Developer Considerations for B2B Stores Created with an Aura Template

Salesforce, Winter '26

Last updated: November 10, 2025

© Copyright 2000–2025 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

DEVELOPER CONSIDERATIONS FOR B2B STORES CREATED WITH AN AURA TEMPLATE 1

Integration Architecture for B2B Stores (Aura) 1

B2B Commerce Checkout Flow (Aura) 3

DEVELOPER CONSIDERATIONS FOR B2B STORES CREATED WITH AN AURA TEMPLATE

This guide includes information for developers working with B2B stores created using an Aura template.

Integration Architecture for B2B Stores (Aura)

Understand how the integration platform and the various components interact for a B2B store created with the Aura template.

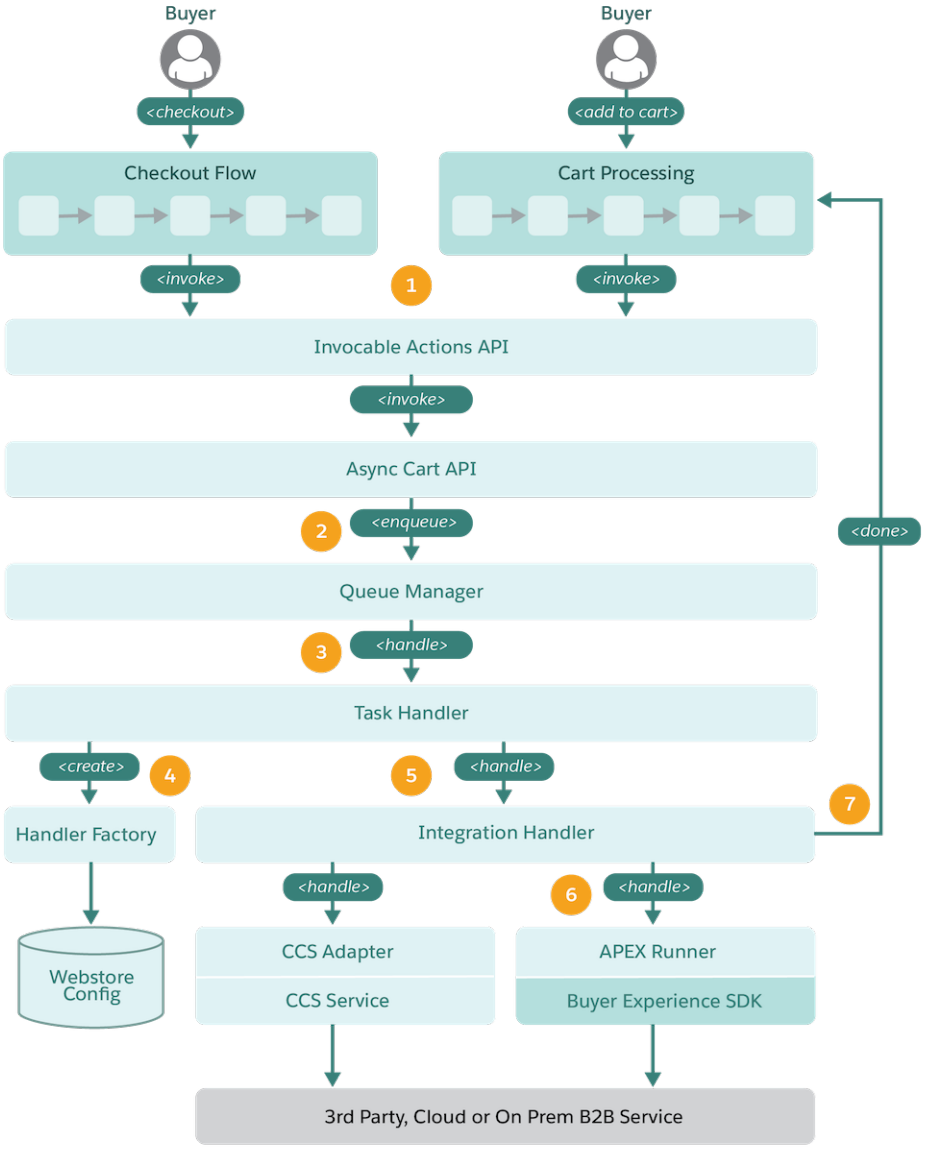
> Note: In the Winter '24 release, we introduced Commerce extensions for pricing, inventory, shipping, taxes, and other services. While the checkout integrations framework is still supported, we recommend extensions over integrations because they offer more targeted customizations for your B2B or D2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

The diagram demonstrates how the B2B (Aura) components work together to form the integration engine.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce



1. **Checkout flow**:— A managed checkout flow that processes cart shipping, inventory, pricing, and taxation integrations, and converts the cart to an order. Install and deploy the B2B checkout flow, then customize the flow in Experience Builder and optionally Flow Builder.
2. **Cart processing** —A predefined set of steps that process cart shipping, inventory, pricing, and tax integrations. Cart processing is executed on select cart actions.
3. **Async Cart API**— A cart service that accepts the processing requests for shipping, inventory, price, and tax integrations. This service executes tasks asynchronously and returns a jobId to the caller.
4. **Queue manager**— The Async Cart API offloads execution of cart integrations by placing a task on the message queue (MQ). The queue manager is responsible for scheduling the future task execution, which provides the cart service a thread pool for potentially longer running async tasks.

5. **Task handler**—The task handler is implemented as an MQ handler and invoked by the queue manager when the integration task is picked up for processing. The integration handler is responsible for delegating the integration task to the integration implementation, which the store admin specifies when setting up the store.
6. **Handler factory**—Responsible for creating an integration handler that maps to the implementation chosen by the store admin when setting up the store.
7. **Integration handler**—Responsible for adapting the Java processing to Apex, managing the lifecycle, and customer code error handling.

SEE ALSO:

B2B Commerce Commerce Developer Guide

B2B Commerce Checkout Flow (Aura)

If you created your B2B store with an Aura template, use Flow Builder to create dynamic checkout flows. Use simple GitHub scripts to stand up a scratch org with a fully functioning checkout, ready to test with preconfigured buyers. Customize the checkout sequence in Experience Builder. Optionally, you can customize your checkout in Flow Builder by adding, replacing, or reordering the flow.

Note: These topics are for setting up checkout flows for B2B stores created with an Aura template. B2B stores created with an LWR template use the Salesforce Commerce Checkout that's installed when you create the store. For B2B stores created with an LWR template, select checkout in Experience Builder, and then click to reorder or add steps and enhance the layout with your custom functionality.

SEE ALSO:

B2B Commerce Commerce Developer Guide

B2B Checkout Flows

B2B checkout implementations are built in Flow Builder, the Salesforce declarative programming interface that requires no coding. The included applications execute each task in the checkout sequence. The Commerce app checkout applications use APIs to trigger interactions between carts, orders, and external providers that take shoppers through all the sequential states—inventory check, shipping, tax fee calculation, payment authorization, order creation, and so on—to support your store checkout.

Main Checkout Flow

When the data and processing necessary to complete integration tasks are tightly coupled with local objects (cart, pricing, inventory), the Main Checkout flow is preferable. This mostly synchronous checkout implementation elicits immediate feedback from objects when triggered by the checkout UI. The Main Checkout flow is a full-featured checkout implementation that provides a responsive experience.

Pricing and Promotions Flow

When you install the Main Checkout flow, the Pricing and Promotions flow is also installed. Use this flow to support B2B promotions.

Re-entrant Checkout Flow

The Re-entrant Checkout flow supports:

- Multiple carts per shopper in various checkout states
- Shopper navigation back to a cart that was closed during checkout by a browser, with no loss of data
- Access to eligible promotions for cart items

Like the Main Checkout and Pricing and Promotions flows, the Re-entrant Checkout flow consists of Apex classes that execute subflow logic. You use an SFDX GitHub script to install the Re-entrant Checkout flow.

Note: The separate, legacy B2B checkout flow template that is also provided does not support tokenization for credit card payments with external providers.

B2B Checkout Flow Tasks

Many components create a working checkout flow. Use this list as you configure your checkout .

Basic Tasks

Complete these tasks to install, test, and configure a checkout.

- **Create an Org and install the checkout flow**—Use an SFDX script to install and deploy a checkout flow to a scratch test org. Then save a copy of the installed and deployed checkout flow to create a customizable flow for your store.
- **Test the checkout flow**—Log in as a buyer (a buyer, store items, and mock shipping and payment integrations are installed with the SFDX script) to explore the checkout journey.
- **Configure the Checkout component in Experience Builder**—On the Checkout page, configure the Checkout component by choosing a flow.
- **Configure checkout using Flow Builder**—Optionally, add, remove, or reorder sublfows, change a subflow from synchronous to asynchronous, or make other customizations.
- **Set the Security setting**—For your cloned checkout flow, select **System Context with Sharing-Enforces Record-Level Access**.
- **Activate the flow**—After you select the appropriate security setting, activate your flow.
- **Configure third-party integrations**—Your checkout sample includes checkout integrations that require configuration. You can also replace these integrations, but doing so requires a developer to create custom integrations. Reference integration samples for tax, shipping, and payment are available in GitHub.
- **Map the checkout flow to your store**—Using Developer Console, map the checkout flow to the store using the StoreIntegrationService object junction table. `ServiceProviderType` is a MappingProviderTypeEnum.

Functional Checkout Elements

Although these elements aren't required, they're important checkout items that we recommend using.

- **Shipping address**—To add addresses to your checkout flow, create them in your buyer account. Addresses can't be added during checkout using the provided shipping component.
- **Inventory check**—Use custom Apex to perform inventory checks to determine if your inventory meets the line item quantities.
- **Product pricing confirmation**—Use the B2B pricing engine or add custom Apex to create a pricing strategy.
- **Shipping cost calculation**—Use custom Apex to calculate shipping charges and connect to a third-party shipping service to determine the cost for each delivery group.
- **Tax cost calculation**—Use custom Apex to calculate taxes per line item.
- **Cart summary**—Provide a high-level summary of the prices, shipping, and taxes in the cart per delivery group. The limit is one cart per delivery group.
- **Payment collection information**—Authorize the payment amount specified in the order summary.
- **Order confirmation**—After converting a cart to an order, let buyers track their order and redirect to the order detail page for further action.

Additional Features

To enhance your checkout experience and provide for complex scenarios, such as third-party integrations or large carts, consider adding these flows.

- **Provide re-entrant checkout**—Allow buyers to browse away from a checkout and return where they left off without losing previously entered data. The out-of-the-box flow provides a re-entrant checkout experience.
- **Add an asynchronous checkout process for long-running tasks**—Allow buyers to browse away from the current checkout process or optionally close the browser tab without losing data. The Re-entrant checkout flow provides an asynchronous checkout experience.
- **Enhance order management**—Export orders from Salesforce to an external ERP, including all order line items, taxes, line item adjustments, tax adjustments, and the payment summary.

Create a B2B Commerce Org and Checkout Flow

Using SFDX scripts, you can deploy a Lightning B2B testing environment that includes checkout flows, sample products, and a buyer.

You can use the configuration files and scripts in the SFDX Git repositoryto quick start your project. If you use the SFDX scripts to create an org, a checkout flow and the checkout subflows are included. You can then select and configure a checkout flow in Experience Builder or optionally in Flow Builder.

- To create a scratch org for testing, see SFDX.
- For example integrations and tests, see Reference Implementation.
- For checkout notifications, see Checkout Notifications.

Configure a B2B Checkout Flow

Use Experience Builder or Flow Builder to configure and customize your checkout flow.

Add a Checkout Flow to a B2B Store

Launch Experience Builder for your store and choose one of the installed checkout flows.

To choose the B2B checkout flow for your store in Experience Builder:

1. On your store's home page, click the **Experience Builder** tile.
2. Navigate to the Checkout page.
3. Select the Checkout Flow component.
4. For Checkout Flow Name , select a flow that you installed, for example,**(Checkout) Main with Salesforce Pricing and Promotions**.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: Available in: B2B Commerce

Update a Checkout Flow to Handle Promotions

If you created a custom checkout flow before Winter '22, it didn't include support for promotions. You can update your existing flow to make it capable of handling promotions.

Note: This section assumes that your existing checkout flow is based on the legacy checkout flow template. In previous releases, this flow was the most frequently used checkout flow. The legacy B2B checkout flow template does not support tokenization for credit card payments with external providers.

Create a Promotion Subflow for an Existing Checkout Flow

Checkout flows created before Winter '22 didn't support promotions. To add promotion capability to a flow created before Winter '22, create a subflow.

Your flow must be based on the legacy checkout flow template.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find box, enter `Flows`, and then select **Flows**.
3. Click **New Flow**.
4. Click **All + Templates**.
5. Click **Checkout Flow**, select **Checkout Flow**, and then click **Next**.
6. Click **Freeform**.
7. Drag the **Action** element onto the canvas.
8. In the New Action screen, locate Promotions, and select **Calculate Cart Promotions**.
9. Define the action.
   a. For the label, enter `Calculate Cart Promotions`.
   b. For Cart Id, select **+ New Resource**.
   c. For Resource Type, select **Variable**.
   d. For API Name, enter `cartId`.
   e. For Data Type, select **Text**.
   f. Select **Available for input**.
   g. On the New Resource screen, click **Done**.
   h. On the New Action screen, click **Done**.
10. Drag another **Action** element onto the canvas.
11. In the New Action screen, search for "Session" and select **Update Checkout Session Action**.
12. Define the action.
    a. Enter `Update Checkout Session` as the label.
    b. In the Checkout Session Id field, select **+ New Resource**.
    c. Select **Variable** as the Resource Type.
    d. Enter `checkoutSessionId` as the API Name.
    e. Select **Text** as the Data Type.
    f. Select **Available for input.**
    g. Click **Done** in the New Resource screen.
    h. In the Next State field, select **+ New Resource**.
    i. Select **Variable** as the Resource Type.
    j. Enter `nextState` as the API Name.
    k. Select **Text** as the Data Type.
    l. Select **Available for input.**

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

m. Click **Done** in the New Resource screen.

n. Select **Include** for the Expected Current State field.

o. In the Expected Current State field, select **+ New Resource**.

p. Select **Variable** as the Resource Type.

q. Enter `currentState` as the API Name.

