**USER EVENT SCRIPT ASSIGNMENT**

**1. What is user event script?**

1. A User Event Script is a type of script in NetSuite SuiteScript, where we can perform some action when records are **created**, **updated** and **deleted**.
2. It is executed in **NetSuite application Server**.
3. It manipulates the **database** to operate some operations like Create, Edit and Delete.
4. These scripts are triggered when some events are applied on some record like when a user creates a new customer, updates something in the sales order or deletes some invoice.

**2. What is the purpose of user event scripts?**

The purpose of user-event script can be -

1. Let we want to **automate** some tasks when a record like sales order got saved.
2. It will help to **update** some field value based on the client’s requirement.
3. We have to **validate** the data we entered in a field, which should meet the business requirement.
4. It implements real time **data synchronization**.
5. It **triggers workflows** when some event occurs. For ex – A mail should be automatically sent to the customer when the invoice gets generated.

**3. What are the user event script functions?**

There are three main functions in a User Event Script, each of which will get triggered by some certain event that we write in the script.

* ***beforeLoad*** :
  + - It runs before the record is loaded or displayed in the user interface.
    - It helps to modify the fields in a record.
    - It also helps to hide some fields from the user.

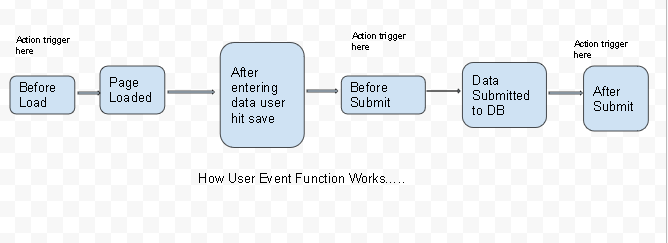
Ex – We can hide some sensitive fields for specific roles or users.

* ***beforeSubmit*** :
  + - It runs before the record is saved in the Database.
    - We use this function to validate or modify some data before it gets saved to the database through NetSuite application server.

Ex – We can validate all the fields we made the change upon using certain criteria.

* ***afterSubmit*** :
  + - It runs after the record is saved in the Database.
    - It is used to trigger some process or actions after the record gets saved in the function like sending mails.
    - Ex – Sending an email after a sales order is created.

You can refer to the image below for better understanding.



**4. Can we call a user event script from other scripts?**

Yes, we can call a user event script from other scripts as well but there are some limitations over there.

* We generally know that user event script gets triggered when we do some operation like create, edit, delete with the record and they are not designed to directly invoke as we invoke a function in our code.
* The user event script is typically associated with specific actions with the record that we do like beforeSubmit, afterSubmit etc.
* So, if you are working with multiple scripts, make sure to manage the module, entry points and dependencies correctly using the define function in SuiteScript 2.X.

By following these steps, we can indirectly call the user event script.

