B2B Commerce and D2C Commerce Developer Guide

Version 60.0, Spring '24

@salesforcedocs

Last updated: January 12, 2024

CONTENTS

CHAPTER 1 B2B Commerce and D2C Commerce Developer Guide

In this chapter ...

Design a B2B Commerce or D2C Commerce solution that uses the power of Lightning Experience. Starting with the Commerce Experience Builder template, create a customized platform where retailers, wholesalers, distributors, and consumers can make purchases. Design your platform to meet your business requirements and connect with all your third-party apps.

B2B and D2C Commerce Data Model

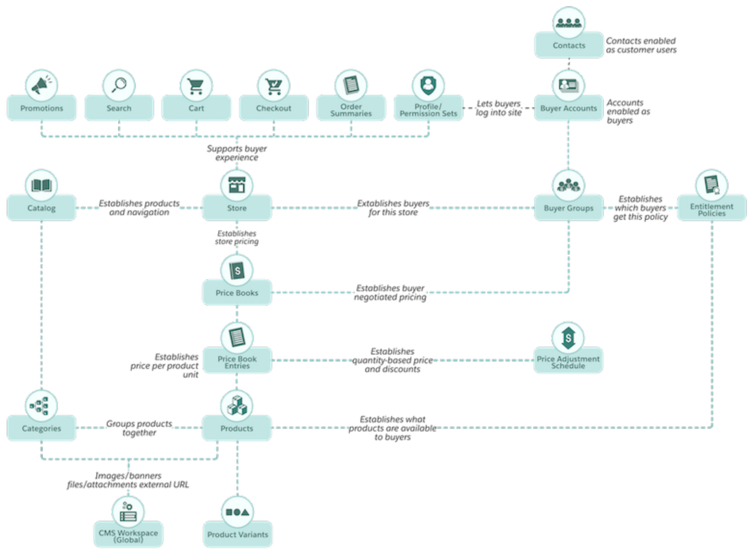
EDITIONS

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

Available in: B2B Commerce and D2C Commerce

The Commerce store template is built on a pre-configured data model. The data model supports standard and customizable business objects for a multitude of business relationships and interoperability with B2B and B2C stores, Salesforce Order Management, and Service Cloud.

The Commerce data model connects the store objects. Default object relationships support a full-featured B2B or B2C store experience.


Contacts
Contacts enabled as customer users
Promotions
Search
Cart
Checkout
Order Summaries
Profile/ Permission Sets
Lets buyers log into site
Buyer Accounts
Accounts enabled as buyers
Supports buyer experience
Catalog
Establishes products and navigation
Store
Extablishes buyers for this store
Buyer Groups
Establishes which buyers get this policy
Entitlement Policies
Establishes store pricing
Price Books
Establishes buyer negotiated pricing
Establishes price per product unit
Price Book Entries
Establishes quantity-based price and discounts
Price Adjustment Schedule
Categories
Groups products together
Products
Establishes what products are available to buyers
Images/banners files/attachments external URL
CMS Workspace (Global)
Product Variants

Among the object relationships in the Commerce app data model are those that you use to quickly:

- Add product catalogs to provision your stores.
- Configure product variants (product size, shape, color, and so on), categories, bundles, and more.
- Select customer search capabilities and how your store displays results.
- Differentiate buyer groups and associate them with specific products and volume-discounted prices.
- Configure tax, shipping, and payment for cart checkout.
- Create promotional campaigns for targeted products.

Here's a list of some of the default data model objects.

Data Model Object	Description	API Name
Store	A website where buyers and shoppers complete wholesale and retail transactions. Includes the fields and properties that define your store. For example, supported currencies, languages, and price books. Many fields are customizable.	WebStore
Cart	Represents an online shopping cart in a store built with B2B or D2C Commerce on Lightning, with total amounts for products, shipping and handling, and taxes.	WebCart

Data Model Object	Description	API Name
Catalog	A catalog is a collection of the products that you sell, organized into different categories. The Commerce Admin or Merchandiser uses data import to set up the catalog.	ProductCatalog
Category	Categories and subcategories organize and group products in your catalog and on your storefront. The Commerce Admin or Merchandiser uses B2C data import to fill in a default compact layout, which includes name, catalog, category, search order, and so on. Layout is customizable.	ProductCategory
Entitlement Policy	Entitlement policies are simple entities that bring together buyer groups and products. Filtered by BuyerGroup membership. Includes CanViewPrice and CanViewProduct fields, which are customizable.	CommerceEntitlementPolicy
Product	The items and services you sell. The Commerce Admin or Merchandiser uses data import to fill in a default compact layout, which includes a variety of customizable fields (name, family, and so on).	Product2
Buyer Account	The buyer's or shopper's financial information, including credit and order limits, some of which pertain only to B2B. A B2C BuyerAccount is established when a shopper self-registers.	BuyerAccount
Buyer Group	A group of buyers with the same assigned entitlement policies, price books, and products. Buyer Group name and description are customizable. For D2C stores, one buyer group per store is created by default during D2C data import.	BuyerGroup
Buyer Group Member	An individual buyer associated with a buyer group.	BuyerGroupMember
Price Book	A price book contains price definitions for a group of products. Typically added during setup with your store's data import, but you can add custom fields.	PriceBook2
Price Book Entry	A product entry (an association between a Pricebook2 and Product2) in a price book.	PricebookEntry

[Cart Data Model]
The Cart data model connects objects used to support shopping cart and checkout functionality in a B2B or B2C store. This data model includes objects used to process shipping, taxes, and promotions.

[Product and Catalog Data Model]
The Product and Catalog data model connects objects used to support product, catalog, and category organization in a B2B or B2C store.

[Product and Category Media Data Model]
The Product and Category Media data model connects objects and relationships that support adding images and other attachments to provide more information to customers.

[Product Attributes Data Model]
The Product Attributes data model connects objects and relationships that support product attribute definitions. These attributes determine how products appear to the customer in places like the product details page and search.

Cart Data Model

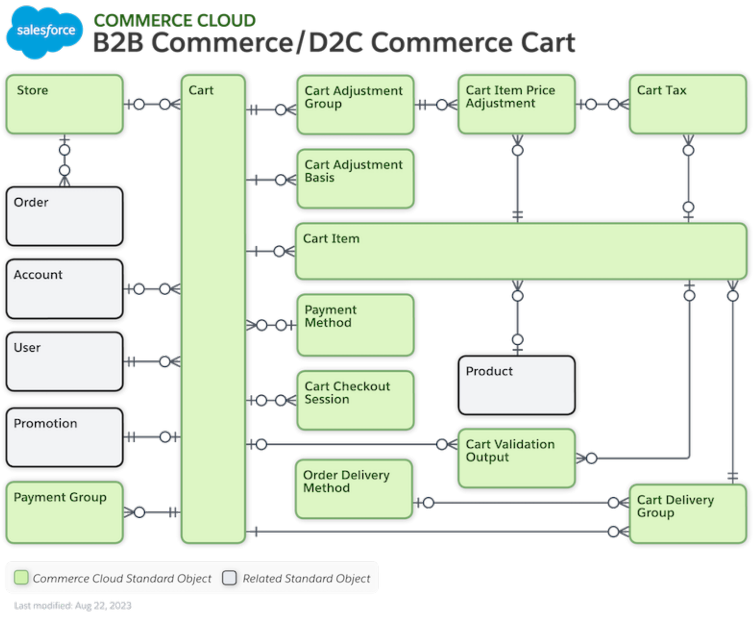
The Cart data model connects objects used to support shopping cart and checkout functionality in a B2B or B2C store. This data model includes objects used to process shipping, taxes, and promotions.

Commerce Cloud standard objects in the cart data model require at least one of the following licenses: B2B Commerce, D2C Commerce, Salesforce Order Management, or Salesforce Payments.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce



- Cart (WebCart)
- Cart Adjustment Basis (WebCartAdjustmentBasis)
- Cart Adjustment Group (WebCartAdjustmentGroup)
- Cart Checkout Session (CartCheckoutSession)
- Cart Delivery Group (CartDeliveryGroup)
- Cart Item (CartItem)
- Cart Item Price Adjustment (CartItemPriceAdjustment)

- Cart Tax (CartTax)
- Cart Validation Output (CartValidationOutput)
- Payment Group (PaymentGroup)
- Payment Method (PaymentMethod)
- Store (WebStore)

Product and Catalog Data Model

EDITIONS

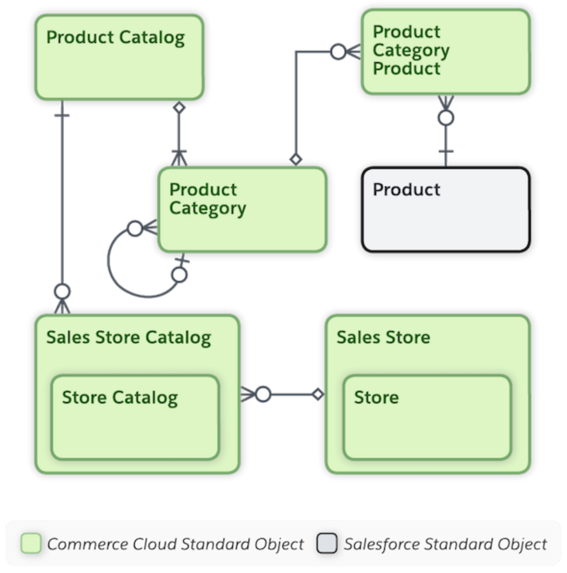
Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce and D2C Commerce

The Product and Catalog data model connects objects used to support product, catalog, and category organization in a B2B or B2C store.

A product is assigned to a product category, and each product category is assigned to a product catalog. A store can be associated with only one catalog, but a catalog can be associated with multiple stores.

Commerce Cloud standard objects in the Product and Catalogs data model require at least one of the following licenses: B2B Commerce, D2C Commerce.