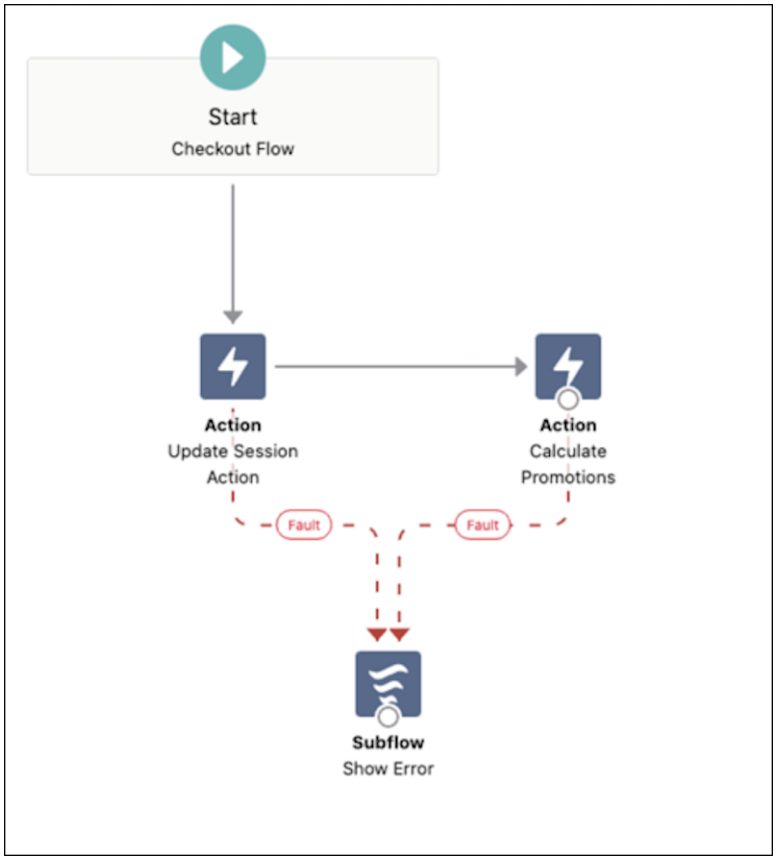
r. Select **Text** as the Data Type.

s. Select **Available for input.**

t. Click **Done** in the New Resource screen.

u. Click **Done** in the New Action screen.

13. Drag the **Subflow** element onto the canvas.
14. In the New Subflow screen, search for "Error" and select **Subflow - Error**.
15. Configure the subflow.
    a. Enter `Show Error` as the Label.
    b. Select **Include** for the cartId field.
    c. Select the cartId variable for the cartId field.
    d. Select **Include** for the ErrorMessage field.
    e. Select the Flow.FaultMessage variable for the ErrorMessage field.
    f. Click **Done** in the New Subflow screen.
16. Drag a connector line from the Start node to the Update Checkout Session action.
17. Drag a connector line from the Update Checkout Session to the Calculate Cart Promotions action.
18. Drag a connector line from the Update Checkout Session action to the Show Error subflow.
19. Drag two connector lines from the Calculate Cart Promotions action to the Show Error subflow, and delete the solid line.
    The first connector line you drag is solid, and the second is a dashed fault line. Delete the solid line.
    Your subflow looks like this.



20. Click **Save**.
21. On the Save the flow screen, enter a label (for example, `Calculate Promotions`) and click **Save**.
22. Click **Activate**.
    You can now add the subflow to your primary checkout flow.

Add a Promotions Subflow to a Checkout Flow

To add promotion capability to a checkout flow created before Winter '22, add a subflow.

Your flow must be based on the legacy B2B checkout flow template. If you haven't created a promotion subflow, see Create a Promotion Subflow for an Existing Checkout Flow.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

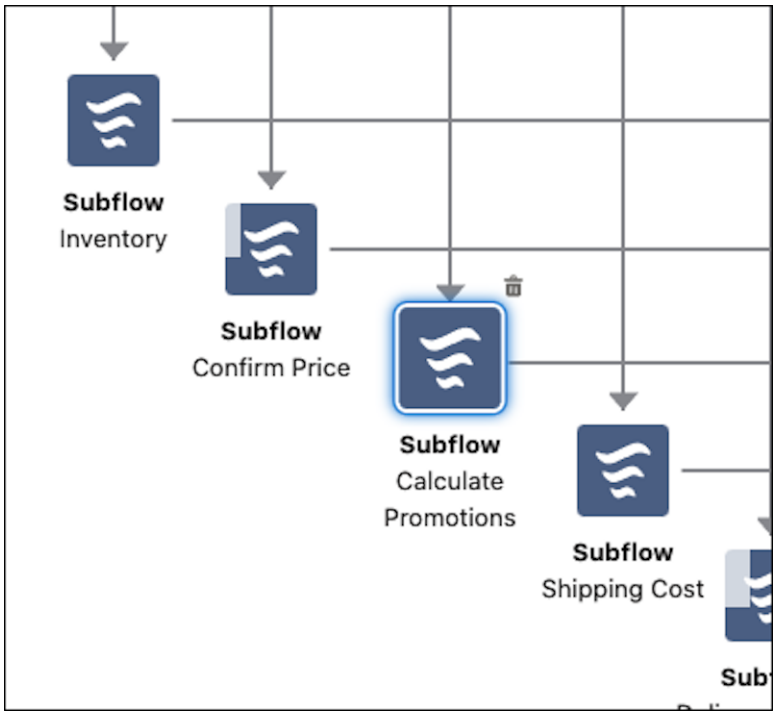
1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find box, enter `Flows`, and then select **Flows**.
3. In Flow Builder, open your main checkout flow.
4. Configure when the subflow is triggered.
   a. Double-click the **Main Decision Hub** node.
   b. On the Edit Decision screen, click **+**.
   c. For the label, enter `Promotions`.
   d. For Conditions Requirement, select **All Conditions Are Met (AND)**.
   e. For Resource, select **mainCheckoutSession > state**.

f. For Operator, select **Equals**.

g. For Value, enter `Promotions`.

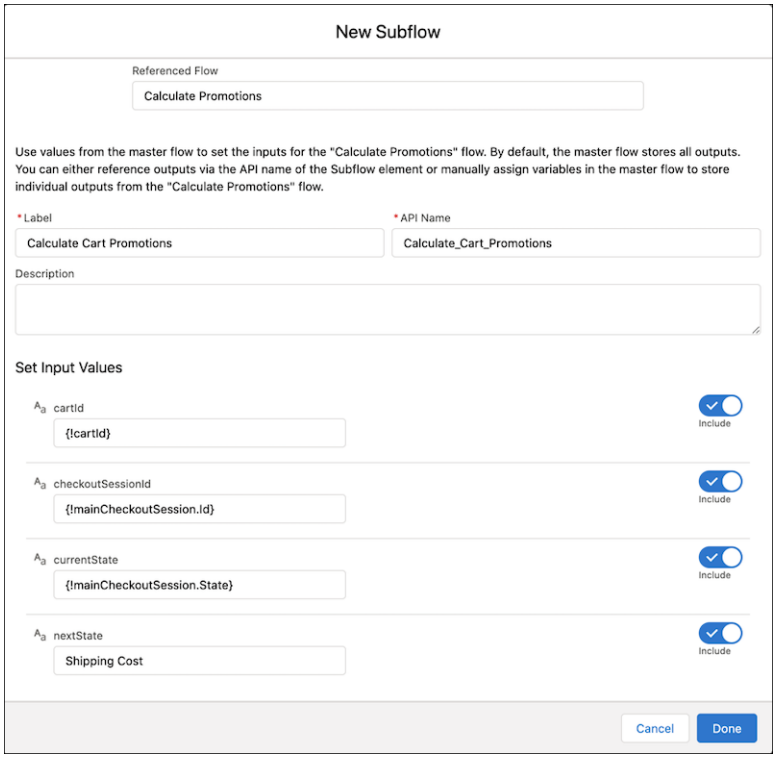
h. Click **Done**.

5. Drag a Subflow element onto the canvas, placing it between the Confirm Price and Shipping Cost subflow nodes.



6. On the New Subflow screen, search for the promotion subflow that you created, and select it.

7. Modify the settings for the new subflow, and then click **Done**.



8. Drag a connector line from the Main Decision Hub to your promotions subflow.
9. On the Select outcome for decision connector screen, select **Promotions**.
10. Drag a connector line from your promotions subflow to the Assignment node.
11. Double-click the **Confirm Price** subflow.
12. On the Edit "Subflow - Confirm Price" Subflow screen, enter `Promotions` as the nextState.
13. Click **Save As** and select **A New Version**.
14. Click **Activate**.

Map the promotions integration to the store.

Map a Promotions Integration to a B2B Store

After you add a promotion subflow to your legacy checkout flow, you map the promotions integration to the store. This task applies only to custom checkout flows created before Winter '22.

1. From the Developer Console, click **Query Editor**, and enter this query.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

```
SELECT Id,Integration, ServiceProviderType, storeId FROM StoreIntegratedService
```

2. Click **Execute**.
3. Click **Insert Row**, and enter these values in the columns.
   a. In the Integration column, enter `Promotions__b2b_storefront__StandardPromotions`.
   b. In the ServiceProviderType column, enter `Promotions`.
   c. In the StoreId column, enter your store's ID.

Configure Purchase Order or Credit Card B2B Flows

After you deploy your flow, choose a purchase order or credit card implementation.

Note: This topic assumes that you deployed the Main, Pricing and Promotions, or Re-Entrant checkout flow.

The Main, Sales and Promotion, and Re-Entrant Flows support purchase order or credit card transactions. Buyers click to choose one or the other on the order summary page. When choosing purchase order, buyers are prompted to enter a PO Number and Billing Address.

Change a B2B Subflow: Asynchronous or Synchronous

You can change a subflow to run synchronously or asynchronously in Flow Builder.

The example used in this task shows how to modify a synchronous Check Inventory subflow to run asynchronously.

Note: This topic assumes that you deployed the Main, Pricing and Promotions, or Re-Entrant checkout flows.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find box, enter `Flows`, and then select **Flows**.
3. Click to select a checkout subflow that runs asynchronously .
4. Drag a New Action element to the canvas, enter `Inventory` in the Action pane, and choose the asynchronous checkCartInventoryAction- checkCartInventoryAction. (The synchronous counterpart is apex_B2BSyncCheckInventory.)
5. Enter a label and API name (such as Check Inventory) for the New Action element.
6. Under cartId, add `{!cartId}`. This variable represents the ID value of the current cart.
7. Click **Done**.
8. Delete the subflow action you want to replace.
9. Drag and connect the new action.
10. Click **Save as** and click **New Flow**.
11. Enter `Asynch Check Inventory` in the Flow Label and Flow API Name fields.
12. To make the new action available, reload the checkout flow.
13. Drag a new subflow onto the canvas.
14. Enter `Inventory` in the Referenced flow pane and choose the new asynchronous subflow.
15. Enter `Asynch Check Inventory` in the label and API Name fields for this subflow.
16. In the Set Input Values, choose `{!cartId}` and also `checkoutsessionId`.
17. Click **Done**.

18. Delete the old Apex Action Check Inventory (the synchronous node).
19. Connect the new Async Check Inventory subflow.
20. Click **Save**.
21. Reload the page.

You can now choose this modified checkout flow in Experience Builder.

Customize a B2B Subflow

You can use the default checkout subflows or modify them by removing, adding, and reordering steps or exchanging part of a default subflow with a custom subflow.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

Note: The default checkout flow doesn't show a Previous button during checkout. Instead, the flow references the checkout state variable to determine which screen to show or which integration to run. If you add a Previous button, make sure that the button resets to the checkout state that corresponds to the screen shown and doesn't revert to the previous checkout state. For example, if an integration runs between screens, reverting to the previous checkout state can result in unexpected behavior or errors.

1. To replace an existing subflow with a custom subflow, drag a new subflow element onto your flow canvas.

   Note: We recommend that you keep the Shipping Address subflow first and the Payments subflow at the end. The Shipping Address subflow determines the information related to shipping that the rest of the flow requires. Ending with the Payments subflow minimizes the number of unused payment authorizations on the payment gateway.

2. Enter or search for the type of subflow that you want to add.
3. For the nextState and cartId input values, select **Include**.
4. Under cartId, add `{!cartId}`.

   This variable represents the ID value of the current cart.

5. Under nextState, add the subflow that comes next.

   Each subflow contains a currentState and a nextState variable. If the nextState variable doesn't accurately reflect the next subflow, the flow can't continue, and the checkout hangs. If you delete or reorder a subflow, make sure to update the nextState variable to the correct state value.

   For example, if you replace the Shipping Address subflow and you want inventory to come next like it does in the default flow, select **Inventory**.

6. Delete the subflow that you want to replace.
7. On the Main Decision Hub element, drag the connector to your new subflow, and then drag the connector on your new subflow to the subflow that follows.
8. Save and activate your flow.

