Flow Structural Conventions - Common Core

As detailed in the General Notes section, these conventions are heavily opinionated towards maintenance and scaling in large organizations. The conventions contain:

- a "common core" set of structural conventions that apply everywhere (this page!)
- conventions for Record Triggered Flows specifically
- conventions for Scheduled Flows specifically

Due to their nature of being triggered by the user and outside of a specific record context, Screen Flows do not require specific structural adaptations at the moment that are not part of the common core specifications.

Common Core Conventions

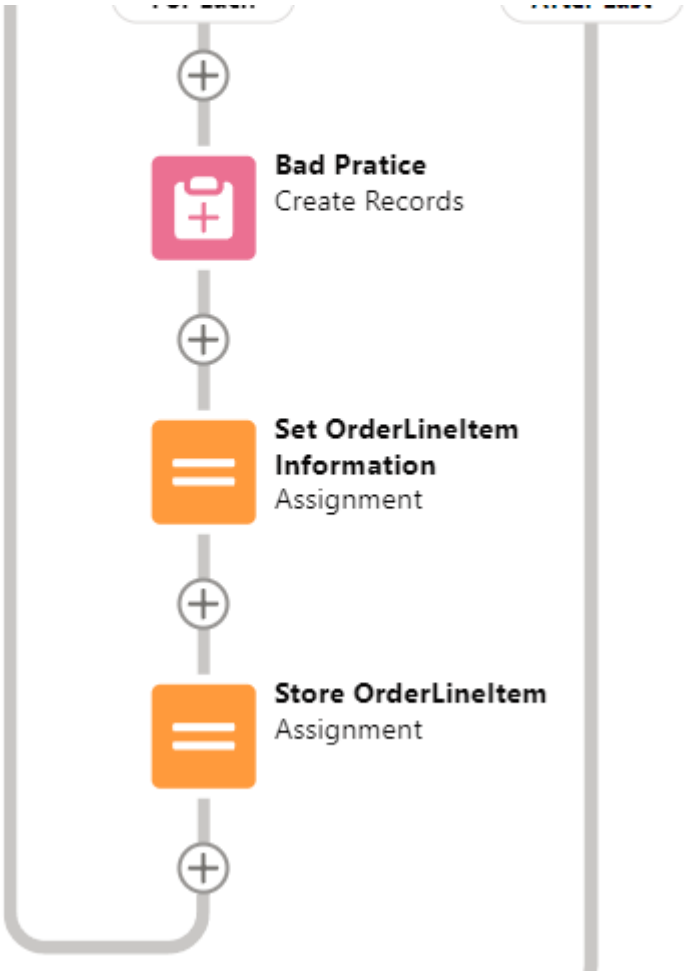
On System-Level Design

Do not do DMLs or Queries in Loops.

> **Simpler: No pink squares in loops.**

> **DML** is Data Manipulation Language. Basically it is what tells the database to change stuff. DML Operations include Insert, Update, Upsert, and Delete, which you should know from Data Loader or other such tools.

Salesforce now actually warns you when you're doing this, but it still bears saying.



Don't do this

You really **must** not do this because:

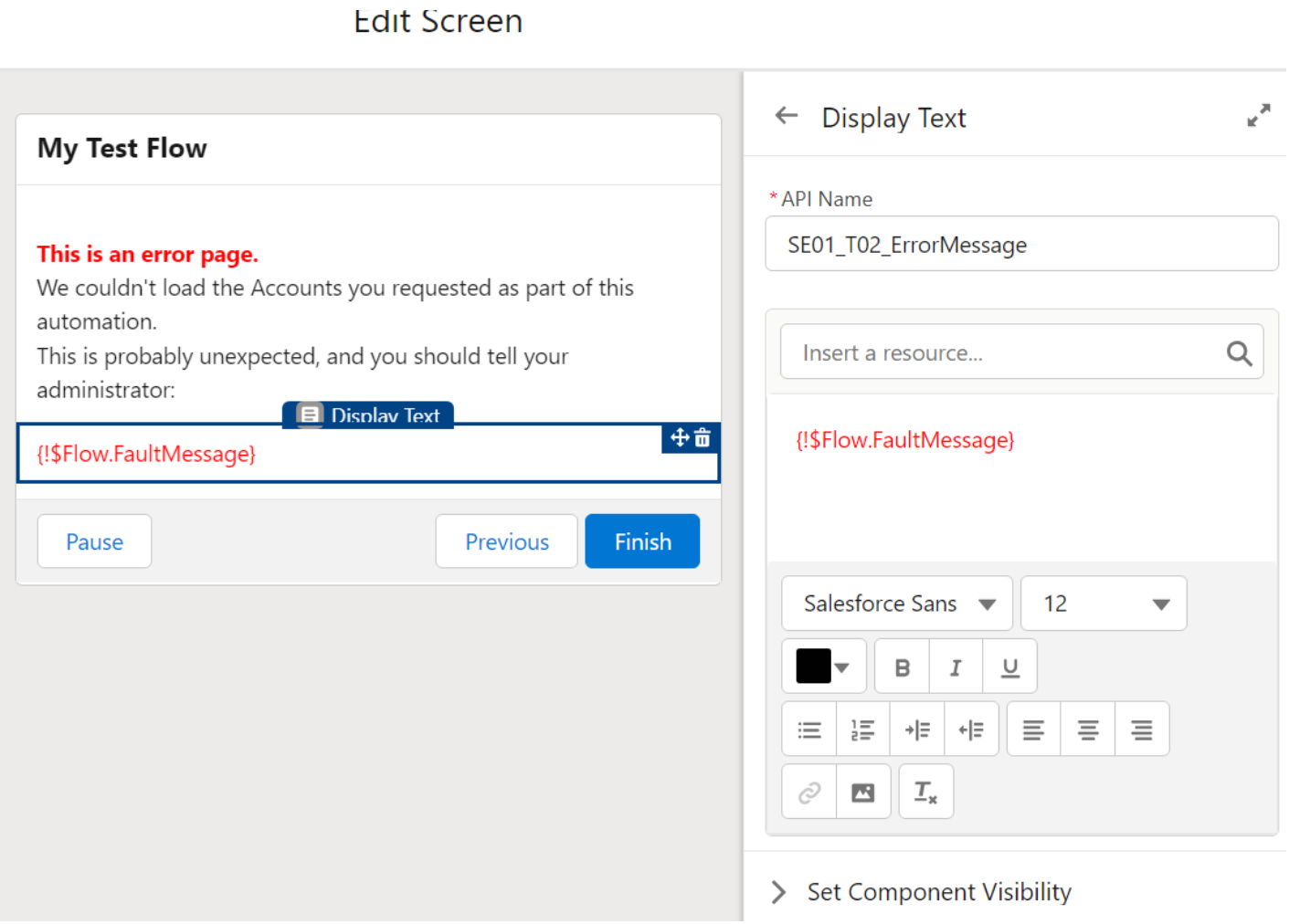
- it can break your Flow. Salesforce will still try to optimize your DML operations, but it will often fail due to the changing context of the loop. This will result in you doing one query or update per record in your loop, which will send you straight into Governor Limit territory.
- even if it doesn't break your Flow, it will be SLOW AS HELL, due to the overhead of all the operations you're doing
