

Basic Queries with SuiteScript 2.1

written by [Eric T Grubaugh](#)

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

published by [Stoic Software, LLC](#)

Basic Queries 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 Queries 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 building basic queries with the ``N/query`` module. The Query API is the SuiteScript equivalent of the Analytics Workbook in the UI.

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


- The fundamentals of creating and executing a Query with SuiteScript
- How to iterate over and read data from Query results
- How to get the number of results for a particular Query
- How to specify Joins to related records in your Query Conditions and Columns
- How to sort Query results
- How to load and execute a Saved Workbook in SuiteScript

Conventions in this Book

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

The ``N/query`` module is always imported as ``q``.

``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.

For more on how to test SuiteScript in your browser's console, watch my  [How-To video](#).

The Anatomy of a SuiteScript Query

In our first example, we create a Customer Query that retrieves the Entity ID and Email Address of all Customers which are marked as Individuals:

```
/**
 * Creates and executes a Customer query that finds all Individual Customers.
 *
 * Uses the most verbose syntax.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const customerQuery = q.create({ type: q.Type.CUSTOMER })

  customerQuery.condition = customerQuery.createCondition({
    fieldId: 'isperson',
    operator: q.Operator.IS,
    values: true
  })

  customerQuery.columns = [
    customerQuery.createColumn({ fieldId: 'email' }),
    customerQuery.createColumn({ fieldId: 'entityid' })
  ]

  const customerResultSet = customerQuery.run()

  const printCustomerNameAndEmail = (result) => {
    console.log(`${result.values[1]} - ${result.values[0]}`)
  }

  customerResultSet.results.forEach(printCustomerNameAndEmail)
})
```

All querying functionality in SuiteScript is provided by the [N/query module](#).

```
require(['N/query'], (q) => { /* ... */ });
```

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

1. Create a new [Query instance](#) OR load an existing [Workbook](#)
2. Specify the Record Type, Conditions, and Columns of our Query
3. Execute the Query
4. Retrieve the Results of the Query

5. Process the Results

Creating a Query

Here we import the `N/query` module as `q` and use its [create method ↗](#) to accomplish steps 1 and 2 of our basic formula. We create a Query instance by specifying its Record Type, Conditions, and Columns.

We start by defining the Query's Record Type with the `type` property of `create`. We provide it a value using the `N/query` module's [Type enumeration ↗](#) for native records.

```
const customerQuery = q.create({ type: q.Type.CUSTOMER })
```

For custom records, we instead enter the Custom Record's ID as a literal `string` (e.g. `type: 'customrecord_my_rec'`).

Specifying Conditions

Next, we provide our Query's [Condition ↗](#) (also called "Criteria" in the UI) using the `createCondition` method of the `Query` instance.

`Conditions` are the equivalent of [Filters on a Search ↗](#).

```
// ...
customerQuery.condition = customerQuery.createCondition({
  fieldId: 'isperson',
  operator: q.Operator.IS,
  values: true
})
// ...
```

Each `Condition` must have:

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

Valid operators are provided by the [query.Operator enumeration ↗](#).

Later, we will explore more properties of `Condition`'s and several other ways to specify them.

Specifying Columns

To finish off the creation of our `Query` instance, we specify the [Columns](#) that will be included in the results. We specify Columns using the `columns` property of the `Query` instance.

`Columns` are the equivalent of [Columns on a Search](#).

```
// ...
customerQuery.columns = [
  customerQuery.createColumn({ fieldId: 'email' }),
  customerQuery.createColumn({ fieldId: 'entityid' })
]
// ...
```

Each element of the Array defines one `Column` by providing at least a `fieldId` to identify the field to be included in the results.

At least one `Column` *must* be defined for a Query. Without any Columns, a Query will throw an `UNEXPECTED ERROR` upon execution.

Later, we will explore more properties of `Column`'s and several other ways to specify them.

Executing the Search

Creating the `Query` instance is not enough to actually execute the Query. In order to do that, we need to invoke the [run method](#) on our Query instance.

```
// ...
const customerResultSet = customerQuery.run()
// ...
```

This will execute our Query on the NetSuite server and save the results there for when our script is ready to retrieve them. The output of the `run()` method is a [ResultSet](#), which we store as `customerResultSet`; this object will contain the properties and methods for iterating over our Query results.

Processing the `results` Array

The `ResultSet` contains a `results` property, which is an Array of [Result](#) instances, which we can use to process our Query results. This is a native JavaScript Array, so all native Array methods are available to us.