To complete your checkout flow, create Apex classes for pricing, tax, shipping, and inventory. For information about creating Apex classes and customizing your flow, see Lightning B2B Commerce Checkout Flow.

Time Limits and Active Checkouts

You can put a time limit on an active checkout to prevent users from checking out with outdated information. For example, a buyer can start a checkout and then leave to complete other tasks. In the meantime, inventory and pricing can change. With the Time to Live feature, you can set an expiration limit so that when the user returns to their cart, the checkout integrations run again and provide current information.

Configure B2B Checkout Time to Live with Developer Console

You can put a time limit on an active checkout to prevent users from checking out with outdated information.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

1. Locate your commerce store ID.
   a. Navigate to the Commerce app and select your store.
   b. In the URL, find your store ID.

      In this example, the store ID is the string of numbers and letters before /view.

      ```
      https://examplestore.lightning.force.com/lightning/r/WebStore/0TERR00000004XG4AY/view
      ```

   a. From Developer Console, select **Query Editor**.
   b. Copy the following SOQL query to the Query Editor panel, and replace *storeID* with the 15- or 18-digit Salesforce ID of the store.

      ```
      SELECT CheckoutTimeToLive, CheckoutValidAfterDate, Id FROM WebStore WHERE Id =
      'storeID'
      ```

   c. Click **Execute**.
   d. Double-click the value in the `CheckoutTimeToLive` column, and enter the number of minutes that the checkout stays active. The default number of minutes is 2880 (48 hours). The maximum value for CheckoutTimeToLive is 525600 minutes (365 days).
   e. Double-click the value in the `CheckoutValidAfterDate` column, and enter a date. If a checkout starts before this date, it's considered expired. Example format: 2020-07-14T14:27:00.000Z
   f. Click **Save Rows**.

Configure B2B Checkout Time to Live with APIs

You can put a time limit on an active checkout to prevent users from checking out with outdated information.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

1. Locate your commerce store ID.
   a. Navigate to the Commerce app and select your store.
   b. In the URL, find your store ID.

      In this example, the store ID is the string of numbers and letters before /view.

```
https://examplestore.lightning.force.com/lightning/r/WebStore/0TERR00000004XG4AY/view
```

2. After you have your store ID, use the WebStore object to configure TTL.

```
GET
https://yourstore.salesforce.com:6101/services/data/v50.0/sobjects/WebStore/<Your Store
 ID>
```

3. Update the WebStore object.

```
PATCH
https://yourstore.salesforce.com:6101/services/data/v50.0/sobjects/WebStore/<Your Store
 ID>
{
"CheckoutTimeToLive" : <value in minutes, e.g. 5>,
"CheckoutValidAfterDate": "<timestamp, e.g. 2020-07-14T14:27:00.000Z>"
}
```

- CheckoutTimeToLive is the number of minutes that the checkout stays active. If you enter *Null*, the checkout never expires. If you enter *0*, checkout is disabled. The default number of minutes is 2880 (48 hours).
- CheckoutValidAfterDate is a timestamp in the default server timezone (GMT). Example format: 2020-07-14T14:27:00.000Z. If a checkout starts before this date, it's considered expired. A Null value means all checkouts are valid.

Test Your B2B Checkout Flow

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B Commerce

Before you go live, make sure that you thoroughly test your B2B checkout.

Test these areas of the checkout flow.

- Buyer facing components in Experience Builder
- Checkout flow definition and implementation
- Action APIs used in the checkout flow and data mapping

Test the Experience Builder components and the checkout flow manually.

To test the API actions, complete these prerequisites:

- Publish a fully configured and populated store with products, buyers, entitlements, prices, and other necessary store elements.
- Configure a test account with test buyers.
- Ensure that the test orders generated by testing are separated logically from real orders and don't have a physical or financial impact downstream.

1. Create an Autolaunched Flow.
   a. Select **Run as System Context with Sharing-Enforces Record-Level Access**.
   b. Apply the input parameter TestBuyerId.
   c. Activate your flow.
2. In the new test flow:
   a. Remove old tests that ran or failed, and clear the cart for the buyer specified by TestBuyerId.
   b. Use SOAP inserts to add items to the cart for the buyer specified by TestBuyerId.

3. Use the Checkout APIs as actions within the test flow.
   a. Create a checkout session.
   b. Check inventory, and wait for `BackgroundOperation` to complete.
   c. Check prices, and wait for `BackgroundOperation` to complete.
   d. Calculate taxes, and wait for `BackgroundOperation` to complete.
   e. Calculate shipping charges, and wait for `BackgroundOperation` to complete.
   f. Process cart to order, and wait for `BackgroundOperation` to complete

B2B Checkout Flow Notifications

The checkout flow runs through stages, such as pricing, shipping, inventory, taxation, order activation, and confirmation. You can modify the flow to trigger notifications during the various stages and send updates to the buyer throughout the checkout process.

Order Confirmation Notifications

To notify users when their order is successfully placed, use email or in-app notifications.

Create Email Order Confirmation Notifications

To email buyers when their order is confirmed, create an email template, an email alert, a flow that uses the email alert, and a process to invoke your flow.

You can perform all of these tasks declaratively in the Commerce app using templates.

1. Go to GitHub for Commerce on Lightning.
   Contact Salesforce for access to this repository.
2. Open and review the Order Confirmation Email Notification Implementation README file.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce and D2C Commerce

App, Push, and Bell Order Confirmations

Use app, push, or bell notifications to notify buyers that their order is confirmed.

Create an App, Push, or Bell Notification

To send bell, app, or push notifications to buyers when their order is confirmed, create a custom notification to use in a flow.

Note: Bell notifications aren't store specific. If a buyer user is a member of multiple stores, the buyer receives a bell notification in all stores after placing an order, not just the store where the order was placed.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter *`Notification Builder`*, and select **Custom Notifications**.
3. Click **New**.
4. Enter the information in the New Custom Notification Type form.
   a. For Custom Notification Name, enter *`Order Summary Bell Notification Type`*.
   b. For API Name, enter *`Order_Summary_Bell_Notification_Type`*.
   c. Under Supported Channels, select **Desktop** and **Mobile**.

5. Click **Save**.

Add this notification to a flow.

Create a Notification Flow

After you create a custom notification, you can use it in an autolaunched flow.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Process Automation`, and select **Flows**.
3. Create a flow.
4. Select **Autolaunched Flow**, and click **Create**.

Add resources to the autolaunched flow. For each resource listed in the Record Resources table, navigate to the Manager tab, click **New Resource**, and fill in the fields accordingly.

Field	Resource One	Resource Two	Resource Three
Resource Type	Variable	Variable	Variable
API Name	OrderSummaryRecord	orderSummaryRecordID	customNotificationRecordId
Data Type	Record	Record	Record
Object	Order Summary	Order Summary	Custom Notification Type
Allow Multiple Values?	No	No	No
Availability Outside the Flow	Available for Input	None	None

Table 1: Text Resources

Field	Resource One	Resource Two
Resource Type	Variable	Variable
API Name	customNotificationTypeDevName	RecipientId
Data Type	Text	Text
Allow Multiple Values?	No	Yes
Default Value	Order_Summary_Bell_Notification_Type Note: Order_Summary_Bell_Notification_Type refers to the name of your custom notification type.	N/A

On the Element tab, create two Get Records elements using this information.

Field	Element One	Element Two
Label	getOrderSummaryRecord	getCustomNotificationTypeRecord

Field	Element One	Element Two
API Name	getOrderSummaryRecord	getCustomNotificationTypeRecord
Object	Order Summary	Custom Notification Type
Conditions	• Condition requirements: Conditions are met • Field: ID • Operator: Equals • Value: {!OrderSummaryRecord.Id} Note: OrderSummaryRecord refers to the API name of the first variable that you created in this flow.	• Condition requirements: Conditions are met • Field: DeveloperName • Operator: Equals • Value: {!customNotificationTypeDevName} Note: customNotificationTypeDevName refers to the API name of the third variable that you created in this flow.
Sort Order	Not Sorted	Not Sorted
How Many Records to Store	Only the first record	Only the first record
How to Store Record Data	Choose fields and assign variables (advanced)	Choose fields and assign variables (advanced)
Where to Store Field Values	Together in a record variable	Together in a record variable
Record	orderSummaryRecordID	customNotificationRecordId
Select [Object] Fields to Store in Variable	• ID • OwnerId	• ID

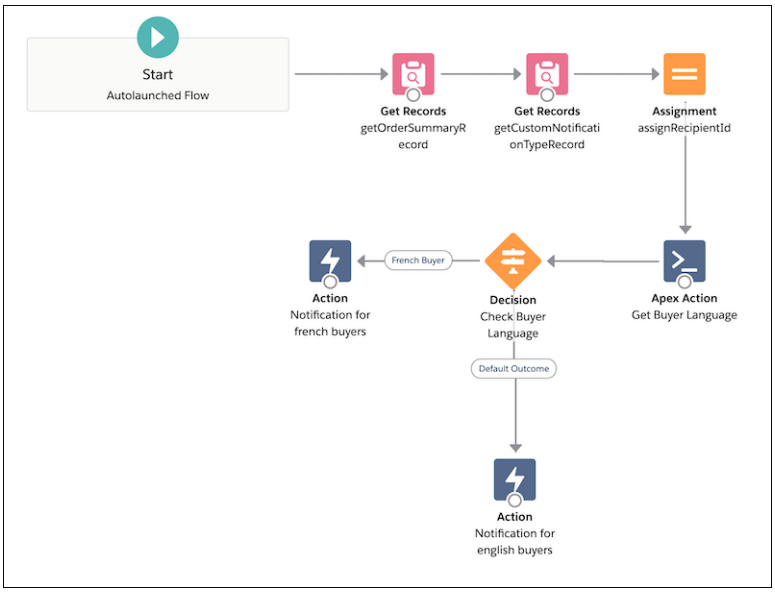
1. On the Element tab, drag an **Assignment** element to the canvas.
2. Fill in the Assignment element form.
   a. For Label, enter *assignRecipientId*.
   b. For **API Name**, enter *assignRecipientId*.
   c. For **Variable**, select **RecipientId**.
   d. For **Operator**, select **Add**.
   e. For **Value**, select **orderSummaryRecordID** and **ownerId**. The field value is `{!orderSummaryRecordID.OwnerId}`.
3. On the Element tab, drag another **Action** element to canvas.
4. Search all actions, and select **Send Custom Notification**.
5. Fill in the Action element form.
   a. For Label, enter *Send Notifications*.
   b. For **API Name**, enter *Send_Notifications*.
   c. For **Custom Notification Type ID**, enter *{!customNotificationRecordId.Id}*.
   d. For **Notification Body** and **Notification Title**, enter your message, such as *Your order is confirmed.*.

e. For **Recipient IDs**, enter `{!RecipientId}`.

f. For **TargetID**, enter `{!orderSummaryRecordID.Id}`.

6. Connect all your elements, and click **Save**.
7. Save your flow.
   a. For Label, enter `Order Summary Bell Notification Flow`.
   b. For API Name, enter `Order_Summary_Bell_Notification_Flow`.
8. Click **Done**, and activate your flow.

Example: Here's an example flow with the elements connected.



Invoke a Notification Flow

After you create a notification flow, add an action to invoke the flow by cloning an existing process.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
3. Select **Notify On Order Summary Created Process**.
4. Select **Clone**, choose **Save Close As**, and select **Version of current process**.
5. Click **Save**.
6. Select **+ Add Action**.
7. Fill in the action form.
   a. For Action Type, select **Flows**.
   b. For **Action Name**, enter `Order Summary Bell Notification Flow Action`.
   c. For **Flow**, select **Order Summary Bell Notification Flow**.
   d. Click **+ Add Row**.
   e. For **Flow Variable**, select **OrderSummaryRecord**.

f. For **Type**, select **Field Reference**.

g. For **Value**, select **Select the OrderSummary record that started your process**, and click **Choose**.

h. Click **Save**.

8. Activate your process.

Checkout Stage Notifications

The checkout process contains various intermediate stages, including pricing, inventory, tax, and shipping. Some of these stages connect to a third party, so they can take more time to complete. Using platform events, Process Builder, and Flow Builder, you can notify the buyer when a stage is complete.

Intermediate Checkout Stage Notifications Overview

Using platform events, Process Builder, and Flow Builder, you can notify the buyer when an integration is complete.

Intermediate checkout stages depend on the cartCheckoutSession object. When cartCheckoutSession completes a stage, it moves to the next stage. See B2B Checkout States for more information.

There are many ways to implement an intermediate checkout notification. In the example detailed in this section, we create a platform event and use Process Builder to publish and subscribe to the event. The buyer receives notifications after completion of inventory, price confirmation, shipping cost, and taxation stages.

Create a Platform Event for Checkout Notifications

Create a platform event to notify users that an intermediate checkout stage is complete.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Platform Events`, and select **Platform Events**.
3. Create a platform event.
4. Fill in the New Platform Event form.
   a. For Label, enter `Checkout Intermediate Notification`.
   b. For **Plural Label**, enter `Checkout Intermediate Notifications`.
   c. For **Object Name**, select **Checkout_Intermediate_Notification**.
   d. For **Publish Behavior**, select **Publish After Commit**.
   e. Click **Save**.
5. Create a field with a Text data type.
   a. In the Custom Fields and Relationships section, click **New**.
   b. For Data Type, select **Text**, and click **Next**.
   c. For Field Label, enter `Cart Session ID`
   d. For Length, enter `255`.
   e. For Field Name, enter `Cart_Session_ID`
   f. Click **Save**.
6. Create two more custom text fields, and label them `Next State` and `State`.