- it's unmaintainable at best, because trying to figure out the interaction between X individual updates and all the possible automations you're triggering on the records you're updating or creating is nigh impossible.

All Pink (DML or Query) elements should have Error handling

> Error, or Fault Paths, are available both in Free Design mode and the Auto-Layout Mode. In Free mode, you need to handle all possible other paths before the Fault path becomes

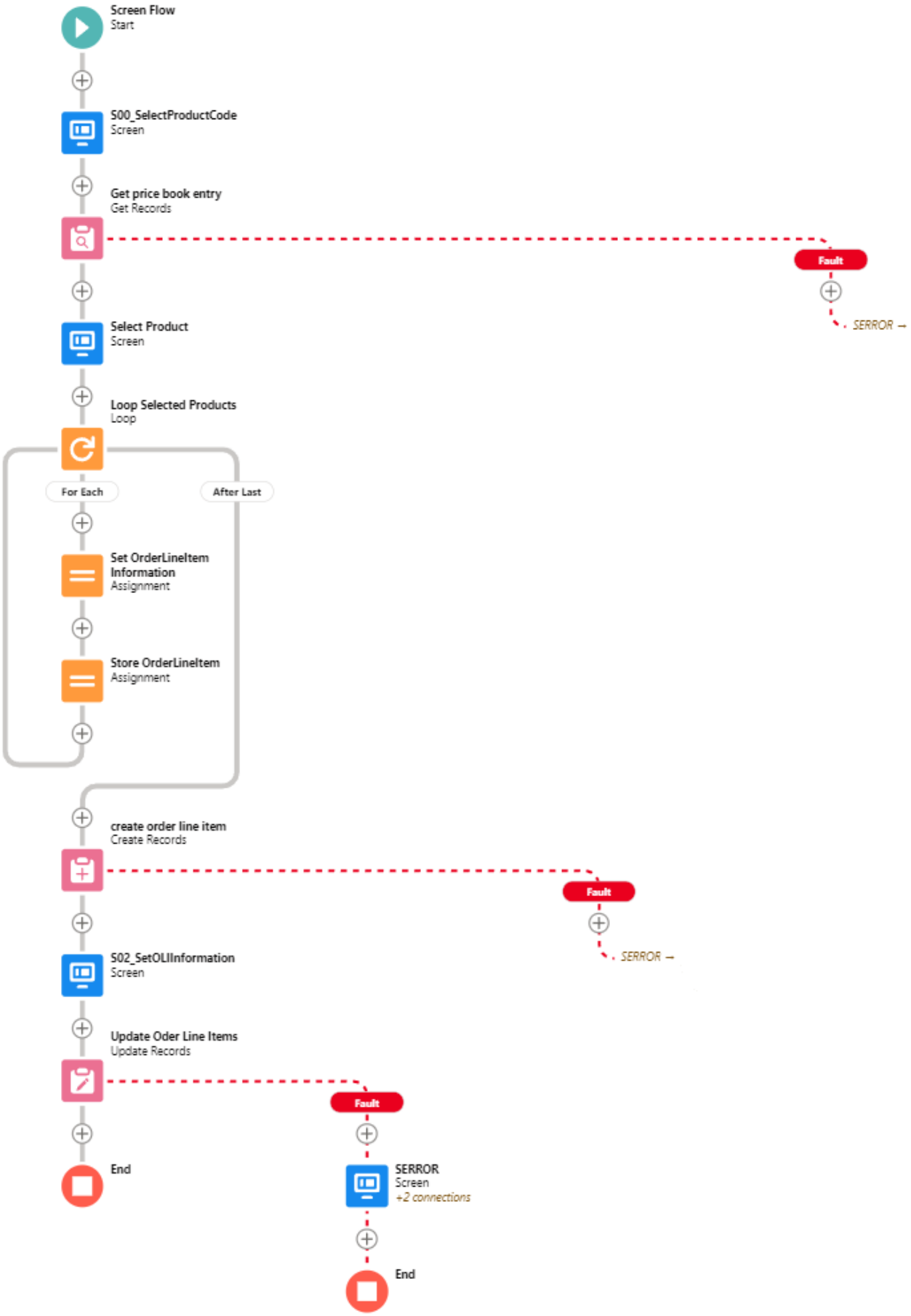
> available. In Auto-Layout mode, you can simply select Fault Path.