- Product (Product2)
- Product Catalog (ProductCatalog)
- Product Category (ProductCategory)
- Product Category Product (ProductCategoryProduct)
- Sales Store (SalesStore)
- Sales Store Catalog (SalesStoreCatalog)
- Store (WebStore)
- Store Catalog (WebStoreCatalog)

Product and Category Media Data Model

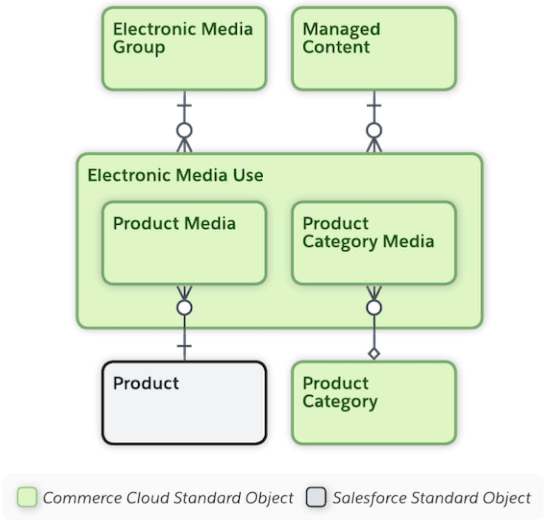
The Product and Category Media data model connects objects and relationships that support adding images and other attachments to provide more information to customers.

Commerce Cloud standard objects in the Product and Category Media data model require at least one of the following licenses: B2B Commerce, D2C Commerce.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce



- Product (Product2)
- Product Category (ProductCategory)
- Product Media (ProductMedia)
- Product Category Media (ProductCategoryMedia)
- Electronic Media Use (ElectronicMediaUse)
- Electronic Media Group (ElectronicMediaGroup)
- Managed Content Info (ManagedContentInfo)

Product Attributes Data Model

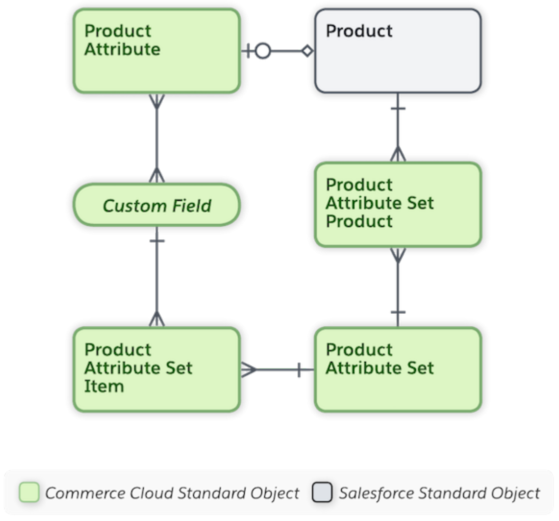
The Product Attributes data model connects objects and relationships that support product attribute definitions. These attributes determine how products appear to the customer in places like the product details page and search.

Commerce Cloud standard objects in the Product Attributes data model require at least one of the following licenses: B2B Commerce, D2C Commerce.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

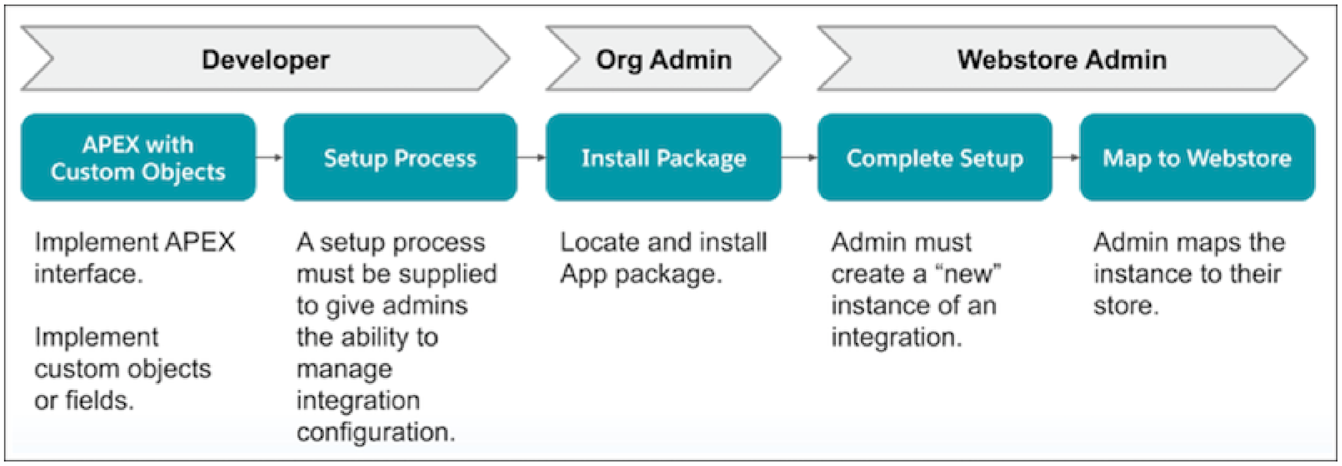

Product Attribute
Product
Custom Field
Product Attribute Set Product
Product Attribute Set Item
Product Attribute Set
Commerce Cloud Standard Object
Salesforce Standard Object

- Product (Product2)
- ProductAttribute
- ProductAttributeSet
- ProductAttributeSetItem
- ProductAttributeSetProduct

Implementation Lifecycle: Personas

In Salesforce Commerce, implementation is a division of labor based on persona skill sets. The Commerce implementation lifecycle relies on three personas: a developer, an org admin, and a store admin.

Ideally, each persona within the implementation lifecycle has a specialty and completes only certain tasks.


Developer
Org Admin
Webstore Admin
APEX with Custom Objects
Setup Process
Install Package
Complete Setup
Map to Webstore
Implement APEX interface.
Implement custom objects or fields.
A setup process must be supplied to give admins the ability to manage integration configuration.
Locate and install App package.
Admin must create a “new” instance of an integration.
Admin maps the instance to their store.

Developer Tasks

Use standard Apex development tools to implement B2B Buyer Experience APIs. Create custom Lightning Web Components to embed payment, product recommendation, and other APIs. Implement or extend the global interfaces included in the Buyer Experience SDK. Create packages and deploy to your org.

Org and Store Admin Tasks

After your developer creates Apex implementations and packages them, deploy them into your Salesforce org. Configure the standard Salesforce integrations so that they can execute properly.

Integrations

B2B and D2C Commerce integration points are embedded into the cart and checkout experience. The integration services work across B2B and B2C stores.

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 B2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

Integration Architecture for B2B and D2C Stores (LWR)
A predefined set of flows simplifies package integration for tax, shipping, and payment providers for B2B and D2C stores created with an LWR template. The integrations are embedded into the cart and checkout experience, triggered by shopper interactions with the storefront UI.

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.

Checkout Integration
Commerce checkout provides integration points to third-party services.

Shipping and Tax Integration
A single API call fetches both shipping and tax costs for cart items in B2B and D2C stores.

Payment Integration
The Commerce app payment architecture combines checkout APIs, the Salesforce Payment Gateway, and an integrated payment package for B2B and B2C stores.

Integration Architecture for B2B and D2C Stores (LWR)

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

A predefined set of flows simplifies package integration for tax, shipping, and payment providers for B2B and D2C stores created with an LWR template. The integrations are embedded into the cart and checkout experience, triggered by shopper interactions with the storefront UI.

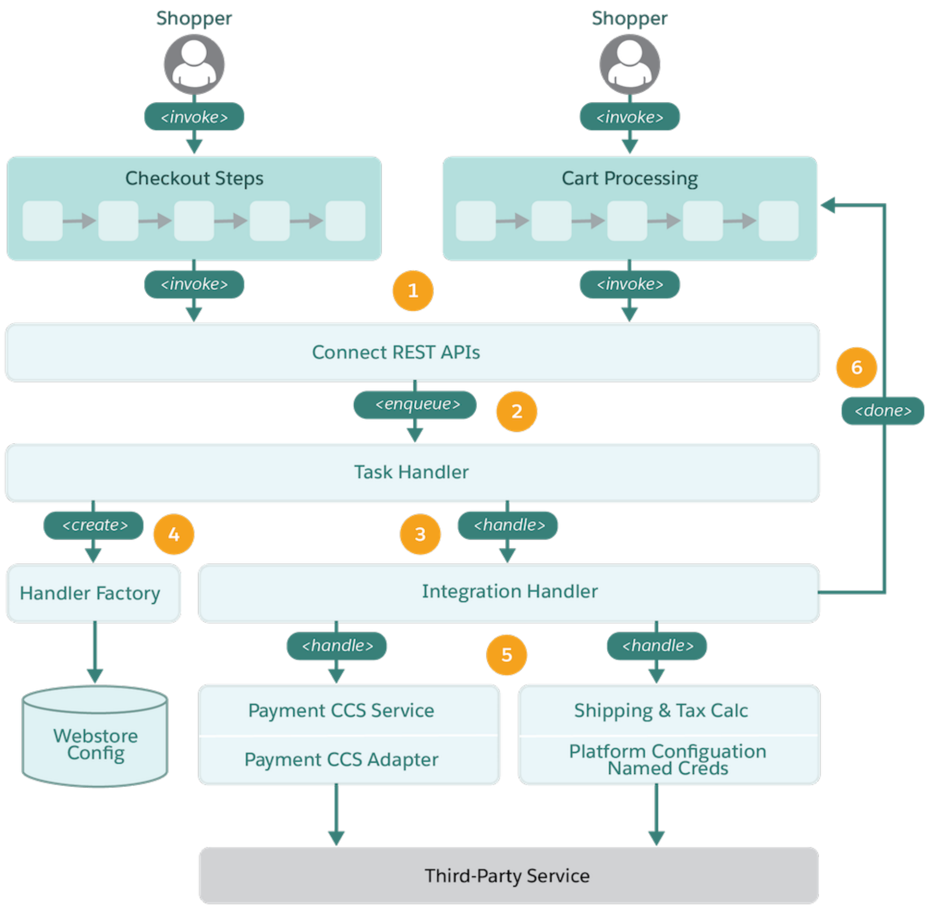
Note: In the Winter '24 release, we introduced Commerce extensions for pricing, inventory, shipping, taxes, and other services. While the checkout integrations framework is still supported, we recommend extensions over integrations because they offer more targeted customizations for your B2B or B2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

B2B and B2C stores created with an LWR template support these third-party integration points and flows.

