

Rendering PDFs with SuiteScript 2.1

written by [Eric T Grubaugh](#)

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

published by [Stoic Software, LLC](#)

Rendering PDFs 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:

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

Introduction

Rendering PDFs with SuiteScript 2.1 is intended to provide you with practical examples for leveraging NetSuite's Advanced PDF engine to generate informative PDF files for distribution.

In this SuiteScript Cookbook, you'll see examples of:

- Displaying a PDF for a user
- Downloading a PDF for a user
- Rendering a PDF using a Template
- Rendering a PDF from custom XML
- Rendering Record data in a PDF Template
- Rendering Saved Search results in a PDF Template
- Rendering Query results in a PDF Template
- Rendering custom data sources in a PDF Template

Patterns in this Book

All code examples are written in *SuiteScript 2.1*.

The `N/render` module is always imported as `render`.

Initial Setup

The ``N/render`` module allows us to interact with NetSuite's built-in PDF generation engine. NetSuite uses the [Freemarker templating engine](#) to merge data into XML templates, and it uses the [Big Faceless Organization](#) (BFO) report generation tool for turning the resulting XML into a PDF. Specifics on these tools and their operation is beyond the scope of this cookbook.

Before we can get to rendering your shiny PDFs, there's a little setup work to do first. PDFs can only be rendered in server-side scripts, meaning we can't drop code in the browser console and expect it to work.

Instead, we'll build a Suitelet that will render our PDFs for us.

We'll start by building a custom module file. This is where we'll be adding and changing the code from the upcoming examples:

```
/**
 * Custom module for executing N/render cookbook examples
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/render'], (render) => {
  const renderPdf = (response) => {
    // This is where our example code will go
  }

  return { renderPdf }
})
```

We'll walk through this code in detail shortly.

1. Create a folder in the File Cabinet at ``/SuiteScripts/render-pdf-cookbook/``
2. Upload the above source code into the new folder in a file named ``render-cookbook.js``.
3. From now on, I will refer to this file as the "``render-cookbook`` module".

```

/**
 * Suitelet for testing PDF rendering
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 * @NScriptType Suitelet
 */
define(['./render-cookbook', 'N/https'], (pdfCookbook, https) => {
  /** @see https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/section_4407
  const onRequest = (context) => {
    log.audit({ title: `${context.request.method} request received` })

    // Ignore POST requests
    if (context.request.method !== https.Method.GET) {
      return
    }

    pdfCookbook.renderPdf(context.response)

    log.audit({ title: 'Request complete.' })
  }

  return { onRequest }
})

```

1. Use the above source code to create a second file in the same folder as before.
2. Use this second file to [create a new `Suitelet`](#) named *Render a PDF*.
3. [Create a Deployment](#) for the Suitelet; leave it in **Testing** status.
4. On the Deployment, [add a new `Link`](#) in the **Links** sublist. Locate it somewhere accessible to the Role you'll be using to test the examples in this book. For me, using the Administrator Role, I chose **Classic Center > Setup > Custom > Render a PDF**.

Script Deployment

[Edit](#)[Back](#)[Actions ▾](#)

SCRIPT

Render a PDF

STATUS

Testing

TITLE

Render a PDF

EVENT TYPE

ID

customdeploy_sl_render

LOG LEVEL

Debug



DEPLOYED

EXECUTE AS ROLE

Current Role



AVAILABLE WITHOUT LOGIN

URL

</app/site/hosting/scriptlet.nl?script=682&deploy=1>[Audience •](#)[Links •](#)[Execution Log](#)[System Notes](#)

CENTER

SECTION

CATEGORY

LABEL

Classic Center

Setup

Custom

Render a PDF

“⚠ If either of these code files is named differently or is not in the same folder, you will likely receive `MODULE_NOT_FOUND` errors when attempting to access the Suitelet.”

