sheet is referenced in Section 3.6.3 Concurrency and Data Bandwidth Considerations.

Concurrency Governance Cheat Sheet

The following governance types are in place simultaneously for each integration.

Account limit
This value 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	**
2	10	10+10	10+20	**
1, 1+	15	15+10	15+20	15+100
0	20	20+10	20+20	20+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.
Each of the current concurrency limits applies to the combined total of SOAP and REST Web Services and RESTlet requests.

Integration limit
Part of the account limit can be allocated to a specific integration. This limit applies to total requests within such integration (Application ID).
The limit is optional and can be configured in integration record.

Account limit = sum (**Integration limits**) + **Unallocated limit**
Unallocated limit applies to all integrations with no limit having set.

Please note that legacy **User limit** applies only to SOAP requests of a user authenticating with deprecated methods (RLC or session based authentication) with endpoints older than 2020.2.

Four sample scenarios – how many concurrent requests can I have?

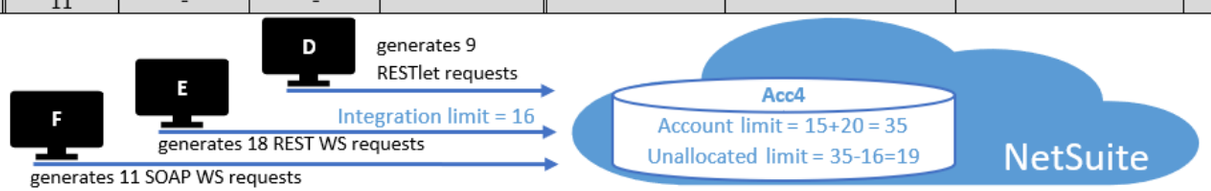
Three different clients/integrations are querying three different accounts with following amount of requests:

Scenario	Integration record		Snapshot of all incoming requests at one time				Account				Requests status	
No.	Integration	Integration limit	SOAP WS	REST WS	RESTlet	Total requests	Service Tier	SC+ License	Total Account Limit	Account ID	Success	Fail*
I	A	-	4	-	-	4	Shared	0	5	Acc1	4	0
II	B	-	-	12	4	16	Shared	1	15	Acc2	15	1
III	C	5			8	8	2	0	10	Acc3	5	3

Scenario IV in detail:

IV	D	-	-	-	9	38	1	2	35	Acc4	35	3
	E	16	-	18	-							
	F		11	-	-							

Three different integrations **D, E, F** send requests simultaneously to account **Acc4.**



Integration limit applies to E
Integration **E** has limit 16 requests. Any two can fail.

Unallocated limit applies to D, F
External applications send concurrently 9+11=20 requests in total.
One of these requests will be rejected.

* Please note: Governance framework can temporarily allow more concurrent requests.

Search Help Center/ SuiteAnswers for other "governance sample scenario"

ORACLE NETSUITE

Concurrency Governance Cheat Sheet

Recommended Actions

1. Periodically check available integration limit with the getIntegrationGovernanceInfo API call.
2. Handle the error codes in client application.
3. Implement retry logic
   a. Retry gradually increasing the delay if more attempts needed.
4. Analyse the frequency and level of concurrency peaks and consider rescheduling requests to be outside of regular peak times.
5. Consider increasing limit by more SC+ licenses.
6. Monitor trends in concurrency usage to prevent broken integrations (see Navigation table below).
7. Consider allocating part of the account limit for your integration.
8. Optionally call the getAccountGovernanceInfo API to get account and unallocated limits and consider changing integration limit.

Code example demonstrates basic handling of SOAP 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
REST WS	HTTP error code: 429 Too Many Requests	
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 Unallocated limit at that moment

* SOAP legacy authentication

NetSuite Navigation

What	Where
Account concurrency limit	Setup > Integration > Integration Management > Integration Governance
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 and REST WS requests that were rejected due to concurrency violation	Execution log on Integration record
Searchable details about SOAP requests	Setup > Integration > SOAP Web Services Usage Log
Web Services performance dashboard	Search for the Application Performance Management SuiteApp in Help Center/ SuiteAnswers
Concurrency Monitor dashboard, monthly/hourly overview (heatmap), charts showing concurrency usage with drill down possibility to seconds	
Scheduling ointegrations	
Decision tree – considering additional license, what is appropriate account concurrency limit	Search Help Center for " concurrency decision tree"

Concurrency Governance Cheat Sheet

The following governance types are in place simultaneously for each integration.

Account limit
This value 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	3 SC+	6 SC+	12 SC+
Standard	5 concurrent requests for account	5+1×10=15	-	-	-
Premium	15	15+10	15+30	-	-
Enterprise	20	20+10	20+30	20+60	-
Ultimate	20	20+10	20+30	20+60	20+120

*The base limit is increased by 10 for each SC+ license. The number of SC+ licences may vary from 1 to maximum allowed.
Each of the current concurrency limits applies to the combined total of SOAP and REST Web Services and RESTlet requests

Integration limit
Part of the account limit can be allocated to a specific integration. This limit applies to total requests within such integration (Application ID).
The limit is optional and can be configured in integration record.

Account limit = sum (**Integration limits**) + **Unallocated limit**
Unallocated limit applies to all integrations with no limit having set.

Please note that legacy **User limit** applies only to SOAP requests of a user authenticating with deprecated methods (RLC or session based authentication) with endpoints older than 2020.2.

Four sample scenarios – how many concurrent requests can I have?