- Shipping—Calculates and writes shipping costs per delivery group.
- Taxes—Calculates and adds tax prices for cart items.

- Payment—Uses the Salesforce Payment Adapter framework to fetch tokens and authorizations (and manage exceptions, such as fraud and insufficient credit) from service providers during checkout via a Payments Gateway. The integrated Salesforce Order Management module handles additional services, including capture and refund.

Here's how the various components work together to form the integration engine.



A predefined set of flows manages shipping, inventory, pricing, and tax integrations.

1. Cart and checkout processing—Entering a delivery address initiates shipping charges and tax calculation. Order placement triggers payment processing. An integrated Salesforce Order Management component processes refund, capture, and more.
2. Connect REST API—Service for these discrete APIs. Shipping, tax, and payment integrations execute tasks asynchronously and are distinct from the Salesforce B2B Checkout subflows implementation for a store built with the Aura template.
3. Task handler—The task handler is implemented as an MQ (Message Queue) handler and invoked by the queue manager when the integration task is picked up for processing. The integration handler is responsible for delegating the task to the integration implementation, which the Commerce Admin or Merchandiser specifies when setting up the store.
4. Handler factory—Responsible for creating an integration handler that maps to the implementation chosen by the Commerce Admin during store setup.
5. Integration services—Using the store's configured named credentials, a gateway conveys requests for third-party tax and shipping calculations. The CCS (Core Commerce Services) Adapter and Service Salesforce Payment Adapter request and receive authorizations, tokens, and exceptions from a third-party service via a Payments Gateway.
6. API responses—Successful results and exceptions with customer-facing help messages are returned to the shopper's browser.

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.

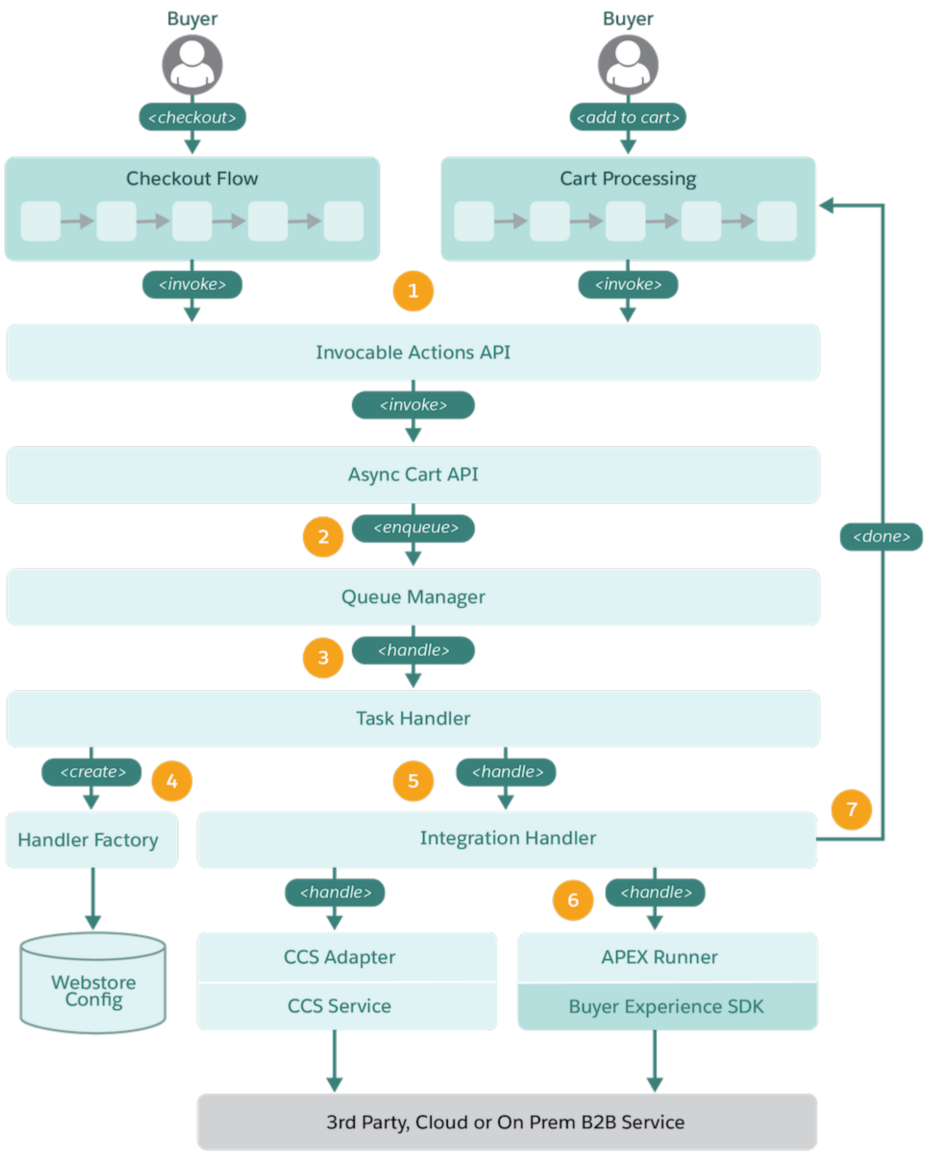
EDITIONS

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

Available in: B2B Commerce

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 B2C 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.



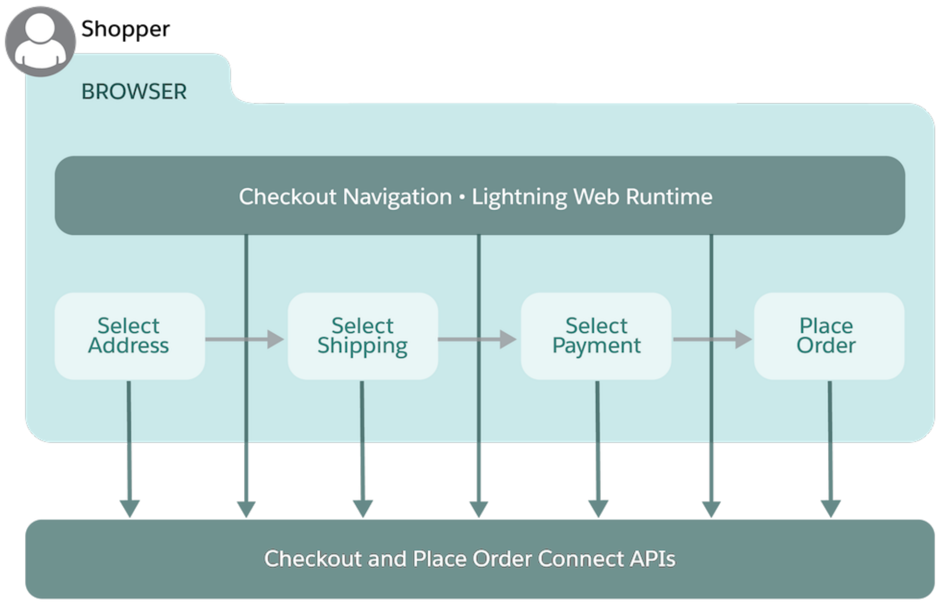
1. **Checkout flow**:— A managed checkout flow that processes cart shipping, inventory, pricing, and taxation integrations, and converts the cart to an order. Install and deploy the B2B checkout flow, then customize the flow in Experience Builder and optionally Flow Builder.
2. **Cart processing** —A 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.

Checkout Integration

Commerce checkout provides integration points to third-party services.

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 B2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

Calls to the API services are triggered when shoppers click Checkout or revisit a checkout session from their browser. Store components that embed the Checkout APIs are implemented in Lightning Web Runtime.


Shopper
BROWSER
Checkout Navigation • Lightning Web Runtime
Select Address
Select Shipping
Select Payment
Place Order
Checkout and Place Order Connect APIs

Shipping and Tax Integration

A single API call fetches both shipping and tax costs for cart items in B2B and D2C stores.

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 B2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

Shipping and Tax API

Because shipping and tax providers require the same inputs to make their respective calculations, a single API call fetches both shipping and tax costs for cart items.

Shipping Reference Package

A reference shipping integration package supports both B2B Commerce and D2C Commerce implementations. You can use it as a template to create your own shipping calculation package.

Tax Reference Package

A reference tax integration package supports both B2B Commerce and D2C Commerce and implementations. You can use it as a template to create your own tax calculation package.

Shipping and Tax API

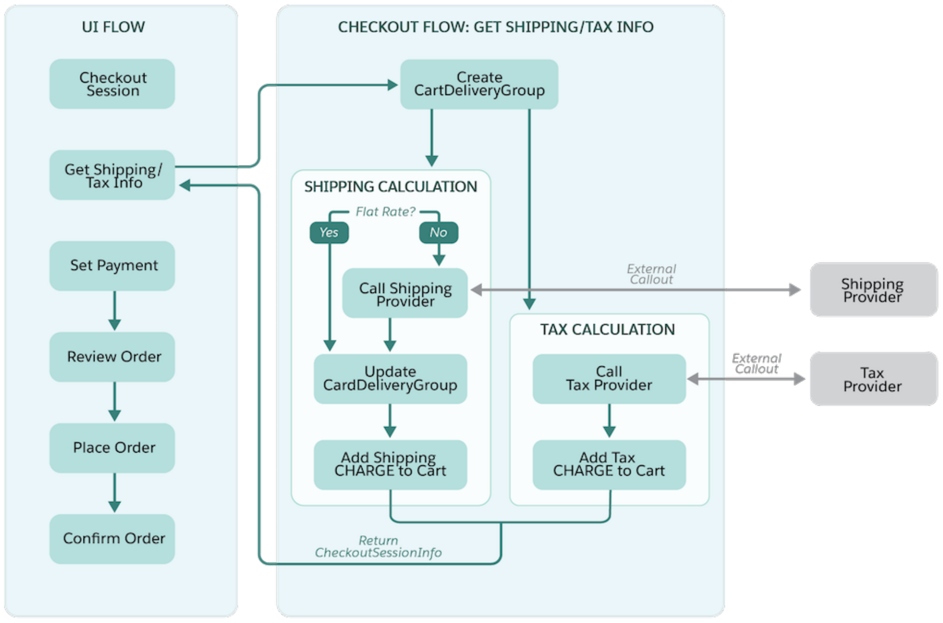
EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Because shipping and tax providers require the same inputs to make their respective calculations, a single API call fetches both shipping and tax costs for cart items.