This Suitelet is our test runner for working with files. Whenever we need to test one of our code examples, we access the link for the Suitelet in NetSuite's main navigation, and we can monitor the resulting [Execution Logs ↗](#) on the Suitelet record.

For the remainder of the cookbook, you should not need to make any modifications to the Suitelet or its source code.

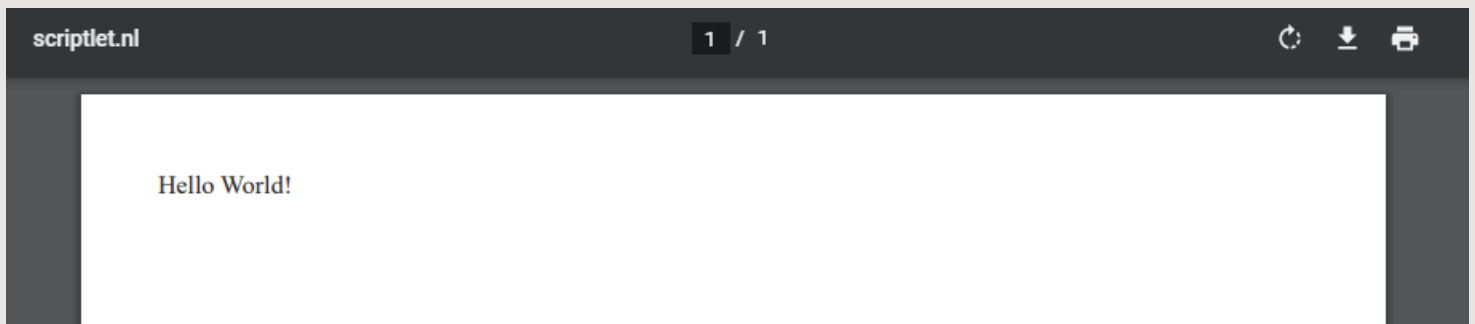
Render and Display a PDF

We start by rendering a PDF consisting of static text:

```
const renderPdf = (response) => {
  const xmlContent = `
    <!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
    <pdf>
      <body>Hello World!</body>
    </pdf>
  `

  const renderer = render.create()
  renderer.templateContent = xmlContent
  renderer.renderToResponse({ response })
}
```

Navigate to the Suitelet via the Link you created on the Deployment, and you should see the message "Hello World!" rendered in a PDF and displayed in your browser.



Create the XML

```
const xmlContent = `
  <?xml version="1.0"?>
  <!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
  <pdf>
    <body>Hello World!</body>
  </pdf>
`
```

Our template is a string of XML boilerplate containing a `<body>` which consists of a static text message: *"Hello World!"*.

The BFO generator accepts markup that is nearly identical to HTML. See the [BFO reference](#) for details.

`TemplateRenderer`

```
const renderer = render.create()  
renderer.templateContent = xmlContent
```

The core of the [`N/render` module's](#) functionality resides in the [`TemplateRenderer` Object](#), which we generate using the [`create\(\)` method](#) of the module. We then assign the raw XML string to the ``TemplateRenderer`'s` `templateContent` property. There are many more capabilities of this Object which we will continue to explore, but for now, our template is set and ready to transform into a PDF.

```
renderer.renderToResponse({ response })
```

The [`renderToResponse\(\)` method](#) of the ``TemplateRenderer`` takes the content we've set as the template and (later) any data we've merged into the template and writes the output to our Suitelet's [`ServerResponse` object](#).

“⚠ There is additionally a [`renderPdfToResponse\(\)` method](#), but at the time of this writing (2024.1), this method was failing with an `UNEXPECTED_ERROR``.”

Download the Rendered PDF

Our previous example creates the PDF and renders it directly in the user's browser. What if we want the PDF to download automatically for the user instead?