Create a Process to Publish a Checkout Notification Platform Event

After you create a platform event, you can use Process Builder to publish it.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
3. Create a process.
4. Fill in the New Process form.
   a. For Process Name, enter `Publish Event On Checkout Intermediate Operation Process`.
   b. For **API Name**, enter `Publish_Event_On_Checkout_Intermediate_Operation_Process`.
   c. For when to start the process, select **A record changes**.
5. Save your process.
6. After Process Builder launches, click **Add Object**.
   a. For Object, select **Cart Checkout Session**.
   b. For when to start the process, select **when a record is created or edited**.
7. Click **Add Criteria**.
   a. For **Criteria Name**, enter `Publish Checkout Intermediate Notification Event`.
   b. For **Criteria for Executing Actions**, select **No criteria-just execute the actions!**
   c. Click **Save**.
8. Click **Add Action**.
   a. For **Action Type**, select **Create a Record**.
   b. For **Action Name**, enter `Publish Intermediate Notification`.
   c. For **Record Type**, select the platform event that you created.
   d. Set field values.
      - For Field, select **Cart Session ID**.
      - For **Type**, select **Field Reference**.
      - Under **Value**, select **Cart Checkout Session ID**, and click **Choose**.
      - Click **Add Row**.
      - Under **Field**, select **Next State**.
      - For **Type**, select **Field Reference**.
      - Under **Value**, select **Next State**, and click **Choose**.
      - Click **Add Row**.
      - Under **Field**, select **State**.
      - For **Type**, select **Field Reference**.
      - Under **Value**, select **State**, and click **Choose**.
   e. Click **Save**.
9. Save and activate your process.

Create an Email Notification

Create a notification to email users when an intermediate checkout stage is complete.

1. Create an email template to notify buyers when an integration phase is complete. For step-by step-instructions, see https://help.salesforce.com/articleView?id=sf.comm_email_templates.htm.
2. Create an email alert. For a step-by-step guide, see https://help.salesforce.com/articleView?id=sf.comm_create_email_alert.htm.
   a. For Object, select **cartCheckoutSession**.
   b. For the recipient, select **Record Creator**.

Example: Here's an example template that you can use.

```
Hello {!OrderSummary.OwnerFullName},

An intermediate checkout stage is complete.

Please click here to continue: https://<Store-url>/s/checkout/{!CartCheckoutSession.Name}

Thanks,
{!Organization.Name}
```

Create a Flow to Send an Email Alert

After you create an email template and an email alert, you can create a flow to trigger the alert.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter *Process Automation*, and select **Flows**.
3. Create a flow.
4. Select **Auto launched Flow**, and click **Create**.
5. In Flow Builder, select the **Manager** tab, and click **New Resource**.
6. Fill in the New Resource form.
   a. Under **Resource Type,** select **Variable**.
   b. For **API Name**, enter *CartCheckoutSessionRecord*.
   c. Under **Data Type**, select **Record**.
   d. Under **Object**, select **Cart Checkout Session**.
   e. For **Availability Outside the Flow**, select **Available for input**.
7. Click **Done**.
8. Back in Flow Builder, select the Elements tab. Drag the Action element to the canvas.
9. Fill in the New Action form.
   a. For **Filter By**, select **Type**.
   b. Select **Email Alert**.
   c. Click **Search email alerts**, and select the alert you created.
   d. Label your action *Checkout Stage Email Notification Alert Action*.
   e. For the **API Name**, enter *Checkout_Stage_Email_Notification_Alert_Action*.

f. Under **Record ID**, enter `{!CartCheckoutSessionRecord.Id}`.

10. Click **Done**.
11. Link the two flow actions by dragging a connection between them.
12. Save and activate your flow.

Create a Flow for Bell Notifications

Create a flow that triggers bell notifications for intermediate checkout steps.

Follow the steps in Create a Notification Flow using these variables and elements.

1. Create the input variable `cartCheckoutSessionRecord` using the Record data type and the Cart Checkout Session object.
2. Create the variable `cartSessionRecordID` using the Record data type and the Cart Checkout Session object.
   This variable stores the output of the getRecord lookup for the Cart Checkout Session object.
3. Create the `Get Records` element on the Cart Checkout Session object, and store the values of the `CreatedById` and `WebCartId` fields in the output.
4. Create the variable `recipientId` using the Text data type, and accept multiple values.
5. Create the `Assignment Logic` element, and add the `{!cartSessionRecordID.CreatedById}` value to recipientId.
6. Create an Action element called `Send Custom Notification`. For Target ID in send custom notification, enter `{!cartSessionRecordID.WebCartId}`. For bell notifications, Target ID accepts only record IDs, not URLs, so you can't redirect the user directly to the checkout flow.

Create a Process to Call a Flow

Create a process that calls your notification flow when your platform event occurs.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
3. Click **New**.
4. Fill in the New Process form.
   a. For Process Name, enter `Notify On Checkout Intermediate Notification Event`.
   b. For API Name, enter `Notify_On_Checkout_Intermediate_Notification_Event`.
   c. For when to start the process, select **A platform event message is received**.
5. Click **Save**.
6. After Process Builder launches, click **Add Trigger**.
   a. For Platform Event, select **Checkout Intermediate Notification**.
   a. For Object, select **Cart Checkout Session**.
   b. Provide matching conditions.
      a. Under Field, select **Cart Checkout Session ID**.
      b. For Operator, select **Equals**.
      c. For Type, select **Event Reference**.
      d. Under Value, select **Cart Session ID**.

c. Click **Save**.

7. Click **Add Criteria**.
   a. For Criteria Name, enter `Checkout Intermediate Operation Criteria`.
   b. For Criteria for Executing Actions, select **Conditions are Met**.
   c. Create five conditions with this criteria.

Field	One	Two	Three	Four	Five
Source	Platform Event	Platform Event	Platform Event	Platform Event	Platform Event
Field	State	State	State	State	Next State
Operator	Equals	Equals	Equals	Equals	Is Null
Type	String	String	String	String	Boolean
Value	Confirm Price	Shipping Cost	Taxes	Checkout Summary	True

8. For Conditions, select **Customize the Logic**.
9. For Logic, select (1 OR 2 OR 3 OR 4) AND 5.
10. Click **Save**.
11. Select **+ Add Action**.
12. Fill in the New Action form.
    a. For Action Type, select **Flows**.
    b. For Action Name, enter `Trigger intermediate checkout notification`.
    c. For Flow, select the flow that you created in the Bell notifications step.
13. Save and activate your process.

Make Notifications Optional for Users

Let users choose whether to receive email or app notifications that you create for the B2B Commerce checkout.

Let Users Opt Out of Checkout Notifications

Add a checkbox to let users choose whether to receive checkout notifications.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Object Manager`, and select **User**.
3. Select **Fields and Relationships**, and click **New**.
4. For Data Type, select **Checkbox**.
5. Under Field Label, enter `Checkout Notifications`.
6. Select the default value.

   If you select **Checked**, users are automatically opted in to checkout notifications. They can choose to opt out by deselecting the box.

7. For Field Name, enter *Checkout_Notifications*, and click **Next**.
8. Choose which profiles see the field, and click **Next**.

   We suggest choosing the buyer user profiles.
9. Add the field to the User Profile Layout.
10. Click **Save**.

Write an Apex Action to Make Notifications Optional

After you add a checkbox to your user profiles, create custom Apex code to create a notification setting for your buyer users.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter *Custom Code*, and select **Apex Classes**.
3. Click **New**.
4. Copy this code sample or write your own.

Example: Here's an example Apex class.

```
public class B2BIfNotifyBuyer {
 @InvocableMethod(label='GetBuyerNotificationSetting' description='Retrieve checkout
notify setting for buyer user')
 public static List<String> getSetting(List<ID> orderSummaryIds) {
 List<String> output = new List<String>();
 List<OrderSummary> orderSummaries = [SELECT TYPEOF OriginalOrder.Owner WHEN User THEN
 Checkout_Notifications__c END FROM OrderSummary WHERE Id in :orderSummaryIds];
 for (OrderSummary orderSummary : orderSummaries) {
 if (orderSummary.OriginalOrder.Owner instanceof User){
 User userOwner = orderSummary.OriginalOrder.Owner;
 output.add(String.valueOf(userOwner.Checkout_Notifications__c));
 }
 }
 return output;
 }
 }
```

Add Optional Notifications to an Email Flow

To let users choose whether to receive email notification, modify your email order notification flow.

This task assumes that you've created an email notification flow. See https://help.salesforce.com/articleView?id=sf.comm_create_a_flow_email.htm.

1. On the Element tab, drag an **Action** element to the canvas.
2. Fill in the New Action form.
   a. Search all actions, and select your Apex action to make notifications optional.
   b. For Label and API Name, enter *GetNotificationSetting*.
   c. Turn on **Set Input Values**.
   d. For orderSummaryIds, enter *{!OrderSummaryRecord.Id}*.
   e. Click **Done**.

3. On the Element tab, drag a **Decision** element to the canvas.
4. Fill in the New Decision form.
   a. For Label and API Name, enter `NotificationDecision`.
   b. Fill in the Outcome Details.
      a. For Label and Outcome API Name, enter `NotificationsEnabled`.
      b. For when to execute the outcome, select **All Conditions are Met**.
      c. For Resource, enter the label and API name used for the Action element.
      d. For Operator, select **Equals**.
      e. For Value, select **{!$GlobalConstant.True}**.
5. Click **Done**.
6. Connect all your elements, and click **Save**.
7. Activate your flow.
8. Repeat these steps for all flows that send email notifications or in-app notifications.

Localize Checkout Notifications

You can set up your B2B Commerce store to provide your users language options. You can create a flow that sends notifications for each language.

Localize a Notification

Create an Apex action that retrieves the buyer's selected language.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Custom Code`, and select **Apex Classes**.
3. Click **New**.
4. Write an Apex action with an invocable method that retrieves the language that the user chooses.

Example: You can use this code in your Apex action.

```
public class B2BGetBuyerUserLanguageAction {
        @InvocableMethod(label='GetBuyerUserLanguageAction' description='Retrieve
language of buyer user')
        public static List<User> getBuyerUserLanguage(List<ID> orderSummaryIds) {
            List<User> output = new List<User>();
            List<OrderSummary> orderSummaries = [SELECT TYPEOF OriginalOrder.Owner
WHEN User THEN LocaleSidKey, LanguageLocaleKey END FROM OrderSummary WHERE Id in
:orderSummaryIds];
            for (OrderSummary orderSummary : orderSummaries) {
                if (orderSummary.OriginalOrder.Owner instanceof User){
                User userOwner = orderSummary.OriginalOrder.Owner;
                output.add(userOwner);
            }
        }
        return output;
```

```
        }
}
```

Create a Flow for Localized Notifications

Create a flow that uses your Apex action to localize notifications.

This task creates a flow for bell notifications, but you can create a similar flow for email notifications. See App, Push, and Bell Order Confirmations.

1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find Box, enter `Process Automation`, and select **Flows**.
3. Create a flow.
4. Select **Autolaunched Flow**, and click **Create**.
5. On the Manager tab, click **New Resource**.
6. Fill in the New Resource form.
   a. For Resource Type, select **Variable**.
   b. For **API Name**, enter `UserLanguage`.
   c. For **Data Type**, select **Record**.
   d. For **Object**, enter `User`.
7. On the Element tab, drag an **Action** element to the canvas.
8. Fill in the New Action form.
   a. For Filter By, select **Type**.
   b. Click **Apex Action**.
   c. Click **Search Apex Actions**, and select **GetBuyerUserLanguageAction**.
   d. For Label, enter `Get Buyer Language`.
   e. For API Name, enter `Get_Buyer_Language`.
   f. Under Set Input Values, include **orderSummaryIds**.
   g. Select **Enter value of search resources**, and enter `{!OrderSummaryRecord.Id}`.
   h. Select **Manually assign variables (advanced)**.
   i. Click **Search variables**, and enter `{!UserLanguage}`.
   j. Click **Done**.
9. On the Element tab, drag a **Decision** element to the canvas.
10. Fill in the New Decision form.
    a. For Label, enter `Check Buyer Language`.
    b. For API Name, enter `Check_Buyer_Language`.
    c. Fill in the Outcome Detail.
       a. For **Label**, enter `French Buyer`.
       b. For **Outcome API Name**, enter `French_Buyer`

c. For **When to Execute Outcome**, select **All Conditions are Met**.

d. For **Resource**, enter `{!UserLanguage.LanguageLocaleKey}`, which uses the output of the Apex class created in Localize a Notification.

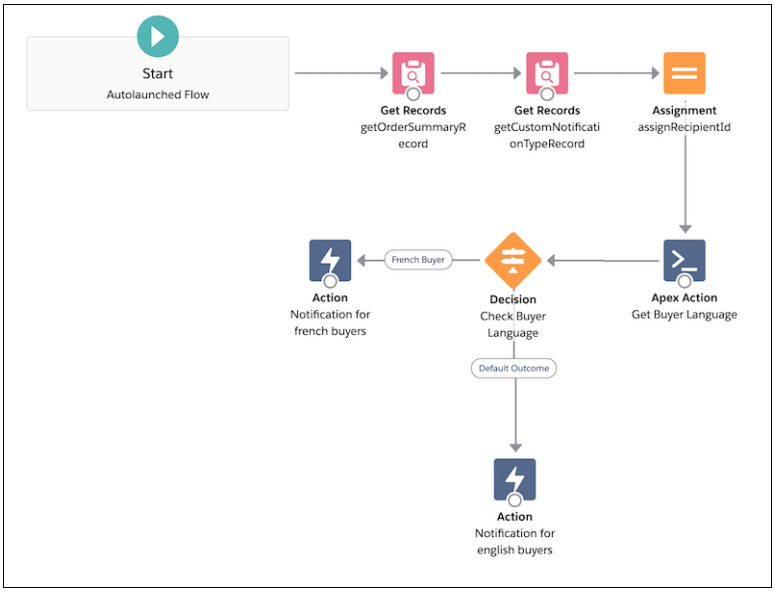
e. For **Operator**, select **Equals**.

f. For **Value**, enter `fr`.

g. Click **Done**.