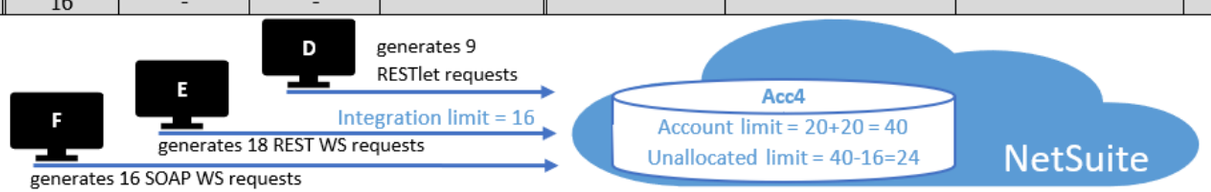
Three different clients/integrations are querying three different accounts with following amount of requests:

Scenario	Integration record		Snapshot of all incoming requests at one time				Account				Requests status	
No.	Integration	Integration limit	SOAP WS	REST WS	RESTlet	Total requests	Service Tier	SC+ License	Total Account Limit	Account ID	Success	Fail*
I	A	-	4	-	-	4	Standard	0	5	Acc1	4	0
II	B	-	-	12	4	16	Standard	1	15	Acc2	15	1
III	C	5			8	8	Premium	0	15	Acc3	5	3

Scenario IV in detail:

IV	D	-	-	-	9	43	Enterprise	2	40	Acc4	40	3
	E	16	-	18	-							
	F		16	-	-							

Three different integrations **D, E, F** send requests simultaneously to account **Acc4.**



Integration limit applies to E
Integration **E** has limit 16 requests. Any two can fail.

Unallocated limit applies to D, F
External applications send concurrently 9+16=25 requests in total.
One of these requests will be rejected.

* Please note: Governance framework can temporarily allow more concurrent requests.

Search Help Center/ SuiteAnswers for other "governance sample scenario"

Concurrency Governance Cheat Sheet

Recommended Actions

1. Periodically check available integration limit with the getIntegrationGovernanceInfo API call.
2. Handle the error codes in client application.
3. Implement retry logic
   a. Retry gradually increasing the delay if more attempts needed.
4. Analyse the frequency and level of concurrency peaks and consider rescheduling requests to be outside of regular peak times.
5. Consider increasing limit by more SC+ licenses.
6. Monitor trends in concurrency usage to prevent broken integrations (see Navigation table below).
7. Consider allocating part of the account limit for your integration.
8. Optionally call the getAccountGovernanceInfo API to get account and unallocated limits and consider changing integration limit.

Code example demonstrates basic handling of SOAP 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
REST WS	HTTP error code: 429 Too Many Requests	
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 Unallocated limit at that moment

* SOAP legacy authentication

NetSuite Navigation

What	Where
Account concurrency limit	Setup > Integration > Integration Management > Integration Governance
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 and REST WS requests that were rejected due to concurrency violation	Execution log on Integration record
Searchable details about SOAP requests	Setup > Integration > SOAP Web Services Usage Log
Web Services performance dashboard	Search for the Application Performance Management SuiteApp in Help Center/ SuiteAnswers
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	Search Help Center for " concurrency decision tree"

Appendix 3: Scripting with Multi-Level Joins using the N/query Module

With SuiteScript 2.x, you can load the N/search module to create and run on-demand or saved searches and analyze and iterate through the search results. However, one of the limitations of the N/search module is that you could not natively utilize more than a two-level record join. The N/query module lets you overcome this limitation by allowing you to create and run queries using the SuiteAnalytics Workbook query engine.

Using the N/query module, you can:

- Use multi-level joins to create queries using field data from multiple record types.
- Create conditions (filters) using AND, OR, and NOT logic, as well as formulas and relative dates.
- Sort query results based on the values of multiple columns.

The following tutorial shows you how to create a SuiteAnalytics Workbook in the UI that includes multi-level joins using the Customer, Employee, and Location records, and how to recreate a query with the same joins in SuiteScript using the N/query module.

Create a SuiteAnalytics Workbook

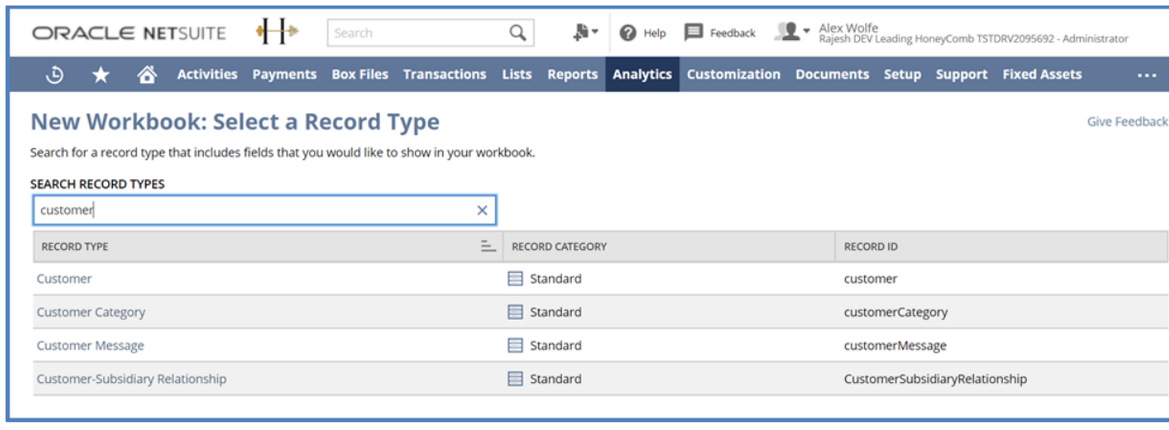
There are two steps involved in creating a SuiteAnalytics Workbook:

- Step 1: Select a Root Record Type
- Step 2: Select Your Source Data
- Step 3: Filter Your Source Data

Step 1: Select a Root Record Type

Every workbook needs a root record type to specify where the base data comes from. This step shows you how to select a root record type for your workbook.