Update the `renderPdf` function like so:

```
const renderPdf = (response) => {
  const xmlContent = `
    <!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
    <pdf>
      <body>Hello World!</body>
    </pdf>
  `

  const renderer = render.create()
  renderer.templateContent = xmlContent

  const pdf = renderer.renderAsPdf()
  response.writeFile({ file: pdf, isInline: false })
}
```

We've replaced the usage of `renderer.renderToResponse()` with:

```
const pdf = renderer.renderAsPdf()
response.writeFile({ file: pdf, isInline: false })
```

The [renderAsPdf\(\) method](#) turns our template into a `File` instance, and then we use the Suitelet's [ServerResponse.writeFile\(\) method](#) to send the `File` in the response.

isInline

The `isInline` option of `writeFile()` is a boolean flag which indicates whether the `File` should be rendered in the browser (`true`) or downloaded (`false`).

“⚠ Unfortunately, at the time of this writing (2024.1), setting `isInline` to `true` does not seem to work correctly for PDF files, and the PDF gets downloaded regardless of the value of `isInline`.”

Render a Record using an existing PDF Template

We've used our ability to specify our own XML and transform it into a PDF, but it's extremely clunky to manage XML (or any code) within a string like we've had to do in our first two examples. Plus, NetSuite already has a robust PDF Template system. How about we leverage that instead?

Replace ``render-cookbook.js`` with:

```
/**
 * Custom module for rendering a Transaction to a PDF
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/file', 'N/render'], (file, render) => {
  const renderPdf = (response) => {
    const pdf = render.transaction({
      entityId: 1280,
      formId: 71
    })

    response.writeFile({ file: pdf, isInline: false })
  }

  return { renderPdf }
})
```

results in:



Quote

Date	Quote #
2017/1/3	QUO00001029

Bill To

Victor Vartan
Franklin Photography
106 W. Kirkwood, Ave
Bloomington IN 47404

Project	Customer Ph...	Ship Via
	(650) 600-1000	Truck

Item	Quantity	Units	Description	Options	Rate	Amount	Logo
Nikon Pix 8.5 Megapixel Digital	5		2 inch display and 10X's Zoom make this steal for the money		1,100.00	5,500.00	
4.8 Megapixels for very clear photographs	2				225.00	450.00	
FujiFilm SharpPix A575 Digital	3		Outstanding pictures, 16MB memory card and 3X's Zoom		250.00	750.00	

`render.transaction()`

The ``N/render`` module gives us several methods for leveraging NetSuite's Advanced PDF/HTML Template system. Here, we demonstrate the [`transaction\(\)` method's`](#) ability to render a Transaction record using the PDF Templates specified on the Transaction Form.

```
const pdf = render.transaction({
  entityId: 1280,
  formId: 71
})
```

The ``entityId`` is - despite the name - actually the *internal ID* of the Transaction record we'd like to render. The ``formId`` is the numeric internal ID of the *Transaction Form* to start from.

In this specific example, ``1280`` is the internal ID of a Quote/Estimate record, and ``71`` is the internal ID of the Standard Quote Transaction Form.

Importantly, ``formId`` is *not* the ID of a specific PDF Template. Instead, NetSuite inspects the Transaction Form you've selected, then uses the ``Print Template`` selected there.

Custom Transaction Form

Save

▼

Cancel

Save & Move Elements

NAME *

Custom Online Quote

ID

TYPE

Quote

PRINTING TYPE ☒ ADVANCED ☐ BASIC

PRINT TEMPLATE

Standard Quote PDF/HTML Template ▼

Refresh the Suitelet's page, and you should see your transaction PDF form rendered in the browser.

Print Modes

By default, `transaction()` and its sibling methods will print the record in either PDF or HTML format, depending on the User or the Company Preference for `PRINT USING HTML`. You can force the printing mode to one or the other by using the `printMode` option of each method and the [render.PrintMode enumeration](#).

Rendering other Record Types

The sibling methods for `transaction()` within `render` are:

- [bom\(\)](#)
- [packingSlip\(\)](#)
- [pickingTicket\(\)](#)
- [statement\(\)](#)

All of these are convenience methods, abstracting the `TemplateRenderer` Object we've used previously and returning the `File` instance for the PDF directly.

