**Welcome** to module 4, Understanding Entry Points. In this module, we're going to look at the different entry points that would get triggered depending on the user action. Now, it's important for us to know what those different events are so that we can control how we can customize our script.

Now here, we're going to look at how we can understand the different kinds of entry points. Now, we're going to concentrate on client side entry points in this module, but basically, understanding how entry points work would be very important for us as developers, because we need to know when our script would be triggered exactly, and what kind of actions the users need to perform to trigger those scripts.

So we're going to look at, also, how we can debug client side scripts because so far, we looked at server side scripts and how to debug it. We also need to know how you can do that on a client side.

So what is an entry point? Again, like what I've mentioned, entry point is simply a trigger on when your script would execute. Now, depending on the kind of action that the user would do, either on the client side or in the browser directly, or on the server side, like before a record is [? closed ?] or after a record is saved, that would determine what kind of function will be triggered.

Now, if you look at this diagram, you can see that there are several points of actions where your script might be triggered. So for example, the first action that the user might do is that the user would open an existing record or maybe create a new record. So every time you perform an action on a record, either by creating one or by loading one, it'll always trigger your before load.

And like what I've mentioned earlier, beforeLoad is the first event that would get triggered on your script, and that's if you are manipulating your record. Now, after a record is loaded or after the script processes on the server, the server would be sending information to your browser, and then your browser page would load. Now, as soon as your browser page would load, that would go into the pageInit action.

Now, as you can see here, the pageInit is happening on the client side. That means that the browser is directly executing your script. Unlike with the beforeLoad or beforeRecordLoad, where your script would be executing on the server and the NetSuite server would be running your script, client side scripts would get triggered directly by the browser.

So that means that you have to be careful with things that you put in on the browser, because the behavior might be different depending on the kind of browser that you're using. And that's kind of how JavaScript really works-- you have to be careful of the browser, because sometimes the browser behavior changes, especially with, like, Internet Explorer. Nowadays, most of the browsers actually give the same behavior, so it's not really that big of a deal compared to a few years ago.

So anyway, so we have beforeLoad on the server side, then pageInit as soon as the page completely loads. Now, once the page loads, of course, the user would do a lot of things with the form. Maybe change some values on the form, maybe delete some values or things like that. So these actions would be fieldChanged, validateField, and postSourcing. We're going to talk about those different actions in a bit. But basically, these actions would be triggered whenever the user does something to a field.

Now, once the user completes the changes that they want to make on the form, the next thing that the user does that the user saves the record. So that's the next action. So once the user saves the record, the user can do that by clicking on the Save button. So that's another entry point or trigger that would execute your script, and this would get executed as soon as the user clicks on the Save button.

So as soon as the user clicks on Save, it'll trigger your saveRecord entry point. And this would be useful if you want to add some validation or if you want to perform a computation before you save the record, or things like that. You can put that on saveRecord.

Now, after the saveRecord, all the information that was processed in the browser would be transferred to the server, and it will trigger now your beforeSubmit. Now, after triggering your beforeSubmit, here you can add additional validation if you want or additional computation. The information would then be processed by the server. It will be saved to the database, and then it's going to trigger afterSubmit.

So so far, we've already seen the beforeLoad event or the beforeLoad entry point, and then we've also seen the afterSubmit entry point. But basically, this is the trigger cycle that's going to happen on your entry point.

Now, to look at it a different way, so this is another illustration that pretty much shows the same thing. Now, what you have to remember here is that whenever you are triggering an event, everything that is happening on the server side-- remember that this script would be running on the server. It's running on one specific environment, which is the serve environment.

Now, anything that's running on the client side, be careful here, because remember, this would be running on the browser, and the browser behavior might change. Now also, try to notice here that in your server side or on your user event, there are two triggers or entry points that you can execute whenever you save the record.

So for example, here on beforeSubmit, you have beforeSubmit and afterSubmit that will be triggered both when you save the record. So what is the difference between beforeSubmit and also your afterSubmit? Now, the difference between beforeSubmit and afterSubmit is the action of the system that saves to the database.