An asynchronous shipping cost and tax API call to service providers is triggered when a shopper enters a shipping address.



Here's the structure of RetrieveDeliveryMethod, the triggering shipping and tax calculation API call.

```
{
   "id" : ID : // the cart delivery group id
   "deliveryMethods" :  DeliveryMethodCollectionRepresentation :
   "deliveryAddress" : AddressRepresentation : // selected delivery address
   "cartItems" : CartItemCollectionRepresentation :
}
```

Here's a sample DeliveryGroupRepresentation.

```
{
   "id":"2Dg4567890123456 78AAA",
   "cartItems":{
       Total: 1,
       carItems:
       [
         {
           "cartItemId":"0a9456789012345678AAA",
           "productId":"01txx0000006i44AAA",
           "name":"shower bar",
           "listPrice":"29.95",
           "salesPrice":"20.00",
           "totalTax":"1.85",
           "totalAmount":"1",
           "totalPrice":"31.80",
           "totalAdjustmentAmount":"31.80"
         }
       ]
   },
   "deliveryMethods":{
     “total: 2”,
     “items” :
     [
       {
         "id":"2Dm456789012345678AAA",
         "shippingFee":14.00,
         "currencyCode":"USD",
         "carrier":"UPS",
         "classOfService":"Next Day Shipping",
         “timeOfArrival” : “2020-11-05T13:15:30Z”
         "selected":true
       },
       {
         "id":"2Dm123789012345678EAA",
         "shippingFee":9.00,
         "currencyCode":"USD",
         "carrier":"UPS",
         "classOfService":"Three Day Shipping",
         “timeOfArrival” : “2020-11-07T16:15:30Z”
         "selected":false
       }
     ]},
   "shippingAddress":{
      "AddressType":"Shipping",
      "City":"Boston",
      "Country":"USA",
      "Id":"81Wxx0000000001EAA",
      "IsPrimary":true,
      "Name":"Home Address",
      "PostalCode":"01234",
      "State":"MA",
      "Street":"1 Milk Street"
   }
}
```

Shipping Reference Package

A reference shipping integration package supports both B2B Commerce and D2C Commerce implementations. You can use it as a template to create your own shipping calculation package.

Clone or download the Shipping Reference Integration Package package.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Tax Reference Package

A reference tax integration package supports both B2B Commerce and D2C Commerce and implementations. You can use it as a template to create your own tax calculation package.

Clone or download the Tax Reference Integration Package package.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Payment Integration

The Commerce app payment architecture combines checkout APIs, the Salesforce Payment Gateway, and an integrated payment package for B2B and B2C stores.

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 B2C store. Plus, they're available for more Commerce domains. See Get Started With Salesforce Commerce Extensions.

Payment Architecture

The Salesforce Payment Adapter framework fetches tokens and authorizations from service providers during checkout.

Payment APIs

B2B and D2C Commerce implements a group of payment APIs to support tokenizing shopper credit cards without storing that information natively.

Payment Gateway

The Commerce Payments Gateway parses key object fields for tokenization, authorization, capture, refund, and other checkout API requests.

Payments Reference Packages

A reference payments integration package supports both B2B Commerce and D2C Commerce implementations. You can use it as a template to create your own payments package.

Payment Architecture

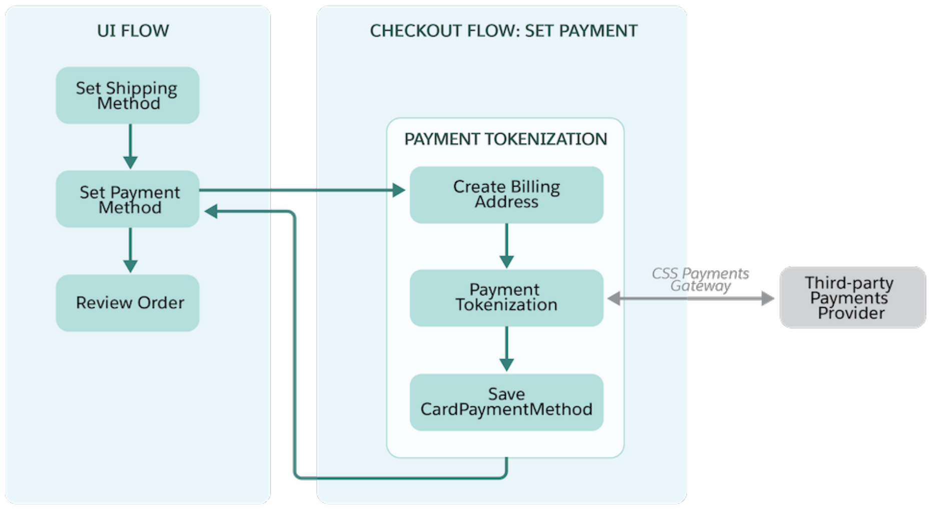
The Salesforce Payment Adapter framework fetches tokens and authorizations from service providers during checkout.

A shopper UI action triggers the SetPaymentMethod, which interacts with an integrated third-party payment provider to reserve funds.

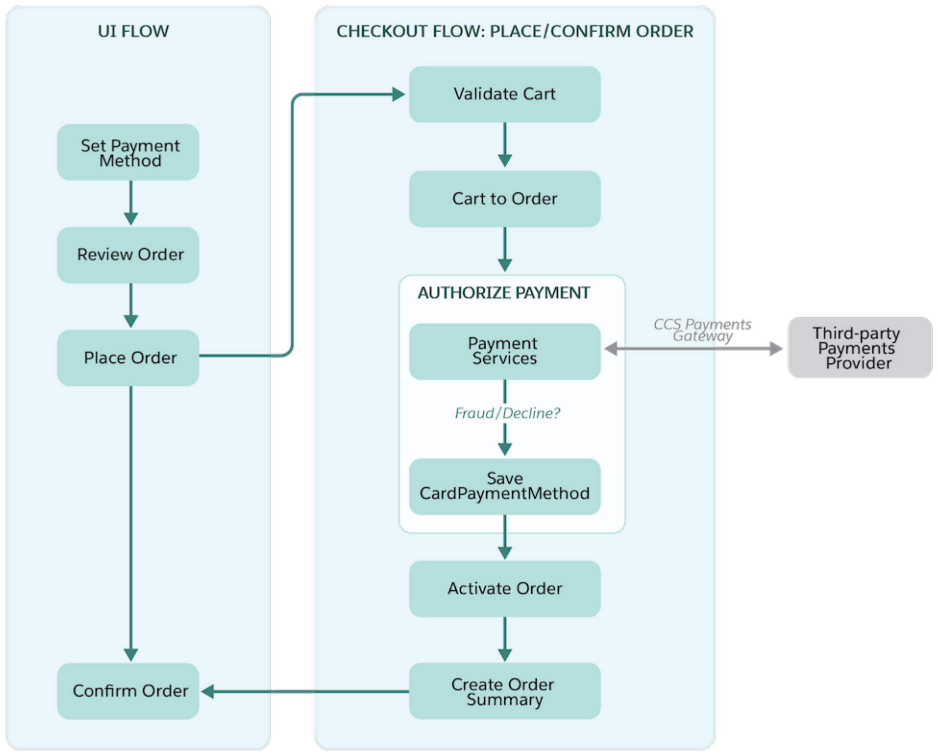
EDITIONS

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

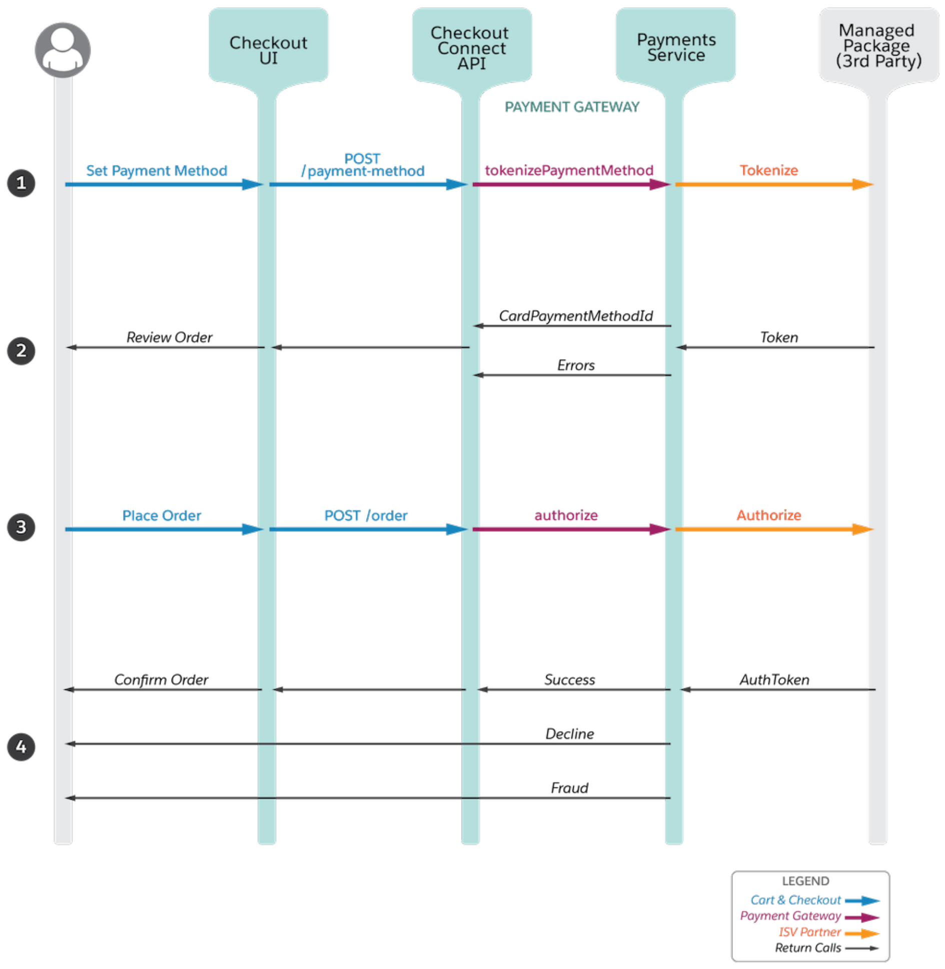
Available in: B2B Commerce and D2C Commerce



After the shopper reviews and approves the order, the PlaceOrder API completes the sale.