Here, we use the JavaScript [forEach](#) method to iterate through the Array and pass each element to a function for processing.

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) => {  
  console.log(`${result.values[1]} - ${result.values[0]}`)  
}  
  
customerResultSet.results.forEach(printCustomerNameAndEmail);
```

Note that using `run()` is limited to, at most, `5,000` results. We will explore the methods for processing larger result sets later.

Reading Result Data with the `values` property

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

```
console.log(`${result.values[1]} - ${result.values[0]}`)
```

`values` is another native Array containing the Column values of the `Result`. The order of the values in the Array corresponds to the order in which the `Columns` were defined on the Query.

For text fields like our Entity ID and Email, the Array element will be the text value of the field.

For Select fields (dropdowns), `values` will always contain the internal ID of the selected record. If instead, you want to display the text displayed in the Select field, you must use a [Field Context](#), which we will explore later.

Promises

Note when you use `N/query`, there is a `Promise` version of the `run()` method: [query.run.promise\(\)](#). While Promises are beyond the scope of this Cookbook, you can find more info [in Help](#) and other links in the *Resources* chapter.

A More Concise, Readable Query

As we saw in the first example, our Query Columns can be created by setting the `columns` property of a `Query` to an Array of `Column` Objects, which in turn are created by passing an Object to the `createColumn` method. This gets *really* verbose for queries with multiple Columns. Fortunately, the `create` method of the Query gives us a slightly more concise way of defining Columns.

We also saw that our results can be read via the `values` Array on the `Result` Object, but this is neither readable nor intuitive as you must remember the order in which your Columns were defined, and you can never change that order.

Don't worry; there is a better way.

```
/**
 * Creates and executes a Customer query that finds all Individual Customers.
 *
 * Uses a more concise syntax for Column creation and a more readable format
 * for reading results.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const customerQuery = q.create({
    type: q.Type.CUSTOMER,
    columns: [
      { fieldId: 'email' },
      { fieldId: 'entityid', alias: 'name' }
    ]
  })

  customerQuery.condition = customerQuery.createCondition({
    fieldId: 'isperson',
    operator: q.Operator.IS,
    values: true
  })

  const customerResultSet = customerQuery.run()

  const printCustomerNameAndEmail = (result) => {
    const resultObject = result.asMap()
    console.log(`${resultObject.name} - ${resultObject.email}`)
  }

  customerResultSet.results.forEach(printCustomerNameAndEmail)
})
```


Functionally, this is an identical Query to that of the previous chapter, but we've saved ourselves a bit of typing, and we've made reading data out of our results a bit easier.

Concise Column Creation

We start by moving our Column definitions into the `create` method instead of using the `columns` property on the `Query` Object.

```
const customerQuery = q.create({
  type: q.Type.CUSTOMER,
  columns: [
    { fieldId: 'email' },
    { fieldId: 'entityid', alias: 'name' }
  ]
})
```

When we do this, we specify exactly the same Objects as before, but now NetSuite will *automatically* invoke `createColumn` on each element of the `columns` Array. It's implied that we are creating Column Objects, so we save ourselves some typing.

The remainder of the examples in this cookbook will utilize this syntax for Column creation.

Alternate Column Creation

If, for some reason, it is not possible for you to use the `columns` property of `create`, we can still make our Column definition a bit more concise. Instead of creating the Array and explicitly calling `createColumn` on each element, we can achieve the same result by a [map over the Array ↗](#):

```
customerQuery.columns = [
  { fieldId: 'email' },
  { fieldId: 'entityid', alias: 'name' }
].map(customerQuery.createColumn)
```

Always remember that the `columns` property of a `Query` is a native JavaScript Array filled with `Column` elements, which means you can build and manipulate that Array using any standard Array methods, so long as the finished result is an Array of `Column` elements.

Results as Objects instead of Arrays

Instead of reading Result values from an Array of strings via the `values` property on each Result, we invoke the [asMap\(\) method](#) of the Result to transform it into a native JavaScript Object:

```
const resultObject = result.asMap()
console.log(`${resultObject.name} - ${resultObject.email}`)
```

This method transforms a `Result` into key-value pairs as a native JavaScript Object.

The key for each Column will be the `alias` specified during Column creation, as we see here in our `entityid` Column. If no `alias` is specified, then the `fieldId` will be used, as we see here in our `email` Column.

From there, we interact with result as we would any other JavaScript Object.

Note that nothing else about how we execute the Query, declare Conditions, or iterate through results has changed.

Even More Concise Results

The transformation of Results to key-value pairs is so handy, NetSuite gives us a convenience method to automatically map all of our Results to Objects, rather than having to call `asMap` on each one ourselves:

```
const printCustomerNameAndEmail = (result) => {
  console.log(`${result.name} - ${result.email}`)
}

customerResultSet.asMappedResults().forEach(printCustomerNameAndEmail);
```

Instead of using the `results` property of our `ResultSet`, we invoke its [asMappedResults method](#). This will automatically invoke `asMap` on each `Result` and return us the resulting Array of plain JavaScript Objects.

The remainder of the examples in this cookbook will utilize this syntax for result iteration.

How Many Results Does My Query Have?

Often, you'll want to determine the number of results your Query has. To do so, we need to use a different method for executing our Query:

```
/**
 * Retrieves the number of results in a Query.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const customerQuery = q.create({
    type: q.Type.CUSTOMER,
    columns: [
      { fieldId: 'email' },
      { fieldId: 'entityid', alias: 'name' }
    ]
  })

  const pagedResultSet = customerQuery.runPaged()

  console.log(`Query Results: ${pagedResultSet.count}`)
})
```

Instead of calling the Query's `run` method which returns a `ResultSet`, we use `runPaged` to get a [PagedData instance](#). The `PagedData` has a `count` property, which will tell us how many total results the Query yields.

Recall that `run` with its `ResultSet` is limited to `5,000` results. We must use `runPaged` and its `PagedData` because it can handle more than `5,000` results.

Note that `runPaged()` also has a Promise version [runPaged.promise\(\)](#), but these are again out of scope for this book.

Multiple Conditions

So far we've only seen how to specify a single `Condition`` in our Query, but we will almost always need multiple Conditions. Given that the `condition`` property of a Query only accepts a single `Condition`` Object, we cannot pass an Array of `Condition`s` like we did with our Query `Column`s`.

Rather than building a list of multiple `Condition`s`, the Query API provides methods for combining multiple `Condition`s` into a single `Condition`` instance using functions that represent the logical operators:

- [`AND` ↗](#)
- [`OR` ↗](#)
- [`NOT` ↗](#)

To illustrate, we'll adjust our previous example to Query for Individual Customers *with a specific Sales Rep*.

We'll even go a step further and encapsulate the Query creation into a function that accepts an arbitrary Sales Rep ID:

```

/**
 * Query for Individual Customers with a specific Sales Rep
 *
 * Shows how to specify multiple Conditions using the logical operator AND.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const findCustomersByRep = (salesRepId) => {
    const customerQuery = q.create({
      type: q.Type.CUSTOMER,
      columns: [
        { fieldId: 'email' },
        { fieldId: 'entityid', alias: 'name' }
      ]
    })

    customerQuery.condition = customerQuery.and(
      customerQuery.createCondition({
        fieldId: 'isperson',
        operator: q.Operator.IS,
        values: true
      }),
      customerQuery.createCondition({
        fieldId: 'salesrep',
        operator: q.Operator.ANY_OF,
        values: salesRepId
      })
    )

    return customerQuery.run().asMappedResults()
  }

  const printCustomerNameAndEmail = (result) => {
    console.log(`${result.name} - ${result.email}`)
  }

  // Replace -5 with the internal ID of any Sales Rep ID in your account
  findCustomersByRep(-5).forEach(printCustomerNameAndEmail)
})

```

Functions for Readability and Reusability

We have moved our Query creation inside a function that accepts a parameter for the Sales Rep:

```

const findCustomersByRep = (salesRepId) => {
  const customerQuery = q.create({
    type: q.Type.CUSTOMER,
    columns: [
      { fieldId: 'email' },
      { fieldId: 'entityid', alias: 'name' }
    ]
  })

  customerQuery.condition = customerQuery.and(
    customerQuery.createCondition({
      fieldId: 'isperson',
      operator: q.Operator.IS,
      values: true
    }),
    customerQuery.createCondition({
      fieldId: 'salesrep',
      operator: q.Operator.ANY_OF,
      values: salesRepId
    })
  )

  return customerQuery.run().asMappedResults()
}

```

The function creates and immediately executes the Query, returning the mapped ``ResultSet``.

This pattern of creating the Query instance and returning it from a function named ``find*`` is a common design pattern I use to isolate my Query logic. I find that this makes it faster to comprehend, modify, and reuse my Queries.

This makes the execution and processing of our Query condense down to:

```

findCustomersByRep(-5).forEach(printCustomerNameAndEmail);

```

which I find to be simpler and more readable than our previous query.

Combine Conditions using Logical Operators

Focusing in on our ``Condition``, you can see that we've added a second ``Condition`` and passed both of them to the [`and` method](#) of our ``Query`` instance:

```
customerQuery.condition = customerQuery.and(
  customerQuery.createCondition({
    fieldId: 'isperson',
    operator: q.Operator.IS,
    values: true
  }),
  customerQuery.createCondition({
    fieldId: 'salesrep',
    operator: q.Operator.ANY_OF,
    values: salesRepId
  })
)
```

The [`and` method](#) provides logical conjunction functionality for Queries, similar to JavaScript's [logical AND \(`&&`\) operator](#). There is a corresponding [`or` method](#) that provides logical disjunction functionality, similar to JavaScript's [logical OR \(`||`\) operator](#).

Both methods accept *any number* of `Condition`` parameters and also return a `Condition`` instance. In this way, multiple `Condition`s` can be combined quickly using the logical operators.

There is also a [`not` method](#) which provides logical negation of a *single* `Condition``, similar to JavaScript's [logical NOT \(`!`\) operator](#).

Complicating Matters

For illustration purposes, we can build a much more complex Query by combining several Conditions and logical operators. We can also make the intent of each `Condition`` clearer by assigning it to a variable before combining it into the logical operator methods.

```

/**
 * Query for Individual Customers which have a specific Sales Rep and either
 * have an overdue balance or a credit hold.
 *
 * Shows how to specify multiple Conditions using multiple logical operators.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const findProblemCustomersByRep = (salesRepId) => {
    const customerQuery = q.create({
      type: q.Type.CUSTOMER,
      columns: [
        { fieldId: 'email' },
        { fieldId: 'entityid', alias: 'name' }
      ]
    })

    const hasGivenSalesRep = customerQuery.createCondition({
      fieldId: 'salesrep',
      operator: q.Operator.ANY_OF,
      values: salesRepId
    })

    const hasOverdueBalance = customerQuery.createCondition({
      fieldId: 'overduebalancesearch',
      operator: q.Operator.GREATER,
      values: 0
    })

    const hasCreditHold = customerQuery.createCondition({
      fieldId: 'creditholdoverride',
      operator: q.Operator.ANY_OF,
      values: ['ON']
    })

    customerQuery.condition = customerQuery.and(
      hasGivenSalesRep,
      customerQuery.or(hasOverdueBalance, hasCreditHold)
    )

    return customerQuery.run().asMappedResults()
  }

  const printCustomerNameAndEmail = (result) => {
    console.log(`${result.name} - ${result.email}`)
  }

  // Replace "17" with the internal ID of any Sales Rep ID in your account
  findProblemCustomersByRep(17).forEach(printCustomerNameAndEmail)
})

```

First, we set up our `Condition`'s as variables to give them readable names.


```

const hasGivenSalesRep = customerQuery.createCondition({
  fieldId: 'salesrep',
  operator: q.Operator.ANY_OF,
  values: salesRepId
})
const hasOverdueBalance = customerQuery.createCondition({
  fieldId: 'overduebalancesearch',
  operator: q.Operator.GREATER,
  values: 0
})
const hasCreditHold = customerQuery.createCondition({
  fieldId: 'creditholdoverride',
  operator: q.Operator.ANY_OF,
  values: ['ON']
})

```

This is not a mandatory step, but does improve the developer experience of our code, in my opinion.

Next, we use the various logical operator methods to combine our several `Condition`s` into one instance and set that as our Query's `condition``.

```

customerQuery.condition = customerQuery.and(
  hasGivenSalesRep,
  customerQuery.or(hasOverdueBalance, hasCreditHold)
)

```

By giving our separate `Condition`s` these understandable names and separating the `Condition`` definitions from their logical combination, we make the code simpler to understand, even if we typed a few more characters with this approach.

If we try to combine both the logical operators and the `Condition`` definitions, it can easily become overwhelming to read.

The equivalent logic expressed with normal JavaScript operators would look like:

```

hasGivenSalesRep && (hasOverdueBalance || hasCreditHold)

```

Once again, we have not changed anything about the way we process the results of the Query.

Retrieving Data from Related Records

Thus far, all of our queries 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 querying are related to Sales Reps, to Contacts, to Transactions. Can we use our queries to retrieve data from these records as well?

We do this using ["joins"](#) in our Conditions and Columns.

"I am no database administrator, and never have been. There are much better resources for in-depth explanations of database joins around the internet. I've linked to a few in the Resources section."

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 retrieves the Primary
 * Contact's Email
 *
 * Shows how to specify Joined Columns to retrieve data from related records
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const findCustomersByRep = (salesRepId) => {
    const customerQuery = q.create({ type: q.Type.CUSTOMER })

    const contactJoin = customerQuery.autoJoin({ fieldId: 'contact' })

    const hasMatchingSalesRep = customerQuery.createCondition({
      fieldId: 'salesrep',
      operator: q.Operator.ANY_OF,
      values: salesRepId
    })
    const hasContactEmail = contactJoin.createCondition({
      fieldId: 'email',
      operator: q.Operator.EMPTY_NOT
    })

    customerQuery.condition = customerQuery.and(
      hasMatchingSalesRep,
      hasContactEmail
    )

    customerQuery.columns = [
      contactJoin.createColumn({ fieldId: 'email', alias: 'contactemail' }),
      customerQuery.createColumn({ fieldId: 'email', alias: 'customeremail' }),
      customerQuery.createColumn({ fieldId: 'entityid' })
    ]

    return customerQuery.run().asMappedResults()
  }

  const printContactEmail = (result) => {
    console.log(result.contactemail)
  }

  // Replace "17" with the internal ID of any Sales Rep ID in your account
  findCustomersByRep(17).forEach(printContactEmail)
})

```

Specifying Joins

The first step in retrieving data from related records is to specify the records we wish to join. There are several different types of joins. Here we start with the [`autoJoin` method](#):

```
const contactJoin = customerQuery.autoJoin({ fieldId: 'contact' })
```

We assign the join to a variable as we will need access to it later in order to add Conditions and Columns on the Join component.

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

Adding Join Conditions and Columns

Now that we have a Join component, we are able to define Conditions and Columns on it in much the same way as we do with our normal Query component:

```
const hasContactEmail = contactJoin.createCondition({
  fieldId: 'email',
  operator: q.Operator.EMPTY_NOT
})

contactJoin.createColumn({ fieldId: 'email', alias: 'contactemail' })
```

Carefully observe that we're calling both `createCondition`` and `createColumn`` on our `contactJoin`` component, *not* on `customerQuery`` as we are with the other `Column`s`. This retrieves the `email`` field from the Contact, as opposed to that from the Customer of our main Query component.

Since we have both the Customer's `email`` Column and the Contact's `email`` Column, we need a way to differentiate the two Columns when reading our mapped results later. We use the `alias`` to do so.

Reading Data from Join Columns

We've renamed our iteration function to `printContactEmail`` and updated it accordingly. Since we defined an `alias`` for both of our email Columns, there's now no difference between the way we read a Join Column and a standard Column:

```
const printContactEmail = (result) => {
  console.log(result.contactemail)
}
```

For this reason, I recommend always specifying an `alias`` for your `Columns``, particularly for Join Columns.

Additional Join Types

The `autoJoin`` method allows us to create very simple joins. There are several other types of joins the Query API allows us to create, but due to their more advanced nature, those joins are not covered in this book.

For the curious, you can investigate the [`join` method ↗](#), [`joinTo` method ↗](#) and [`joinFrom` method ↗](#) of the Query module to learn how to create directional joins.

Sorting Query Results

Often we will need to enforce the order of our Query results by sorting on various Columns.

```

/**
 * Creates and executes a Customer search that sorts results by Overdue
 * Balance then by Name
 *
 * Shows how to sort Query results
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  const findOverdueCustomers = () => {
    const customerQuery = q.create({ type: q.Type.CUSTOMER })

    customerQuery.condition = customerQuery.createCondition({
      fieldId: 'overduebalancesearch',
      operator: q.Operator.GREATER,
      values: 0
    })

    const companyName = customerQuery.createColumn({
      fieldId: 'companyname',
      alias: 'name'
    })

    const overdueBalance = customerQuery.createColumn({
      fieldId: 'overduebalancesearch',
      alias: 'overdueBalance'
    })

    customerQuery.columns = [companyName, overdueBalance]

    customerQuery.sort = [
      customerQuery.createSort({ column: overdueBalance, ascending: false }),
      customerQuery.createSort({
        column: companyName,
        nullsLast: true,
        caseSensitive: false
      })
    ]

    return customerQuery.run().asMappedResults()
  }

  const printCustomerData = (result) => {
    console.log(`${result.name}: ${result.overdueBalance}`)
  }

  findOverdueCustomers().forEach(printCustomerData)
})

```

Specifying Sorts

After we have established the Columns of a Query, we can then apply sorting to them using the `Query` component's `sort` property. This is an Array of [Sort instances](#) which we generate by calling the [createSort method](#).

```
customerQuery.sort = [  
  customerQuery.createSort({ column: overdueBalance, ascending: false }),  
  customerQuery.createSort({  
    column: companyName,  
    nullsLast: true,  
    caseSensitive: false  
  })  
];
```

The order of `sort`'s defined in the Array is the order in which the corresponding Columns will be sorted. Thus in our example, we are sorting first by Overdue Balance, then by Company Name.