11. On the Element tab, drag an **Action** element to the canvas.
12. Fill in the Action element form.
    a. For Search All Actions, select **Send Custom Notifications**.
    b. For **Label**, enter `Notifications for French Buyers`.
    c. For **API Name**, enter `Notifications_for_French_Buyers`.
    d. For **Custom Notification Type ID**, enter `{!customNotificationRecordId.Id}`.
    e. For **Notification Body** and **Notification Title**, enter `Votre commande a été confirmée`. You can also customize this message.
    f. For **Recipient IDs**, enter **{!RecipientId}**.
    g. For **TargetID**, enter `{!orderSummaryRecordID.Id}`.
    h. Click **Done**.
13. On the Element tab, drag another **Action** element to the canvas.
14. Fill in the Action element form.
    a. For **Search All Actions**, select **Send Custom Notifications**.
    b. For **Label**, enter `Notifications for English Buyers`.
    c. For **API Name**, enter `Notifications_for_English_Buyers`.
    d. For **Custom Notification Type ID**, enter `{!customNotificationRecordId.Id}`.
    e. For **Notification Body** and **Notification Title**, enter `Your Order is Confirmed`. You can also customize this message.
    f. For **Recipient IDs**, enter **{!RecipientId}**.
    g. For **TargetID**, enter `{!orderSummaryRecordID.Id}`.
    h. Click **Done**.
15. Connect your elements. Start connects to getOrderSummaryRecord, connects to getCustomNotificationTypeRecord, connects to assignRecipientId, connects to Get Buyer Language, connects to Check Buyer language, which connects to both Notification for English Buyers and Notification for French Buyers. Notification for English Buyers is the default outcome.

Example: Here's an example flow with the elements connected.



Create Third-Party Integrations with Platform Events

To trigger notifications using an external, or third-party, system, subscribe to Salesforce platform events.

You can subscribe to browser events or processes running on the Salesforce platform using Streaming API, which is built on the Bayeux protocol. Subscribe to events upon order confirmation or for intermediate checkout stages.

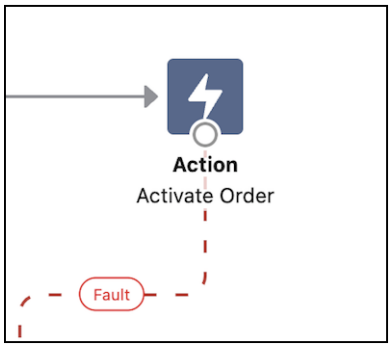
- **Order confirmation**—When an OrderSummary sObject is created successfully, Salesforce triggers the OrderSummaryCreatedEvent. Applications can subscribe to this platform event and send notifications from external systems to Salesforce.
- **Intermediate checkout stages**—Create a custom platform event, such as the event described in Create a Platform Event for Checkout Notifications. Applications can subscribe to your platform event and send notifications from systems external to Salesforce.

Configure B2B Checkout Flows to Create Managed Order Summaries

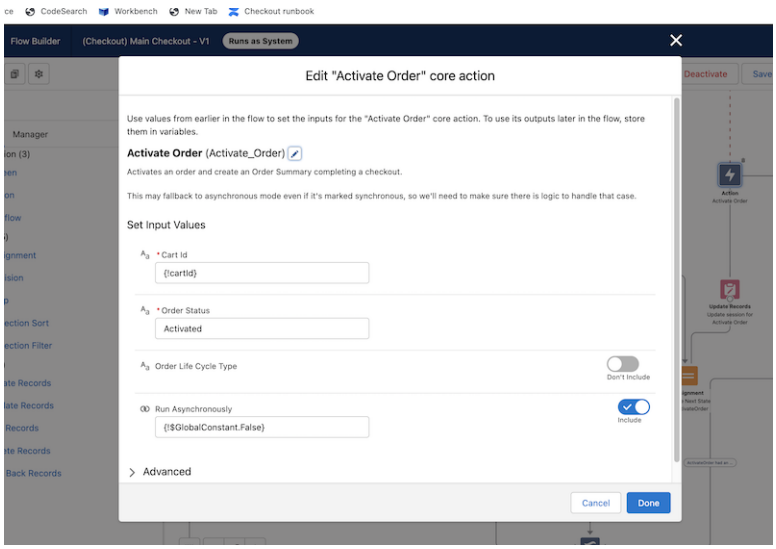
Configure your B2B checkout flow to integrate Salesforce Order Management.

Note: Before using Salesforce Order Management, you must purchase a license and enable it for your org.

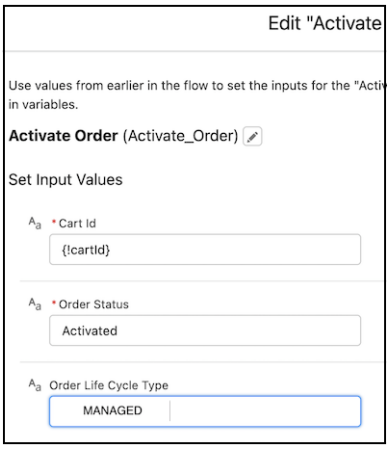
1. Click at the top of the page, and then select **Setup**.
2. In the Quick Find box, enter Flows, and then select **Flows**.
3. Click your checkout flow.
4. In Flow Builder, double-click the **Activate Order** action.



5. If the Order Life Cycle Type field is not visible in the “Activate Order” action, toggle the **Don’t Include** control to display it.



6. In the Order Life Cycle Type input field, enter **MANAGED**.



7. Click **Done**.
8. In Flow Builder, click **Save**.

Import and Export Lightning B2B Commerce Order Summaries

You can export order summaries created by the Lightning B2B Checkout flow to an external order management system and then import them back into Salesforce.

Order Summaries

When a buyer places an order in Lightning B2B Commerce using the Checkout flow, two records are created: an order record and an order summary record.

What Happens When a Customer Places an Order

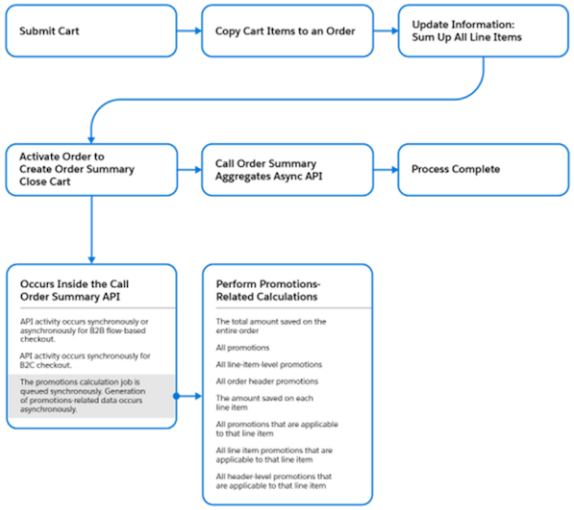
When a customer places an order, several events occur.

EDITIONS

Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

EDITIONS

Available in: B2B and D2C Commerce



1. The order is submitted.
2. The items in the customer's cart are copied to an order.
3. Cart and order information is updated with line item totals.
4. The order is activated to create the order summary, and the cart is closed.

   This step occurs inside the CreateOrderSummary API transaction.
5. The OrderSummaryAdjustmentAggregate API is called.
6. The order placement process is completed.

Managed and Unmanaged Order Summaries

When you use the Order Summary object for Lightning B2B Commerce, the order summary lifecycle is often managed by an external order management system.

When updates are made in your external system, they must be reflected in Salesforce. Use the OrderLifeCycleType field on the Order Summary object to manage external system information by selecting the `unmanaged` value.

- The `managed` value identifies an order summary created by Salesforce Order Management.
- The `unmanaged` value identifies an order summary created by Lightning B2B Commerce.

Note: Users with the Edit Unmanaged Order Summaries or B2B Commerce Integrator User permission can export or update an order summary with an `unmanaged` value.

Map Cart Data to Order Data

During checkout, B2B Commerce and D2C Commerce create orders and order summaries using information from the cart. Many fields in the cart map to order fields and to order summary fields. In addition, fields in other order-related objects are mapped to fields in various other objects, including cart-related objects. If you understand how the various fields are mapped, you can plan accordingly. Also, if you follow certain patterns, you can ensure that custom fields are mapped successfully.

Several objects hold cart-related data and order-related data. We describe which fields are required to create an order record and an order summary record. We also indicate how the fields of order-related objects map to other fields.

Note: Starting in Spring '23, the Cart to Order action supports the automatic mapping of custom fields if you follow certain patterns.

To be mapped successfully, a custom field must exist in the Cart, Order, and OrderSummary objects. It can also optionally exist in the OrderAdjustmentGroup, OrderDeliveryGroup, OrderItem, OrderItemAdjustmentLineItem, and OrderItemTaxLineItem objects.

Across all of these objects, the custom field must have the same API name, same data type, and same field configuration (length, precision, field-level security, and so on). The data type of the field must be one of the following:

- Checkbox
- Currency
- Date
- DateTime
- Email
- Html
- LongTextArea
- Number
- Percent
- Phone
- Text
- TextArea
- Url

If the system finds a custom field that matches across these objects, the field is automatically mapped.

Mapping Order Data

The following tables list fields for order-related objects. The tables don't show a comprehensive list of all fields for each object. However, the tables do include all fields that must be populated to create an order record or an order summary record. The tables also show information about other fields whose values are mapped to other objects.

When an order field is populated, or mapped, the system copies the value from a cart object and inserts it into the order object without modification. If you want to import orders from an external system, use this information to determine which fields are necessary for a functional checkout.

Note:

- If a field is denoted with the `*Generated at runtime` value, the value of the field is unique and generated when the record is created.

- If a field is denoted with `*Provided by shipping integration`, the values can also be provided using your shipping integration. For an example, see our sample shipping integration.

Order Object

Order Field	Required for Order Summaries	Required for Orders	Field Type	Value
`Id`	Yes	Yes	EntityId	*Generated at runtime
`AccountId`	No	Yes	EntityId	*`WebCart.AccountId`*
`EffectiveDate`	Yes	Yes	Date	*`CreatedDate`*
`BillingCity`	No	No	Address	*`WebCart.BillingCity`*
`BillingCountry`	No	No	Address	*`WebCart.BillingCountry`*
`BillingEmailAddress`	No	No	Email	*`WebCart.GuestEmailAddress`*
`BillingLatitude`	No	No	Address	*`WebCart.BillingLatitude`*
`BillingLongitude`	No	No	Address	*`WebCart.BillingLongitude`*
`BillingPhoneNumber`	No	No	Phone	*`WebCart.GuestPhoneNumber`*
`BillingPostalCode`	No	No	Address	*`WebCart.BillingPostalCode`*
`BillingStreet`	No	No	Address	*`WebCart.BillingStreet`*
`BillingState`	No	No	Address	*`WebCart.BillingState`*
`CurrencyIsoCode`	Yes	Yes	CurrencyCode	*`WebCart.CurrencyIsoCode`* Note: This field is only required if MultiCurrency is enabled.
`OrderedDate`	No	No	DateTime	*`CreatedDate`*
`OwnerId`	No	Yes	Reference	*`WebCart.OwnerId`*
`PoNumber`	No	No	String	Collected at checkout.
`SalesStore`	No	Yes	EntityId	*`WebCart.WebStoreId`*
`Status`	Yes	Yes	DynamicEnum	*`Draft`*
`TaxLocaleType`	Yes	Yes	EntityId	*`WebCart.TaxType`*

OrderItem Object

OrderItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`GrossUnitPrice`	Yes	Yes	Currency	*`WebCart.GrossUnitPrice`*
`Id`	Yes	Yes	EntityId	*Generated at runtime
`OrderId`	Yes	Yes	EntityId	*`Order.Id`*
`OrderDeliveryGroupId`	Yes	No	EntityId	ID of `OrderDeliveryGroup` **Note:** *`CartItem.CartDeliveryGroupId`* points to the `CartDeliveryGroup`, which is used to create an `OrderDeliveryGroup`.
`Product2Id`	Yes	Yes	EntityId	*`CartItem.Product2Id`*
`Quantity`	Yes	Yes	Double	*`CartItem.Quantity`*
`TotalLineAmount`	No	Yes	Currency	*`CartItem.TotalLineAmount`*, *`TotalLineNetAmount`*
`Type`	No	Yes	Picklist	*`CartItem.Type`* **Note:** `Product` and `Charge` are the only types allowed. When *`CartItem.Type`* is set to `Product`, *`Order.Type`* is set to `Order Product`. When *`CartItem.Type`* is set to `Charge`, *`Order.Type`* is set to `Delivery Charge`.
`UnitPrice`	Yes	Yes	Currency	*`CartItem.SalesPrice`* or *`CartItem.ListPrice`* if `SalesPrice` is empty, *`NetUnitPrice`*
`ListPrice`	No	No	Currency	*`CartItem.ListPrice`* or *`CartItem.SalesPrice`* if `ListPrice` is empty.