Integration packages for payment providers are interoperable across the B2B and D2C Commerce solution and Salesforce Order Management.



Payment APIs

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

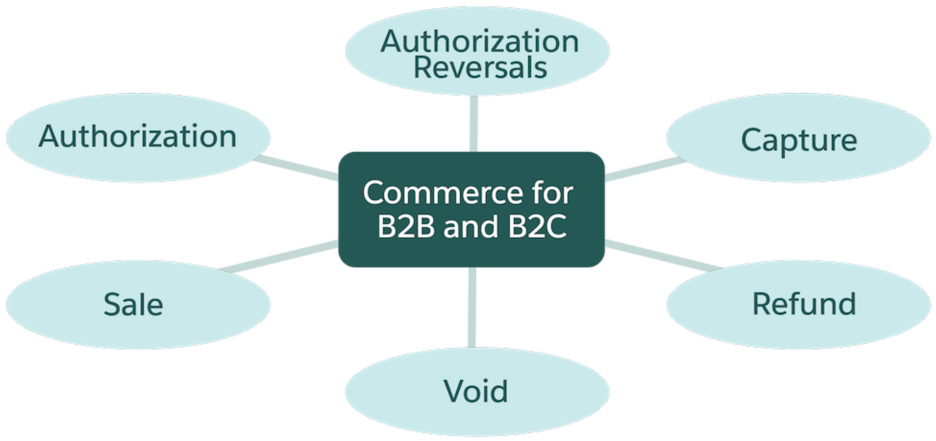
B2B and D2C Commerce implements a group of payment APIs to support tokenizing shopper credit cards without storing that information natively.

Shopper actions trigger payment API calls to a third-party service provider via the Payment Adapter framework. The APIs support tokenizing shopper credit cards so that no personal information is stored natively. Together with the licensed Salesforce Order Management (SOM) or 1C OM, the Payment Adapter framework exposes additional APIs. SOM manages the capture and refund flows and provides an integrated customer order lifecycle, including fulfillment and service.

These APIs initiate payments processing.

- **SetPaymentMethod** tokenizes the shopper's credit card and returns a token.
- **PlaceOrder** (returns Auth Code) validates the cart, reserves inventory, converts cart to order, authorizes payment, and activates the order in SOM. The payment authorization reserves funds from the available credit of the credit card, but it's not a payment. To the shopper, it can display as pending.

After the transaction is tokenized and the order placed, any of these APIs and corresponding transactions can occur.



- Authorization reversals release the funds reserved by the payment authorization, removing them from pending transactions.
- Capture consumes the funds reserved by the payment authorization.
- Sale is a transaction type where Authorization and Capture are executed as part of a single request. If successful, the order is fulfilled immediately.
- Void cancels the transfer of funds to the merchant account before settlement. When a payment is processed, funds are held. The balance is deducted from the customer's credit limit, but not yet transferred to the merchant account. At a later point, all transactions are batched for settlement, and the funds are transferred to the merchant's account. A transaction can be voided after purchase but before settlement.
- Refund is a transaction request that transfers the amount from the merchant's account to the customer's account.

For more information on these APIs, see Commerce Connect REST Payment APIs.

Payment Gateway

The Commerce Payments Gateway parses key object fields for tokenization, authorization, capture, refund, and other checkout API requests.

Payments Gateway uses the CommercePayments Apex namespace. For more information, see the Apex Reference Guide.

The Commerce Payments Gateway parses key object fields passed by the UI Checkout components. These fields include data to tokenize and authorize the payment request. For example, the Payments Gateway requires a tokenize request to pass this information.

EDITIONS

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

Available in: B2B Commerce and B2B2C Commerce

```
cardPaymentMethod: {
    cardHolderName: <string>,
    cardNumber: <string>,
    expiryMonth: <string>,
    expiryYear: <string>,
    cvv: <string>
}
address: {
    street: <string>,
    city: <string>,
    state: <string>,
    postalCode: <string>,
    country: <string>
}
```

Authorization, capture, refund, and other requests also pass these objects. Review the Stripe adapter samples in the Payments Reference Package to see how objects for transmission to Stripe are handled.

Examples for the Salesforce Payment Adapter include:

- `tokenizeRequest → commercePayments.PaymentMethodTokenizationRequest`
- `authRequest → commercePayments.AuthorizationRequest`
- `captureRequest → commercePayments.CaptureRequest`
- `refundRequest → commercePayments.ReferencedRefundRequest`

Payments Reference Packages

A reference payments integration package supports both B2B Commerce and D2C Commerce implementations. You can use it as a template to create your own payments package.

Clone or download the package: Payments Reference Integration Package

This guide describes how to set up development environments and provides step-by-step procedures for creating and deploying a payments package to your store.

Set Up Payment Processing

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

To collect credit card payment information in your B2B store using a synchronous payment gateway, create a payment gateway and connect it to your checkout. Payment processing in the checkout flow is limited to payment authorization. The checkout flow doesn’t collect or store payment information in the provided payment components. Payment authorization takes place between cart to order and order activation to limit the chance of an error occurring before credit card authorization.

Create your store and configure checkout before you set up your payment system.

Note: B2B Checkout and payment APIs don’t support authorization cancellation. Payment authorization takes place between cart to order and order activation to limit the chance of an error occurring before credit card authorization.

1. Create named credentials.
   a. From Setup, in the Quick Find box, enter *`NamedCredential`*, and then select **New Named Credential**.
   b. Enter the username, password, and URL for your payment gateway. These fields are dependent on your company’s external payment system.

      Example named credential

      - Label: My Named Credential
      - Name: My_Named_Credential
      - URL: https://api-cert.payeezy.com or https://api.stripe.com
      - Identity Type: Named Principal
      - Authentication Protocol: Password Authentication
      - Username: store@example.com
      - Password: password123

   c. Select **Allow Merge Fields in HTTP Header**.

   d. Save your changes.

2. Create a payment gateway adapter.

   A payment gateway adapter connects your payment platform in Salesforce and an external payment gateway. The Apex class that you create references your named credential.

   You can create your own adapter using the code samples and instructions in Payment Gateway Adapters. For example implementations associated with specific payment platforms, see Sample Payment Gateway Implementation for CommercePayments.

3. Create a payment gateway provider. See Set Up a Synchronous Payment Gateway Adapter.

   PaymentGatewayProvider is an object that stores details about the payment gateway that Salesforce communicates with when processing a transaction. It defines the connection to a payment gateway Apex adapter.

   a. Fill in the fields using your payment gateway adapter information.

      Example

      - ApexAdapterId: <Payment Gateway Adapter ID from Step 2>
      - DeveloperName: MyPaymentGatewayProvider
      - IdempotencySupported: Yes
      - Language: en_US
      - MasterLabel: MyPaymentGatewayProvider

4. Create a payment gateway object.

   The PaymentGateway object stores information that Salesforce uses to communicate with the payment gateway.

   a. From the Commerce app, in the object finder, enter `Payment Gateway` and select **Payment Gateway**.
   b. Click **New**.
   c. Enter a name.
   d. For Payment Gateway Provider, enter the ID of the payment gateway provider that you created in Step 3.
   e. For Merchant Credentials, enter your named credential.
   f. For Status, select **Active**.
   g. Save your work.

5. Connect your payment gateway to your checkout using the integrated store service.

   a. In the Commerce app navigation menu, select **Stores**, and select your store.
   b. Under Manage Your Store, select **Administration**.
   c. Select **Store Integrations**.
   d. On the Card Payment Gateway tile, select **Link Integration**.
   e. Choose the payment gateway that you created.

Test your payment gateway to make sure that it's working as expected. If you have trouble, check the Payment Gateway logs in your Commerce app. The PaymentGatewayLogs object creates records for every payment attempt and provides information when errors occur.

Handle Currency Changes for Active Carts

Commerce carts don't automatically support currency changes. If your store supports multiple currencies, we recommend that you make the currency picker inactive when a customer places an item in the cart. However, you can create a custom process to handle currency changes when items are in the cart.

1. Create a secondary cart in the new currency. See Commerce Webstore Carts.
2. Add items from the previous cart to the new cart. Make sure that products and price books are set up correctly in each currency that your store supports. See Commerce Webstore Cart Items and Commerce Webstore Cart Items, Batch.
3. Delete the previous cart. See Commerce Webstore Cart.
4. Make the secondary cart the primary cart. See Commerce Webstore Cart, Make Primary.

Commerce SFDX Environment Setup

Salesforce recommends the Salesforce Developer Experience (SFDX) environment for building Lightning web components (LWC), and for custom tax, shipping, and payment integration packages and extensions. SFDX provides easy access to Salesforce extensions and GitHub repositories containing LWC templates and reference packages. SFDX also integrates with Salesforce CLI, the Visual Studio Code editor with the Salesforce Extension Pack, and plug-ins to quickly deploy packages and components to scratch orgs and stores.

Install the SFDX CLI
The Salesforce DX (SFDX) CLI synchronizes your source code between the Salesforce orgs that you deploy to and your version control system.

Install the Visual Studio Code Editor
The free Visual Studio (VS) Code editor is an ideal development environment for creating, debugging, and deploying Salesforce Lightning web components and integration packages and extensions.

Get Salesforce Extensions for VS Code Editor
The VS Code editor provides access to Salesforce extensions that support creating integration packages and custom components for B2B and B2C stores created with an LWR template. The extensions include features for working with scratch orgs, sandboxes, and DE orgs, Lightning web components (LWC), and more.

Install the SFDX CLI

The Salesforce DX (SFDX) CLI synchronizes your source code between the Salesforce orgs that you deploy to and your version control system.

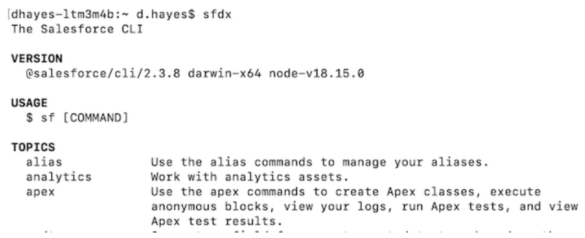
EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Install the CLI. See Install the Salesforce SFDX CLI.

