

Basic Searching with SuiteScript 2.1

written by [Eric T Grubaugh](#)

part of the ["SuiteScript by Example" ↗](#) series

published by [Stoic Software, LLC](#)

Basic Searching with SuiteScript 2.1

by [Eric T Grubaugh](#)

Copyright (c) 2017- [Stoic Software, LLC](#). All rights reserved.

Published by Stoic Software, LLC, PO Box 129, Wellington, CO 80549.

NetSuite and SuiteScript are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Neither the author nor the publisher have any affiliation with Oracle Corporation or NetSuite, Inc. This product is neither endorsed nor sponsored by Oracle Corporation or NetSuite, Inc.

Using Code Samples

This book is here to help you learn. In general, you may use the code presented herein in your own code. You do not need to contact me unless you are reproducing or redistributing large portions of the code.

I appreciate, but do not require, attribution. An attribution usually includes the title, author, and publisher:

*"Basic Searching with SuiteScript 2.1, by Eric T Grubaugh (Stoic Software, LLC).
Copyright 2017 Stoic Software, LLC."*

Introduction

This SuiteScript cookbook is intended to provide you with simple, practical examples of performing basic searches with the SuiteScript API.

In *Basic Searching with SuiteScript 2.1*, you'll see examples of:

- The fundamentals of creating and executing a Search with SuiteScript
- The various methods of iterating over and processing search results
- How to load and execute a Saved Search in SuiteScript
- How to specify Joins to related records in your search filters and columns
- How to use Summaries like summing in your search results
- How to create Filter Expressions for your searches
- How to get the number of results for a particular search

Conventions in this Book

All code examples in this book use the ``require`` function for defining modules. This allows you to copy and paste the snippets directly into the debugger or your browser's developer console and run them.

The ``N/search`` module is always imported as ``s``.

``console.log`` is used for writing output to the browser console. If desired, you can replace these with calls to the [`N/log` module ↗](#) for writing to the Execution Log for the debugger.

The Anatomy of a SuiteScript Search

In our first example, we create a Customer Search that retrieves all Customers located within the state of California:

```
/**
 * Creates and executes a Customer search that finds all Customers located
 * within California.
 *
 * Uses the most verbose syntax for specifying filters and columns.
 *
 * Uses the `each` method for iteration through search results.
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      {
        name: 'state',
        operator: s.Operator.ANYOF,
        values: ['CA']
      }
    ],
    columns: [
      {
        name: 'entityid'
      }, {
        name: 'email'
      }
    ]
  })

  const printCustomerNameAndEmail = (result) => {
    const customerName = result.getValue({ name: 'entityid' })
    const email = result.getValue({ name: 'email' })

    console.log(`${customerName} - ${email}`)

    return true
  }

  customerSearch.run().each(printCustomerNameAndEmail)
})
```

All searching functionality in SuiteScript is provided by the ``N/search`` module.

```
require(['N/search'], (s) => {
```

The basic formula for searching with the SuiteScript API goes like this:

1. *Create* a new ``Search`` instance OR *load* an existing Saved Search
2. Specify the Record Type, Filters, and Columns of our Search
3. Execute the Search
4. Retrieve the Results of the Search
5. Process the Results

Creation of a search with ``s.create``

Here we import the ``N/search`` module as ``s`` and use its ``create`` method to accomplish steps 1 and 2 of our basic formula. We create a Search instance by specifying its Record Type, Filters, and Columns.

We start by defining the Search's Record Type with the ``type`` property of ``create``. We provide it a value using the ``N/search`` module's ``Type`` enumeration for native records. For custom records, we instead enter the Custom Record's ID as a literal String (e.g. ``type: 'customrecord_my_rec'``).

```
const customerSearch = s.create({  
  type: s.Type.CUSTOMER,  
  // ...
```

For a list of all possible values for the ``Type`` enumeration, see the Help page [search.Type](#).

Specifying Filters

Next, we provide our Search's Filters (also called "Criteria" in the UI) using the ``filters`` property. We will explore a couple different ways to specify Filters. The first way that you see here is by creating an Array of Objects:

```
// ...
filters: [
  {
    name: 'state',
    operator: s.Operator.ANYOF,
    values: ['CA']
  }
],
// ...
```

Each element of the Array defines one Filter by providing:

- a `name` to identify the field to be filtered
- an `operator` for how the field will be compared
- the `values` to compare the field to

We could optionally include in each Filter:

- a `join` to filter on fields from related records
- a `summary` to filter on summarized values like `COUNT`'s or `SUM`'s

If you provide multiple Filters in this Array, they will all have a logical `AND` relationship, so a Record must match *all* of your Filters to be included in the results. With this syntax, there is no way to specify an `OR` relationship for Filters. Later, we'll explore a different syntax that does enable `OR` relationships.