Screen Flow? Throw a Screen, and display what situation could lead to this. Maybe also send the Admin an email explaining what happened.



Record-triggered Flow? Throw an email to the APEX Email Exception recipients, or emit a Custom Notification.
Hell, better yet throw that logic into a Subflow and call it from wherever.

> (Note that if you are in a sandbox with email deliverability set to System Only, regular flow emails and email alerts will not get sent.)

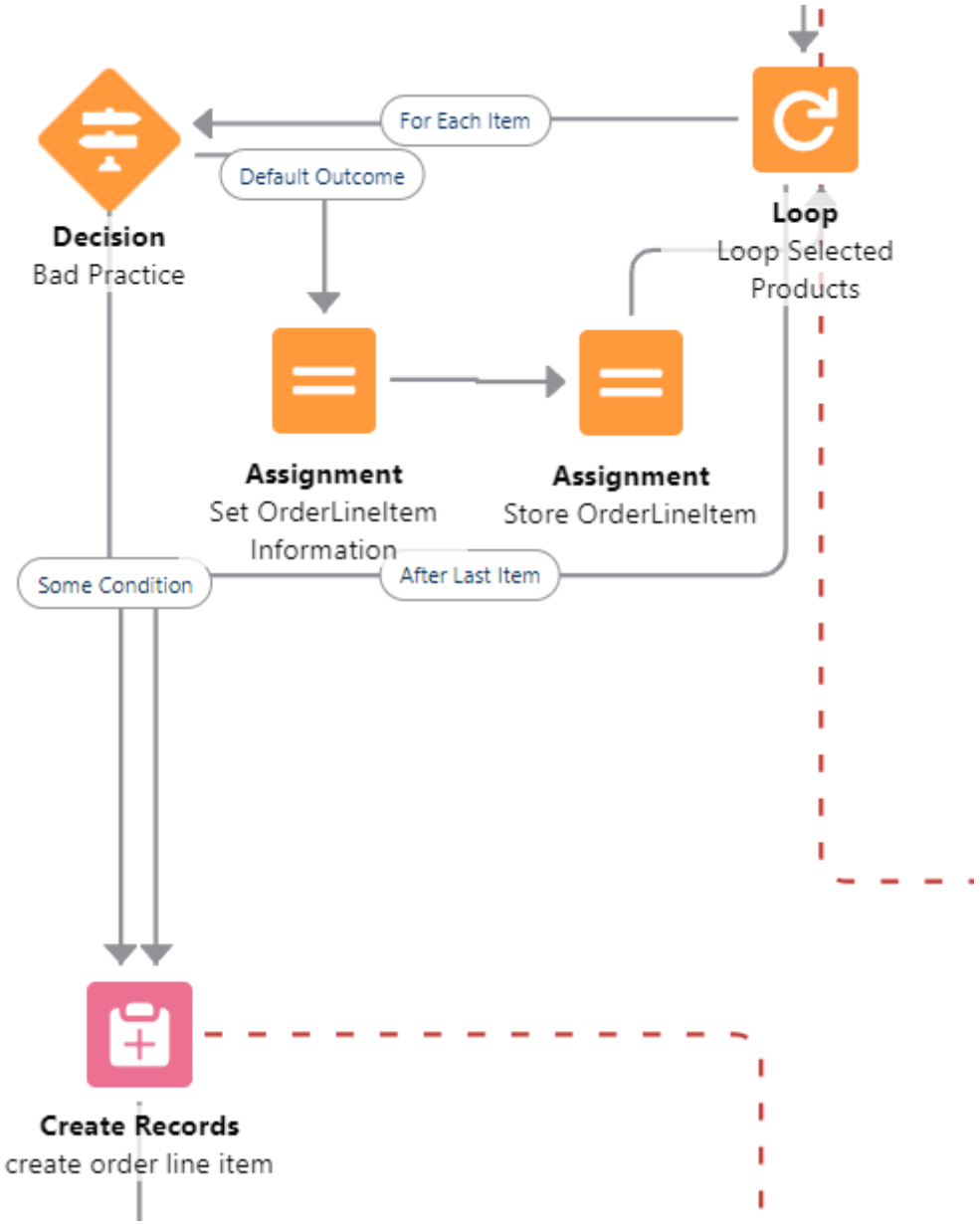

Screen Flow
Start
S00_SelectProductCode
Screen
Get price book entry
Get Records
Fault
SERROR
Select Product
Screen
Loop Selected Products
Loop
For Each
After Last
Set OrderLineItem Information
Assignment
Store OrderLineItem
Assignment
create order line item
Create Records
Fault
SERROR
S02_SetOLIInformation
Screen
Update Oder Line Items
Update Records
Fault
End
SERROR
Screen
+2 connections
End

> Handling Errors this way allows you to:
> - not have your users presented with UNEXPECTED EXCEPTION - YOUR ADMIN DID THINGS BADLY
> - maybe deflect a few error messages, in case some things can be fixed by the user doing things differently
> - have a better understanding of how often Errors happen.

You want to supercharge your error handling? Audit Nebula Logger to see if it can suit your needs. With proper implementation (and knowledge of how to service it, remember that installed code is still code that requires maintenance), Nebula Logger will allow you to centralize **all** logs in your organization, and have proper notification when something happens - whether in Flow, APEX, or whatever.

Don't exit loops based on decision checks

The Flow engine doesn't support that well and you will have weird and confusing issues if you ever go back to the main loop.



Don’t do this either - always finish the loop

Issues include variables not being reset, DML errors if you do come back to the loop, and all around general unpredictable situations.
You *can* still do this if you absolutely NEVER come back to the loop, but it's bad design.

Do not design Flows that will have long Wait elements

> This is often done by Admins coming from Workflow or Process Builder space, where you could just say "do that 1 week before contract end date" or "1 day after Opportunity closure". This design is sadly as outdated as the tools that permitted it.
> Doing this will have you exceed your Paused Interview limits, and actions just won't be carried out.

A proper handling of "1 day before/after whenever", in Flow, is often via a Scheduled Flow.
Scheduled Flows execute once daily (or more if you use plugins to allow it), check conditions, and

execute based on these conditions. In the above case, you would be creating a Scheduled Flow that :

- Queries all Contract that have an End Date at `TODAY()-7`
- Proceeds to loop over them and do whatever you need it to

> Despite it not being evident in the Salesforce Builder, there is a VERY big difference between the criteria in the Schedule Flow execution start, and an initial GET.
> - Putting criteria in the Start Element has less conditions available, but effectively limits the scope of the Flow to only these records, which is great in **big environments**. It also fires **One Flow Interview per Record**, and then bulkifies operations at the end - so doing a **GET** if you put a criteria in the Start element should be done after due consideration only.
> - On the opposite, putting no criteria and relying on an initial Get does a single Flow Interview, and so will run less effectively on huge amounts of records, *but* does allow you to handle more complex selection criteria.

Do not Over-Optimize your Flows

When Admins start becoming great at Flows, everything looks like a Flow.
The issue with that is that sometimes, Admins will start building Flows that shouldn't be built because Users should be using standard features (yes, I know, convincing Users to change habits can be nigh impossible but is sometimes still the right path)... and sometimes, they will keep at building Flows that just should be APEX instead.

> If you are starting to hit CPU timeout errors, Flow Element Count errors, huge amounts of slowness... You're probably trying to shove things in Flow that should be something else instead.

APEX has more tools than Flows, as do LWCs. Sometimes, admitting that Development is necessary is not a failure - it's just good design.

On Flow-Specific Design

Flows should have one easily identifiable Triggering Element

This relates to the Naming Conventions.

Flow Type	Triggering Element
Record-Triggered Flows	It is the Record that triggers the DML
Event-based Flows	It should be a single event, as simple as possible.

Screen Flows	This should be either a single recordId, a single sObject variable, or a single sObject list variable. In all cases, the Flow that is being called should query what it needs by itself, and output whatever is needed in its context.
Subflows	The rule can vary - it can be useful to pass multiple collections to a Subflow in order to avoid recurring queries on the same object. However, passing multiple single-record variables, or single text variables, to a Subflow generally indicates a design that is overly coupled with the main flow and should be more abstracted.