They all behave similarly, but I leave researching the specifics of each as an exercise for you.

Render a Record using a Custom Template

We can combine what we've learned so far, leveraging the more sophisticated Freemarker templates and merging records into them, to render PDFs from our own custom Templates, as opposed to ones created via the UI.

Replace `render-cookbook.js` with:

```
/**
 * Custom module for rendering a PDF with a custom template
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/file', 'N/record', 'N/render'], (file, r, render) => {
  const renderPdf = (response) => {
    const xmlContent = file.load({ id: './custom-record.ftl' }).getContents()

    const renderer = render.create()
    renderer.templateContent = xmlContent

    renderer.addRecord({
      templateName: 'quote',
      record: r.load({ type: r.Type.ESTIMATE, id: 1280 })
    })

    renderer.renderToResponse({ response })
  }

  return { renderPdf }
})
```

Name the following file `custom-record.ftl`, and upload it to the same folder as `render-cookbook.js`:

```

<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
<pdf>
  <head>
  </head>
  <body size="Letter">
    <form>
      <h4>${quote.tranid@label}: ${quote.tranid}</h4>
      <p>
        <span>${quote.probability@label}:</span>
        ${quote.probability}
      </p>
      <p>
        <span>${quote.expectedclosedate@label}:</span>
        ${quote.expectedclosedate}
      </p>
      <p>
        <span>${quote.total@label}:</span>
        ${quote.total}
      </p>
      <p>
        <span>Sales Team:</span>
        <#list (quote.salesteam)![]>
          <table>
            <tr>
              <th>Name</th>
              <th>Role</th>
              <th>Contribution</th>
            </tr>
            <#items as rep>
              <tr>
                <td>${rep.employee}</td>
                <td>${rep.salesrole}</td>
                <td>${rep.contribution}</td>
              </tr>
            </#items>
          </table>
        <#else>
          No associated Sales Team.
        </#list>
      </p>
    </form>
  </body>
</pdf>

```

Rendering the PDF results in something like:

Quote #: QUO00001029

Probability: 50%

Exp. Close: 2017/1/3

Total: \$6,080.00

Sales Team:

Name	Role	Contribution
E0016 Neil Thomson	Sales Rep	100%

Our custom template exists like any other file in the File Cabinet, so we can load its contents into our script accordingly using the ``N/file`` module.

```
const xmlContent = file.load({ id: './custom-record.ftl' }).getContents()
```

We then create a new ``TemplateRenderer`` as usual and assign the template file contents to ``renderer.templateContent``. Keeping our Template XML in its own dedicated file and not in a string within our script is highly preferable and will make it far easier to maintain and update.

```
const renderer = render.create()  
renderer.templateContent = xmlContent
```

As we saw with the ``transaction()`` method, we can merge existing records into *custom* templates as well by using the [`addRecord\(\)` method](#) of the ``TemplateRenderer``. We use the ``N/record`` module to load a reference to a specific ``Record`` - in this case, the exact same Quote/Estimate from the previous example - and we pass that reference in to ``addRecord()`` via the ``record`` property.

```
renderer.addRecord({  
  templateName: 'quote',  
  record: r.load({ type: r.Type.ESTIMATE, id: 1280 })  
})
```

When we merge a ``Record`` instance into the template, we need to tell the engine which name to assign the data from the ``Record`` instance to using the ``templateName``. Looking at our template, it expects to read data from the Quote record using the name ``quote`` (e.g. ``${quote.tranid}``), thus we set the ``templateName`` to ``quote``.

From there, we render and display the PDF exactly as we have in previous examples.

```
renderer.renderToResponse({ response })
```


Render Search Results

We just saw how to use a `Record` instance as a template datasource, and there are several more options we have for supplying data to our template.