If you've heard of the term transactional rollback, that is usually a process that other systems employ so that just in case a process breaks in the middle of a transaction, all of the changes that were made by your automation would be rolled back so as if that action has not happened. OK?

Now, in NetSuite, we don't really employ transactional rollback, but what we do is that we have two events that you can use to make sure that your changes are clean before actually pushing that to the database. And that's why we have two events, the beforeSubmit and afterSubmit.

So if there's anything that you need to do, for example, validation or anything that would allow you to check your data so that you can verify the data or the information is intact before you save to the database, you would put that on beforeSubmit. Now, if you want to finalize your changes-- for example, if you want to send a notification or maybe you want to change the status or things like that-- then you would put that on afterSubmit.

Now, if you think about it, if we only had one and we didn't have any transactional rollback, then that would be a problem, because if we did not have afterSubmit and we only had beforeSubmit, what if you had an order that you want to send a confirmation for? On beforeSubmit, maybe you sent your customer a notification saying that, OK, we received your order and we're already processing your order. And then your script breaks on beforeSubmit.

Now, the thing here is that your customer already received a confirmation saying that, OK, the order has been processed. But in fact, it has not been processed yet because the process broke at this particular point, meaning it has not even reached the server yet. So that will be very problematic.

Now, if you have an afterSubmit and you did not have beforeSubmit, and let's say you are doing a process where you are maybe adding a validation to your script-- so let's say you want to make sure that the data is correct before it enters the database, now, that really wouldn't work, because you've already passed the database. Even if you added the validation on afterSubmit, doesn't matter, because your data has already been saved.

So again, that's kind of why we have those two events, OK? So beforeSubmit, again, for validation or if you want to perform any computation, or just making sure that the data is clean before pushing to the database. And afterSubmit, so anything that is involved with finalizing your transaction, like maybe if you want to send an email, set the status to completed, or anything like that.

Also, aside from bearing beforeSubmit and afterSubmit, there are also some actions that you can compare-- for example, beforeLoad and pageInit. So these two things are similar in a way that, for example, if you want to set values, you can set the values for default values on both beforeLoad and pageInit. It just really depends on you and where you want to put that. Just remember that beforeLoad is executing on the server, and then pageInit is executing on your browser or on the client side.

Same thing with the saveRecord. So you can use saveRecord for validation, as well as beforeSubmit. The difference between the saveRecord would be the kind of message that you can display on the screen. On a saveRecord, since that's running on the client side, you can display a pop-up on the browser because remember, it's running directly on the browser. beforeSubmit, on the other hand, would be running on the server, so you can't really display a pop-up, because it's already on the server. You've already submitted that.

Now, if you want to add a validation here, what the server would do is that on beforeSubmit, instead of going down to the database, they'll actually transfer the information back to the client, and you will get a page that will tell you that there's an error. You have to deal with that error before submitting that form again.

So again, those are the entry points, and it's very important for you to know the cycle because if you are going to trigger your event, you need to know where that event is triggered. So beforeLoad, pageInit, and then your field level events, you will have your saveRecord, beforeSubmit, and then it goes to the database and then goes down to afterSubmit.

So in this walkthrough, what we're going to do is that we're going to script for client side entry points. We've seen all of the entry points and we've seen how we can do that on a user event script. So we're going to try to change that up a little bit by using client side script.

So what do we want to do here is that we want to copy the fax number to the phone number field, if the phone number field is empty. Another thing that we want to automate is that we want to validate the coupon code value whenever the user saves the record. And then lastly, we're going to extend that so that we are going to also validate as soon as the user types a value in on the employee code.

Now normally, if you have a validation, you wouldn't really be duplicating validation on user save or when the user is typing the record. But in this case, we want to explore the different kinds of events, and that's why we're adding those two validations. Now, what we want to learn in this walkthrough is that we want to learn how we can trigger client side scripts, specifically the field change event, the save record event, and also the validate field event. And we're also going to look at how we can debug client side scripts.