Flow Definitions
All Flows (Extended)

16 items • Sorted by Flow API Name • Filtered by All flow definitions - Flow Namespace • Updated 2 hours ago

Trigg...	Process T...	Trigger	Flow Label	Flow API Name	Flow Description
Account	Autolaunched...	Record—Run Afte...	Account_AftercreateAftersave_SetStatusClientA...	Account_AftercreateAftersave_SetStatus...	When [redacted] is changed ...
Account	Autolaunched...	Record—Run Bef...	Account_BeforeSave_InvoicingClientStatusChan...	Account_BeforeSave_InvoicingClientStat...	Triggers when the Client invoicing Sta...
Account	Autolaunched...	Record—Run Bef...	Account_BeforeSave_InvoicingStatusChange	Account_BeforeSave_InvoicingStatusCha...	Triggers when the invoicing Status ch...
Account	Autolaunched...	Record—Run Afte...	Account_BeforeSave_SetClientNumber	Account_BeforeSave_SetClientNumber	Triggers when the record is new or up... Sets the Client Number based on [redacted]
Account	Autolaunched...	Record—Run Bef...	Account_BeforeSave_SetAccountAddress	Account_BeforeSave_SetCountryMarketI...	Triggers when record is new and a co... Sets the ShippingCountry and the [redacted]
	Screen Flow		Account_SCR_RequestChanges	Account_RequestChanges	A Flow to request changes on an Acc...

Fill in the descriptions

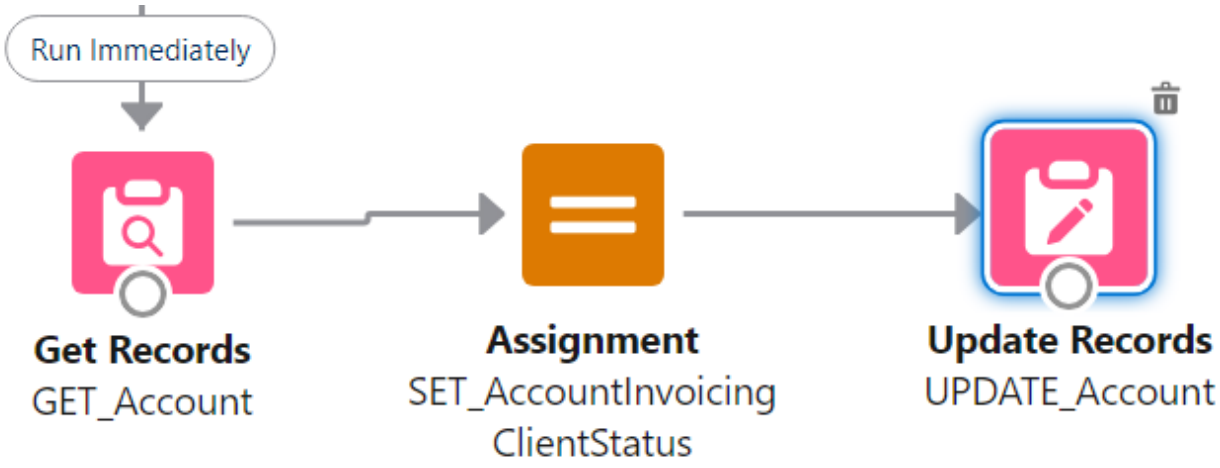
You'll thank yourself when you have to maintain it in two years.
Descriptions should not be technical, but functional. A Consultant should be able to read your Flow and know what it does technically. The Descriptions should therefore explain what function the Flow provides within the given Domain (if applicable) of the configuration.

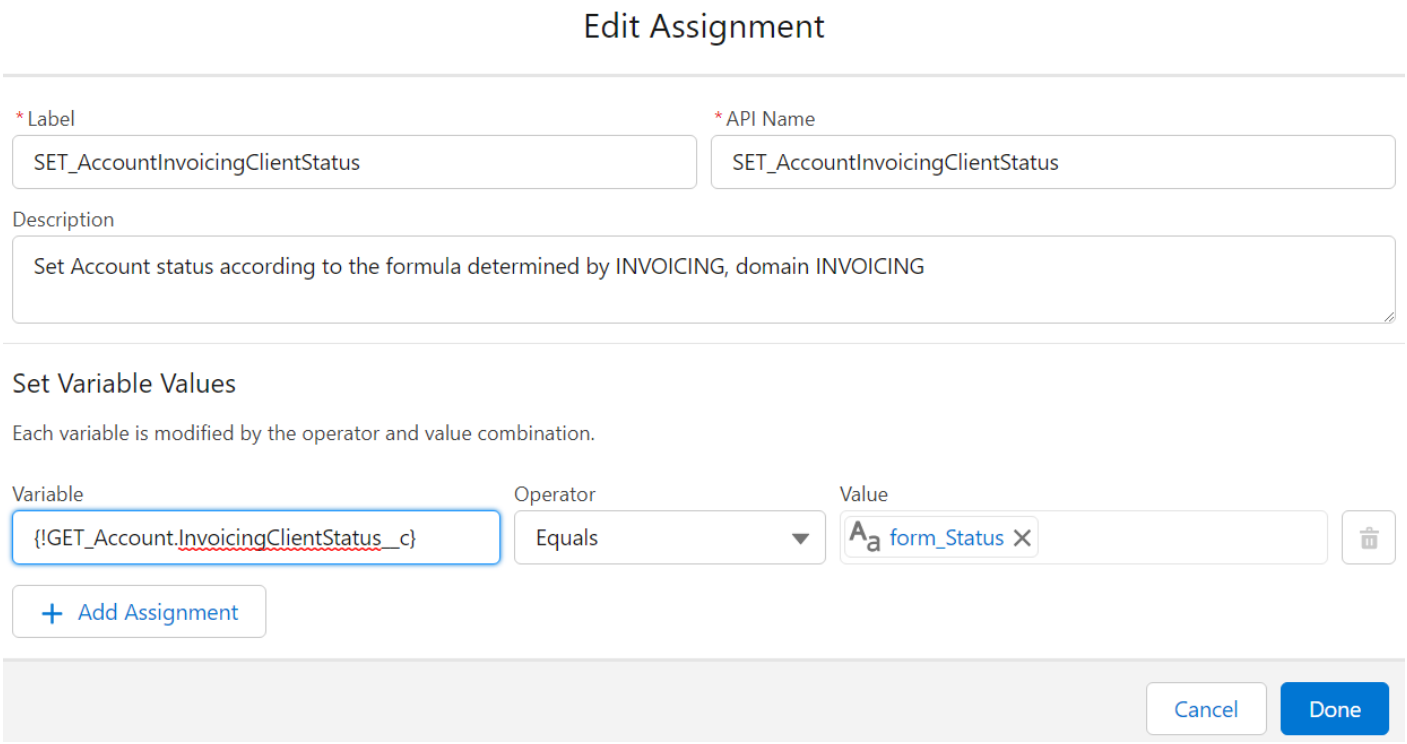
Flow Description
Triggers when OrderItems are Created. Launches automations / publishes events as needed.
Screen Flow which queries all Children Products for Product2 called, then offers the Parent Fields for modification, then sets the new information on all related Children.
Platform event driven flow that sets the order item number if needed