1. Click the **Analytics** tab in the NetSuite navigation menu.
2. On the Workbook Listing Page, click **New Workbook**.
3. For the purposes of this tutorial, select the Customer record type.



By default, the Data Grid displays preselected fields based on the root record type selected for the workbook. Any fields that you add to the grid appear highlighted at the top of the Fields list.

Step 2: Select Your Source Data

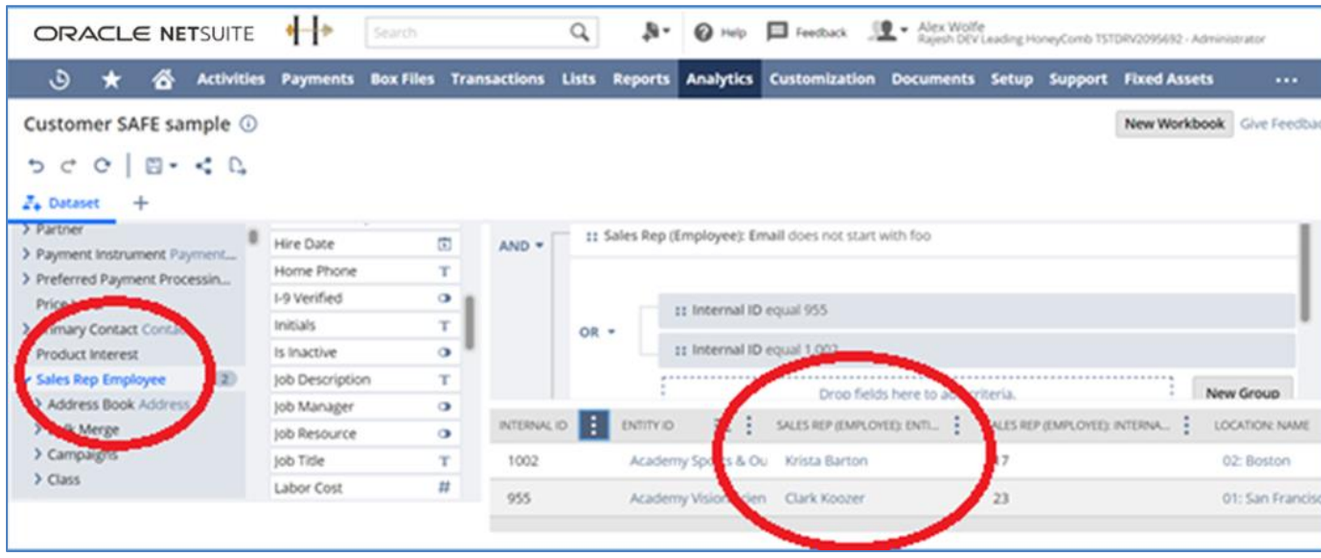
To select your source data, you add fields from the root record type to the Data Grid. You have three options:

- Drag the fields from the Fields list to the Data Grid
- Double-click the fields in the Fields list
- Type the name of the fields in the search bar at the top of the Fields list, then double-click or drag them to the Data Grid

By default, the Records list on the DataSet tab shows all related record types that you have access to in your account. Click the arrow next to any record type in the Records list to view additional related record types.

- Click the desired record type name to update the Fields list
- Double-click or drag the desired fields to the Data Grid

In this tutorial, you will be using the Sales Rep (Employee) record and the email field from this record.



Step 3: Filter Your Source Data

This step show you how to filter the source data for your workbook. Filters applied to the Data Grid from the Criteria Builder determine the values that are available for the workbook. For example, if you create a filter to exclude any Customers that have an internal ID in a certain range, data from those sales orders is not presented on the Pivot or Chart tabs.

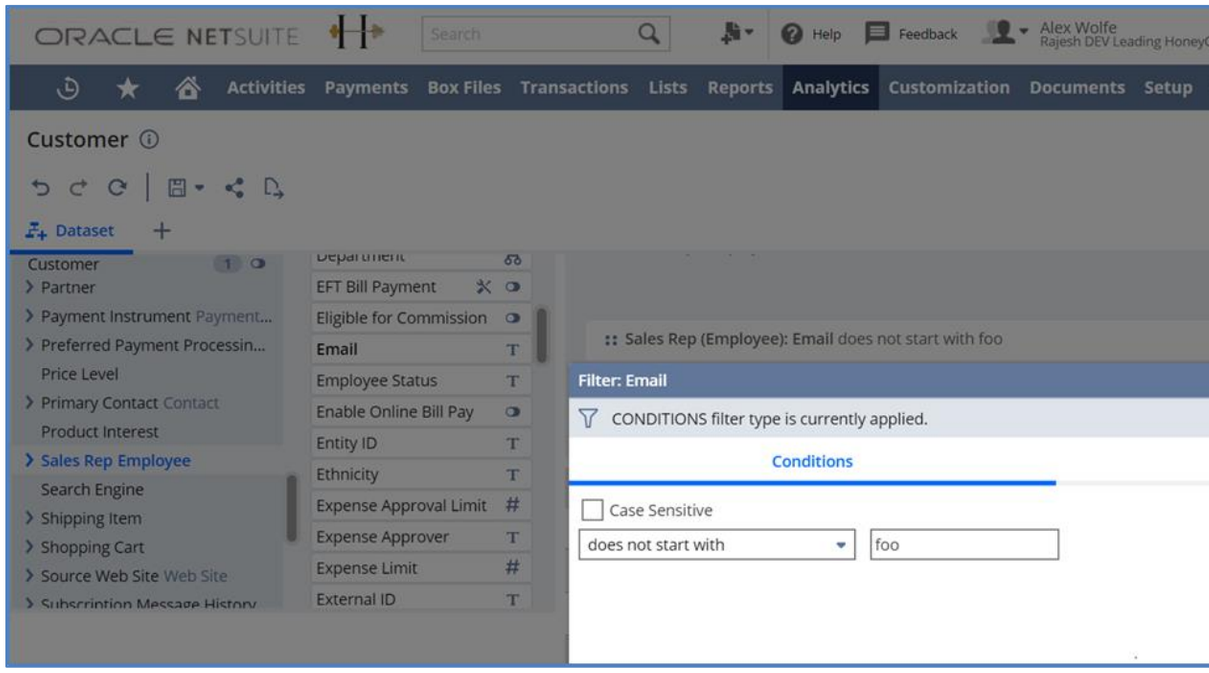
1. On the DataSet tab, drag the desired record field or formula field from the Fields list to the Criteria Builder above the Data Grid.