So let's go ahead and build our script. So we don't need this user event script anymore, and this time, we are going to build a client side script. So for the script type, I am going to select client side script. And then for the filename, since this would be also deployed to my employee record, I'm just going to use similar naming conventions, so sdr\_cs-- because we're building a client side script-- underscore employee.

When I click on Finish, you notice that there are a lot of information here and the template has already been provided to us. Now, what we want to start building would be first our fieldChanged. So we already know that there are some functions that we're going to use. So for the functions that we're not going to use-- for example, these things-- I'm just going to comment these out.

Now, the reason why I'm commenting this out is because once I start uploading my file to the server, my script record would identify these scripts, and I don't want the system to look at these scripts and say, OK, I do have a pageinit event, but I'm not really doing anything with that pageInit event because that's a waste of resources. And also, as a developer, I don't want to get the impression that I actually have a function when in reality, I'm not implementing that function. So I want to comment that out.

Going back to our function, I'm going to start with my fieldChanged. Now, if you remember with our discussion, the fieldChanged event would get triggered as soon as the user leaves the field or as soon as the user finishes typing on a field. OK? So what we want to do first is that we want to get our employee record. So our employee then equals to context.

By the way, so here, instead of using context, I should use scriptContext, but I find that scriptContext is a little too long, so maybe I'll just rename this the context so I can just use context in my actual implementation. So anyway, so here we have context.

And since we're working on a client side script this time, instead of using newRecord-- so if you remember, previously we were working on newRecord-- and that will only be on the server side, specifically for a user event script. But this time, since this is not a user event script, this is a client side script, then I would be using currentRecord. OK? So the property will change depending on the function or the script type that you're using.

Now that we have our script record, the next thing that you have to do is that you have to add in a filter so that you can check which field the user is currently manipulating. Because if you don't, what will happen is that the system will try to execute your script, regardless of the field that the user is changing. For example, if the user is changing the email address or the phone number or any other field, then the script would be open.

Now, if that's the case, the system would get an error saying you have an overload or some kind of similar error, because the system really doesn't know when to execute your script. So what I need to do here is that I'll just add an if statement, and then on that if statement, I need to extract the field that the user is currently using from my context object.

So here, I'm going to say context.fieldId, and then on that field ID, I'll just add in a condition to say, OK, I'm checking for this particular field. Now, what we want to do is that we want to automatically copy the phone number field to the fax field if the fax field is empty. So if the user types in a phone number, then that phone number would be copied to the fax field. So here, since the user will be changing the value of the phone number field, I need to make sure that I filter based on the phone number field.

And if I want to add in an automation on another field, I can add another if statement and say, if context.fieldId. So for example, if the user changes the supervisor field, something else happens, then I can just type in supervisor, and then this would do something else. Now, in our case, we really don't need this, so I'll just remove that.

And then for the field, I want to get the value of the fax field, because we only want to perform this automation if the fax field is empty. If it's not empty, then we don't want to touch it. To get the fax field, we already know how to do that. We have our employee object .getValue, and then fax.

Now that we have that value, we can check. So if-- OK? Now, in this case, this is something that's very interesting with JavaScript. So JavaScript has something that's called the falsy check, where it will try to check if the value is zero, false, undefined, not a number, or an empty string. Now, if you want to get that falsy check, so just in case you want to know if the value is null, then what you can do is that you can just say exclamation point and then fax.

So here, I'm saying that if my fax number is not empty, or is not filled out, then what I want to do is that I want to grab my phone number value, so employee.getValue, then here I want to get my phone number. And then once I get my phone number, I'm just going to set that value, employee.setValue. I want to set my fax number value to my phone field. That's it.

If you want to know more information about the falsy check, we do have information on that on the Appendix. So if you go to your student guide, jump over to Appendix A and look for the truthy and also the falsy checks. So that's it. So let me upload this script. And then let's go back to our account here, and-- yeah, we don't need that debugger anymore.

So I'll just create a new script file. And then here, I'm going to select the script that I've uploaded, sdr\_cs\_employee, create my script record. So for this one, I'm going to name this SuiteDreams CS Employee. So naming convention, pretty much the same-- cs\_employee-- and then also deploy this to my employee record. So for the ID, I'll just copy this. Paste that right over here, click on Add, and then click on Save.