OrderDeliveryGroup Object

OrderDeliveryGroup Field	Required for Order Summaries	Required for Orders	Field Type	Value
`Id`	Yes	Yes	EntityId	*Generated at runtime
`DeliverToStreet`	No	No	Address	*`CartDeliveryGroup.DeliverToStreet`*
`DeliverToCity`	No	No	Address	*`CartDeliveryGroup.DeliverToCity`*
`DeliverToState`	No	No	Address	*`CartDeliveryGroup.DeliverToState`*
`DeliverToPostalCode`	No	No	Address	*`CartDeliveryGroup.DeliverToPostalCode`*
`DeliverToCountry`	No	No	Address	*`CartDeliveryGroup.DeliverToCountry`*
`DeliverToLatitude`	No	No	Address	*`CartDeliveryGroup.DeliverToLatitude`*
`DeliverToLongitude`	No	No	Address	*`CartDeliveryGroup.DeliverToLongitude`*
`DeliveryInstructions`	No	No	TextArea	*`CartDeliveryGroup.ShippingInstructions`*
`DesiredDeliveryDate`	No	No	Date	*`CartDeliveryGroup.DesiredDeliveryDate`*
`DeliveryMethodId`	Yes	Yes	EntityId	*`CartDeliveryGroup.OrderDeliveryMethodId`*
`OrderId`	Yes	Yes	EntityId	*`Order.Id`*
`DeliverToName`	Yes	Yes	Text	*`CartDeliveryGroup.DeliverToName`* Note: Defaults to “Deliver To”

OrderItemAdjustmentLineItem Object

OrderItemAdjustmentLineItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`Id`	Yes	Yes	EntityId	*Generated at runtime
`Amount`	Yes	Yes	Currency	*`CartItem.NetAdjustmentAmount`*
`Name`	No	Yes	Text	“Price Adjustment”
`OrderItemId`	Yes	Yes	EntityId	*`OrderItem.Id`*

When you enable promotions, the following OrderItemAdjustmentLineItem fields are also mapped.

Note: The `Amount` and `Name` fields are mapped from a different source than when promotions are disabled

OrderItemAdjustmentLineItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`AdjustmentCause`	No	No	EntityId	*`CartItemPriceAdjustment.PriceAdjustmentCause`*
`Amount`	Yes	Yes	Currency	*`CartItemPriceAdjustment.TotalAmount`*, *`TotalNetAmount`*
`Description`	No	No	TextArea	*`CartItemPriceAdjustment.Description`*
`Name`	No	Yes	Text	*`CartItemPriceAdjustment.Name`*
`OrderAdjustmentGroup`	No	No	EntityId	*`CartItemPriceAdjustment.WebCartAdjustmentGroup`*
`Priority`	No	No	Integer	*`CartItemPriceAdjustment.Priority`*

OrderItemTaxLineItem Object

OrderItemTaxLineItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`Id`	Yes	Yes	EntityId	*Generated at runtime
`Amount`	Yes	Yes	Currency	*`CartTax.Amount`* or *`CartItem.AdjustmentTaxAmount`* if this field contains a tax adjustment.
`Name`	Yes	Yes	Text	*`CartTax.Name`* or "Tax Adjustment" if field contains a tax adjustment.
`OrderItemId`	Yes	Yes	EntityId	*`Orderitem.Id`*
`OrderItemAdjustmentLineItem`	No	No	EntityId	• *`OrderItemAdjustmentLineItem.Id`*, if this field contains a tax adjustment • Otherwise, this field is empty.
`TaxEffectiveDate`	Yes	Yes	Date	*`CartTax.TaxCalculationDate`* or a past date if `TaxEffectiveDate` is empty. If this field contains a tax adjustment, it's the current date.
`Type`	Yes	Yes	StaticEnum	*`CartTax.TaxType`* or "Estimated" if this field contains a tax adjustment.
`Rate`	No	No	Percent	*`CartTax.TaxRate`* or empty if this field contains a tax adjustment.
`Description`	No	No	Text	*`CartTax.Description`* or empty if this field contains a tax adjustment.

OrderAdjustmentGroup Object

Note: The OrderAdjustmentGroup object is available only when you enable promotions.

OrderAdjustmentGroup Field	Required for Order Summaries	Required for Orders	Field Type	Value
`Id`	No	No	EntityId	*Generated at runtime
`Description`	No	No	TextArea	*`WebCartAdjustmentGroup.Description`*
`Name`	No	Yes	Text	*`WebCartAdjustmentGroup.Name`*
`OrderId`	No	Yes	EntityId	*`Order.id`*
`PromotionAdjustmentCause`	No	No	EntityId	*`WebCartAdjustmentGroup.PriceAdjustmentCause`*
`Type`	No	No	Picklist	*`WebCartAdjustmentGroup.AdjustmentTargetType`*

Disable Cart to Order Mappings for Custom Fields

Starting in Spring '23, the Cart to Order action supports the automatic mapping of custom fields. This feature is enabled by default, but you can disable it if needed.

You use Developer Console to disable the automatic mapping of custom fields supported by the Cart to Order action.

1. Locate your commerce store ID.
   a. Navigate to the Commerce app and select your store.
   b. In the URL, find your store ID.
      In this example, the store ID is the string of numbers and letters before /view.
      `https://examplestore.lightning.force.com/lightning/r/WebStore/0TERR00000004XG4AY/view`
2. Set the value of the OptionsCartToOrderAutoCustomFieldMapping to false.
   a. From Developer Console, select **Query Editor**.
   b. Copy the following SOQL query to the Query Editor panel, and replace *`storeID`* with the 15- or 18-digit Salesforce ID of the store.

      ```
      SELECT OptionsCartToOrderAutoCustomFieldMapping, Id FROM WebStore WHERE Id = 'storeID'
      ```

   c. Click **Execute**.
   d. Double-click the value in the `OptionsCartToOrderAutoCustomFieldMapping` column, and set the value to false.
   e. Click **Save Rows**.

Exportable and Updatable Summary Objects

You can update and export a variety of order-related objects in Lightning B2B Commerce.

- OrderSummary
- OrderItemSummary

- OrderItemAdjustmentLineSummary
- OrderItemTaxLineItemSummary
- OrderAdjustmentGroupSummary
- OrderDeliveryGroupSummary
- OrderPaymentSummary

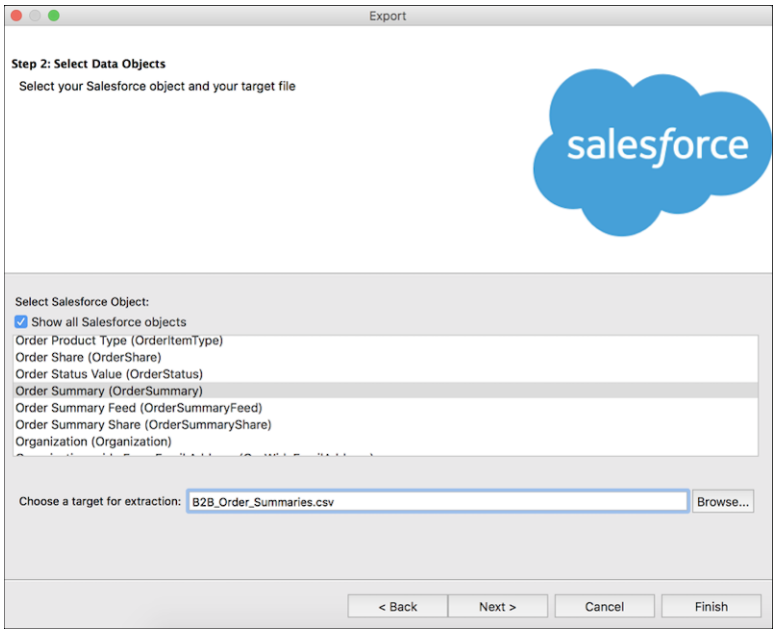
Export or Update Unmanaged Order Summaries

Use Salesforce Data Loader to bulk export or update unmanaged order summaries created by Lightning B2B Commerce. When you export data with Data Loader, you can use a comma-separated values (CSV) file or a database connection. Data Loader exports to a CSV file. You can also use other solutions, like dataloader.io, MuleSoft, or direct API access.

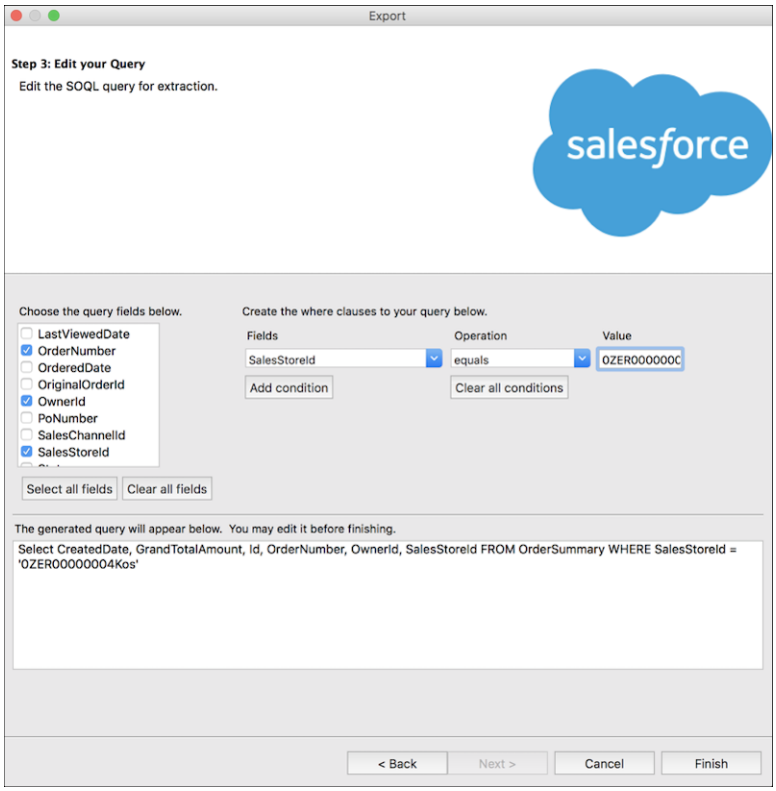
Export Unmanaged Order Summaries with Data Loader

Export your B2B Commerce order summary data out of Salesforce using Data Loader.

1. Log in to Data Loader.
2. Select **Export**.
3. Enter the credentials for the org where your B2B order summaries are.
4. Under Select Salesforce Object, select an order summary or a related summary object.
5. For the extraction target, enter the name of your output CSV file and click **Next**.



6. On the Edit Your Query page, select the fields that you want to export, and under Fields, select the ID that you want to export from.



7. Add the `OrderLifeCycleType = 'UNMANAGED'` clause. This clause is required if you want to limit your export to B2B Commerce order summaries.
8. To edit or create a Where statement when querying data from a summary object rather than an order summary, include the `OrderSummaryId`.
9. When you export data periodically, use the `CreatedDate` field with the SOQL supported values. For example, `CreatedDate < LAST_90_DAYS`.

   Data Loader generates a query that you can edit.
10. Click **Finish** and view the extracted results.

Example: These example queries illustrate how you can export data from order summaries.

Object	Example Query
OrderSummary	Select Id, CreatedDate, OrderNumber, OrderedDate, OwnerId, Status, TotalAmount, FROM OrderSummary **WHERE SalesStoreId = '0ZER00000004Kos' AND OrderLifeCycleType = 'UNMANAGED'**
OrderItemSummary	Select Id, CreatedDate, Product2Id, ProductCode, Quantity, Status, Type, UnitPrice FROM OrderItemSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderItemAdjustmentLineSummary	Select Id, Amount, CreatedDate, TotalTaxAmount FROM OrderItemAdjustmentLineSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**

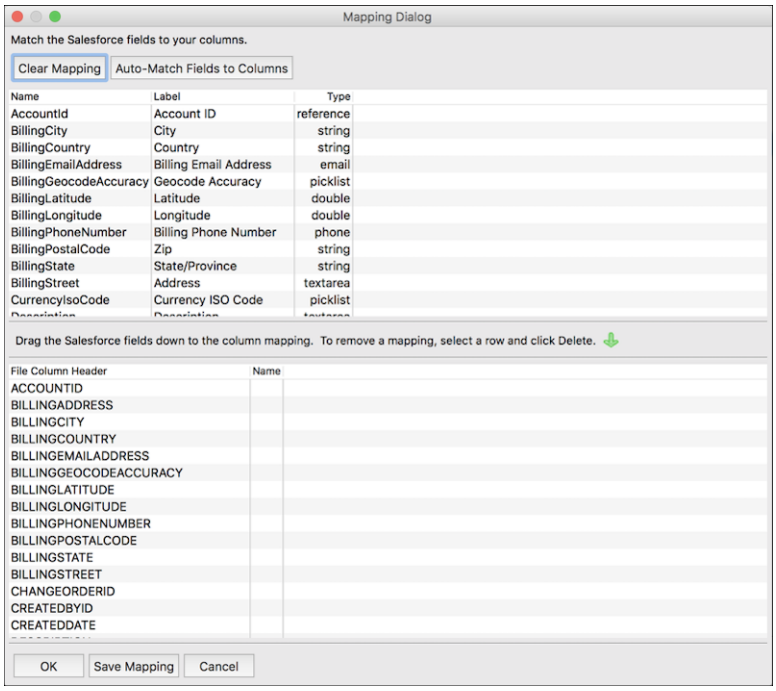
Object	Example Query
OrderItemTaxLineItemSummary	Select Id, Amount, CreatedDate, Rate, Type FROM OrderItemTaxLineItemSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderAdjustmentGroupSummary	Select Id, CreatedDate, TotalAmount, TotalTaxAmount, Type FROM OrderAdjustmentGroupSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderDeliveryGroupSummary	Select Id, CreatedDate, DeliverToCity, DeliverToCountry, EmailAddress, PhoneNumber, TotalAmount FROM OrderDeliveryGroupSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderItemSummaryChange	Select Id, ChangeOrderItemId, ChangeType, OrderItemSummaryId, Reason FROM OrderItemSummaryChange **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderPaymentSummary	Select Id, CreatedDate, FullName, AuthorizationAmount OwnerId, PaymentMethodId, RefundedAmount, Type FROM OrderPaymentSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**

Update Unmanaged Order Summaries with Data Loader

Use Data Loader to make bulk updates to your B2B Commerce order summary data.