Descriptions shouldn’t be too technical.

Don't use the "Set Fields manually" part of Update elements

Yes, it's possible. It's also bad practice. You should always rely on a record variable, which you Assign values to, before using Update with "use the values from a record variable". This is mainly for maintenance purposes (in 99% of cases you can safely ignore pink elements in maintenance to know where something is set), but is also impactful when you do multi-record edits and you *have* to manipulate the record variable and store the resulting manipulation in a record collection variable.

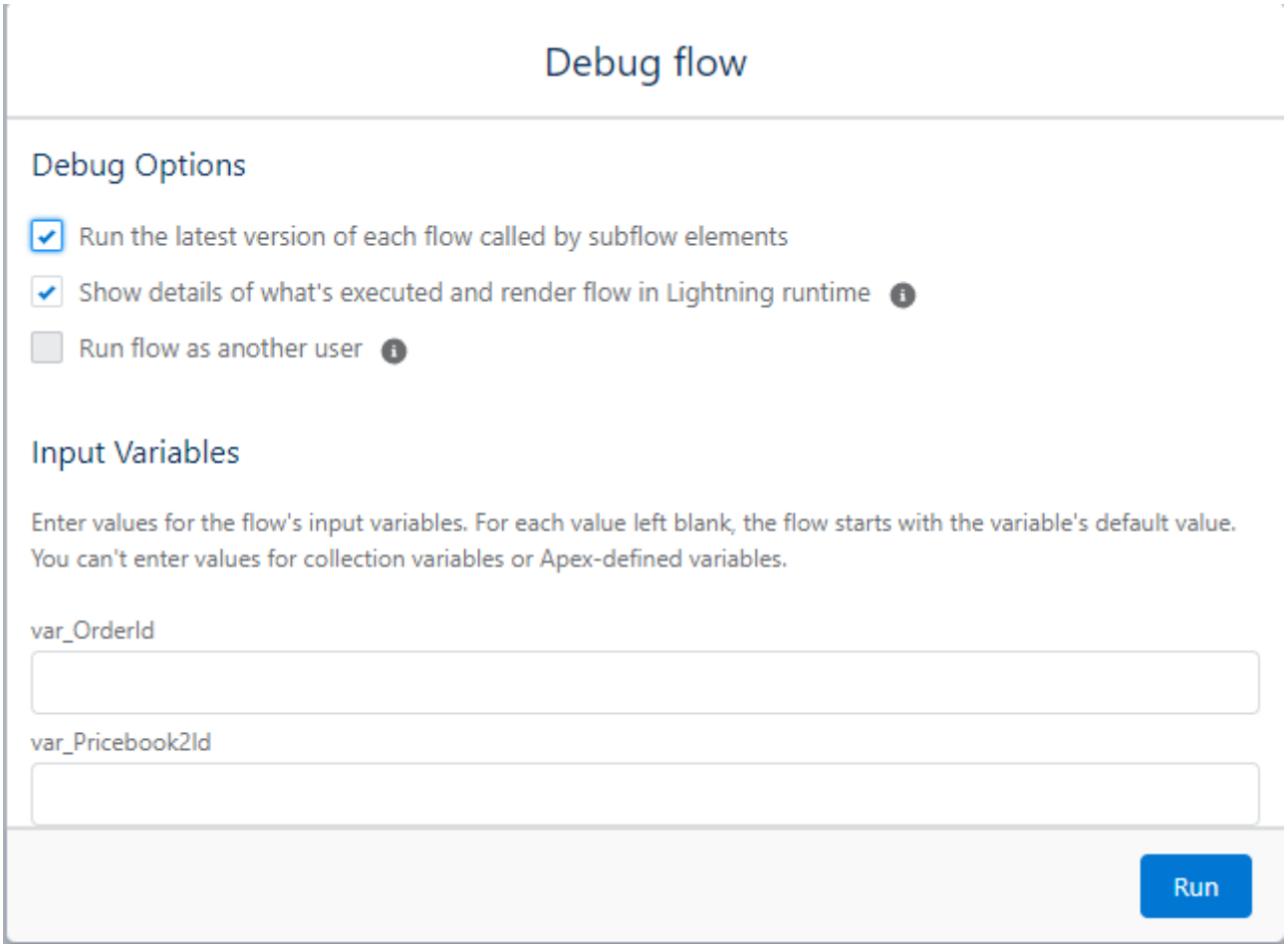




Try to pass only one Record variable or one Record collection to a Flow or Subflow

See "Tie each Flow to a Domain".
Initializing a lot of Record variables on run often points to you being able to split that subflow into different functions. Passing Records as the Triggering Element, and *configuration information* as variables is fine within reason.

> In the example below, the Pricebook2Id variable should be taken from the Order variable.



Try to make Subflows that are reusable as possible.