2. The Filter window appears. In the Filter window, select the filter conditions you want to apply to the field. Up to four options are available for filtering the data, depending on the type of field that is selected.

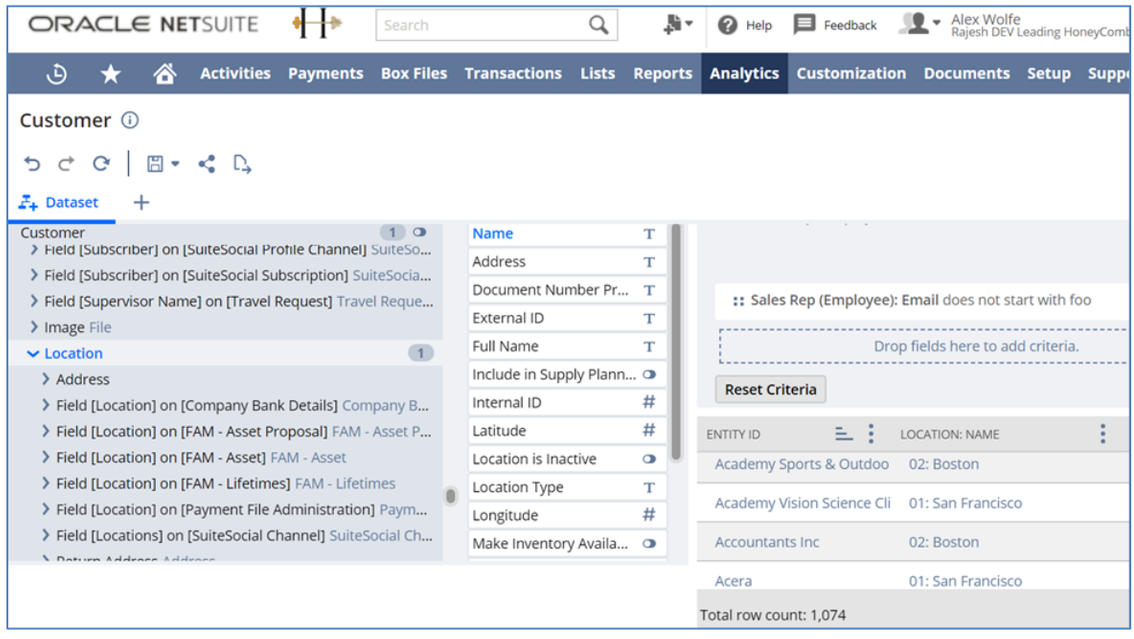
    - Values: Existing values or dates from the source data, or custom values
    - Ranges/Date Ranges: Range of values or dates available in the source data
    - Relative Conditions/Relative Dates: Conditions relative to the existing values in the source data

- Conditions/Specific Dates: Specific value or date and an expression

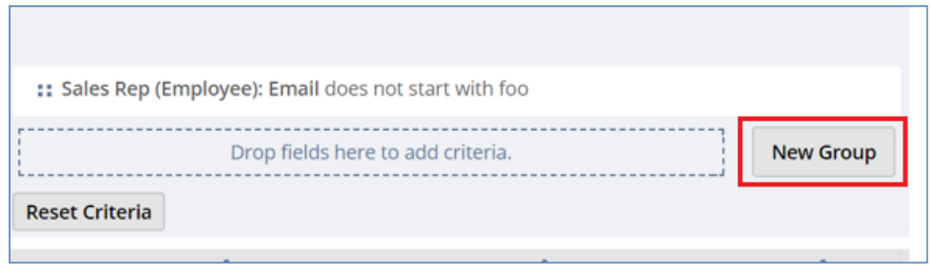
3. For this tutorial, select the Sales Rep (Employee) Record, then drag the Email Field into the filter section and select “Does not start with = ‘foo’”.



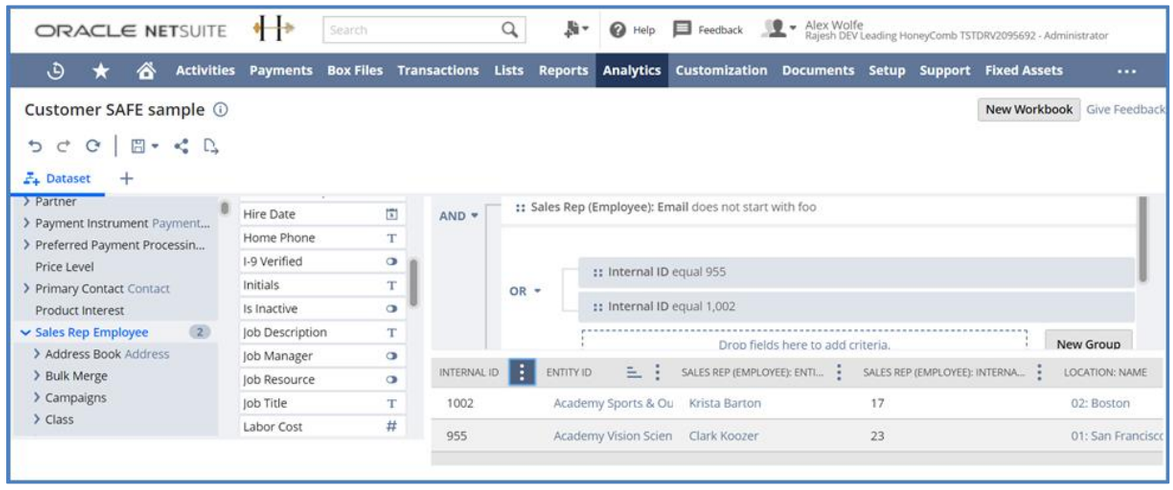
4. Select the Location Record and add the Name field to the Results.