Now, on our deployment record, you can see here that I have defined something for my saveRecord, validateField, and fieldChanged, when in reality, the only thing that I have actually implemented is the fieldChanged event. But since we're also going to use the two, we've added those two definitions.

But if you don't comment out some of the functions-- for example, if I just kept all of the function definitions here, if I scroll down at the bottom, if I did not comment this out, then this would just all be checked. And again, this can be confusing to any developer who would be picking this project up.

So let's say I did not create this, but this was assigned to me so that I could maintain this, then I would be confused and say, OK, it looks like I have a pageInit event, but in reality, if I look at my script, then I really don't have a pageInit event. So that's something that you want to make sure that you avoid.

So let's go ahead and test this. Let me open an employee record. In fact, we can try to just create a new one. Let's just do that. On this new employee record, so if I go to my phone number field, so if I type in 555-5555, and the field change event would be triggered as soon as I move away from the field. So from my phone number field, if I click on any other field-- let's say if I click on my mobile phone field, you can see that the fax number value has automatically been copied.

Now, if I try to change this value to, say, 111-1111 and if I click on, say, my email field, you can see that the fax number field was not changed because we added a condition to say that the fax field value would only be changed if the fax field is empty. OK? So the field change event would be very convenient if you want to change the behavior, whereas a field would automatically trigger an action after a user changes the value of a field.

Now, the next thing that we're going to do here is that we're going to see what's going to happen or if we want to add an automation when the user clicks on the Save button. So as soon as I click on Save, I want to trigger an action. You can do this for validation or you can do this for maybe if you want to automatically compute for certain field values. Or anything that you want to happen when the user saves the record, we can put that in our Save Record event.

Now, going back to our script here-- and by the way, what I did was that I commented out all of the lines so that we can do that one function at a time. Now, moving on here to our script record, so I can just uncomment that out and then go to my saveRecord definition over here. OK?

Now, like what we did in our fieldChanged, so I want to get my employee record reference, employee equals to context, I renamed this to make it shorter. You can just use scriptContext if you want to. So here, context.currentRecord so that we can get that employee record. And then the next thing that I want to do is maybe I want to validate my employee code value. So that's one of the requirements that we have in our walkthrough.

So if I want to validate my employee code, well, first I need to get my employee code value. So empCode, this equals to employee.getValue. And then I need to get the internal ID of my employee code value. So open the field level help, copy that ID, and then paste that inside my code. OK.

So now that I have that employee code value, what I need to do is that I just need to add in an if statement so that I can validate. So here, if and then I could say if empCode, this equals to-- I can't think of any invalid employee code value. So maybe my invalid employee code value is x. OK? So I'll just use that.

And then-- so if the employee code value is x, then I want to add in an alert statement to say-- just maybe say that "Invalid Employee Code value." Something simple like that. Maybe just add in "Please try again."

Now here's the key if you are going to use a saveRecord function. So with a saveRecord function, you need to make sure that you return a Boolean value. Return true if you want to allow the user to save the record. If you want to prevent the record from saving the record, you have to return false. Now, in this case, since the user has typed in an invalid employee code value, I'm going to say to return false, and then at the bottom, I'm going to say otherwise, return true, or just allow the user to save the value.

So let me upload this and then go back to my form, and then I'll refresh my form. Now, something that you also have to remember if you're dealing with client side scripts is that if you're going to refresh the form, refresh the form by pressing Control-F5 if you're using Windows, or if you're using a Mac, press Command-R.

The reason why you want to do that is because sometimes the browser doesn't really get all of the files from the server. So instead of getting the files from the server, the browser would be just getting information from the cache, especially if the page did not significantly change.

So sometimes what happens is that if you just press F5, then the browser picks the script from the cache, then you have an older copy of the script. And you might be thinking, OK, why isn't this working, when in fact, it's just working. It's just that it's an older version of your script. So again, make sure you press Control-F5 if you're using Windows, or Command-R if you're using a Mac.