A Subflow that does a lot of different actions will probably be single-use, and if you need a subpart of it in another logic, you will probably build it again, which may lead to higher technical debt.
If at all possible, each Subflow should execute a single function, within a single Domain.

Yes, this ties into "service-based architecture" - we did say Flows were code.

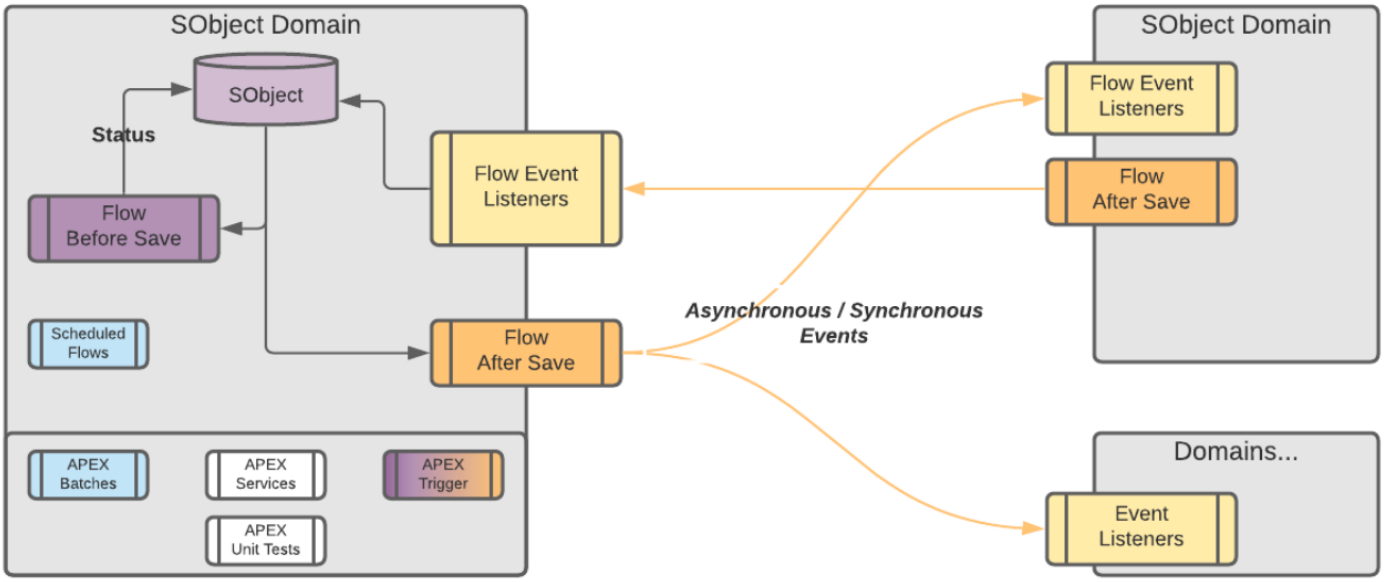
Do not rely on implicit references

This is when you query a record, then fetch parent information via `{MyRecord.ParentRecord__c.SomeField__c}`. While this is *useful*, it's also very prone to errors (specifically with fields like `RecordType`) and makes for wonky error messages if the User does not have access to one of the intermediary records.
Do an explicit Query instead if possible, even if it is technically slower.

Tie each Flow to a Domain

This is also tied to Naming Conventions. Note that in the example below, the Domain is the Object that the Flow lives on. One might say it is redundant with the Triggering Object, except Scheduled Flows and Screen Flows don't have this populated, and are often still linked to specific objects, hence the explicit link.

