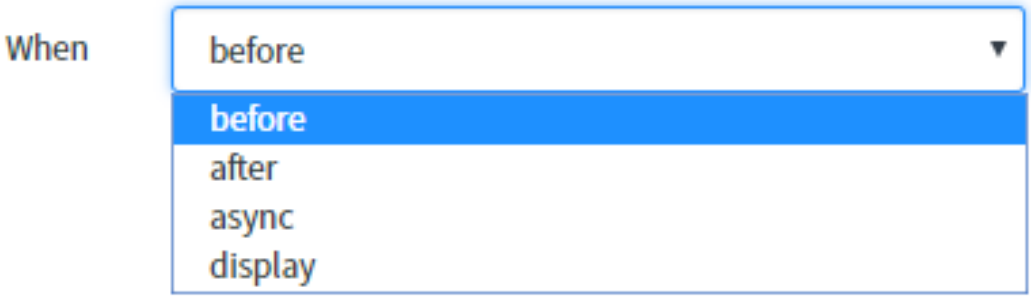
BUSINESS RULE ORDER & .UPDATE()

One common point of confusion when dealing with scripts in Business Rules, is when to use `current.update()`. The answer to this question becomes intuitive once you have a clear understanding of how the different types of Business Rules work. There are four ways a Business Rule can run, indicated by the **When** field on the form:

- Before
- After
- Async
- Display



The **When** field on the Business Rule dictates several things about its behavior, and what parameters are available to scripts. First, recall that Business Rules are triggered by database options (*insert*, *update*, *delete*, or sometimes on *query*[1]). With this in mind, the values in the **When** field make a bit more sense.

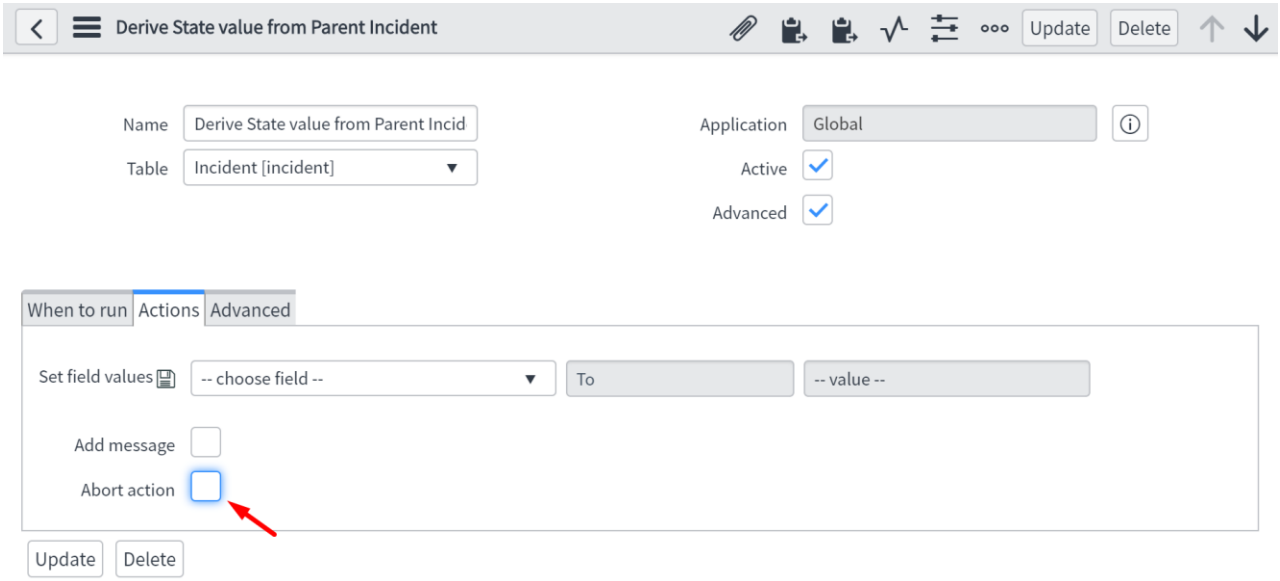
Let's discuss each of the options for "when" a Business Rule runs, how they alter the behavior of our Business Rule, and what we can and can't do with each:

BEFORE

The **before** option means that the Business Rule will run **before** the *insert*, *update*, or *delete* operation is committed to the database. This means that you can still stop the operation by

[1] *For more information on Query Business Rules and why they can be tricky and (if used incorrectly) even harmful, see my article on the subject on SN Pro Tips: https://qbr.snc.guru.*

using `current.setAbortAction(true);`. This also means that you **do not** need to call the `current.update()` API in order to commit any changes that you make, because the record is already "on its way to the database", and will be committed after all "*before*" logic is finished running, assuming that the operation is not aborted by either using current.setAbortAction(true), or by executing a business rule with the "Abort action" checkbox checked.