Specifying Columns

To finish off the creation of our `Search` instance, we specify the Columns (also called "Results" in the UI) that will be included in the results. We specify Columns using the `columns` property of `create`.

```
// ...
columns: [
  {
    name: 'entityid'
  }, {
    name: 'email'
  }
]
// ...
```

Each element of the `Array` defines one `Column` by providing:

- a ``name`` to identify the field to be included in the results

We could optionally include in each Column:

- a ``join`` to retrieve fields from related records
- a ``summary`` to summarize that field with say a ``COUNT`` or ``SUM``

Notice we also specify the ``summary`` with an enumeration. To see the possible Summary types, see the Help page [search.Summary.2](#).

Executing the Search

Creating the ``Search`` instance is not enough to actually *execute* the Search. In order to do that, we need to invoke the ``run`` method on our ``Search`` instance.

```
// ...
customerSearch.run()
// ...
```

This will execute our Search on the NetSuite server and save the results there for when our script is ready to retrieve them, but it *still* doesn't actually give us the search results. There is one more step to actually process the results.

Iterating with ``each``

In this case, we want to retrieve our Search's results immediately, so we can directly chain our ``run`` call with a call to the ``each`` iterator. We use ``each`` to process results one at a time.

We define a function ``printCustomerNameAndEmail`` that contains the logic for processing a single result. As the name implies, all we want to do is print each Customer's Name and Email to the browser console.

```
// ...
const printCustomerNameAndEmail = (result) => {
  const customerName = result.getValue({ name: 'entityid' })
  const email = result.getValue({ name: 'email' })

  console.log(`${customerName} - ${email}`)

  return true
}

customerSearch.run().each(printCustomerNameAndEmail)
// ...
```

The callback function for `each` *must* return a Boolean value:

- `true` to continue iterating to the next result
- `false` to stop iterating

*“⚠ Returning nothing is the same as returning `false` and will **stop** iteration. ⚠”*

We can use this behaviour to conditionally stop processing our results once certain conditions are met.

If you are trying to process results with `each`, but only see one result getting processed, it is very likely you forgot the `return` statement in your callback function.

Note that using `run()` and `each` will only iterate through, at most, `4,000` results.

Reading Result Data with `getValue`

As the `each` method iterates over our Search Results, it passes them individually into the callback function we specified, `printCustomerNameAndEmail`. The parameter passed in is an instance of `N/search.Result`, which has a `getValue` method for reading the value from a particular `Column`:

```
const customerName = result.getValue({ name: 'entityid' })
const email = result.getValue({ name: 'email' })
```

For Select fields (dropdowns), `getValue` will always return the internal ID of the selected record. If instead, you want to display the text displayed in the Select field, you can use `getText`.

For example, if we were to add the ``salesrep`` Column to our Search, we would retrieve the Sales Rep's ID and Name values this way:

```
const salesRepId = result.getValue({ name: 'salesrep' })
const salesRepName = result.getText({ name: 'salesrep' })

console.log(`Sales Rep ID: ${salesRepId}`) // Sales Rep ID: 45
console.log(`Sales Rep Name: ${salesRepName}`) // Sales Rep Name: Eric Grubaugh
```

A More Concise Search

As we saw in the first example, our Search Filters and Columns can be created as Arrays of Objects, but this gets really verbose for searches with multiple Filters and Columns. For simple searches, there is a much more concise way of specifying our Filters and Columns:

```
/**
 * Creates and executes a Customer search that finds all Customers located
 * within California.
 *
 * Uses minimal syntax for specifying filters and columns.
 *
 * Uses the `each` method for iteration through search results.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['state', s.Operator.ANYOF, ['CA']]
    ],
    columns: ['entityid', 'email']
  })

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  customerSearch.run().each(printCustomerName)
})
```

Functionally, this is an identical search to that of the first chapter, but we've saved ourselves a bit of typing.