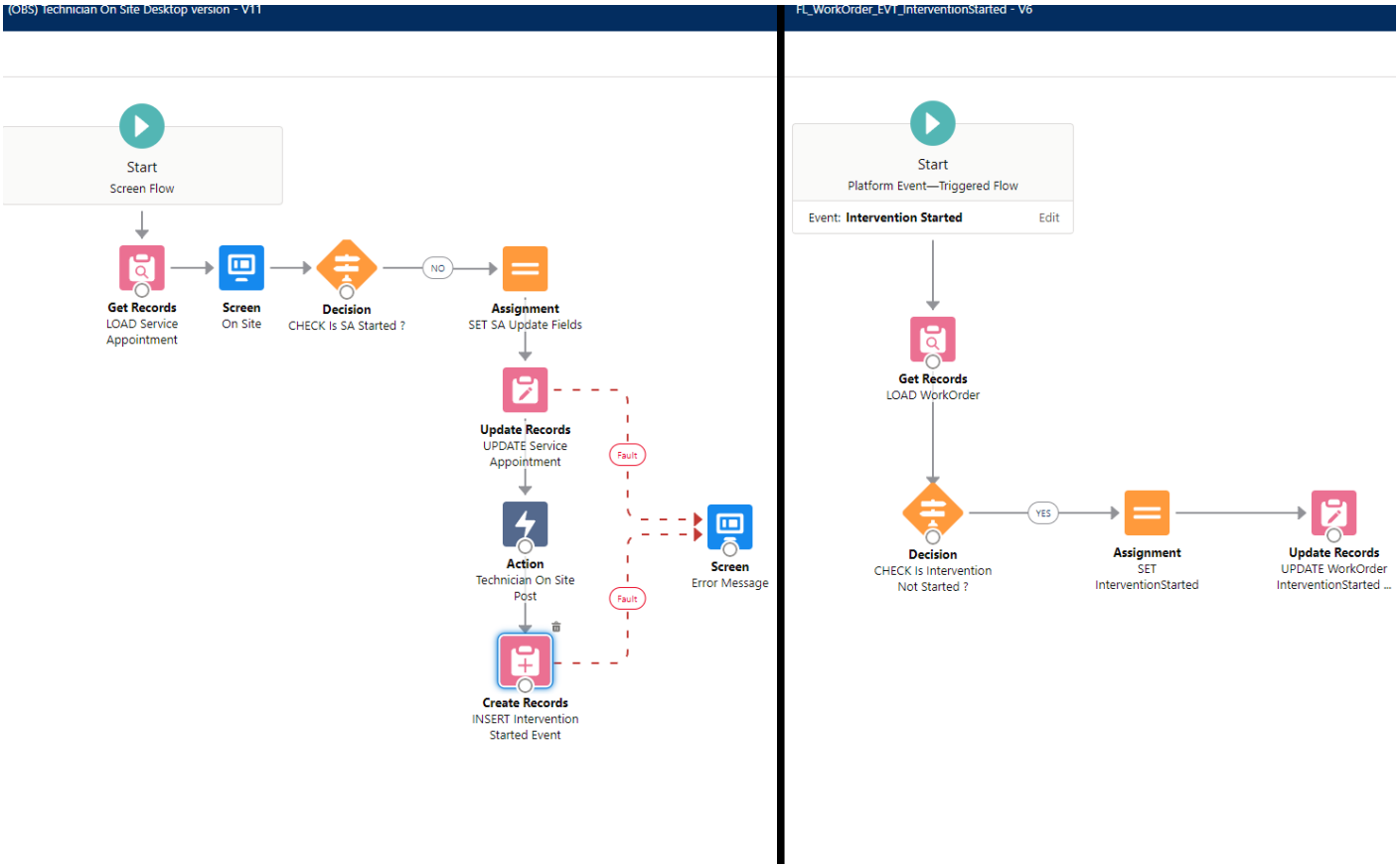
Domains are definable as **Stand-alone groupings of function which have a clear Responsible Persona.**



Communication between Domains should ideally be handled via Events

In short, if a Flow starts in Sales (actions that are taken when an Opportunity closes for example) and finishes in Invoicing (creates an invoice and notifies the people responsible for those invoices), this should be two separate Flows, each tied to a single Domain.

> Note that the Salesforce Event bus is mostly built for External Integrations.
> The amount of events we specify here is quite high, and as such on gigantic organisations it might not be best practice to handle things this way - you might want to rely on an external event bus instead.
>
> That being said if you are in fact an enterprise admin I expect you are considering the best usecase in every practice you implement, and as such this disclaimer is unnecessary.



Example of Event-Driven decoupling

Avoid cascading Subflows wherein one calls another one that call another one

Unless the secondary subflows are basically fully abstract methods handling inputs from any possible Flow (like one that returns a collection from a multipicklist), you're adding complexity in maintenance which will be costly

---

Revision #14
Created 2021-01-14 10:32:16 UTC by Windyo
Updated 2023-08-28 07:11:00 UTC by Windyo