To confirm installation, open a command or terminal window and enter `sfdx`.



```
[dhayes-ltm3m4b:~ d.hayes$ sfdx
 The Salesforce CLI

 VERSION
   @salesforce/cli/2.3.8 darwin-x64 node-v18.15.0

 USAGE
   $ sf [COMMAND]

 TOPICS
   alias            Use the alias commands to manage your aliases.
   analytics        Work with analytics assets.
   apex             Use the apex commands to create Apex classes, execute
                    anonymous blocks, view your logs, run Apex tests, and view
                    Apex test results.
```

Install the Visual Studio Code Editor

The free Visual Studio (VS) Code editor is an ideal development environment for creating, debugging, and deploying Salesforce Lightning web components and integration packages and extensions.

The free VS Code editor is open-source and optimized for cloud and web coding. The project folder combines, compiles, and displays the output of the HTML, JavaScript, and CSS files for your component.

- Install the open source Visual Studio Code editor.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Get Salesforce Extensions for VS Code Editor

The VS Code editor provides access to Salesforce extensions that support creating integration packages and custom components for B2B and B2C stores created with an LWR template. The extensions include features for working with scratch orgs, sandboxes, and DE orgs, Lightning web components (LWC), and more.

EDITIONS

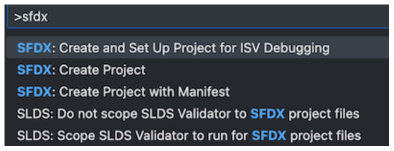
Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce and D2C Commerce

For more information, see the Salesforce Extensions for VS Code Overview.

1. Open the VS Code app.
2. Navigate to **View** > **Extensions**.
3. In the search field, enter `Salesforce`.
4. To install the package, click **Salesforce Extension Pack** (v. 51.4.0 or later).
5. To confirm installation in the VS Code Editor, open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS) and enter `sfdx`.

Confirmation looks like this:

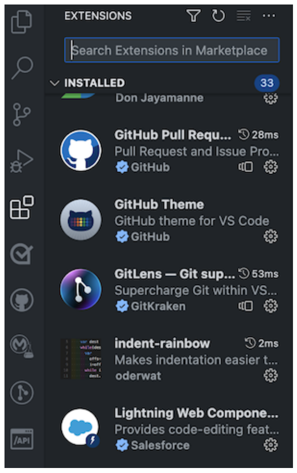


```
>sfdx
SFDX: Create and Set Up Project for ISV Debugging
SFDX: Create Project
SFDX: Create Project with Manifest
SLDS: Do not scope SLDS Validator to SFDX project files
SLDS: Scope SLDS Validator to run for SFDX project files
```

The Salesforce Extension Pack includes:

- Salesforce SFDX CLI integration for VS Code
- ESLint JavaScript integration for VS Code
- LWC for VS Code, which uses the VS Code HTML server to provide code-editing features for the LWC program model, including syntax highlighting, code completion, and file outlining

To display a list of the installed extensions, click the Extensions icon in the left navigation pane or press Ctrl+Shift+X (Windows) or Cmd+Shift+X (macOS) and scroll to view them all.


EXTENSIONS

Search Extensions in Marketplace

INSTALLED 33

Don Jayamanne

GitHub Pull Requ... 28ms
Pull Request and Issue Pro...
GitHub

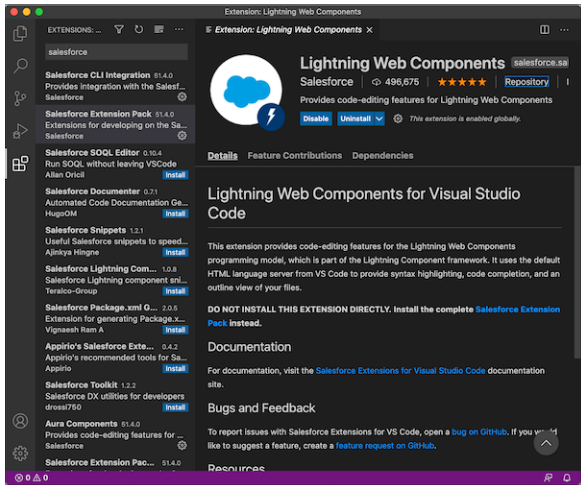
GitHub Theme
GitHub theme for VS Code
GitHub

GitLens — Git sup... 53ms
Supercharge Git within VS...
GitKraken

indent-rainbow 2ms
Makes indentation easier t...
oderwat

Lightning Web Compone...
Provides code-editing feat...
Salesforce

For an overview, see Salesforce Extensions for LWC


Extension: Lightning Web Components

EXTENSIONS: ...

salesforce

Salesforce CLI Integration 51.4.0
Provides integration with the Salesf...
Salesforce

Salesforce Extension Pack 51.4.0
Extensions for developing on the Sa...
Salesforce

Salesforce SOQL Editor 0.10.4
Run SOQL without leaving VSCode
Allan Oricil Install

Salesforce Documenter 0.7.1
Automated Code Documentation Ge...
HugoOM Install

Salesforce Snippets 1.2.1
Useful Salesforce snippets to speed...
Ajinkya Hingne Install

Salesforce Lightning Com... 1.0.8
Salesforce Lightning component sni...
Teralco-Group Install

Salesforce Package.xml G... 2.0.5
Extension for generating Package.x...
Vignaesh Ram A Install

Appirio's Salesforce Exte... 0.4.2
Appirio's recommended tools for Sa...
Appirio Install

Salesforce Toolkit 1.2.2
Salesforce DX utilities for developers
drossi750 Install

Aura Components 51.4.0
Provides code-editing features for ...
Salesforce

Salesforce Extension Pac... 51.4.0

Extension: Lightning Web Components

Lightning Web Components salesforce.sa
Salesforce | 496,675 | ★★★★★ | Repository
Provides code-editing features for Lightning Web Components
Disable Uninstall This extension is enabled globally.

Details Feature Contributions Dependencies

Lightning Web Components for Visual Studio Code

This extension provides code-editing features for the Lightning Web Components programming model, which is part of the Lightning Component framework. It uses the default HTML language server from VS Code to provide syntax highlighting, code completion, and an outline view of your files.

DO NOT INSTALL THIS EXTENSION DIRECTLY. Install the complete Salesforce Extension Pack instead.

Documentation

For documentation, visit the Salesforce Extensions for Visual Studio Code documentation site.

Bugs and Feedback

To report issues with Salesforce Extensions for VS Code, open a bug on GitHub. If you would like to suggest a feature, create a feature request on GitHub.

Resources

Build Custom Components

In addition to the Commerce App standard components, you can add custom components to your B2B or D2C store created with an LWR template.

Lightning Web Components
Custom Lightning web components (LWC) are easy to build, and they perform well in the web browser that hosts your B2B or D2C store created with an LWR template.

Storefront APIs
B2B and B2C stores built with LWR templates support Storefront APIs that help you build custom Lightning Web Components (LWCs) including headers, footers, and banners. Storefront APIs connect LWCs with data and execute processes, like adding a product to a wishlist or a cart, or applying a coupon to an order. Storefront APIs simplify LWC creation and help integrate components into larger page contexts.

Custom Payment Components

You can integrate a client-side payment capability as a Lightning web component in your B2B or B2C store created with an LWR template. Registered and guest buyers interact with the component during checkout to make direct or proxied calls to process the payment with an external provider.

Create a Custom Payment Component

Use the same credit card payment methods and order processing across stores by creating a custom payment component for your B2B or B2C store created with an LWR template.

Create Commerce Einstein Recommendations Components

Use Commerce Einstein Activity Tracking and Product Recommendations APIs to create custom components for your Salesforce Commerce B2C and B2B storefronts.

Create a Custom Checkout Component for a B2B or B2C Store (LWR)

You can create custom checkout components to extend the default checkout processing for a B2B or B2C store created with an LWR template.

Lightning Web Components

Custom Lightning web components (LWC) are easy to build, and they perform well in the web browser that hosts your B2B or D2C store created with an LWR template.

Build a Lightning web component by using HTML, JavaScript, and CSS. HTML provides the structure. JavaScript defines the core business logic, event handling, API calls to fetch page data, and related metadata. CSS provides the look, feel, and animation. Then deploy the component to Experience Builder with Commerce metadata. From start to finish, including deploying the component to your store, an SFDX project framework and related utilities assist you.

Note: Your LWR-based B2B or D2C store has dozens of standard Lightning web components. Before building a custom component, review the already-built LWCs described in LWR Store Components or, to view all the components available in your store template, see View All Components.

Create an SFDX Project for the Custom Component

Create a Salesforce Developer Experience (SFDX) project to store your custom component files.

Authorize an Org for an SFDX Project

To streamline deployment, authorize the org that SFDX deploys custom objects to.

Create a Sample Lightning Web Component

Create a sample custom component to deploy from SFDX to an authorized org. You can then reuse content from this Lightning web component file structure.

Deploy a Custom Component

Deploy a Lightning web component to use it on pages in your B2B or D2C store built with an LWR template.

Add a Custom Component to Your Store

Add a component to the canvas of your B2B or D2C store created with an LWR template. Optionally edit its properties.

Public Commerce LWR Library

Accelerate the development process with ready-to-use Lightning web components that help you customize your Commerce storefronts.

Create an SFDX Project for the Custom Component

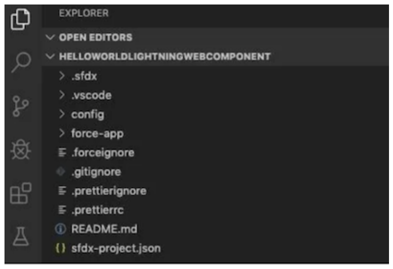
EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Create a Salesforce Developer Experience (SFDX) project to store your custom component files.

1. In the Visual Studio Code editor, open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS).
2. Enter *`SFDX`*.
3. Select **SFDX: Create Project**.
4. Click Enter.
5. Name the project, for example, *`HelloWorldLightningWebComponent`*, and click **Enter**.
6. Select a folder to store the project.
7. Click **Create Project**.