Before Business Rules are useful for when you need to alter something about the record itself before it ever touches the database or prevent the database operation from happening at all. They should *not* (generally) be used to update **other** records – that is what "*after*" and "*async*" Business Rules are for.

To reiterate: "**before**" Business Rules are generally for **updating the record which triggered them to run**. You should *not* use a "before" Business Rule on the Requested Item [*sc_req_item*] table to create or update a Catalog Task [*sc_task*].

AFTER

The **after** option means that the Business Rule will run **after** the database operation, but *before* the user's browser is refreshed.

If you make a change to the record in an "*after*" Business Rule, it will **not** be committed to the database unless you call `current.update()`. However, if you need to update the current record in this way, it's almost always best to use a "*before*" Business Rule. It is *extremely rare* to have a good use-case for calling `current.update()` in any Business Rule. Making a change to the "`current`" record and then calling `current.update()` in an "*after*" Business Rule would actually re-trigger any business logic (including your "*after*" Business Rule), since it constitutes a *new* database operation. This could even lead to an infinite loop!

"So," you might be asking, "what, then, *are* '*after*' Business Rules used for?"

Although they should not be used to update the `current` record, "a*fter*" Business Rules are fantastically useful for updating **related** records; especially records that might be shown in **related lists** on the record that triggered the Business Rule. For example, if you're on a Request Item [*sc_req_item*] (RITM) record, and you have a Business Rule that runs based off of an event in that table, but it updates one or more child Catalog Task [*sc_task*] records (which are shown in the related list at the bottom of the RITM record), then an **after** Business Rule is usually the way to go. However, if you need to update another type of record that *doesn't* need to be displayed when the form reloads after an update, an **async** rule is best.

Why? Because while "*before*" and "*after*" Business Rules are *running*, the user is *waiting*! These types of Business Rules run **synchronously**, meaning that the user has to sit around and wait for *entire milliseconds* while those operations finish running.

Async Business Rules, as we'll learn below, are different.

ASYNC

The **async** option in the **When** field, indicates that a business rule should run asynchronously, whenever the system scheduler is able to run it (usually within just a few seconds). This means that after saving a record, the server **does not wait** for async Business Rules to finish running before returning control to the user, and reloading the form (or whatever page the user will be directed to).

Async business rules are an extremely useful tool for maximizing performance and creating a positive and snappy user experience, but it is important to be aware of the side-effects of using Async over a different sort of Business Rule. For example, if you use an **async** Business Rule to update a record that will be displayed in a related list on the form that the

user will see when the page refreshes, then when you load the form and view that related list, you will (usually) not see those changes. This is because when the related record was loaded from the database when the form reloads, it's likely that the async Business Rule would not yet have had a chance to run.

The up-side to **async** Business Rules, is that it is much more efficient and user-friendly to perform certain operations asynchronously (and therefore provide a much better user experience, when used correctly!) For example, updating a peripheral or related record which is **not** shown on the form, or triggering some other operation such as a REST API call to update or trigger some logic in some external system that doesn't need to be done *immediately* upon the triggering record being updated. If the operation can stand to wait a few seconds, make it async! The only exception here, is that the `previous` object (the state of the triggering record *before* the update that triggered the BR was made) is not available in async BRs, so if you *need* access to that, you might still need to use an "after" Business Rule to perform that operation.

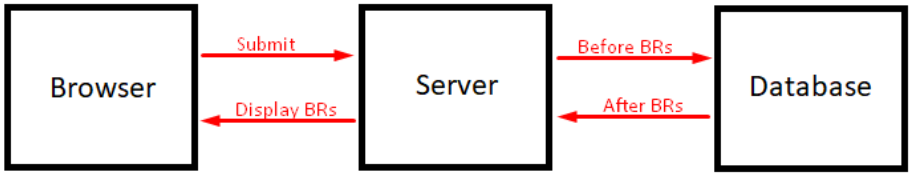
DISPLAY

Finally, we come to **display** Business Rules. These are a little bit different than the other types we discussed above.

Display Business Rules run *after* the record is **retrieved** from the database. This means that they are not triggered by any changes to the record itself. Instead, they are triggered by the record being loaded from the database to be displayed in a form or interacted with in a script.

The results of these Business Rules can sometimes be a little bit unintuitive, since they modify data that's sent from the server, *without modifying the record itself* (unless you call `current.update()` in a "*display*" Business Rule, which... don't). This means that it's possible that you'd end up seeing data in the form which does not match the data in the database! For this reason, **display** Business Rules are not often used to modify any records. Instead, they're most commonly used to populate the **scratchpad**[2], which is an object that's used to make certain data available on the client; usually for client scripts. This is extremely useful from a performance perspective, so we go into detail about the scratchpad in the **AJAX & Display Business Rules** section!