**5. Can we execute user event script as admin role?**

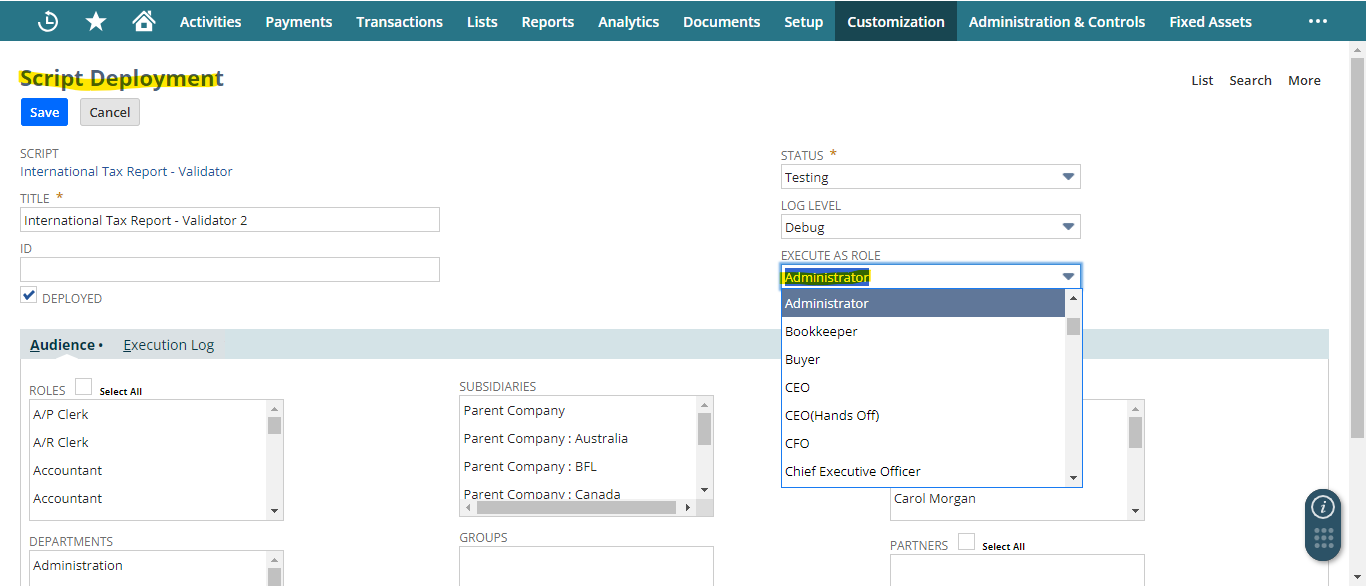
Yes, we can execute a User Event Script as then admin role in NetSuite.

* User Event Script runs in the context of the user who triggered the event. So, if a non admin user performed some action like create, update or delete, that will also execute by the user’s permission only.
* We can set the execution role for the script deployment to Administrator during the script deployment process.

***Step 1*** – Go to Customization -> Scripting -> Script Deployments.

***Step 2*** – Edit the deployment record for user event script.

***Step 3*** – In the Execute as role field, select Administrator and click on Save.



**6. What is the governance limit for user event scripts?**

* The governance limit for a User Event Script is **1,000 units.**
* If the script exceeds this limit, it will throw a “**SSS\_USAGE\_LIMIT\_EXCEEDED**” error.
* To stay in the limit, optimize the script as much as you can, remove unnecessary operations.

**7. How can we create custom button using user event script?**

***Step 1*** - Here I used beforeLoad entry point so that the button will be visible once the record load.

***Step 2*** - Make sure It should be loaded in CREATE mode, so I used UserEventType.CREATE .

***Step 3*** - I used scriptContext.form to select the current form.

***Step 4*** - Then I have addButton API to create a new button in the current form

***Code:***

function beforeLoad(scriptContext) {

var rec = scriptContext.record;

if(scriptContext.type === scriptContext.UserEventType.CREATE) {

var form = scriptContext.form; //

form.addButton({

id: 'custpage\_buttonid',

label: 'Click Me'

});