Filter Expressions

We condense our Filters down by using what NetSuite calls a *Filter Expression* instead of our previous Array of Objects:

```
// ...
filters: [
  ['state', s.Operator.ANYOF, ['CA']]
],
// ...
```

The Filter Expression includes exactly the same data as before:

- a ``name`` to identify the field to be filtered
- an ``operator`` for how the field will be compared
- the ``values`` to compare the field to

Instead of being an Array of Objects, we now have an Array of Arrays. It may look a bit confusing at first, but once you are accustomed to it, I feel it is a very compact, readable expression of Filters.

The remainder of the examples in this cookbook will utilize the Filter Expression syntax.

Columns by Name Only

We also condense our Columns down to string literals. For any Column that doesn't involve a Join or a Summary, we can specify the name of the column as a string literal:

```
// ...
columns: ['entityid', 'email']
// ...
```

Note that nothing else about how we execute the search, iterate through results, or retrieve the values of our columns has changed.

How Many Results Does My Search Have?

Because `run().getRange()` is limited to `1,000` results and `run().each()` is limited to `4,000` results, we cannot rely on either method to return *all* results that match our criteria.

For the most convenient and accurate way to count the number of results that a particular Search will return, we need to execute our search using the Search module's Paging API. The Paging API is primarily intended to page through large result sets, but it also contains the quickest method of getting a total result count using its `count` property.

```
/**
 * Counts all Customers located within California.
 *
 * Uses `runPaged` to execute the search instead of `run`.
 *
 * Uses the `count` property of a `PagedData` instance to get its total result count.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], function (s) {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['state', s.Operator.ANYOF, ['CA']]
    ]
  })

  const customerCount = customerSearch.runPaged().count
  console.log(`# Customers in CA = ${customerCount}`)
})
```

We create our `Search` instance with the `create` method just as we did before. This time, instead of executing our search using the `run()` method, we use the `runPaged()` method, which returns a `PagedData` instance as opposed to a `ResultSet` instance.

```
// ...
const customerCount = customerSearch.runPaged().count
console.log(`# Customers in CA = ${customerCount}`)
// ...
```

The `PagedData` object has a `count` property which contains the total number of results that match the search criteria.

We'll investigate the Paging API and large datasets later in our exploration of the Search module.

AND Relationships in Filter Expressions

So far we've only seen how to specify a single Filter in our Filter Expression. Let's make our previous example a little more generic. We'll create a function that searches for Customers by a given State and Sales Rep that we provide:

```
/**
 * Creates and executes a Customer search that counts all Customers located
 * within California for a specific Sales Rep.
 *
 * Shows how to specify AND relationship between multiple criteria in Filter Expression
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], (s) => {
  const findCustomersByStateAndRep = (state, salesRepId) =>
    s.create({
      type: s.Type.CUSTOMER,
      filters: [
        ['state', s.Operator.ANYOF, [state]], 'and',
        ['salesrep', s.Operator.ANYOF, salesRepId]
      ],
      columns: ['entityid', 'email']
    }).run()

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  // Replace "45" with the internal ID of any Sales Rep ID in your account
  findCustomersByStateAndRep('CA', '45').each(printCustomerName)
})
```

We have moved our Search creation inside a function that accepts parameters for the State and Sales Rep:

```
const findCustomersByStateAndRep = (state, salesRepId) =>
  s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['state', s.Operator.ANYOF, [state]], 'and',
      ['salesrep', s.Operator.ANYOF, salesRepId]
    ],
    columns: ['entityid', 'email']
  }).run()
```

The function creates and immediately runs the search, returning the `ResultSet` instance created by `run()`.

This pattern of creating the Search instance and returning it from a function named `find*` is a convention I use to isolate my search logic. I find that this makes it easier to comprehend, modify, and reuse my Searches.

Focusing in on our Filter Expression, you can see that we've added a second filter and placed an `'and'` between them:

```
// ...
filters: [
  ['state', s.Operator.ANYOF, [state]], 'and',
  ['salesrep', s.Operator.ANYOF, salesRepId]
],
// ...
```

This shows us a more general pattern of Filter Expressions:

```
[
  filter1,
  logicalOperator1,
  filter2,
  logicalOperator2,
  filter3,
  ...
]
```

Where `filterN` is in the format `[fieldName, operator, values]`, and `logicalOperatorN` is one of `'and'` or `'or'`, depending on the logical relationship between the filters.

The UI equivalent of SuiteScript's Filter Expressions is checking the *Use Expressions* box in a Saved Search's Criteria tab.