Authorize an Org for an SFDX Project

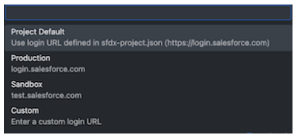
EDITIONS

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

Available in: B2B Commerce and D2C Commerce

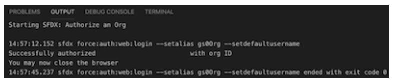
To streamline deployment, authorize the org that SFDX deploys custom objects to.

1. In the Visual Studio Code editor, open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS).
2. Enter *`SFDX`*.
3. Select **SFDX: Authorize an Org**



Project Default
Use login URL defined in sfdx-project.json (https://login.salesforce.com)
Production
login.salesforce.com
Sandbox
test.salesforce.com
Custom
Enter a custom login URL

4. Choose the Project Default (org URL in your project.json file), or a Production, Sandbox, or Custom org login URL.
   SFDX opens the Salesforce login portal to your org in a separate browser window.
5. Log in using your admin credentials.
6. If you're prompted to allow access, click **Allow**.
   After you authenticate in the browser, the CLI remembers your credentials. The success message looks like this:



```
Starting SFDX: Authorize an Org

14:57:12.152 sfdx force:auth:web:login --setalias gs0Org --setdefaultusername
Successfully authorized                          with org ID
You may now close the browser
14:57:45.237 sfdx force:auth:web:login --setalias gs0Org --setdefaultusername ended with exit code 0
```

Create a Sample Lightning Web Component

Create a sample custom component to deploy from SFDX to an authorized org. You can then reuse content from this Lightning web component file structure.

EDITIONS

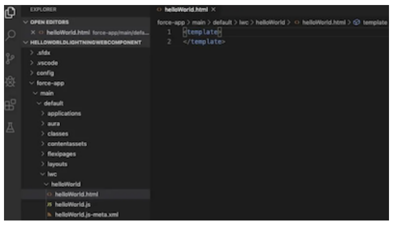
Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce and D2C Commerce

Note: To display your LWR component in Experience Builder, you must specify target configuration values in the .js-meta.xml file. You can also make component properties editable in Experience Builder.

1. In the Visual Studio Code editor, open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS).
2. Enter *`SFDX`*.
3. Select **SFDX: Create Lightning Web Component**.

   Don't select SFDX: Create Lightning Component, which creates an Aura component.
4. Name the component, for example *`helloWorld`*.
5. To accept the default `force-app/main/default/lwc` location, press **Enter**.
6. To view the new files in Visual Studio Code Explorer, press Enter.



```
<template>
</template>
```

7. Open the HTML file (for example, `helloWorld.html`), and write or copy and paste the HTML code for your project.
8. Save the file.
9. Open the JavaScript file (for example, `helloWorld.js`), and write or copy and paste the JavaScript code for your project.
10. Save the file.
11. Open the XML file (for example, `helloWorld.js-meta.xml`), and write or copy and paste the XML code for your project.
12. In the XML file, define the component design configuration values for Experience Builder.

Option	Description
`lightningCommunity__Page`	For a custom drag-and-drop component on an LWR page in Experience Builder (appears in the Components panel).
`lightningCommunity__Page_Layout`	For a custom page layout component in an LWR site in Experience Builder (appears in the Page Layout Window).
`lightningCommunity__Theme_Layout`	For a custom theme layout in an LWR site in Experience Builder (appears in Settings in the Theme area).
`lightningCommunity__Default`	Enables editable component properties when the component is selected in Experience Builder. Supported property attribute types include integer, string, boolean, picklist, and color.

   a. To make your component usable in Experience Builder, set `isExposed` to `true`, and define ***`lightningCommunity__Page`***, ***`lightningCommunity__Page_Layout`***, or ***`lightningCommunity__Theme_Layout`*** in `targets`.

b. To include properties that are editable when the component is selected in Experience Builder, define ***`lightningCommunity__Default`*** in targets and define the properties in `targetConfigs`.

Example: The snippet specifies a custom component for an LWR page in Experience Builder and also enables component property editing.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentHelloWorld xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>59.0</apiVersion>
     <isExposed>true</isExposed>
    <targets>
        <target>lightningCommunity__Page</target>
        <target>lightningCommunity__Default</target>
    </targets>
</LightningComponentHelloWorld>
```

Deploy a Custom Component

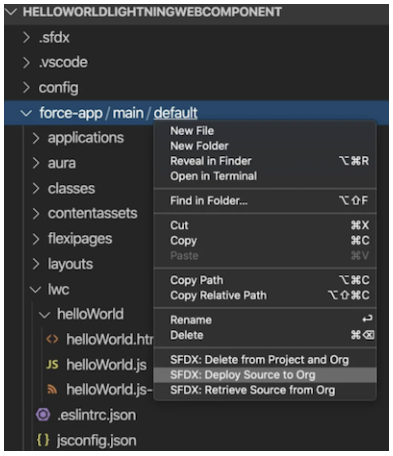
Deploy a Lightning web component to use it on pages in your B2B or D2C store built with an LWR template.

EDITIONS

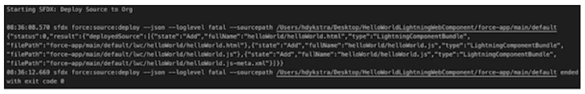
Available in: **Enterprise**, **Unlimited**, and **Developer** Editions

Available in: B2B Commerce and D2C Commerce

1. From the component project directory in Visual Studio Code, right-click the default folder under force-app/main, and select **SFDX: Deploy Source to Org**.



2. On the Output tab of the integrated terminal, view the results of your deployment. SFDX displays a deployment status notice that includes an exit code, such as "SFDX: Deploy Source to Org ... ended with exit code 0." Exit code 0 means that the command ran successfully.