}

}

return {

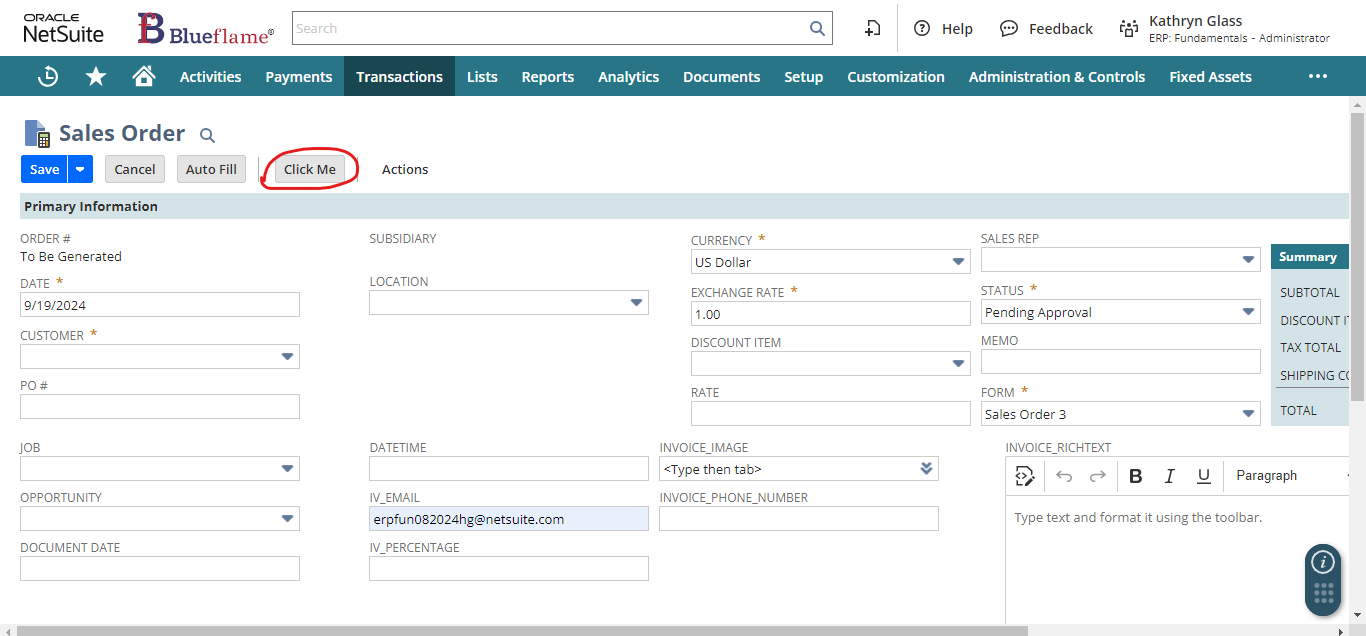
beforeLoad: beforeLoad,

beforeSubmit: beforeSubmit,

afterSubmit: afterSubmit

};

***Output:***



**8. How can we prevent users from submitting a record?**

To prevent a user from submitting a record, we can use beforeSubmit entry point and throw an error when certain condition meet.

***Step 1*** - I have used **beforeSubmit** entry point, so that it will prevent the user from submitting the record.

***Step 2*** – I used **N/error** module to create your own custom SuiteScript errors.

***Step 3*** – There is a **create** API that helps to Creates a new error.

***Code:***

define(['N/record', 'N/search', 'N/error'],

(record, search, error) => {

const beforeSubmit = (scriptContext) => {

log.debug('beforeSubmit triggered ✅');

if (scriptContext.type === scriptContext.UserEventType.CREATE ||

scriptContext.type === scriptContext.UserEventType.EDIT) {

log.debug('Error is showing');

throw error.create({

name: 'MY\_ERROR\_CODE',

message: 'This is my error details.',

notifyOff: true

});

}

else {

log.debug('Error is not showing');

}

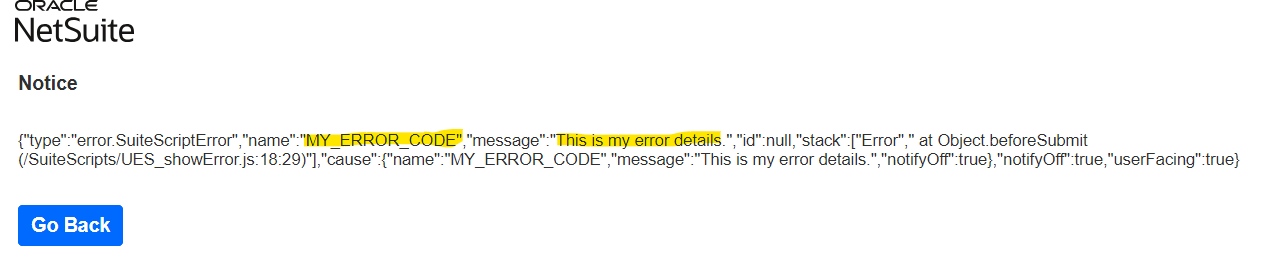
}

return {

beforeSubmit: beforeSubmit

};