`OR` Relationships and Grouping Criteria

In addition to **`AND`** relationships between our Filters, we can also have **`OR`** relationships between them. Beyond that, we can build complex Filters by logically grouping Filters together.

Let's create a Search with a more complex Filter structure of **`A AND (B OR C)`**:

```
/**
 * Creates and executes a Customer search that finds all Customers for a specific
 * Sales Rep that are either on Credit Hold or have an Overdue Balance.
 *
 * Shows how to specify OR relationship between multiple criteria in Filter Expression
 *
 * Shows how to logically group criteria like using "Parens" in the UI
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], (s) => {
  const findProblemCustomersByRep = (salesRepId) =>
    s.create({
      type: s.Type.CUSTOMER,
      filters: [
        ['salesrep', s.Operator.ANYOF, salesRepId], 'and',
        [
          ['overduebalance', s.Operator.GREATERTHAN, 0], 'or',
          ['credithold', s.Operator.ANYOF, 'ON']
        ]
      ],
      columns: ['entityid', 'email']
    }).run()

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  // Replace "17" with the internal ID of any Sales Rep ID in your account
  findProblemCustomersByRep('17').each(printCustomerName)
})
```

We've encapsulated our Search in the **`findProblemCustomersByRep`** function.

`OR` Relationships

First let's focus on how we specify the `OR` relationship between Filters. Ultimately, all we're doing is changing the `'and'` we have been using to an `'or'` where appropriate:

```
['overduebalance', s.Operator.GREATERTHAN, 0], 'or',  
['credithold', s.Operator.ANYOF, 'ON']
```

Now our Search will find Customers that meet *either* (or both) of these criteria.

Logically Grouping Criteria

We have the `(B OR C)` portion of our Filters, but we also want to filter our results down by a specific Sales Rep to get the `A AND` portion.

We do that by first nesting our `overduebalance` and `credithold` filters within their own Array, thus logically grouping them together:

```
[  
  ['overduebalance', s.Operator.GREATERTHAN, 0], 'or',  
  ['credithold', s.Operator.ANYOF, 'ON']  
]
```

Although it's not necessary to write it out so verbosely, it might help to walk through it this way:

```
const salesRepFilter = ['salesrep', s.Operator.ANYOF, salesRepId] // => A  
const overdueFilter = ['overduebalance', s.Operator.GREATERTHAN, 0] // => B  
const creditFilter = ['credithold', s.Operator.ANYOF, 'ON'] // => C  
  
const problemCustomerFilter = [overdueFilter, 'or', creditFilter] // => B OR C  
  
customerSearch.filters = [salesRepFilter, 'and', problemCustomerFilter]  
// => A AND (B OR C)
```

Note that nothing at all has changed about the way we iterate through and process our results.

Favor Filter Expressions

There is no way to create an `OR` relationship using the Object syntax for Filters; there is also no way to logically group Filters using the Object syntax. These can only be accomplished with Filter Expressions.

Because of this power and flexibility that Filter Expressions have over the Object syntax, and the more concise nature of Filter Expressions, I only use the Object syntax when absolutely necessary. In all other cases, I use Filter Expressions to define my Search Filters.

Retrieving Data from Related Records

So far all of our searches have only retrieved data from the exact records that show up in our results. We know from navigating the UI that those records are related to many other records.

For instance, the Customers we have been searching are related to Sales Reps, to Contacts, to Transactions. Can we use our searches to retrieve data from these records as well?

Of course we can! We do this using "joins" in our Filters and Columns.

Let's expand our previous example to retrieve not only the Customer's main email address, but also the email address of the Primary Contact:

```
/**
 * Creates and executes a Customer search that finds all Customers for a specific
 * Sales Rep that are either on Credit Hold or have an Overdue Balance.
 *
 * Shows how to specify Joined Columns to retrieve data from related records
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], (s) => {
  const findProblemCustomersByRep = (salesRepId) =>
    s.create({
      type: s.Type.CUSTOMER,
      filters: [
        ['salesrep', s.Operator.ANYOF, salesRepId], 'and',
        ['salesrep.isinactive', s.Operator.IS, 'F'], 'and',
        [
          ['overduebalance', s.Operator.GREATERTHAN, 0], 'or',
          ['credithold', s.Operator.ANYOF, 'ON']
        ]
      ],
      columns: ['entityid', 'email', 'contactprimary.email']
    }).run()

  const printPrimaryContactEmail = (result) => {
    console.log(result.getValue({ name: 'email', join: 'contactprimary' }))
    return true
  }

  // Replace "17" with the internal ID of any Sales Rep ID in your account
  findProblemCustomersByRep('17').each(printPrimaryContactEmail)
})
```

Specifying Join Columns

Here, we are adding a Column to our results:

```
columns: ['entityid', 'email', 'contactprimary.email']
```

The format for a joined column uses a simple dot syntax: ``joinId.columnId``.