So anyway, let's see if our validation works. Let me scroll down here and let me set the subsidiary value, because that's required. For the employee code value, I'll use x. So again, x would be our invalid value. And I'll go up here and put a name. I'll just say Test 1 User. OK. Then I'll click on Save.

And as soon as I click on Save, as you can see, I have a pop-up saying "Invalid Employee Code value. Please try again." So if I click on OK, I am back to my form. If I click on OK, again, I would not be able to save this record, because on my script, I said that OK, return false or prevent the user from saving if they use an invalid value for my employee code.

Now, if I use the proper value-- let's say I put this as EMP001. And if I save this, it allowed me to save the value. OK, so it looks like we have an error here. Oh yeah, so this is just an issue with the old script. So we can just ignore that for now.

So anyway, if I go to my records, so if I look for my test user, as you can see, I was able to save my user information. So here, Test 1 User, and we were able to properly perform the save. OK? So if I edit this, here you can see that it allowed me to save EMP001. But if I change that again to x, I click on Save, I get that validation.

So that's it. So one fieldChanged is used if you want to automate an action after you change the value of a field, and then saveRecord would be used if you want to trigger something, either a validation or any kind of action as soon as the user clicks on the Save button.

Now, the last thing here that we have would be the validateField event. So for my validateField event, I'm going to uncomment this, and then go to my validateField function definition. So on my validateField, again, I want to just change this to make it shorter. So here, employees equals to employee.currentRecord. OK.

And what we want to do here is that I'll just do something similar to my saveRecord event, where after the user types in an incorrect value, which is x, then it would just display that error message. Now, the difference between validateField and saveRecord is that validateField would get triggered when the user moves away from the field, kind of similar to your fieldChanged. And it's also similar to your saveRecord because you can allow or prevent the user from moving away from that field if they typed in an incorrect value.

Now, you can use this if you have a field that is very important that they need to make sure that they properly fill out. Otherwise, there might be something wrong with the business process or that might affect the business process. So for example, if you have a field that says this is my identifier for this particular item, and they have to make sure that they use the proper value for that identifier, otherwise, it might cause confusion when the item gets picked up from the warehouse or something like that.

So if that's the case, then you want to make sure that you add a validateField. Now, a validateField would be similar to a combination between a fieldChanged event and also a saveRecord event. It's similar to a fieldChanged event because like all other fields, you need to make sure you add in a filter so that you know that the user is modifying a particular field.

Now, in this case, since we want to trigger this when the user is changing the employee code value, I can add then if context.fieldId, and then I can just check, OK, if the user is modifying my employee code-- so here, I'll just grab that employee code value and then put that right over here. OK.

And again, if the user is modifying the employee code value, then first, get my employee code value. So this is just a simple employee.getValue call. Sorry about that. So getValue, and then want to get that, paste that right over there. OK.

Now that we have our employee code value, we can just use the same validation that we used in our saveRecord. I'll just copy this, including my return statement, then put that right over here and maybe just add in that return statement at the end of my condition.

OK? So that's it. So again, as you can see, it's kind of like a combination of my fieldChanged saveRecord. So I have an if statement, the filter that this would only execute if the user is changing the value of my employee code. If they typed in an incorrect value, prevent the user from moving away from this field. They have to fix this first.

And again, this can be annoying if you have a lot of fields like this. But if this is a very important field, you have to make sure it's populated. Otherwise, I'm saying return true and allow the user to move away from that field.

So let me upload this script and let's see if this works. Let me check if I uncommented that. OK, that looks good. And let me refresh this. Again, I'm refreshing by pressing Control-F5 so that we can get the latest copy of the script. So moving down here, I'm going to change the employee code value. I'm going to use x, which is an invalid value. And then I'm going to click on, say, the office phone number.

So if I click on that, you can see that-- OK, it looks like I might have mistyped something on my script. Let me just go ahead and fix that really quick. employe dot-- oh, not employee, but context.currentRecord. There I go. Let me just go ahead and fix that OK, let me refresh this again. OK, so just try that.