First, we'll look at how to add Saved Search results to our template.

Replace `render-cookbook.js` with:

```
/**
 * Render a PDF with Search Results
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubbaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/file', 'N/render', 'N/search'], (file, render, s) => {
  const renderPdf = (response) => {
    const xmlContent = file.load({ id: './custom-search.ftl' }).getContents()

    const renderer = render.create()
    renderer.templateContent = xmlContent

    renderer.addSearchResults({
      templateName: 'quotes',
      searchResult: findQuotes()
    })

    renderer.renderToResponse({ response })
  }

  const findQuotes = () =>
    s.create({
      type: s.Type.ESTIMATE,
      filters: [
        ['mainline', s.Operator.IS, true], 'AND',
        ['amount', s.Operator.GREATERTHAN, 10000]
      ],
      columns: [
        'tranid',
        'salesrep',
        'entity',
        'amount'
      ]
    }).run().getRange({ start: 0, end: 1000 })

  return { renderPdf }
})
```

Name the following file `custom-search.ftl`, and upload it to the same folder as `render-cookbook.js`:

```
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
<pdf>
  <head>
  </head>
  <body size="Letter">
    <form>
      <h4>Quotes > 10k</h4>
      <p>
        <#list quotes![]>
          <table>
            <tr>
              <th>Quote #</th>
              <th>Sales Rep</th>
              <th>Customer</th>
              <th>Amount</th>
            </tr>
            <#items as quote>
              <tr>
                <td>${quote.tranid}</td>
                <td>${quote.salesrep}</td>
                <td>${quote.entity}</td>
                <td>${quote.amount}</td>
              </tr>
            </#items>
          </table>
        <#else>
          No Quotes > 10k
        </#list>
      </p>
    </form>
  </body>
</pdf>
```

Rendering the PDF results in something like:

scriptlet.nl 1 / 1

Quotes > 10k			
Quote #	Sales Rep	Customer	Amount
QUO00001002	E0004 Krista Barton	C000907 B-Sharp Music	\$16,814.68
QUO00001019	E0006 Mark Grogan	C000903 Jennings Financial	\$10,948.50
QUO00001025	E0001 Eric T Grubaugh	C000675 San Francisco Design Center	\$22,475.00
QUO00001026	E0001 Eric T Grubaugh	C000675 San Francisco Design Center	\$23,975.00
QUO00001033	E0010 Clark Koozer	C000272 Advanced Machining Techniques Inc.	\$39,031.25

Supplying the Search Results

Here we use a similar approach to the ``addRecord()`` example. We load a template file from the File Cabinet and apply it to the contents of our ``TemplateRenderer``. Then, we invoke the [`addSearchResults\(\)` method ↗](#) of the ``TemplateRenderer``.

```
renderer.addSearchResults({  
  templateName: 'quotes',  
  searchResult: findQuotes()  
})
```

The parameters are almost identical to those of ``addRecord``; we pass the Array of ``Result`` instances to render via the ``searchResult`` parameter, and the name by which the template references those results via the ``templateName`` parameter.

I happened to write a separate function which creates an ad-hoc ``Search`` and runs it, but you could instead load an existing Saved Search and run that. The only requirement is that you pass an Array of [`N/search.Result` instances ↗](#) to the ``searchResult`` parameter.

Accessing Search Columns in the Template

Given the ``Columns`` definition of our Search:

```
columns: [  
  'tranid',  
  'salesrep',  
  'entity',  
  'amount'  
]
```

to access the ``Columns`` within our template, we use the ``templateName`` from ``addSearchResults()`` combined with the ``name`` of the ``Column`` instance:

```
<#items as quote>  
<tr>  
  <td>${quote.tranid}</td>  
  <td>${quote.salesrep}</td>  
  <td>${quote.entity}</td>  
  <td>${quote.amount}</td>  
</tr>  
</#items>
```

Summary Limitation / Workaround

Note that it is *not possible* as of this writing (2024.1) to access summarized Columns (groups, sums, etc) from a template. You can work around this limitation using the custom

datasource approach which we will see later. You would run your summarized search, map the ``Result`` instances to plain ``Object`` instances, then render those in the template using the ``OBJECT`` datasource.