To see the available Joins for a specific Record Type, find the Record Type in the Records Browser and explore the "Search Joins" section.

Reading Data from Join Columns

We've renamed our iteration function to ``printPrimaryContactEmail`` and updated it accordingly.

When we go to read the value of a Join column, there is a slight addition to our ``getValue`` call:

```
result.getValue({ name: 'email', join: 'contactprimary' })
```

We must specify the additional ``join`` property to properly retrieve the value for the Join column. The same addition would apply to the ``getText`` method as well.

Specifying Join Filters

We can also filter our Searches based on data from related records using Join Filters. The syntax for a Join Filter is exactly the same as a Join Column, with the dot syntax:

```
`joinId.filterId`.
```

Notice in our example, we also wanted to make sure our Sales Rep record was not inactive, so we added a ``salesrep.isinactive`` Filter:

```
// ...  
['salesrep.isinactive', s.Operator.IS, 'F'], 'and',  
// ...
```

Filtering by Empty Fields

It is often the case that we will need to search for an empty field. How we filter for empty fields will vary depending on the type of field.

Empty Text-Like Fields

Let's look first at text fields by finding all Customers that don't have an email address:

```
/**
 * Creates and executes a Customer search that finds all Customers with no email
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['email', s.Operator.IEMPTY, '']
    ],
    columns: ['entityid', 'email']
  })

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  customerSearch.run().each(printCustomerName)
})
```

Text fields have an Operator called `IEMPTY` that matches empty fields.

Note that the empty String `''` is required even when we use `IEMPTY`. If you forget this value, you will receive an Error with a message similar to

“Wrong parameter type: filters is expected as Array.”

Empty Select Fields

Select fields (dropdowns) do not have the same `IEMPTY` Operator, so we need to specify our filter differently. NetSuite has made up a special value for empty Select fields. Let's modify our example to locate all Customers with no assigned Sales Rep:

```

require(['N/search'], (s) => {
  const customerSearch = s.create({
    type: s.Type.CUSTOMER,
    filters: [
      ['salesrep', s.Operator.ANYOF, '@NONE@']
    ],
    columns: ['entityid', 'email']
  })

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  customerSearch.run().each(printCustomerName)
})

```

We use the special `@NONE@` value to filter on an empty Select field. If we want records where the specified field *is empty* to match, we use `ANYOF` and `@NONE@`. If instead we want records where the specified field *is not empty* to match, we would use `NONEOF` and `@NONE@`.

To determine which Search Operators are available for specific Field Types, see the Help article [SuiteScript 2.x Search Operators ↗](#).

Loading and Executing a Saved Search

Thus far we've been creating our own SuiteScript Searches from scratch, but we can also leverage Saved Searches within our Scripts.

Instead of using `s.create` to make a new Search, we can instead use `s.load` to load an existing Saved Search:

```
/**
 * Loads and executes a Saved Customer search.
 *
 * Shows how to modify Filters and Columns on a Saved Search without saving them
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/search'], function (s) {
  const customerSearch = s.load({ id: 'customsearch_customers_in_ca' })

  customerSearch.filters = [
    ...customerSearch.filters,
    {
      name: 'salesrep',
      operator: s.Operator.ANYOF,
      values: ['@NONE@']
    }
  ]

  customerSearch.columns = [
    ...customerSearch.columns,
    'email',
    'contactprimary.email'
  ]

  const printCustomerName = (result) => {
    console.log(result.getValue({ name: 'entityid' }))
    return true
  }

  customerSearch.run().each(printCustomerName)
})
```

We only need to provide the Internal ID of the Saved Search to `load`.

```
const customerSearch = s.load({ id: 'customsearch_customers_in_ca' })
```

Let's presume our Saved Search here finds all Customers in California, but in our Script we need to add some additional criteria and result data.