So for my employee code value, I'll type in x, and then I'll click on, say, the fax number. Then as soon as I click on the fax number, notice here that I have that message appear. So "Invalid Employee Code value. Please try again." If I click on OK, it'll go back to this field. And it'll always return to this field and it would not allow me to move to any other field. For example, if I go to my phone number field, I click on phone, gives me that error message. And then the focus goes back to my employee code.

If I click on the fax number, displays my message. If I click on OK, goes back to my employee code. But if I change this to a valid value, not even EMP001. I can just say 123, because the only thing that we're validating for is x. So if I click on, say, office phone, you can see now I'm able to go to my office phone field.

So that's an event that you can use. Again, you can use the validateField if you want to validate a very important field. Or if you want to validate multiple fields at the same time, instead of using validateField, then I would recommend that you use saveRecord, because it'll be easier for your users to use saveRecord, or it'll be less annoying for your users to use saveRecord.

Because if you validate one field after the other and if it's not really that important-- like, if they skip that, it's not going to affect the business process adversely-- then just put that in the saveRecord. Because if they keep seeing a pop-up on the screen like, this is an incorrect value, this is an incorrect value, this is an incorrect value, that's just really annoying for a user and you don't really want that. You want to make sure that your automation will be as user-friendly as possible.

So that's about it for this walkthrough. Let's do a review of the different entry points that we have in our client side script and user event script. So remember that whenever the user does something to the record, either open the record by viewing the record or editing a record or creating a new record, it will always trigger your beforeLoad. So beforeLoad would happen on the server before the information gets transferred to your browser.

Once the information gets transferred to the browser, then it would use pageInit if you are either creating a record, copying a record, or editing an existing record. Remember that client side scripts would only get triggered if the user is interacting with the form. If you're viewing the record, you're not really interacting with the form, because you're just looking at the record.

Again, if you are creating, copying, or editing an existing record, once the page completely loads, then you have your pageInit event. If the user edits a field, you have three functions that would get triggered, or any of these three functions. You have your fieldChanged event, you have your validateField event, and you also have your postSourcing event.

Now, the fieldChanged event and the postSourcing event would be very similar. The difference is that the postSourcing field is used for fields that are being sourced in from another field. An example of that would be your Currency field. Your Currency field is not set by the user. But what happens is that whatever subsidiary you choose, the currency would be based on that subsidiary.

For example, if I choose a US subsidiary, then your currency would be US dollars. If I choose a Japanese subsidiary, then my currency would be Japanese yen, and so on. OK? So if that's something that you want to automate, since the user is not really changing that currency value, then you would be using postSourcing. But in terms of when the event is used, it would be the same as fieldChanged. validateField, of course, is used if you want to validate one particular field.

Now, if you want to validate multiple fields when the user clicks on the Save button, or perform any action when the user clicks on Save, that'll be your saveRecord event. And as soon as the information gets transferred to the browser, like everything is OK on the client side, then it would trigger your beforeSubmit event. On beforeSubmit, you can add additional validation. You can have some computation. You can have some data checks to make sure that everything is clean before processing through the database.

And then once you're done with beforeSubmit, the information would get saved on the database, and then it will trigger your afterSubmit, where your afterSubmit script would be triggered. And that's when you would trigger all of your finalization events or automation, such as maybe sending email or setting a field value or anything like that.

Forms can be created in NetSuite, depending on the user, and this is something that you can do. And as an administrator, if you are going to be a full time or if this is a role that you will be having, think of making your forms easier for your users to use. So if you have a certain form, say the employee record, or in this case, the customer record, now let's say the customer record is used by several departments.

So that means that maybe there are certain fields that your departments use, or then there are other fields that other departments use. And what people tend to do is just throw in all of the fields into form. So when the user opens, say, the customer record, you have hundreds of fields and it's really overwhelming to use. That's not a good idea.

So what you want to do is that you want to look at how your users are using NetSuite, and then if there are certain fields that they really wouldn't use, like OK, this field is specific to the sales department-- nobody else would be using this field aside from the sales department-- then go ahead and create a form so that the sales department would see that field and the rest of the company would be using another form.

