Week 6

Suitelets and Map/Reduce Scripts

Week 6 Overview

You will learn:

1. Suitelets
    a. Usage
    b. Basic functionality
2. Map / Reduce Scripts
    a. Usage
    b. Hands on Exercise

Introduction to Suitelets

What Are Suitelets?

- Suitelets are server-side scripts in NetSuite that allow developers to create custom web pages or extend NetSuite's UI.

- They provide a way to build dynamic HTML, forms, or both.

- Often used for custom workflows, dashboards, or data entry forms.

Why use Suitelets?

- Fully customizable user experience.

- Can integrate with external systems or serve as REST-like endpoints.

- Useful for creating bespoke solutions that standard NetSuite forms cannot achieve.

Examples of Suitelete use cases

- Custom Data Entry Form
  - Enter data, press submit, create associated record(s) in NetSuite
- Generate / Print PDF Forms Using Custom Data
  - Add buttons to transactions that call a suitelet to render custom PDF forms with extra data not natively available
- Custom Approval Workflow Page
  - Create a custom page with business specific approval logic

Commonly Used Modules

- N/ui/serverWidget
  - Create custom UI elements such as:
    - Buttons
    - Fields
    - Sublists
    - Tabs
    - And more!
- N/https
  - Communicate between RESTlets and SuiteTalk REST APIs without having to reauthenticate

Suitelet Structure Overview

```
define(['N/ui/serverWidget', 'N/https', 'N/log'], function(serverWidget, https,
log) {
    function onRequest(context) {
        if (context.request.method === 'GET') {
            // Handle GET request (render the form)
        } else if (context.request.method === 'POST') {
            // Handle POST request (process submitted data)
        }
    }

    return {
        onRequest: onRequest
    }
});
```

Hands-on exercise

The purpose of this hands on exercise is just to familiarize you with handling simple Suitelet functionality including: creating a form, submitting data.

What you will be doing:

- Create a Suitelet that takes a Sales Order and submits a value for a field on the Sales Order
    - For example. You enter the Sales Order ID and a Memo and press submit. That Sales Order will then have the updated Memo.

Solution

Link to Solution:

https://github.com/DerekEsonus/SampleSuiteScripts/blob/main/SampleSuitelet.js

Introduction to Map / Reduce Scripts

What Are Map / Reduce Scripts?

- A type of server-side SuiteScript designed for processing and transforming large volumes of data.

- The data is sourced, grouped, and then processed in defined stages

Why use Map/Reduce when Scheduled Scripts are available?

Map/Reduce scripts are the preferred choice when:

1. The data set is large and may exceed governance limits
2. Tasks can benefit from parallel processing for speed
3. The workflow involves data aggregation

For smaller, simpler tasks, scheduled scripts work great

Examples of Map/Reduce Scripts

- Updating fields across a large set of records

- Exporting large data sets to external systems

Understanding the Map/Reduce Structure

Stages:

1. getInputData - Define the data source

2. map - Process each data point individually

3. reduce - Aggregate results from the map state

4. summarize - Final cleanup and logging

Hands-On Exercise

We want to use a Map/Reduce script to do the following:

1. Search all customers
2. Total up all Sales Order amounts
3. Log the results to the script deployment
4. Log the number of customers we processed
5. Log any customers that encountered errors during the process

The following slides contain function-by-function steps for the above exercise

```
define(['N/search', 'N/record', 'N/log'], function(search, record, log) {
    function getInputData() {
    }
    function map(context) {
    }
    function reduce(context) {
    }
    function summarize(summary) {
    }
    return {
        getInputData: getInputData,
        map: map,
        reduce: reduce,
        summarize: summarize
    };
});
```

getInputData

How it works:

- Acts as the starting point for the script.
- Provides the input data for the `map` stage.

Example:

- If you're processing all customer records, `getInputData` might return a saved search that lists those customers.

```
    // Get the data you are working with.This example is a saved search that gets all sales transactions. Can Add filters for date, location, etc..
function getInputData() {
        return search.create({
            type: search.Type.TRANSACTION,
            filters: [['type', 'anyof', ['SalesOrd']], 'AND', ['mainline', 'is', 'T']],
            columns: ['entity', 'amount']
        });
    }
```

map

Purpose: This stage processes each individual item from the input data one at a time. You can add logic to perform specific actions for each data point (e.g., updating records, transforming data, etc.).

How it works:

- Takes one "chunk" of data from `getInputData` and processes it.
- Outputs intermediate results that may go to the `reduce` stage.

Example:

- For each customer, you might update their email address or calculate a discount.

```
    // Map the data - Group transactions by customer. For other map/reduce scripts you can perform your logic here (for example updating fields on all the records returned by getInputData)
    function map(context) {
        var transactionData = JSON.parse(context.value);
        var customerId = transactionData.values.entity.value; // Customer ID
        var amount = parseFloat(transactionData.values.amount); // Transaction Amount

        // Group transactions by customer ID
        context.write({
            key: customerId,
            value: amount
        });
    }
```

reduce (optional)

Purpose: This stage is for combining or aggregating results from the `map` stage. It's used when you need to group data or calculate summaries.

How it works:

- Processes grouped data from the `map` stage.
- Useful for tasks like summing values, merging records, or generating reports.

Example:

- If your `map` stage outputs sales data, the `reduce` stage could calculate total sales for each region.

```
    // Reduce - calculate the total sales for each customer (since the key in this example
is the customer)
    function reduce(context) {
        var customerId = context.key; // Customer ID
        var totalSales = 0;

        // Sum all amounts for this customer
        context.values.forEach(function(amount) {
            totalSales += parseFloat(amount);
        });

        log.audit('Customer Total Sales', 'Customer ID: ' + customerId + ', Total Sales: ' +
totalSales);
    }
```

summarize (optional)

Purpose: This is the final stage where you clean up, log results, and perform any final actions after all data has been processed.

How it works:

- Runs after all `map` and `reduce` operations are complete.
- Provides access to processing statistics (e.g., how many items were processed, errors, etc.).

Example:

- Log how many records were updated or create a summary report of the processed data.

```
// Summarize - Log the total number of customers processed and any errors.
function summarize(summary) {
    let totalCustomers = 0;

    // Iterate over the keys to count total customers and log errors simultaneously
    summary.reduceSummary.keys.iterator().each(function(key, error) {
        totalCustomers++; // Increment the counter for each key
        if (error) {
            log.error('Error Processing Customer', 'ID: ' + key + ', Error: ' + error);
        }
        return true; // Continue iterating
    });

    log.audit('Summary', 'Total Customers Processed: ' + totalCustomers);
}
```

Link to sample

https://github.com/DerekEsonus/SampleSuiteScripts/blob/main/SampleMapReduce.js