Start by opening the Data Loader client application.

1. Select **Update**.
2. Search for an order summary or a related summary object, and enter the name of the CSV file where orders are updated.
3. Map fields in your CSV file to the corresponding object columns in Salesforce.



4. Select the directory where your success or error files are saved.
5. Click **Finish**, and view the extracted results.

Create Unmanaged Order Summaries

To directly create an Order Summary record, you must create an activated order record and generate an order summary from it.

Creating an order summary is a three-step process:

1. Import a draft order into Salesforce.
2. Activate the order.
3. Create an order summary in one of four ways.
   a. Use the Order Summaries Rest API to create an OrderSummary record based on an order.
   b. Use the OrderSummaryCreation Apex class.
   c. Use Process Builder to call an invocable action when an order is activated.
   d. Use a flow that runs when an order is activated and a SalesStoreId is provided.

B2B Legacy Checkout Reference

Understand the legacy B2B checkout flow and subflow architecture, elements, and states to create custom buyer experiences.

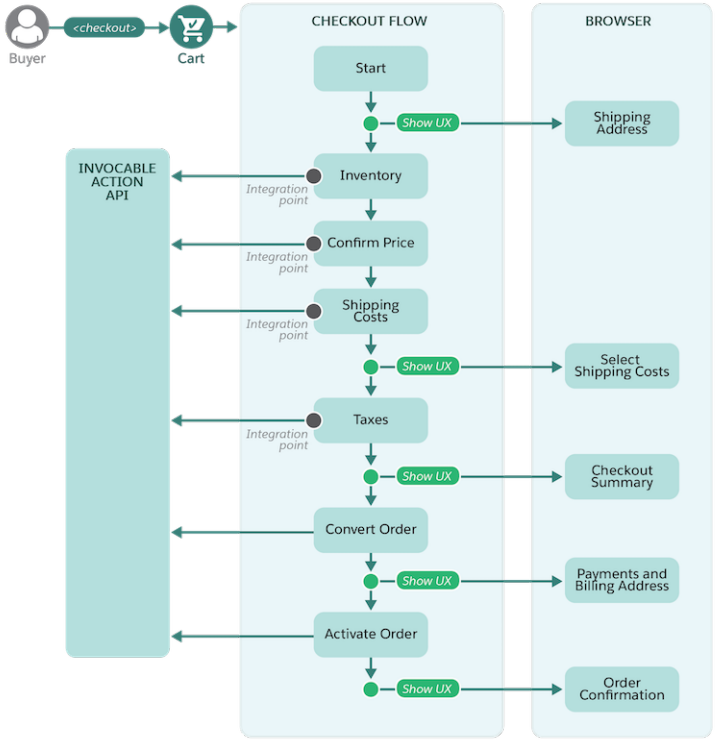
Checkout Flow Architecture

The Checkout flow is available only from the Checkout component in Experience Builder and uses only specific subflows.

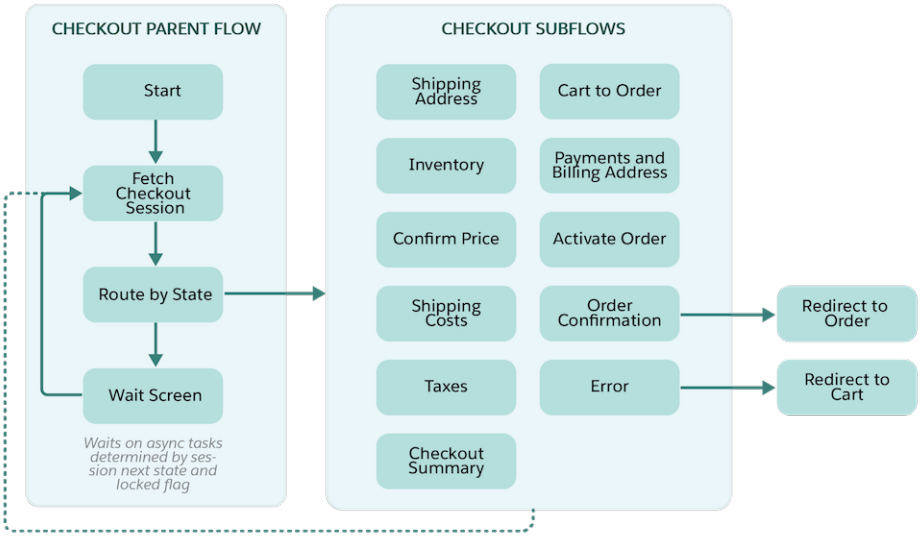
B2B Checkout Flow Design

The B2B Checkout flow is triggered by, and interacts with, browser-based shopper UI actions.

The flow, which provides integration points to third-party services, is triggered when a customer clicks **Checkout** or revisits a previously started checkout.



The flow is split into two tiers: the parent flow and subflows. The parent flow controls the flow through the checkout, managing the state, collecting errors, and waiting on async tasks to complete. The subflows make it easy to add, move, or remove steps in the checkout process.



B2B Checkout Flow Elements

The B2B checkout flows include flow elements that structure the buyer experience.

Several core elements create the basic structure of your checkout flow.

- **Start**—Your checkout flow execution begins here.
- **Fetch Checkout Session**—Returns the `CartCheckoutSessionId`. The session identifies the current state of the checkout and whether an async task is running in the background. Based on that information, the element determines whether to display the wait screen. The checkout process routes to the next step based on current state and async task status. This process provides a re-entrant checkout and executes one time per loop in the default implementation.
- **Route by State**—Routes to the next element based on the value of `CartCheckoutSession.State` and transitions to `CartCheckoutSession.IsProcessing`. It then routes to the wait screen until the background operation completes and sets `CartCheckoutSession.IsProcessing` to false.
- **Wait Screen**—A screen that polls `CartCheckoutSession.IsProcessing` for changes and routes back to Fetch Checkout Session when `CartCheckoutSession.IsProcessing` is false.

B2B Checkout Subflows

Each B2B checkout flow contains subflows Working with subflows makes it easier to move, delete, or change steps. Some of the subflows are screen flows, which require input from, or display information to, the buyer. Other subflows are system flows, which complete actions that make your checkout work.

Screen Subflows

- **Shipping Address**—Requires buyer input to determine the shipping address. Addresses are pulled from the buyer account.
- **Checkout Summary**—Redirects the user to the Checkout Summary component, which outlines the total cart cost, prices, taxes, and shipping information.
- **Payments and Billing Address**—Requires buyer input to determine the billing address and payment method. Addresses are pulled from the buyer account. The component in the browser integrates with the Salesforce Payments API to provide payment authorization services.
- **Order Confirmation**—Redirects the user to the Order Confirmation component, which outlines the final order details, prices, taxes, and shipping charges. Checkout is complete at this point.
- **Error**—Presents errors to the buyer and then redirects to the cart to resolve those errors. If the buyer reaches this point, the checkout is complete with errors. To complete the checkout as intended, the user must cancel the checkout and start again from the beginning.

System Subflows

- **Inventory**—Triggers a check inventory request for each cart line item. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen.
- **Confirm Price**—Triggers a price check request for each cart line item. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen.
- **Shipping Costs**—Triggers a request to calculate shipping costs for each cart delivery group. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen.

- **Taxes**—Triggers a request to calculate taxes for each cart line item. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen.
- **Cart to Order**—Converts a cart to an order in Draft state, pending activation. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen. `CartCheckoutSession.OrderId` is populated with the ID of the created order. You can extend your checkout flow to update order fields after the Cart to Order step.
- **Activate Order**—Activates an order previously created in Draft state. Executing this step updates `CartCheckoutSession.BackgroundOperation` and `CartCheckoutSession.IsProcessing` and redirects the flow to the Wait Screen.

Subflows allowed in the Checkout Flow include:

- FlowProcessType.Flow
- FlowProcessType.AutoLaunchedFlow
- FlowProcessType.CheckoutFlow

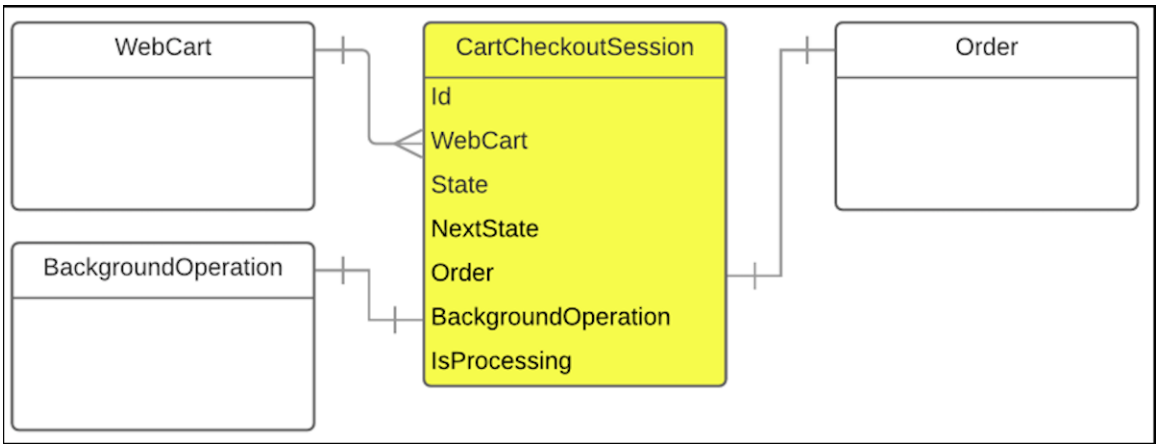
Note: The Cart to Order and Activate Order subflows run synchronously by default. To run these actions asynchronously, change the `Run Asynchronously` parameter from within the flow.

B2B Checkout States

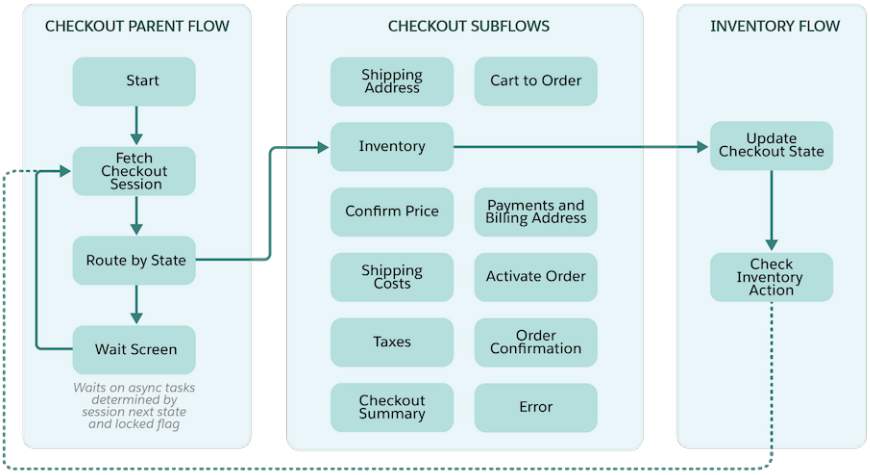
The CartCheckoutSession object manages checkout progress.

The State and NextState fields on CartCheckoutSession drive the progression through a checkout. State identifies and completes an async task, and NextState identifies the value that State automatically transitions to. The values are stored as strings, and the Checkout flow manages state transitions.

The BackgroundOperation object identifies the async checkout task running in the background and provides a status of the tasks. `CartCheckoutSession.IsProcessing` also tracks whether the background operation is still processing, and `CartCheckoutSession.IsError` indicates whether an error occurred during checkout.



Example: In this example, `State=Inventory` transitions to `NextState=Confirm Price`, which confirms the price after inventory processing completes. `InProcessing` indicates that a checkout task is operating in the background, and the checkout Wait Screen displays until the task is finished or canceled.



When an async task completes:

- `IsProcessing` is set to `false`.
- If an error occurs, `If error State` is set to `Error`, and NextState is set to `null`.
- Else, NextState is copied to State.

SEE ALSO:

CartCheckoutSession

B2B Checkout Task Modes

The B2B checkout flow executes tasks in two distinct modes, each suited to different action types.

A screen subflow contains a screen component, which is displayed to the buyer when the element executes. A screen subflow uses buyer input and progresses to the next step when the buyer clicks **Next**.

An asynchronous checkout API action triggers a checkout task. The APIs return the job ID, which the checkout flow wait screen uses to determine if the async task is complete.

When a synchronous operation executes in the checkout flow, the buyer must stay on the checkout flow screen for the task to complete. If the buyer leaves the screen, the checkout information is lost.

To add a sync checkout step, create a custom invocable action using the InvocableMethod Apex annotation. Include the action in your checkout flow. You can also execute Apex directly with the Flow Apex element.

B2B Checkout Error Handling

If an error occurs during the checkout process, the Checkout flow routes the checkout to the error subflow. The error subflow then presents the user an error screen with an explanation.

When an error occurs, State is set to `error`, and NextState is set to `null`.