5. Under the Filter section, add a New Group.



6. The new filter should match as show below.



Now that you have completed your multi-level join in the Workbook UI, you will use the N/query SuiteScript module to create and run a similar query.

Use N/query Module to Create Query with Joins

In this tutorial, you will use the following objects from the N/query module to create a query with joins:

- Query and Component Objects
- Condition Object
- Column Object
- Sort Object
- ResultSet and Result Objects

Query and Component Objects

To create a query with the N/query module:

1. Use the `query.create(options)` method to create your initial query definition (the `query.Query` object). The initial query definition uses one search type.

```
// Create a query definition for Customer records
const myCustomerQuery = query.create({
    type: query.Type.CUSTOMER
});
```

2. After you create the initial query definition, use `Query.autoJoin(options)` to create your first join.

The query definition always contains at least one `query.Component` object. Each new component is created as a child of the previous component, and all components exist as children of the query definition. You can think of a component as a building block; each new component builds on the previous component created. The last component created encapsulates the relationship between it and all of its parent components.

Condition Objects

A condition narrows the query results. The `query.Condition` object performs the same function as the `search.Filter` object in the N/search Module. The primary difference is that `query.Condition` objects can contain other `query.Condition` objects.

To create conditions:

- Use `Query.createCondition(options)` to create conditions for the initial query definition created with `query.create(options)`.
- Use `Component.createCondition(options)` to create conditions for the join relationships created with `Query.autoJoin(options)`, `Query.joinFrom(options)`/`Query.joinTo(options)`, `Component.autoJoin(options)`, or `Component.joinFrom(options)`/ `Component.joinTo(options)`.

```
// Create a join with the Sales rep (Employee) record
const mySalesRepJoin = myCustomerQuery.autoJoin({
    fieldId: 'salesrep'
});
```

- If you have multiple conditions, use `Query.and()`, `Query.or()`, and `Query.not()` to create a new nested condition.

```
myCustomerQuery.condition = myCustomerQuery.and(
    thirdCondition, myCustomerQuery.or(firstCondition, secondCondition));
```

Column Object

The `query.Column` object is the equivalent of the `search.Column` object in the N/search Module. The `query.Column` object describes the field types (columns) that are displayed from the query results.

To create columns:

- Use `Query.createColumn(options)` to create a column on the initial query definition created with `query.create(options)`.

```
// Create query columns
myCustomerQuery.columns = [
    myCustomerQuery.createColumn({
        fieldId: 'entityid'
    }),
    myCustomerQuery.createColumn({
        fieldId: 'id'
    })
]
```

Sort Object

The `query.Sort` object describes how query results are sorted (for example, ascending or descending, case sensitive or case insensitive, and so on).

To create a sort:

- Use `Query.createSort(options)` to create a sort on the initial query definition created with `query.create(options)`.

```
//Sort the query results based on query columns
myCustomerQuery.sort = [
    myCustomerQuery.createSort({
        column: myCustomerQuery.columns[3]
    }),
    myCustomerQuery.createSort({
        column: myCustomerQuery.columns[0],
        ascending: false
    })
]
```

ResultSet and Result Objects

When you are ready to execute your query. call `Query.run()`. This method returns a `query.ResultSet` object, which encapsulates the metadata for the set of results returned by the query.

```
//Run the query
const resultSet = myCustomerQuery.run();
```

The following sample creates a query for Custom records, joins the query with another query type, and runs the query as a paged query:

```
/**
 * @NApiVersion 2.1
 */
require (['N/query'], function(query) {
    // Create a query definition for Customer records
    const myCustomerQuery = query.create({
        type: query.Type.CUSTOMER
    });
    // Join the original query definition based on the salesrep field of a Customer
record.
    // In a Customer record, the salesrep field contains a reference to an Employee
record.
    // When you join based on this field, you are joining the query definition with
the
    // Employee query type, which allows you to access the fields of the joined
Employee
    // record in your query.
    const mysalesRepJoin = myCustomerQuery.autoJoin({
        fieldId: 'salesrep'
    });
```

```
    // Join the joined query definition based on the location field of an Employee
record.
    // In an Employee record, the location field contains a reference to a Location
record.
    const myLocationJoin = mySalesRepJoin.autoJoin({
        fieldId: 'location'
    });
    // Create conditions for the query
    // This condition checks for an id = 1022
    const firstCondition = myCustomerQuery.createCondition({
        fieldId: 'id',
        operator: query.Operator.EQUAL,
        values: 1022
    });
    // This condition checks for an id = 955
    const secondCondition = myCustomerQuery.createCondition({
        fieldId: 'id',
        operator: query.Operator.EQUAL,
        values: 955
    });
    // This condition checks for an email field that does not start with 'foo'
    const thirdCondition = myCustomerQuery.createCondition({
        fieldId: 'email',
        operator: query.Operator.START_WITH_NOT,
        values: 'foo'
    });
    // Combine conditions using Query.and() and Query.or() operator methods. In this
    // sample, the combined condition states that the id field of the Custom record
must
    // have a value of either 1022 or 955, and the email field of the Employee record
(the
    // record that is referenced in the salesrep field of the Customer record) must
not
    // start with 'foo'.
    myCustomerQuery.condition = myCustomerQuery.and(thirdCondition,
        myCustomerQuery.or(firstCondition, secondCondition));
    // Create the query columns
    myCustomerQuery.columns= [
        myCustomerQuery.createColumn({
            fieldId: 'entityid'
        }),
        myCustomerQuery.createColumn({
            fieldId: 'id'
```

```
        }),
        mySalesRepJoin.createColumn({
            fieldId: 'entityid'
        }),
        mySalesRepJoin.createColumn({
            fieldId: 'email'
        }),
        mySalesRepJoin.createColumn({
            fieldId: 'hiredate'
        }),
        myLocationJoin.createColumn({
            fieldId: 'name'
        })
    ];
    // Sort the query results based on the query columns created above
    myCustomerQuery.sort = [
        myCustomerQuery.createSort({
            column: myCustomerQuery.columns[3]
        }),
        myCustomerQuery.createSort({
            column: myCustomerQuery.columns[0],
            ascending: false
        })
    ];
    // Run the query
    const resultSet = myCustomerQuery.run();
    // Retrieve and log the query results
    const results = resultSet.results;
    for (let i = results.length - 1; i >= 0; i--) {
        log.debug(results[i].values);
    }
    log.debug(resultSet.types);
}
```