[2] *More info on the scratchpad, and how to use it with "display" Business Rules, in the "AJAX & Display Business Rules" section.*



AJAX & DISPLAY BUSINESS RULES

There are a multitude of reasons you might need to communicate between the client, and the server (such as in a *Client Script*, *Catalog Client Script*, or *UI Policy*), including:

- Retrieving a **system property** to determine whether a field should be visible (*or for whatever reason you want, I'm not your dad*)
- Checking some value on another record in another table, possibly to determine how some logic on the front-end should behave
- Retrieving the GlideRecord object for a record being *referenced* on a form, to get the value in some field on that referenced record[3]
- Dealing with date/time objects, and calculating system time

With the notable exception of certain Client Scripts which must run on **submit**, it's always important to make sure we're performing **asynchronous** server requests in client-side code. In most cases, this can be done by specifying a **callback function** (such as when using the GlideRecord `.query()` method client-side), or by using **GlideAjax**.

***Pro-tip**: Need to perform a query in an onSubmit Client Script, but not able to run it synchronously (such as if your code needs to run on the Service Portal)? I've written a method for doing that, which you can see in my article: Asynchronous onSubmit Client Scripts:*
http://onsubmit.snc.guru/

[3] *It isn't possible to "dot-walk" using the `g_form` APIs in a Client Script, so you'll have to use something like `g_form.getReference()` (asynchronously) in the Client Script to retrieve a record before you can retrieve its field values. However, a common mistake is to use `g_form.getReference()` to get a GlideRecord object for a referenced record, in order to retrieve its sys_id. This is unnecessary, because the actual value of the reference field itself, is the referenced record's sys_id!*
You can retrieve a referenced record's sys_id by simply using `g_form.getValue('some_reference_field');`.
Orlando API docs for the g_form.getReference() API can be found at:
https://developer.servicenow.com/dev.do#!/reference/api/orlando/client/c_GlideFormAPI#r_GlideFormGetReference_String_Function

If you know **in advance** that you're going to need certain information nearly every time a given form is loaded, you can use a **display Business Rule** to grab the data from the server as the form loads, and pass it up to the client. Display Business Rules are a special type of script that run on the **server** when you request a record's data from the database, in order to load a form.

***Pro-tip**: It's not within the scope of this handbook, but if you're not sure how to make your request run asynchronously, I've written an entire article on the topic over at SN Pro Tips: http://ajax.snc.guru/*

Once your browser requests the page and record from the database to display in the form, the display Business Rule runs **on the server** *if* the record you're loading matches the condition in the Business Rule. In that script, you could modify the data in the "`current`" object if you want to change how the record is shown in the form, but you'll also have access to the `g_scratchpad` object. You can **add properties** to this object in the *display Business Rule*, and these properties will be available to any *Client Script*, *UI Policy*, or *UI Action* running on that form via the same `g_scratchpad` object.

***Note**: This does not work for Catalog Client Scripts. You'll have to rely on asynchronous AJAX requests to the server in that case.*

Display Business Rules are probably the most effective and **efficient** way of retrieving data from the server, but they are sometimes overkill. For example, if you only need the data that you would load into `g_scratchpad` in one special circumstance based on some rare client-side event, it might make sense to use a **GlideAjax** call instead. Just make sure that it's asynchronous!

GlideAjax is the second most efficient means of retrieving data from the server. However, unlike display Business Rules, GlideAjax typically requires both a client-side, *and* a server-side component, so it can be a bit more complicated.

***Pro tip**: If you look up the documentation on GlideAjax, you'll find that you can add "nodes" to the returned AJAX, which is useful for returning multiple or complex values from a single call. However, a much easier and cleaner way of doing this, is to return an **object** populated with the values you*

need returned, in properties of that object. The object will then be available when you get the '`answer`*' node of the returned XML, and you can work with it directly. Much simpler!*

More details on this, GlideAjax, and asynchronicity in general, in my article at: http://ajax.snc.guru/

Finally, the *least* efficient method that's *still acceptable* in a pinch, is an **asynchronous** GlideRecord call. The same article linked above, will walk you through how to make a GlideRecord call happen asynchronously. You can also find documentation on all of these APIs in the ServiceNow developer site (https://developer.servicenow.com/), for specifics on how to use them.

To reiterate: If you know exactly what data you need to retrieve from the server in advance, and the conditions under which you want to retrieve the data are known on load, a **Display Business Rule** is probably the best option.

If that isn't the case, then **GlideAjax** is likely the best option.

For small, efficient queries (such as only returning one record in a script that runs only occasionally), an *asynchronous* **GlideRecord** query can be used.

For getting a GlideRecord object from a **reference** field, calling `g_form.getReference()` asynchronously with a **callback function** makes the query asynchronous, and is also acceptable (especially if you're only retrieving a single record).