***Output:***



**9. Is there any limit for the number of user event scripts that can be deployed in one record?**

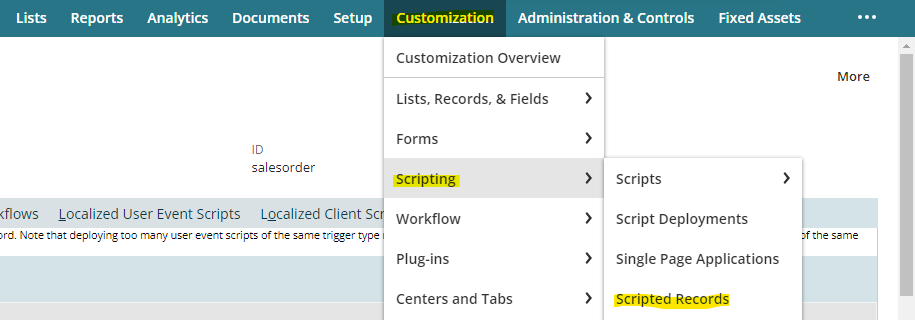
No, there is no limit explicitly available on the number of User Event Script that can be deployed on a single record. But we have to keep some points in our mind before we do so.

* If multiple user event scripts are deployed in the same record, we may not control the flow of execution.
* It may show some unexpected behavior.
* Deploying too many scripts may negatively impact performance.
* If multiple scripts use multiple APIs, then the collective governance unit will also increase. So, we may face governance errors.
* So, we can inactive the script that we no longer use.

**10. How to execute one user event script after or before another?**

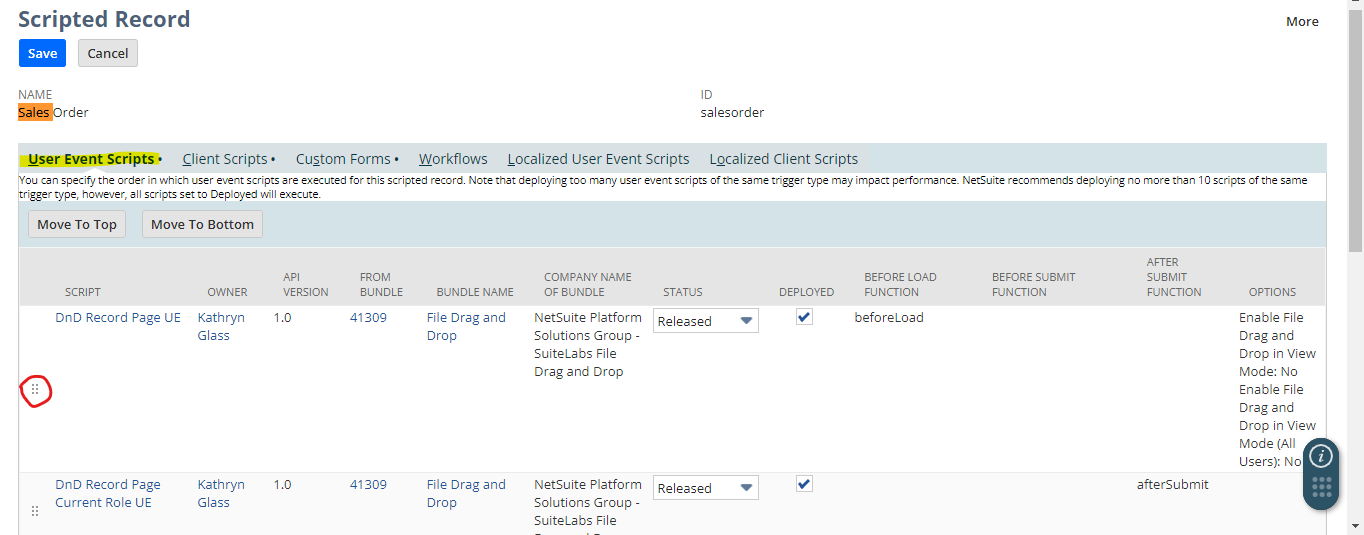
We can reorder the scripts file in the User Event Script in the way we want to execute all. To do so follow the steps below.