When you run the script, you will see the same result as the SuiteAnalytics Workbook in the UI.

Appendix 4: N/Cache Full Code Sample

This appendix provides the full sample code of the example referenced in section 3.1.6 Use a Cache.

```
/**
 * @NApiVersion 2.1
 * @NScriptType Suitelet
 */
define(function (require, exports) {
    const url = require('N/url');
    const query = require('N/query');
    const cache = require('N/cache');
    const format = require('N/format');
    const record = require('N/record');
    const runtime = require('N/runtime');
    const message = require('N/ui/message');
    const widget = require('N/ui/serverWidget');

    /**
     *@param  {Object} context.request
     *@param  {Object} context.response
     */
    exports.onRequest = function (context) {
        if (context.request.method === 'GET' || context.request.method === 'POST') {
            const currentUser = runtime.getCurrentUser();
            const form = widget.createForm({
                title: 'Approve Journal Entries'
            });
            form.addField({
                id: 'custpage_user',
                type: widget.FieldType.TEXT,
                label: 'Journal Approver'
            }).updateDisplayType({
                displayType: widget.FieldDisplayType.INLINE
            }).defaultValue = currentUser.name
            const jeApprovalPerm = currentUser.getPermission({
                name: 'TRAN_JOURNALAPPRV'
            });
            if (jeApprovalPerm === 0) {
                form.addPageInitMessage({
                    type: message.Type.ERROR,
                    message: `You do not have permissions to approve journal entries.
                              Please contact your NetSuite Administrator.`
                });
```

```
                return context.response.writePage(form);
            }
            let params = context.request.parameters;
            if (context.request.method === 'GET') {
                setSearchJournalsForm(form, params);
            } else {
                if (params.custpage_search) {
                    setJournalsPendingApprovalForm(form, params);
                } else {
                    setJournalSubmissionForm(context.request, form, params);
                }
            }
            context.response.writePage(form);
        }
    };

    /**
     *@param  {Object} form
     *@param  {Object} params
     */
    const setSearchJournalsForm = (form, params) => {
        if (params.qr) {
            purgeCache('JournalApprovalCache', params.qr);
        }
        if (params.expired === 'T') {
            form.addPageInitMessage({
                type: message.Type.WARNING,
                message: `The maximum allowed time for your previous approval
submission was reached. <br/>
                Select a Subsidiary and click the Search button to view the list of
pending approval journals`
            });
        } else {
            form.addPageInitMessage({
                type: message.Type.INFORMATION,
                message: 'Select a subsidiary and click the search button to view the
list of journals'
            });
        }
        form.addSubmitButton({
            label: 'Search'
        });
        const subsidiaryFld = form.addField({
            id: 'custpage_subsidiary',
```

```
            type: widget.FieldType.SELECT,
            label: 'Subsidiary',
            source: 'subsidiary'
        });
        subsidiaryFld.defaultValue = params.qr ? params.qr :
runtime.getCurrentUser().subsidiary;
        subsidiaryFld.isMandatory = true;
        form.addField({
            id: 'custpage_search',
            type: widget.FieldType.CHECKBOX,
            label: 'Search Journals'
        }).updateDisplayType({
            displayType: widget.FieldDisplayType.HIDDEN
        }).defaultValue = 'T';
    };

    /**
     *@param  {Object} form
     *@param  {Object} params
     */
    const setJournalsPendingApprovalForm = (form, params) => {
        //Get the end time of the allowed submission time (5 min), which corresponds to
the cache
        //ttl value
        const endDateTime = getEndDateTime();
        const subsidiaryId = params.custpage_subsidiary;

        //Get cache, if it does not exist it will be automatically created
        const appCache = cache.getCache({
            name: 'JournalApprovalCache',
            scope: cache.Scope.PROTECTED,
        });

        //Get journals for the subsidiary selected by the user that are currently in
the cache
        const cacheValues = getCacheValues(appCache, subsidiaryId);

        //Get journals available for the subsidiary selected by the user, excluding the
ones in
        //the cache
        const journals = getPendingApprovalJournals(subsidiaryId, cacheValues);
        if (journals.length > 0) {
            //Add current user journal entries into cache
            updateCacheValues(appCache, subsidiaryId,
cacheValues.concat(journals.map((a) => a.id)));
```

```
            form.addPageInitMessage({
                type: message.Type.INFORMATION,
                message: `Select the journals that you want to approve by ticking the
select box.
                Only 25 journal entries can be approved at a time.<br/>
                You have until <b>${endDateTime}</b> to complete your submission. If
you exceed the maximum allowed time on the page,
                the transactions you are holding will be released to other users.`
            });
            form.addSubmitButton({
                label: 'Submit'
            });
            setCancelButton(form, subsidiaryId, true);
            addRecordSublist(form, 'Pending Approval', journals, 'transaction', false);
        } else {
            form.addPageInitMessage({
                type: message.Type.ERROR,
                message: `There are no journals to approve based on the criteria set in
your journal approval workflow(s)`
            });
            setCancelButton(form, '', false);
        }
        form.addField({
            id: 'custpage_subsidiary',