Render Query Results

In addition to Search results, we can also use Query results as a datasource for our template.

Replace `render-cookbook.js` with:

```

/**
 * Render a PDF with Query Results
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/file', 'N/render', 'N/query'], (file, render, q) => {
  const renderPdf = (response) => {
    const xmlContent = file.load({ id: './custom-query.ftl' }).getContents()

    const renderer = render.create()
    renderer.templateContent = xmlContent

    renderer.addQuery({
      templateName: 'individuals',
      query: findIndividuals(),
      pageIndex: 0,
      pageSize: 10
    })

    renderer.renderToResponse({ response })
  }

  const findIndividuals = () => {
    const customerQuery = q.create({
      type: q.Type.CUSTOMER,
      columns: [
        { fieldId: 'email' },
        { fieldId: 'firstname' },
        { fieldId: 'lastname' }
      ]
    })

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

    return customerQuery
  }

  return { renderPdf }
})

```

Name the following file `custom-query.ftl`, and upload it to the same folder as `render-cookbook.js`:

```
<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
<pdf>
  <head>
  </head>
  <body size="Letter">
    <form>
      <h4>Individuals</h4>
      <p>
        <#list individuals![]>
          <table>
            <tr>
              <th>#</th>
              <th>Individual</th>
              <th>Email Address</th>
            </tr>
            <#items as ind>
              <tr>
                <td>${ind?counter}</td>
                <td>${ind[1]} ${ind[2]}</td>
                <td>${ind[0]}</td>
              </tr>
            </#items>
          </table>
        <#else>
          None.
        </#list>
      </p>
    </form>
  </body>
</pdf>
```

Rendering the PDF results in something like:

scriptlet.nl 1 / 1

<i>Individuals</i>		
#	Individual	Email Address
1	Iain Bennett	ibennett@netsuite.com
2	Gus Lee	ali@netsuite.com
3	Test	test@test2.com
4	tester1	tester1@tester1.com
5	Gary Underwood	gary@underwoodws.com
6	Frank Edwards	frank@edwards.net
7	Alex Fabre	info@fabreart.com
8	Aaron Abbott	aa@fake-email.com
9	Mike Miller	mm@mmltd.us
10	Greg Muller	greg@originaldraftsltd.net

Supplying the Query

```
renderer.addQuery({
  templateName: 'individuals',
  query: findIndividuals(),
  pageIndex: 0,
  pageSize: 10
})
```

Similar to the Search example, we `load` our template file, assign it as the contents of the `TemplateRenderer`. Then, we call the [addQuery\(\) method](#), providing it with a `templateName` for our Query results and the `Query` Object itself.

```
const findIndividuals = () => {
  const customerQuery = q.create(/*...*/)
  // ...
  return customerQuery
}
```

Note one critical difference is that we do not actually execute our Query; we only create the `Query` instance, and that is what we pass to the `query` parameter of `addQuery`.

Additionally, see how we limit the number of results and/or select a specific page of results to render via the `pageIndex` and `pageSize` parameters.

Accessing Query Columns

```
columns: [
  { fieldId: 'email' },
  { fieldId: 'firstname' },
  { fieldId: 'lastname' }
]
```

```
<tr>
  <td>...</td>
  <td>${ind[1]} ${ind[2]}</td>
  <td>${ind[0]}</td>
</tr>
```

Observe we access Query columns by the order in which they were defined on the Query.

We cannot, unfortunately, access Query columns by their `fieldId` as we can with Search results.