Or you can even expand that, and instead of just creating two forms, you can have multiple forms. For example, you have a customer generic form, where everything is there, and then you have a support form that is used specifically by your customer support team, and then you have a project manager form, where you have certain fields that are available only for project managers. And then you have, again, customer sales form that is specific to your sales team. So that's something that you can do.

Now typically, as developers, if you're only going to be a developer and you're not going to maintain the actual NetSuite account, this is not something that you need to worry about, because everything would be handled by your NetSuite administrator. Now, for some companies, the NetSuite administrator and SuiteScript developer are one and the same. So if that's the case, this is something that you need to look at.

Now, if you are going to create multiple forms, you can also create scripts that are attached to a single form instead of the whole record. Because so far, what we've done is that we've created a script, and it's always going to be for the whole record. Like, we've deployed our script to the customer record and regardless of the form that they would use, it would still trigger the script.

Now, for the most part, users would be creating record level scripts, because customizations would normally apply to the whole company and not just for a certain department. Like, if I add in an automation, I want everybody to benefit from that automation. Or if I want a field validated, I want all of my users to have that field validated, instead of just the one department.

But there are some things that would be unique to a department, say, sales. Sales has a unique computation that they need to worry about in terms of commission and things like that. So you don't want that automation to happen on other forms. So what you can do is that instead of deploying your customization on the record, you can actually deploy that on your form. OK?

But again, this is very uncommon. In my six years in NetSuite, I've only seen this twice, I think. Just know that if that's something that you need, then you can actually deploy the script to a form.

Now, to deploy a script to the form, what you need to do is you just need to go to the form configuration and then configure that similar to how you would configure a script record. You have to upload the script, and then after you upload the script, everything would be handled by the system. You don't need to specify a name or an ID, because that would depend on the form. So the form has the ID, but not the script.

Some things to remember before we go on to our exercise. Be mindful of the execution order. So remember that your entry point triggers depending on an order. It starts with your beforeLoad and then goes to pageInit and then your field level events, then your saveRecord and then beforeSubmit and afterSubmit. And it always goes through that cycle.

What people tend to forget is that they try to perform an automation where there's a certain process, and they're not really aware of how that execution cycle goes. And that way, they're expecting a certain value to happen when in fact, it hasn't happened yet because they've placed the automation in the wrong place. So just try to be careful with that.

Also, entry points can only be triggered on create, copy, or edit. So remember that client side scripts trigger only when you are directly interacting with the form. So if you're not directly interacting with the form, such as when you're viewing the record, I mean, you're looking at the record, but you're not doing anything to the form. You're just looking at it.

In that case, if you view a record, client side script would not get triggered. So again, this will only be triggered when the user is typing something on the form, and that'll happen when you are creating a record, copying, or editing a record.

Also, if you are using a client side script, make sure that you always use Control-F5 on Windows or Command-R, if you're on a Mac, to refresh your page. The reason for this is because your browser can behave differently, especially if the page did not change significantly.

Because some browsers, what they do is that instead of loading pages from the server, to make the performance faster, they would just get the information from cache. That means that the information might be outdated, especially for your script. So to make sure that you get the latest copy of the script, press Control-F5 or Command-R.

Also, if you are debugging your client side script, so aside from using the debugger statement, you can use an alert statement. If you do use an alert statement, it can be annoying, even for developers. If you need to get certain values, you can have four different pop-ups, and it's very annoying if you use alert.

An alternative to that is by using console.log. So if you console.log, instead of displaying that on the screen, you can just open your browser debugger and that would display that on the log. But this is only applicable for client side scripts. On a server side script, since you don't have access to the server console, or since the script is running on the server, then the alert statement would also not work.

And then lastly here, if you are going to combine form level scripts and record level scripts, then you have to remember that form level scripts execute first before triggering record level scripts. OK? Now, this is only an issue if you are going to combine form level scripts and record level scripts. Otherwise, it's always going to be record level script anyway, so you don't have to worry about that. But again, just in case that's something that you want to do.

Now it's your turn. Please complete the following activity.

This concludes module 4, Understanding Entry Points.