            type: widget.FieldType.SELECT,
            label: 'Subsidiary',
            source: 'subsidiary'
        }).updateDisplayType({
            displayType: widget.FieldDisplayType.HIDDEN
        }).defaultValue = params.custpage_subsidiary;
    };

    /**
     *@param  {Object} request
     *@param  {Object} form
     *@param  {Object} params
     */
    const setJournalSubmissionForm = (request, form, params) => {
        let errorCount = 0;
        let notProcessed = [];
        const subsidiaryId = params.custpage_subsidiary;
        //Get journals that were selected for approval by the user
        let journals = getSublistValues(request, 'custpage_transaction', ['select',
'id', 'intercompany'], 'custpage_')
```

```
            .filter((a) => {
                if (a.select === 'T') {
                    return true;
                }
                notProcessed.push(a.id);
                return false;
            }).map((b) => {
                return {
                    id: b.id,
                    type: b.intercompany === 'Yes' ? 'advintercompanyjournalentry' :
'journalentry',
                    values: {approvalstatus: 2}
                };
            });
        journals.forEach((journal) => {
            try {
                record.submitFields(journal);
                journal.message = 'Approved';
            } catch (e) {
                journal.message = e.name + ' ' + e.message;
                notProcessed.push(journal.id);
                errorCount += 1;
            }
        });

        //Get cache, if it does not exist it will be automatically created
        const appCache = cache.getCache({
            name: 'JournalApprovalCache',
            scope: cache.Scope.PROTECTED,
        });

        //Get journals for the subsidiary selected by the user that are currently in
the cache
        const cacheValues = getCacheValues(appCache, subsidiaryId);

        //Remove journal entries that failed to be processed from the cache
        if (notProcessed.length > 0) {
            updateCacheValues(appCache, subsidiaryId, cacheValues.filter((a) =>
!notProcessed.includes(a)));
        }
        if (errorCount > 0) {
            form.addPageInitMessage({
                type: message.Type.ERROR,
```

```
                message: `Journal entry approval process was completed with errors.
Please review the messages below.`
            });
            setCancelButton(form, subsidiaryId, false);
        } else {
            form.addPageInitMessage({
                type: message.Type.CONFIRMATION,
                message: `Journal entries were successfully approved.`
            });
            setGoBackToMainButton(form, subsidiaryId);
        }
        addRecordSublist(form, 'Processed', journals.map((a) => {
            delete a.type;
            delete a.values;
            return a;
        }), 'transaction', true);
    };

    /***
     *@param  {Object} form
     *@param  {String} subsidiaryId
     *@param  {Boolean} addTimer
     */
    function setGoBackToMainButton(form, subsidiaryId) {
        const scriptObj = runtime.getCurrentScript();
        form.addButton({
            id: 'custpage_btn_goback',
            label: 'Go Back',
            functionName: "(function(){window.location.href='" + url.resolveScript({
                scriptId: scriptObj.id,
                deploymentId: scriptObj.deploymentId,
                returnExternalUrl: false
            }) + "'; return false})();"
        });
    };
    /***
     *@param  {Object} form
     *@param  {String} subsidiaryId
     *@param  {Boolean} addTimer
     */
    function setCancelButton(form, subsidiaryId, addTimer) {
        const scriptObj = runtime.getCurrentScript();
        let suiteletUrl = url.resolveScript({
            scriptId: scriptObj.id,
```

```
            deploymentId: scriptObj.deploymentId,
            returnExternalUrl: false
        }) + "&qr=" + subsidiaryId;
        form.addButton({
            id: 'custpage_btn_cancel',
            label: addTimer ? 'Cancel' : 'Go Back',
            functionName: "(function(){window.location.href='" + suiteletUrl + "';
return false})();"
        });
        if (addTimer) {
            suiteletUrl += "&expired=T";
            form.addField({
                id: 'custpage_auto_refresh',
                type: widget.FieldType.INLINEHTML,
                label: '_'
            }).updateDisplayType({
                displayType: widget.FieldDisplayType.INLINE
            }).defaultValue = `<html><script>
                    (function () {
                        window.setTimeout(function () {
                        window.location.href='${suiteletUrl}';
                    }, 300000);
                    })();
                </script></html>`;
        }
    }

    /***
     *@param  {Object} form
     *@param  {String} label
     *@param  {Array} values
     *@param  {Array} recordType
     *@param  {Boolean} readOnly
     */
    const addRecordSublist = (form, label, values, recordType, readOnly) => {
        const columns = Object.keys(values[0]);
        const sublist = form.addSublist({
            id: 'custpage_' + recordType,
            type: widget.SublistType.LIST,
            label: label
        });
        if (!readOnly) {
            sublist.addMarkAllButtons();
            sublist.addField({
```

```
                id: 'custpage_select',
                type: widget.FieldType.CHECKBOX,
                label: 'Select'
            });
        }
        columns.forEach((id) => {
            if (id === 'id') {
                sublist.addField({
                    id: 'custpage_' + id,
                    type: widget.FieldType.SELECT,
                    label: recordType,
                    source: recordType
                }).updateDisplayType({
                    displayType: widget.FieldDisplayType.INLINE
                });
            } else {
                sublist.addField({
                    id: 'custpage_' + id,