An error can occur during checkout for these reasons.

- A checkout flow element contains errors.

- The checkout flow API returns an HTTP error status.
- The checkout async integration indicates that a background operation failed.
- The checkout async integration contains write errors during execution to `CartValidationOutput`.

Buyer Experience SDK

The Buyer Experience Software Development Kit (SDK) includes all the Apex interfaces that support B2B Commerce cart and checkout. The reference content presented here exposes the interfaces that support the legacy B2B checkout template.

Apex APIs

These Apex interfaces form the Buyer Experience SDK. Repeated execution of all these integrations must be idempotent.

Inventory Validation

Inventory checks guarantee that cart item quantities are available by delegating checks to the cart inventory interface. Quantity availability errors are written to `CartValidationOutput`.

```
global interface checkout_CartInventoryValidation
   extends checkout_AsyncCartFunction {
}
```

Price Calculations

The price calculation integration guarantees that all cart items have a transactional price assigned to each line item in the cart.

```
global interface checkout_CartPriceCalculations
   extends checkout_AsyncCartFunction {
}
```

Shipping Charges

The shipping integration guarantees that all cart delivery groups have the correct identifying carrier, class of service, and cost.

```
global interface checkout_CartShippingCharges
   extends checkout_AsyncCartFunction {
}
```

Tax Calculations

The tax calculation integration guarantees that each cart item has associated tax line items added with tax amounts.

```
global interface checkout_CartTaxCalculations
```

```
    extends checkout_AsyncCartFunction {
}
```

SEE ALSO:

Salesforce Commerce Apex Reference

Shared SDK Interface Properties

Inventory, pricing, shipping, and tax integrations share interface properties.

Each SDK interface shares the following properties:

- Published under reserved namespace `sfdc_checkout`.
- Extends `AsyncCartProcessor`.
- Executes asynchronously.
- Accept `cartId` as an input parameter.

Integration Use Case	Interface Name
Inventory Validation	`CartInventoryValidation.apex`
Pricing	`CartPriceCalculations.apex`
Shipping Charges	`CartShippingCharges.apex`
Calculate Taxes	`CartTaxCalculations.apex`

Asynchronous Checkout Interface

The cart processing and checkout integrations share a base method structure. These interfaces extend the base interface.

Base Interface

```
global interface AsyncCartProcessor {

    // Integration used for async processing.
    IntegrationStatus startCartProcessAsync(
         IntegrationInfo integrationInfo,
         Id cartId);
}
```

Input Type: IntegrationInfo

The `IntegrationInfo` input param provides values that checkout APIs use to map requests to responses and the necessary metadata and context. Integration code also needs instance details because code can be reused across stores.

```
global class IntegrationInfo {

    // Identifies specific to the Salesforce Background Operation framework.
    global String jobId;
```

```
    // ID of this integration.
    Webservice String integrationId;

    // Site language to be used by third party services.
    Webservice String siteLanguage;
}
```

Return Type: IntegrationStatus

`IntegrationStatus` supports only synchronous execution of Apex integrations. The implementation must return the status of the execution.

- `Success` indicates that the integration executed successfully.
- `Failed` indicates a transient, unknown error managed by the implementor. The buyer can retry this action.

> Note: If `IntegrationStatus` returns `Failed`, the checkout session state is set to `Error`. In turn, the Checkout flow fails, presents an error to the buyer, and redirects the buyer back to the cart to resolve errors or retry.

```
global class IntegrationStatus {
    //Indicates that integration processing is complete.

    global enum Status {SUCCESS, FAILED}
    global Status status;
}
```

Return Type: Exception

If a runtime exception occurs during an Apex integration execution, the integration execution is rolled back and the system fires a platform error event. This rollback preserves data integrity and marks the integration task as failed. The checkout session state is updated to `Error`, which causes the Checkout flow process to fail. When the checkout fails, an error displays to the buyer, and they're redirected back to the cart to resolve the errors or retry. To receive more information about an error, subscribe to the platform error event.

Error Handling

Cart and checkout integrations can provide errors and warnings that give the buyer who triggered the integration the ability to resolve the problem and retry. Examples of resolvable errors include lack of inventory, pricing changes, and transient errors, such as unreachable external services.

Integration developers can catch runtime errors and communicate those errors to users via the CartValidationOutput standard object. This example code is executed in a synchronous style, indicating to the caller that the integration completed with errors. You can also execute these errors asynchronously by adding a callback mechanism.

```
public class CartInventoryValidationImpl implements checkout_CartInventoryValidation {

  IntegrationStatus startCartProcessAsync(IntegrationInfo jobInfo, ID cartId) {

     //Look up cart items and process inventory availability
     ID cartItemId = firstCartItemWithoutInventory(cartId);

     // Insert cart item error
```

```
    insert new CartValidationOutput(
       jobInfo.getJobId(),
       cartItemId,
       "Inventory",
       "Error",
       "Item out of stock");

    // Indicate to the service that the work
    // completed with errors that still need to be resolved
    return IntegrationStatus.Failed;
  }
```

Note: This example code is executed in a synchronous style, indicating to the caller that the integration completed with errors. You can also execute these errors in an async style with the addition of a callback mechanism.

Errors that occur during checkout execution are presented to the buyer within the cart. They can present as a cart header, a cart item error, or a cart delivery group error.

Idempotent Execution

API calls that trigger Integration services often run multiple times for the same cart. The integration APIs are designed to clean up and resolve the results of prior calls (idempotentcy) before updating the cart. When a failure occurs, the cart executes a retry. Usually the integration service or a buyer action causes errors.

An integration can be executed using partial results from a previous execution. Partial results are a mix of null and non-null values. Before writing updated values, the code cleans up the integration results from previous executions.

This example integration clears the price of all items with a bulk write and then attempts to recalculate the prices.

```
public class CartPriceCalcImpl implements checkout_CartPriceCalc {

  IntegrationStatus startCartProcessAsync(IntegrationInfo jobInfo, ID cartId) {

    // clean previous executions
    cleanItemPrices(jobInfo, cartId);

    // bulk update prices
    bulkUpdateItems(jobInfo, cartId);

    // indicate to the service, that the work is complete
    IntegrationStatus.complete();
  }

  public void cleanItemPrices(IntegrationInfo jobInfo, ID cartId) {

    // execute integration work on the calling thread
    List<CartItem> cartItems =
         [ SELECT CartItemId, SKU, Quantity, TotalLineAmount
           FROM CartItem
           WHERE WebCartId = cartId ];

    // for each set price
    for(Cartitem cartItem : cartItems) {
       cartItem.TotalLineAmount = null;
```

```
    }

    // bulk update of cart items
    update cartItems;
  }

  ...
}
```

Set Up Custom B2B Checkout Integrations

After you create your Apex classes, set up your checkout integrations.

Register Your Apex Class

Insert your Apex class into `RegisteredExternalService`. Use the `ApexClassId` as `ExternalServiceProviderId`, and use the `ExternalServiceProviderType` included in the code sample.

```
// WebStore query values
String webstoreName = 'store1';

// ApexClass query values
String apexClassname = 'B2BPricingSample';
String status = 'Active';
Double ApiVersion = 54.0;

// RegisteredExternalService insert values
String registeredProviderType = 'Price';
String registeredDevName = 'COMPUTE_PRICE';
String registeredLabel = registeredDevName;

// StoreIntegratedService insert values
String devname = registeredDevName;
String prefix = registeredProviderType;
String prefixedName = prefix + '__' + devname;

// locate webstore
WebStore webStore = Database.query('SELECT Id FROM WebStore WHERE Name = :webstoreName
LIMIT 1');
String webStoreId = webStore.Id;
System.debug('webStoreId:' + webStoreId);

// locate apex class Id
ApexClass apexClass = Database.query('SELECT Id FROM ApexClass WHERE Status=:status AND
ApiVersion=:apiVersion AND Name=:apexClassname LIMIT 1');
String apexClassId = apexClass.Id;
System.debug('apexClassId:' + apexClassId);


// locate apex in RegisteredExternalService
String registeredIntegrationId = null;
try {
 RegisteredExternalService registeredExternalService = Database.query('SELECT Id FROM
RegisteredExternalService WHERE ExternalServiceProviderId=:apexClassId AND
```

```
DeveloperName=:registeredDevName AND ExternalServiceProviderType=:registeredProviderType
LIMIT 1');
 registeredIntegrationId = registeredExternalService.Id;
 System.debug('apex class registration: FOUND ' + registeredIntegrationId);
 //delete registeredExternalService; // optionally remove if needed

} catch (QueryException q) {
  System.debug('apex class registration: MISSING ' + apexClassId);
  insert new RegisteredExternalService(DeveloperName = registeredDevName, MasterLabel =
registeredLabel, ExternalServiceProviderId = apexClassId, ExternalServiceProviderType =
registeredProviderType);
  RegisteredExternalService registeredExternalService = Database.query('SELECT Id FROM
RegisteredExternalService WHERE ExternalServiceProviderId = :apexClassId  LIMIT 1');
  registeredIntegrationId = registeredExternalService.Id;
  System.debug('apex class registration: INSERTED ' + registeredIntegrationId);
}
```

Example: Configure a provider for each type, and include these entries.

Id	ExternalServiceProviderId	ExternalServiceProviderType	DeveloperName
1uuxx00000001s9AAA	01pxx0000004cGRAAY	Inventory	CHECK_INVENTORY
1uuxx00000001sAAAQ	01pxx0000004cGQAAY	Shipment	COMPUTE_SHIPPING
1uuxx00000001sBAAQ	01pxx0000004cGPAAY	Tax	COMPUTE_TAXES
1uuxx00000001sCAAQ	01pxx0000004cGOAAY	Price	COMPUTE_PRICE

Map to Your Store

Insert the DeveloperName into `StoreIntegratedService`.

```
// WebStore query values
String webstoreName = 'store1';

// ApexClass query values
String apexClassname = 'B2BPricingSample';
String status = 'Active';
Double ApiVersion = 48.0;

// RegisteredExternalService insert values
String registeredProviderType = 'Price';
String registeredDevName = 'COMPUTE_PRICE';
String registeredLabel = registeredDevName;

// StoreIntegratedService insert values
String devname = registeredDevName;
String prefix = registeredProviderType;
String prefixedName = prefix + '__' + devname;

// locate webstore
```

```
WebStore webStore = Database.query('SELECT Id FROM WebStore WHERE Name = :webstoreName
LIMIT 1');
String webStoreId = webStore.Id;
System.debug('webStoreId:' + webStoreId);

// locate apex class Id
ApexClass apexClass = Database.query('SELECT Id FROM ApexClass WHERE Status=:status AND
ApiVersion=:apiVersion AND Name=:apexClassname LIMIT 1');
String apexClassId = apexClass.Id;
System.debug('apexClassId:' + apexClassId);

// locate apex in RegisteredExternalService
RegisteredExternalService registeredExternalService = Database.query('SELECT Id FROM
RegisteredExternalService WHERE ExternalServiceProviderId=:apexClassId AND
DeveloperName=:registeredDevName AND ExternalServiceProviderType=:registeredProviderType
LIMIT 1');
String registeredIntegrationId = registeredExternalService.Id;
System.debug('apex registration:' + registeredIntegrationId);

// locate and map in StoreIntegratedService
try {
  StoreIntegratedService registeredMappingObj = Database.query('SELECT Id FROM
StoreIntegratedService WHERE Integration=:prefixedName AND ServiceProviderType=:prefix AND
 StoreId=:webStoreId LIMIT 1');
  System.debug('registered class mapping: FOUND ' + registeredMappingObj);
  // delete registeredMappingObj; // optionally remove if needed

} catch (QueryException q) {
  System.debug('registered class mapping: MISSING ' + prefixedName);
  insert new StoreIntegratedService(Integration = prefixedName, ServiceProviderType =
prefix, StoreId = webStoreId);
  StoreIntegratedService registeredMappingObj = Database.query('SELECT Id FROM
StoreIntegratedService WHERE Integration=:prefixedName AND ServiceProviderType=:prefix AND
 StoreId=:webStoreId LIMIT 1');
  System.debug('registered class mapping: INSERTED ' + registeredMappingObj);
}
```

Example: Use the ID from `RegisteredExternalService` for the Integration field and the `ServiceProviderType` matching the `ExternalServiceProviderType` from `RegisteredExternalService`.

Id	Integration	ServiceProviderType	StoreId
1ffxx000000021pAAA	1uuxx00000001s9AAA	Inventory	0ZExx00000002rRGAQ
1ffxx000000021qAAA	1uuxx00000001sAAAQ	Shipment	0ZExx00000002rRGAQ
1ffxx000000021rAAA	1uuxx00000001sBAAQ	Tax	0ZExx00000002rRGAQ
1ffxx000000021sAAA	1uuxx00000001sCAAQ	Price	0ZExx00000002rRGAQ
1ffxx000000021tAAA	2abxx00000001sDAAQ	Payment	0ZExx00000002rRGAQ

Set Up the Payment Gateway

Create a payment gateway, and locate the `PaymentGatewayId`.

```
String paymentGatewayId = [SELECT Id FROM PaymentGateway WHERE PaymentGatewayName = 'Name'
 LIMIT 1].Id;
System.debug('paymentGatewayId:' + paymentGatewayId);
```

Create Named Credentials

A custom Apex call that makes outbound HTTP connections must reference a named credential to avoid creating hard-coded credentials within the code. Configure corresponding named credentials, referenced by the Apex class, after you finish your integration installation.

```
//Prepare HTTP POST
HttpRequest req = new HttpRequest();
req.setEndpoint('callout:My_Named_Credential/some_path');
```