***Step 1*** – Navigate to Customization -> Scripting -> Scripted Records.



***Step 2*** - Now search the record (for ex – Sales Order) and click on Edit.

***Step 3*** - Now under User Event Script subtab, click and hold the red circular mark script and reorder according to you.



**11. Can I attach a client script in a user event script?**

No, we cannot directly attach a Client Script within a User Event Script. But we can load it dynamically and attach the client script in the user event script using the **form.clientScriptField** or **form**.**clientScriptModulePath** properties within the **beforeLoad** function of the User Event Script.

**12. How can we get the execution context?**

* We can get the execution context by using the **N/runtime** module which helps us to view runtime settings for the script, the session, or the user.
* Then we can use **runtime**.**executionContext** API trigger on the current script. which helps us to provide information about how a script is triggered to execute.
* It runs both on Client Script and User Event Script.

**13. Is there any API that can't be used in a user event script?**

As we all know the basic concept, i.e. if we won't import the module, then we can't use its APIs. So, there are certain modules that we can't import while we are working with user event script.

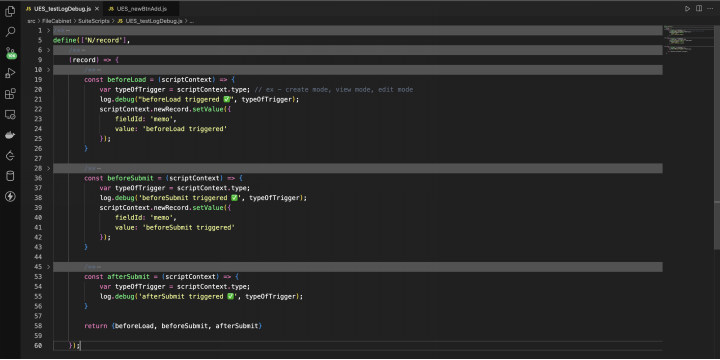
i.e. - **N/currentRecord, N/message, N/dialog**

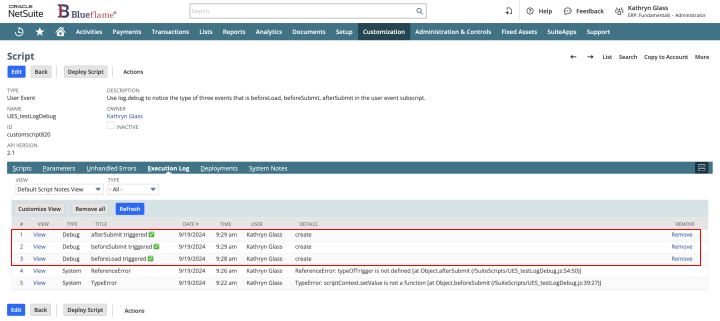
**14. Use log.debug to notice the type of three events that is beforeLoad, beforeSubmit, afterSubmit in the user event subscript.**

Here I have implemented the three entry points of user event script and deployed this on ‘invoice’ record in the memo field. Then I used **scriptContext.type** to see in which mode we are accessing that record.

Now, we can see everything under execution log subtab in that particular script.

***Code:***

***Output:***



**15. Differentiate newRecord and oldRecord.**

***newRecord:***

* It is the record that is being currently processed.
* It is used in all triggered types such as **CREATE**, **EDIT**, **DELETE**, **VIEW**.
* It represents the new values in the fields that are not yet committed to the database.
* Holds the newRecord data.
* It allows modification of the field values.