                    type: widget.FieldType[id === 'total' ? 'CURRENCY' : 'TEXT'],
                    label: id
                });
            }
        });
        values.forEach((a, k) => {
            columns.forEach((b) => {
                if (a[b] !== undefined && a[b] !== null && a[b] !== '') {
                    sublist.setSublistValue({
                        id: 'custpage_' + b.toLowerCase(),
                        line: k,
                        value: a[b]
                    });
                }
            });
        });

    };

    /***
     * @param  {Object} form
     * @param  {String} sublistId
     * @param  {Object} fields
     * @param  {String} prefix
     * @return {Array}  lines
     */
```

```
    const getSublistValues = function (request, sublistId, fields, prefix) {
        let lines = [];
        const lnc = parseInt(request.getLineCount(sublistId));
        if (lnc === 0) {
            return lines;
        }
        for (let k = 0; k < lnc; k++) {
            let line = {};
            if (Array.isArray(fields)) {
                for (let m = 0; m < fields.length; m++) {
                    line[fields[m]] = request.getSublistValue({
                        group: sublistId,
                        name: prefix ? prefix + fields[m] : fields[m],
                        line: k
                    });
                }
            } else {
                for (let id in fields) {
                    if (fields.hasOwnProperty(id)) {
                        line[id] = request.getSublistValue({
                            group: sublistId,
                            name: prefix ? prefix + fields[id] : fields[id],
                            line: k
                        });
                    }
                }
            }
            lines.push(line);
        }
        return lines;
    };

    /**
     *@return  {String} endDateTime
     */
    const getEndDateTime = () => {
        const userTMZ = runtime.getCurrentUser().getPreference('TIMEZONE');
        const tmzEnum = Object.entries(format.Timezone).filter((a) => a[1] ===
userTMZ)[0][0];
        const endTimeStr = format.format({
            value: new Date(format.parse({
                value: new Date(),
                type: format.Type.DATETIME,
                timezone: format.Timezone[tmzEnum]
```

```
            }).getTime() + 5 * 60000),
            type: format.Type.DATETIME,
            timezone: format.Timezone[tmzEnum]
        });
        return endTimeStr.split(' ').map((a, i) => {
            if (i === 0)
                return '';
            const s = a.split(':');
            return s.length <= 1 ? a : s[0] + ':' + s[1];
        }).join('');
    };

    /**
     *@param  {Object} appCache
     *@param  {Object} cacheKey
     *@return {Array} cacheValues
     */
    const getCacheValues = (appCache, cacheKey) => {
        const cacheValues = appCache.get({
            key: cacheKey
        });
        return cacheValues ? cacheValues.split(',') : [];
    };

    /**
     *@param  {Object} appCache
     *@param  {String} cacheKey
     *@param  {Array} cacheValues
     */
    const updateCacheValues = (appCache, cacheKey, cacheValues) => {
        const uniqueValues = cacheValues.map((d) =>
            parseFloat(d)).reduce((u, b) => u.includes(b) ? u : [...u, b],
[]).join(',');
        if (uniqueValues.length > 0) {
            appCache.put({
                key: cacheKey,
                value: uniqueValues,
                ttl: 300
            });
        } else {
            purgeCache(appCache.name, cacheKey);
        }
    };
```

```
    /**
     *@param  {String} cacheName
     *@param  {String} cacheKey
     */
    const purgeCache = (cacheName, cacheKey) => {
        cache.getCache({
            name: cacheName,
            scope: cache.Scope.PROTECTED
        }).remove({
            key: cacheKey
        });
    };

    /**
     *@param   {String} subsidiaryId
     *@param   {Array} cacheIds
     *@return  {Array} res
     */
    const getPendingApprovalJournals = (subsidiaryId, cacheIds) => {
        const idsFilter = cacheIds.length > 0 ? ` AND (Transaction.ID NOT IN(` +
cacheIds.map((id) => {
            return `'${id}'`;
        }) + `))` : '';
        return query.runSuiteQL({
            query: `
              SELECT DISTINCT
                 Transaction.Id AS id,
                 MAX(Transaction.TranDate) AS date,
                 MAX(BUILTIN.DF( Transaction.PostingPeriod )) AS period,
                 BUILTIN.DF(TransactionLine.Subsidiary) AS subsidiary,
                 MAX(BUILTIN.DF( Transaction.Currency )) AS currency,
                 SUM(ABS(NVL(TransactionLine.debitforeignamount,0))) as total,
                 MAX(BUILTIN.DF( Transaction.CreatedBy )) AS createdby,
                 MAX(CASE WHEN NVL(TransactionLine.Eliminate, 'T') = 'T' THEN 'Yes'
ELSE 'No' END) AS intercompany
             FROM
                Transaction
                  INNER JOIN TransactionAccountingLine ON
                  ( TransactionAccountingLine.Transaction = Transaction.ID )
                  INNER JOIN TransactionLine ON
                  ( TransactionLine.Transaction = Transaction.ID )
            WHERE
                 ( Transaction.Type = 'Journal' )
                 AND (TransactionLine.Subsidiary = ? )
```

```
                AND ( BUILTIN.DF(Transaction.ApprovalStatus) = 'Pending Approval' )
                AND NVL( TransactionAccountingLine.Debit,0 ) <> 0
                AND (( TransactionAccountingLine.Debit IS NOT NULL )
                OR ( TransactionAccountingLine.Credit IS NOT NULL ))
                ${idsFilter}
            GROUP BY
               Transaction.Id,
               BUILTIN.DF(TransactionLine.Subsidiary)
            ORDER BY
               MAX(Transaction.TranDate) DESC
            `,
            params: [subsidiaryId]
        }).asMappedResults().slice(0,25);
    };
});
```

Appendix 5: Announcements/Advisory

Starting with version 2024.2, the Advanced Revenue Recognition Management plugin is unsupported.