Render Custom Datasources

Finally, we can use several formats of custom datasources for our template:

- JavaScript Objects
- JSON strings
- XML strings
- XML `Document` Objects

Replace `render-cookbook.js` with:

```

/**
 * Render a PDF with multiple custom data sources
 *
 * @NApiVersion 2.1
 * @NModuleScope SameAccount
 *
 * @author Eric T Grubaugh <eric@stoic.software> (https://stoic.software/)
 */
define(['N/file', 'N/render', 'N/xml'], (file, render, xml) => {
  const renderPdf = (response) => {
    const xmlContent = file.load({ id: './custom-datasource.ftl' })
      .getContents()

    const objectData = gatherMetrics()
    const xmlData = `
      <script>
        <name>Render a PDF</name>
        <id>customscript_render_pdf</id>
      </script>
    `

    const renderer = render.create()
    renderer.templateContent = xmlContent

    renderer.addCustomDataSource({
      alias: 'metricsXml',
      format: render.DataSource.XML_STRING,
      data: xmlData
    })
    renderer.addCustomDataSource({
      alias: 'metricsDoc',
      format: render.DataSource.XML_DOC,
      data: xml.Parser.fromString(xmlData)
    })
    renderer.addCustomDataSource({
      alias: 'metricsObj',
      format: render.DataSource.OBJECT,
      data: objectData
    })
    renderer.addCustomDataSource({
      alias: 'metricsJson',
      format: render.DataSource.JSON,
      data: JSON.stringify(objectData)
    })

    renderer.renderToResponse({ response })
  }

  const gatherMetrics = () => ({
    start: new Date(),
    runtime: 7.8903,
    governanceUsed: 883,
    author: ['Eric', 'T', 'Grubaugh']
  })
})

```

```
    return { renderPdf }  
  })
```

Name the following file `custom-datasource.ftl`, and upload it to the same folder as `render-cookbook.js`:

```

<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">
<pdf>
  <head>
  </head>
  <body size="Letter">
    <form>
      <h4>Script Metrics</h4>
      <p>XML string:
        <strong>${metricsXml.script.name}</strong>
      </p>
      <p>XML Document:
        <em>${metricsDoc.script.id}</em>
      </p>
      <p>
        <em>OBJECT</em>
        datasource
      </p>
      <table>
        <tr>
          <td>
            <strong>Start Time</strong>
          </td>
          <td>${metricsObj.start?datetime.iso?string.long}</td>
        </tr>
        <tr>
          <td>
            <strong>Runtime</strong>
          </td>
          <td>${metricsObj.runtime} seconds</td>
        </tr>
      </table>
      <p>
        <em>JSON</em>
        datasource
      </p>
      <table>
        <tr>
          <td>
            <strong>Governance Used</strong>
          </td>
          <td>${metricsJson.governanceUsed}</td>
        </tr>
        <tr>
          <td>
            <strong>Author</strong>
          </td>
          <td>${metricsJson.author?join(" ")}</td>
        </tr>
      </table>
    </form>
  </body>
</pdf>

```

Rendering the PDF results in something like:

Script Metrics

XML datasources

Render a PDF

customscript_render_pdf

OBJECT datasource

Start Time November 19, 2020 8:04:03 PM MST

Runtime 7.8903 seconds

JSON datasource

Governance Used 883

Author Eric T Grubaugh

We have yet another similar API in the [`addCustomDataSource\(\)` method](#). We provide an ``alias`` by which the data will be accessed within the template and the ``data`` to render. Additionally, we provide one of four datasource ``format``s from the [`render.DataSource` enumeration](#).

Rendering data from an XML string

```
const xmlData = `  
<script>  
  <name>Render a PDF</name>  
  <id>customscript_render_pdf</id>  
</script>  
`  
  
// ...  
renderer.addCustomDataSource({  
  alias: 'metricsXml',  
  format: render.DataSource.XML_STRING,  
  data: xmlData  
})
```

In this particular example, we write the XML as a string literal directly within our script, but it could have been loaded from the File Cabinet or retrieved from an external web service.