***oldRecord:***

* It is the record that represents the previous step of the record.
* It is used only during **EDIT** and **DELETE** events.
* It represents the saved value in the database before any changes.
* Holds the oldRecord data.
* It allows only read-only access to compare the previous field values.

**16. Email and Phone number of customer should be populated on the transaction before the loading of the record.**

Here I used **N/record** and **N/search** module to find that particular record in the context and search module to Search for a single record using keywords.

-> I also used an API called **lookupFields** from search module that Performs a search for one or more body fields on a record. Returns select fields as an object with value and text properties. Returns multiselect fields as an object with value:text pairs.

const beforeLoad = (scriptContext) => {

log.debug('beforeLoad triggered ✅');

var current\_record = scriptContext.newRecord; // Get the current record. i.e. invoice

// var customerId = current\_record.getValue({ // Get the customer

// fieldId: 'entity' // 'entity' is the field where the customer is stored

// });

var customerId = 1126;

log.debug('Customer ID:', customerId);

if (customerId) {

log.debug('Customer found ✅', customerId);

var fields = search.lookupFields({

type: search.Type.CUSTOMER,

id: customerId,

columns: ['email', 'phone'] // Fetch email and phone, it returns an object

});

log.debug('Customer Fields (Email and Phone):', fields); // Log the returned fields for email and phone

// Set the email in the custom field

current\_record.setValue({

fieldId: 'custbody3',

value: fields.email || '' // if email is available then set, else set empty string

});

// Set the phone number in the custom field

current\_record.setValue({

fieldId: 'custbody7',

value: fields.phone || ''

});

} else {

log.debug('Customer not found ❌');

}

}

**17.** **Similar to the above one but now it should get populated dynamically after the record is submitted.**

* This script runs after the record has been submitted (afterSubmit), so the email and phone fields are dynamically populated based on the selected customer.
* The script fetches the customer ID from the entity field, which holds the customer on the transaction (eg. Invoice)
* It then uses **search.lookupFields()** to retrieve the email and phone of the customer dynamically.
* The **record.submitFields()** method is used to update the transaction fields (e.g., custbody3 for email and custbody7 for phone) with the fetched customer information after the record is saved.

***Code:***

const afterSubmit = (scriptContext) => {

if (scriptContext.type !== scriptContext.UserEventType.CREATE &&

scriptContext.type !== scriptContext.UserEventType.EDIT) {

return;

}

log.debug('afterSubmit triggered ✅');

var current\_record = scriptContext.newRecord; // Get the current transaction record (e.g. Invoice)

var customerId = current\_record.getValue({ // Fetch the customer ID from the 'entity' field (the customer on the transaction)

fieldId: 'entity' // 'entity' is the field that holds the customer

});

log.debug('Customer ID:', customerId);

if (customerId) { // Checking if a customer is selected or not

log.debug('Customer found ✅', customerId);

// Lookup customer's email and phone using the customer ID

var fields = search.lookupFields({

type: search.Type.CUSTOMER, // Customer record type

id: customerId, // Use the dynamically fetched customerId

columns: ['email', 'phone'] // Fetch email and phone

});

log.debug('Customer Fields (Email and Phone):', fields);

record.submitFields({ // Update the transaction with the customer's email and phone

type: current\_record.type, // The type of the current transaction (e.g., Sales Order or Invoice)

id: current\_record.id, // The internal ID of the current transaction

values: {

custbody3: fields.email || '',

custbody7: fields.phone || ''

}

});

log.debug('Email and Phone updated ✅');

} else {

log.debug('No customer found ❌');

}

};

**18. Make the above question dynamic (customer chosen's email and phone number needs to be get reflected).**

* I have used **beforeSubmit** event, which occurs before the record is saved (during creation or editing).
* When the user selects a customer, the script retrieves the **customer ID** and looks up their **email** and **phone** using **search.lookupFields.**
* It sets the retrieved **email** and **phone** values into the respective custom fields (custbody3 for email and custbody7 for phone) before saving the record.