Sort Direction

The `ascending` option of the `createSort` method provides us with the means to specify the direction of the sorting. It's a boolean value: `true` for ascending order, `false` for descending.

The default value for `ascending` is `true`.

In our example, we sort Overdue Balance in descending order and Company Name in ascending order.

Sorting of Empty and Null Values

Using the `nullsLast` property, you can specify how empty or null values are handled. They can either be grouped at the beginning or the end of the sorted results. It's another simple boolean value: `true` to put empty values at the end, `false` to put them at the beginning.

The default value for `nullsLast` is whatever value you set for `ascending`. If `ascending` is `true`, then `nullsLast` defaults to `true`.

In our example, we have filtered out empty Overdue Balance fields, and we group any empty Company Names at the end.

Case Sensitivity

The `caseSensitive` property of `createSort` lets us specify how to handle character case in our sorting. It's another simple boolean value: `true` to respect case when sorting, `false` to ignore case when sorting.

The default value for `caseSensitive` is `false`.

In our example, we want to ignore case when sorting by Company Name.

Loading and Executing a Workbook

The Query API is the SuiteScript equivalent of the [Analytics Workbook](#) in the UI. Thus far we've been creating our own SuiteScript Queries from scratch, but we can also leverage existing Saved Workbooks within our Scripts.

Instead of using `q.create` to make a new Query, we can instead use the `q.load` method to load an existing Workbook:

```
/**
 * Loads and executes a Saved Workbook.
 *
 * @author Eric T Grubaugh <eric@stoic.software>
 */
require(['N/query'], (q) => {
  // Replace the id here with the Workbook ID of a Workbook in your account
  const salesOrderedWorkbook = q.load({ id: 'custworkbook6' })

  const printResult = (result) => console.log(result)

  salesOrderedWorkbook.run().asMappedResults().forEach(printResult)
})
```

We only need to provide the Workbook ID of the Workbook to `load`.

Once the Workbook is loaded, we are free to modify the Conditions and Columns beyond what is set in the Workbook using its `condition` and `columns` properties, just as we would with a Query we created ourselves.

Once again, nothing changes in the way we iterate through or process results. You would update the `printResult` method with the specific Columns of your Workbook.

“⚠ Note that there are many conditions and exceptions for what makes a Workbook acceptable to `load()`. These are documented in the [query.load\(\) documentation](#). If you encounter an error when trying to load a Workbook, I recommend reading through these conditions and confirming your Workbook meets all criteria or that you are using the appropriate ID.”

Saving Modifications to a Query

At the time of this writing (2024.1), SuiteScript can only load or execute a Workbook. Saving changes to a created or existing Workbook via SuiteScript is not supported.

Recommendations and Resources

NetSuite Help

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

- [Scripting with the N/query Module ↗](#)
- [N/query Module ↗](#)
- [N/query Module Script Samples ↗](#)
- [query.Type ↗](#)
- [query.Operator ↗](#)
- [Query.createCondition\(options\) ↗](#)
- [Query.createColumn\(options\) ↗](#)
- [`Query.and\(\)` ↗](#)
- [`Query.or\(\)` ↗](#)
- [`Query.not\(\)` ↗](#)
- [Query.autoJoin\(options\) ↗](#)
- [Query.createSort\(options\) ↗](#)

Study Database Joins

- W3Schools [on SQL Joins ↗](#)
- LearnSQL [on SQL Join Types ↗](#)
- Database Star [on SQL Joins ↗](#)

Use the Workbook UI

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

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

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 ↗](#) in the Help documentation and [my tutorial](#).

Mozilla Developer Network

SuiteScript is a library on top of JavaScript, and the best JavaScript reference manual is the [Mozilla Developer Network ↗](#).

While not related specifically to NetSuite, this site is an excellent source of JavaScript reference material, examples, and tutorials.

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 [LinkedIn](#).