The source of the XML is irrelevant, so long as we're passing in a ``string`` of valid XML.

```
<p>  
  <strong>${metricsXml.script.name}</strong>  
</p>
```

The template will parse our XML string into an Object structure. We access the data from our string via the ``alias`` we provided, then we navigate down the XML tree like we would an Object's properties.

Rendering data from an XML Document

```
renderer.addCustomDataSource({
  alias: 'metricsDoc',
  format: render.DataSource.XML_DOC,
  data: xml.Parser.fromString(xmlData)
});
```

Alternatively, we can parse the XML string into a proper ``Document`` instance using the ``N/xml`` module's [Parser.fromString\(\) method](#). This might be useful if we needed to transform the XML somehow beforehand, or only pull specific data points out of it first.

```
<p>XML Document:
  <em>${metricsDoc.script.id}</em>
</p>
```

Similarly to the XML string, the template will parse the ``Document`` into an Object, and we access its values via the ``alias`` we provided.

Rendering data from an Object

```
const gatherMetrics = () => ({
  start: new Date(),
  runtime: 7.8903,
  governanceUsed: 883,
  author: ['Eric', 'T', 'Grubaugh']
})

// ...
const objectData = gatherMetrics()
// ...
renderer.addCustomDataSource({
  alias: 'metricsObj',
  format: render.DataSource.OBJECT,
  data: objectData
})
```

We can render native JavaScript Object data. Here the data is static, but - as with the XML - the source is irrelevant. Whether it comes from a function within our script, another module, or an external service does not matter, so long as we pass it in to ``addCustomDataSource()`` as an ``Object``.

To access ``Object`` data within the template, we use the ``alias`` as the name of the object, and then access its properties like we would any JavaScript object:

```
<td>${metricsObj.start?datetime.iso?string.long}</td>
<!-- ... -->
<td>${metricsObj.runtime} seconds</td>
```

``string``s and ``number``s retain their types within the template.

A ``Date`` instance in JavaScript (e.g. ``start``) gets transformed to a [date](#) in the template.

Rendering data from a JSON string

```
renderer.addCustomDataSource({
  alias: 'metricsJson',
  format: render.DataSource.JSON,
  data: JSON.stringify(objectData)
})
```

Whether we parse an Object into a JSON string or receive it that way from, say, an external service, we can render JSON data in our PDF as well.

Accessing the data from a JSON string is identical to accessing ``Object`` data; use the ``alias`` and then access the property by its ``key``:

```
<td>${metricsJson.governanceUsed}</td>
<!-- ... -->
<td>${metricsJson.author?join(" ")}</td>
```

If our Object or JSON contains an Array (e.g. ``author``), it will get passed into the template as a [sequence](#).

Combining Datasources

Note in this example, we add several datasources to our template. I know of no technical limit - although certainly there are practical limits - on the number or the types of datasources you can add to a single template. Using the same template, you could add a set of Search Results, a set of Query results, four transactions, seven Objects, three JSON strings, 27 XML documents ...

Those numbers are arbitrary. Combine and render as many datasources as is useful and actionable to the consumer of the PDF.

Recommendations and Resources

NetSuite Help

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

- [N/render module ↗](#)
- [N/render ↗](#)
- [N/xml.Parser ↗](#)
- [N/xml Samples ↗](#)
- [N/xml Module Script Samples ↗](#)
- [FreeMarker Syntax ↗](#)
- [FreeMarker Data Model ↗](#)
- [Syntax for Advanced Template Fields ↗](#)
- [Using FreeMarker to Work with Hidden Fields Used in Advanced Templates ↗](#)

Big Faceless Org Generator

For details on what markup tags are allowed in your PDF templates, see the [BFO PDF Tag Reference ↗](#).

Apache Freemarker Template Language

For details on the functions and directives you can use in your templates, see the [Freemarker Language Reference ↗](#)

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](#).