***Code:***

/\*\*

\* @NApiVersion 2.1

\* @NScriptType UserEventScript

\*/

define(['N/record', 'N/search', 'N/log'],

(record, search, log) => {

/\*\*

\* Populates the customer's email and phone before the record is saved.

\* @param *{Object}* *scriptContext*

\* @param *{Record}* *scriptContext.newRecord* - The current record (invoice, sales order, etc.)

\* @param *{string}* *scriptContext.type* - The event type (create, edit)

\*/

const beforeSubmit = (scriptContext) => {

if (scriptContext.type !== scriptContext.UserEventType.CREATE &&

scriptContext.type !== scriptContext.UserEventType.EDIT) {

return;

}

log.debug('beforeSubmit triggered ✅');

var current\_record = scriptContext.newRecord; // Get the current transaction record (e.g., Invoice or Sales Order)

var customerId = current\_record.getValue({

fieldId: 'entity' // 'entity' refers to the Customer field on the transaction

});

log.debug('Customer ID:', customerId);

// If a customer is selected, retrieve their details

if (customerId) {

log.debug('Customer found ✅', customerId);

// Fetch the customer's email and phone using lookupFields

var customerFields = search.lookupFields({

type: search.Type.CUSTOMER, // Customer record type

id: customerId, // The customer ID

columns: ['email', 'phone'] // Fetch email and phone

});

log.debug('Customer Email and Phone:', customerFields);

current\_record.setValue({

fieldId: 'custbody3',

value: customerFields.email || ''

});

current\_record.setValue({

fieldId: 'custbody7',

value: customerFields.phone || ''

});

log.debug('Email and Phone fields updated before save ✅');

} else {

log.debug('No customer selected ❌');

}

};

return {

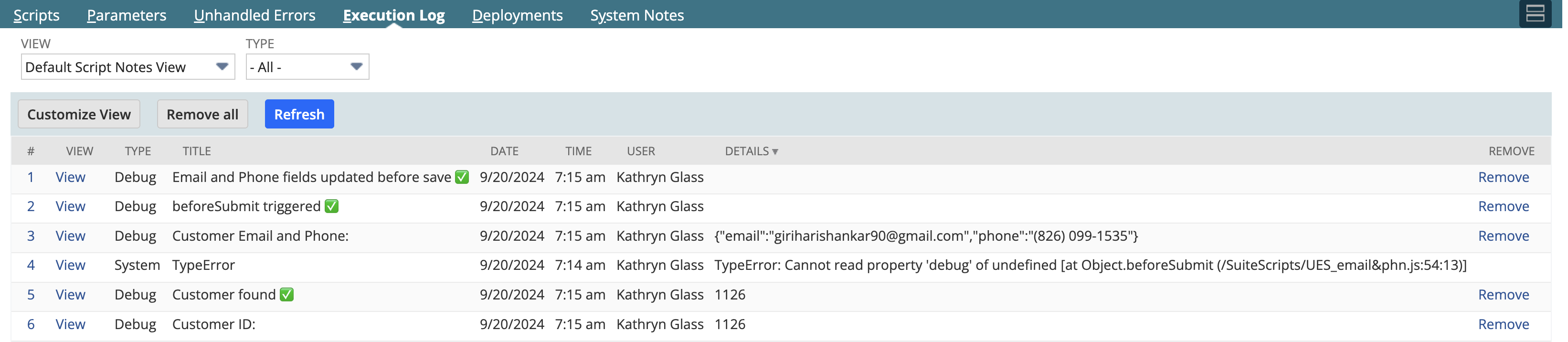
beforeSubmit: beforeSubmit

};

}

);

***Execution Log:***



Now you can see that once I select the customer in the entity field, the email and phone of that customer auto populated dynamically.

***Output:***