Once the Saved Search is loaded, we are free to modify the Search Filters and Columns beyond what is set in the Saved Search using its `filters` and `columns` properties:

```
customerSearch.filters = [
  ...customerSearch.filters,
  {
    name: 'salesrep',
    operator: s.Operator.ANYOF,
    values: ['@NONE@']
  }
]

customerSearch.columns = [
  ...customerSearch.columns,
  'email',
  'contactprimary.email'
]
```

“In case you are newer to SuiteScript 2.1 and its more recent syntax, the `...` is the [destructuring operator](#). It's a powerful operator that you'll see in the wild and in many of my code examples, so it's worth studying intently.”

Here we've added a Filter so that we're only finding the Customers in California *that don't have a Sales Rep*, and we've also added the Email Address and Primary Contact's Email Address as Columns.

The `filters` and `columns` properties are standard JavaScript Arrays, so you can manipulate them using any Array method you would normally use for adding/removing/modifying elements of an Array.

Once again, nothing changes in the way we iterate through or process results.

Saving Modifications to a Saved Search

Changes to Filters and Columns of the loaded Search instance will only apply to the Search in our Script; they will *not* be saved back to the NetSuite UI.

If you want the changes you make in your Script to be saved back to the Saved Search itself, then you can call `save` on the Search instance after you've made the changes:


```
customerSearch.save()
```

Frequently Asked Questions

How do I find the name/ID for a specific Filter?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Filters" section. The value in the *Internal ID* column is what you'll use as your Filter name.

How do I find the name/ID for a specific Column?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Columns" section. The value in the *Internal ID* column is what you'll use as your Column name.

How do I find the name/ID for a specific Join?

Find your Record Type in the [Records Browser ↗](#), and explore the "Search Joins" section. The value in the *Join ID* column is what you'll use as your Join name.

There's a Records Browser, a Schema Browser, and a Connect Browser. What's the difference?

- *Records Browser* - Used for accessing Record data via *SuiteScript*
- *Schema Browser* - Used for accessing Record data via *SuiteTalk*
- *Connect Browser* - Used for accessing Record data via *ODBC*

When you're writing SuiteScript, you can safely focus only on the *Records Browser*.

Recommendations and Resources

NetSuite Help

NetSuite Help is the most definitive reference for the `N/search`` module and all of its capabilities. I recommend studying the following articles and any related sub-articles:

- [N/search Module ↗](#)
- [N/search Module Script Samples ↗](#)
- [search.Type ↗](#)
- [search.Summary ↗](#)
- [SuiteScript 2.x Search Operators ↗](#)
- [search.Filter ↗](#)
- [search.filterExpression ↗](#)
- [search.Column ↗](#)

The Records Browser

The Records Browser is an absolutely crucial tool for creating effective searches. There is a new version of the Records Browser for every version of NetSuite. The 2023.2 version can be found [in NetSuite Help ↗](#).

If you are unfamiliar with the Records Browser, see [SuiteScript Records Browser ↗](#) and [my tutorial article](#).

Use the Search UI

A great way to both learn about and verify your SuiteScript searches is to actually build the search in the UI first, then translate it into SuiteScript.

By doing this, you can quickly verify that the Filters and Columns you're specifying actually give you the correct results before you even start writing code.

NetSuite Search Export Chrome Plugin

There is a very helpful Chrome Plugin called "NetSuite Search Export" built by [in David Smith](#). The Plugin will automatically generate SuiteScript for any Saved Search in your account. You can find it [on the Chrome Plugin Store ↗](#)

To use this plugin:

1. Create a Saved Search in the UI
2. Save the search
3. Click the "Export to Script" link near the top right

About the Author



My name is [in Eric T Grubaugh](#). I run the *Sustainable SuiteScript* community for NetSuite developers. I founded Stoic Software in 2016 to help others lead successful, sustainable careers as NetSuite developers.

The "Sustainable SuiteScript" Community

We are a small community of NetSuite developers who want to deepen their technical skills, expand their professional network, and raise the bar for SuiteScript development. [Join us](#) today.

Questions, Comments, Corrections

If you have any questions, comments, or corrections on this document, please email them to me at eric+cookbooks@stoic.software.

Get in Touch

The best way to keep in regular contact with me is to join the [Sustainable SuiteScript](#) mailing list. I read and respond to all emails I receive there.

I create SuiteScript videos on [YouTube](#).

You can also connect with me on [in LinkedIn](#).