```
Starting SFDX: Deploy Source to Org

08:36:08.570 sfdx force:source:deploy --json --loglevel fatal --sourcepath /Users/hdykstra/Desktop/HelloWorldLightningWebComponent/force-app/main/default
{"status":0,"result":{"deployedSource":[{"state":"Add","fullName":"helloWorld/helloWorld.html","type":"LightningComponentBundle",
"filePath":"force-app/main/default/lwc/helloWorld/helloWorld.html"},{"state":"Add","fullName":"helloWorld/helloWorld.js","type":"LightningComponentBundle",
"filePath":"force-app/main/default/lwc/helloWorld/helloWorld.js"},{"state":"Add","fullName":"helloWorld/helloWorld.js","type":"LightningComponentBundle",
"filePath":"force-app/main/default/lwc/helloWorld/helloWorld.js-meta.xml"}]}}
08:36:12.669 sfdx force:source:deploy --json --loglevel fatal --sourcepath /Users/hdykstra/Desktop/HelloWorldLightningWebComponent/force-app/main/default ended
with exit code 0
```

Add a Custom Component to Your Store

Add a component to the canvas of your B2B or D2C store created with an LWR template. Optionally edit its properties.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

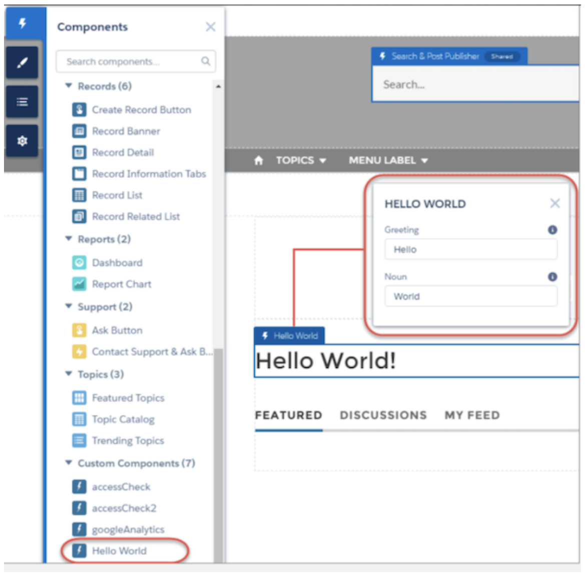
Note: Before placing a custom component, assign a custom theme layout to the page in Experience Builder.

1. In the Visual Studio Code editor, open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS).
2. Enter *`SFDX`*.
3. Select **SFDX: Open Default Org**.

   Your store opens in a separate browser.
4. In the left Navigation panel, click **Components**.

   The custom component displays under **Components** > **Custom Components**.
5. From the Custom area of the Lightning Components list, drag your Lightning web component to the page canvas.
6. To edit the component properties, select the component on the page canvas, and enter changes in the component property editor.

   To include properties that are editable when the component is selected in Experience Builder, open the .js-meta.xml file and add ***`lightningCommunity__Default`*** in targets and then define the properties in `targetConfigs`.



Public Commerce LWR Library

Accelerate the development process with ready-to-use Lightning web components that help you customize your Commerce storefronts.

The Public Commerce LWR Library in GitHub is an open repository of components that look and function like the product components available in Experience Builder. Clone and customize these public components to streamline your workflow and build outstanding Commerce storefronts.

The library is organized as an SFDX (Salesforce DX) project, and the components in this library are continually updated.

Contents include:

- Buttons, links, and windows with styling options

- Product Detail Page (PDP) items such as pricing displays, discount tiers, product variant selectors, and pricing quantity rules
- Add to Cart and item quantity selector controls
- Search results and filters

The PDP and search components in the reference library employ a versatile API to update data in component properties. These Builder-supported component APIs use expressions and data binding to achieve continuous data flows. The expressions and data bindings serve as input for `@api` properties in the reference LWCs.

SEE ALSO:

Public Commerce LWR Library GitHub Repository

Storefront APIs

EDITIONS

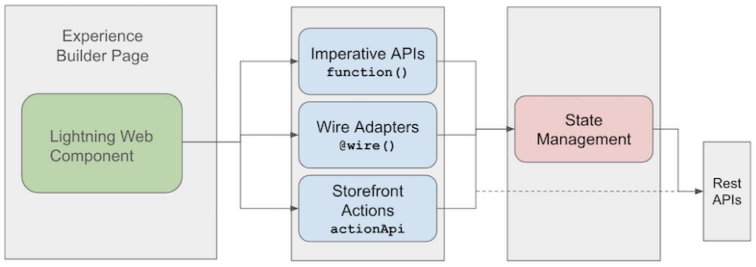
Available in: Lightning communities accessed through Lightning Experience in **Enterprise**, **Unlimited**, and **Developer** editions.

B2B and B2C stores built with LWR templates support Storefront APIs that help you build custom Lightning Web Components (LWCs) including headers, footers, and banners. Storefront APIs connect LWCs with data and execute processes, like adding a product to a wishlist or a cart, or applying a coupon to an order. Storefront APIs simplify LWC creation and help integrate components into larger page contexts.

Implementing components with Storefront APIs delivers distinct advantages over those relying on platform UI APIs and Aura controllers:

- Security - Storefront APIs automatically apply guardrails, such as buyer entitlement, that protect transactions and personalize customer journeys, with minimum risk.
- Performance and Scalability - Storefront APIs ensure that the same data is not requested and retrieved multiple times. Optimized to use multiple cache layers (browser, CDN, server), Storefront APIs retrieve information efficiently. Storefront Actions, for example, are designed to fetch data from the nearest available source.
- Convenience - Storefront APIs aggregate data from multiple sources and objects, like catalogs, or price books, and can execute business logic (fetch a product price, determine promotion eligibility).
- UI Cohesiveness - Components that implement Storefront APIs help ensure a consistent, cohesive customer experience. Cart components that use Wire Adapters, for example, achieve real-time reactivity: adding an item to a cart automatically updates the cart icon in the header.

The Storefront APIs fetch and display data for Commerce, Platform, and CMS products and include Imperative APIs, Wire Adapters, and Storefront Actions. Each of these differ in how and when they retrieve data for your component. A State Management layer provides access to hosted data via REST APIs and also coordinates operations that engage and update builder components. For example, when a buyer adds a product to the cart via a Storefront API, the State Management layer assures that this updates the list of cart items and the cart status icon. Taken together, the Storefront APIs provide a comprehensive framework for a dynamic shopping experience.



- Imperative APIs are process-oriented functions that, for example, call a Connect API method (POST, PUT, PATCH) to update data and return an outcome. Imperative calls typically retrieve hosted data when triggered by a page load or button click. Imperative APIs prompt updates to related entities including Wire Adapters, all orchestrated via State Management.

- Wire Adapters specify a GET method to call and retrieve data from a hosted data source. The adapter automatically updates properties in the LWC when hosted source data changes. When the LWC loads or updates, it executes the adapter call and retrieves the data in real time. For more information, see Understand the Wire Service.
- Storefront Actions are events, such as adding an item to a cart or using a filter to search on a product variant, that trigger updates to closely-related LWCs. Unlike actual backend API endpoint communication, Storefront Actions react to UI clicks more responsively, by making changes to related page-level components in chain-of-sequence fashion within a proximate scope. When a customer clicks a quantity selector component, for example, a `createCartItemsLoadAction` refreshes the quantity displayed in the cart. Tight data binding ensures that the triggered operation gets and sets data from the closest source. That said, Storefront Actions can result in calls (via Wire Adapters or Imperative APIs) to Rest API endpoints

These tables provide information about custom component Imperative APIs and adapters.

Imperative APIs

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint

`commerce/activitiesApi`	`trackAddProductToCart(`
	`product: Product`
	`): void`
		Commerce	55.0	Triggers the addToCart activity when a shopper adds a product to the cart.
				
				If you replace the Product Detail Purchase Options component with a custom component, implement the addToCart activity to ensure that Commerce Einstein Recommendations use cases generate results based on shopper or buyer view behavior.
					Einstein Recommendations API

`commerce/activitiesApi`	`trackClickReco(`
	`recommenderName: string, \\The name of the recommender`
	`recoUUID: string, \\A string representing the unique ID for this recommendation response`
	`product:`
		Commerce	55.0	Triggers the clickReco activity when a recommended product is clicked and the customer is taken to the product detail page.
				
				Implement this activity when building a custom Commerce Einstein recommendations component.
					Einstein Recommendations API

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
	`Product\\An object with an 18-character product ID that represents the product that the customer clicked sku: 'sku123' });\\(Optional) A unique stock keeping unit identifier for the product ): void`				
commerce/activitiesApi	`trackViewProduct({ id: '01t00000000000001',\\ The product that the customer viewed. An object with an 18-character product ID sku: 'sku123' });\\(Optional) A unique stock keeping unit identifier for the product`	Commerce	55.0	Triggers the viewProduct activity when a shopper views a Product Detail page. Don't fire if a product is displayed via a recommendation, search result, or any other means. If you replace the Product Detail Purchase Options component with a custom component, implement the viewProduct activity to ensure that Commerce Einstein Recommendations use cases generate results based on shopper or buyer view behavior.	Einstein Recommendations API

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
commerce/activitiesApi	`trackViewReco(recommenderName: string,\\The name of the recommender. recoUUID: string,\\A string representing the unique ID for this recommendation response products: Products,\\The products displayed to the customer. A list of one or more 18-character product IDs sku: SKU):\\(Optional) A unique stock keeping unit identifier for the product void`	Commerce	55.0	Triggers the viewReco activity when a recommendation is displayed to the customer. Implement this activity when building a custom Commerce Einstein recommendations component. If you calculate a recommendation but don't show it to the customer—for example, it doesn't have as many results as you like—don't fire this activity.	Einstein Recommendations API
commerce/cartApi	`addItemToCart(productId: string, quantity: number) : Promise<Record<string, unknown>>`	Commerce	57.0	Adds an item to a cart.	Commerce Webstore Cart Items
commerce/cartApi	`addItemsToCart(payload: AddItemsToCartActionPayload) : Promise<Record<string, unknown>>`	Commerce	57.0	Adds multiple items to the cart.	Commerce Webstore Cart Items

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
`commerce/cartApi`	`applyCouponToCart(couponCode: string) : Promise<Record<string, unknown>>`	Commerce	57.0	Applies a coupon to the cart.	Commerce Webstore Cart Items
`commerce/cartApi`	`deleteCouponFromCart(couponId: string) : Promise<Record<string, unknown>>`	Commerce	57.0	Deletes an applied coupon from the cart.	Commerce Webstore Cart Items
`commerce/cartApi`	`deleteCurrentCart() : Promise<Record<string, unknown>>`	Commerce	57.0	Deletes an active/current cart.	Commerce Webstore Cart Items
`commerce/cartApi`	`deleteItemFromCart(itemId: string) : Promise<Record<string, unknown>>`	Commerce	57.0	Deletes an item from the cart.	Commerce Webstore Cart Items
`commerce/cartApi`	`refreshCartSummary() : Promise<Record<string, unknown>>`	Commerce	58.0	Refreshes the cart summary.	Commerce Webstore Cart Items
`commerce/cartApi`	`updateCartStatusProcessing(status: boolean) : Promise<void>>`	Commerce	58.0	Sets the isProcessing status field of the cart to the provided processing state.	Commerce Webstore Cart Items
`commerce/cartApi`	`updateItemInCart(itemId: string, quantity: number) : Promise<Record<string, unknown>>`	Commerce	57.0	Updates the item quantity in the cart.	Commerce Webstore Cart Items
`commerce/checkoutApi`	`authorizePayment(checkoutId, tokenResponse.token, billingAddress): Promise<PaymentAuthorizationResponse>`	Commerce	56.0	Authorizes a tokenized payment for a checkout session.	Commerce Webstore Checkout

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
commerce/checkoutApi	`checkoutStatusIsReady(checkoutStatus: CheckoutStatus | undefined): boolean`	Commerce	60.0	Returns true if the supplied checkout status is complete and will accept additional parameters	Commerce Webstore Checkout
commerce/checkoutApi	`createContactPointAddress(address: Address): Promise<Address>`	Commerce	56.0	Creates a contact point address record. This API doesn't affect the checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`loadCheckout(): Promise<CheckoutInformation>`	Commerce	57.0	Loads the checkout session or error and saves it in the store for access by the wire adapter.	Commerce Webstore Checkout
commerce/checkoutApi	`notifyCheckout(state: CheckoutInformation | Error | null): Promise<CheckoutInformation | Error | null>`	Commerce	57.0	Publishes an updated checkout session state to the store so changes can be propagated by the wire adapter to subscribed listeners. Passing an Error replaces the checkout state and checkout ID with the error state. Passing null clears the published data cache. Returns the passed in data unmodified.	Commerce Webstore Checkout
commerce/checkoutApi	`notifyAndPollCheckout(state: CheckoutInformation | Error | null): Promise<CheckoutInformation | Error | null>`	Commerce	57.0	Calls notifyCheckout with supplied data. Additionally, when state indicates an async computation in progress (httpStatus is 202), it calls loadCheckout to poll until async tasks complete.	Commerce Webstore Checkout

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
				Automatically triggers a cart summary refresh if needed.	
commerce/checkoutApi	`paymentClientRequest(paymentsData: PaymentsData): Promise<PaymentsData>`	Commerce	57.0	Handles a variety of payment operations, including getting the metadata that the component needs from a third-party API, initiating a payment session, and making 3DS authentication confirmations.	Commerce Webstore Checkout
commerce/checkoutApi	`placeOrder(): Promise<OrderConfirmation>`	Commerce	56.0	Finalizes the order, completing the active checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`postAuthorizePayment(checkoutId: string, paymentToken: string, billingAddress?: Address): Promise<PaymentAuthorizationResponse>`	Commerce	56.0	Sends a client-side authorization result to the server.	Commerce Webstore Checkout
commerce/checkoutApi	`restartCheckout(): Promise<void>`	Commerce	56.0	Restarts an active checkout process. Before you use this API, clear all cached checkout and address data to prepare for a new "active" session. Clear cache is required anytime the previous checkout session becomes invalid. Attempts to use other checkout APIs after the previous session become invalid and fail until the restart is called.	Commerce Webstore Checkout

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
commerce/checkoutApi	`simplePurchaseOrderPayment(checkoutId: string, tokenResponseToken: string, billingAddress: Address): Promise<PaymentAuthorizationResponse>`	Commerce	57	Sends a simple purchase order number to the server.	Commerce Webstore Checkout
commerce/checkoutApi	`updateContactInformation(contactInfo: ContactInfo): Promise<void>`	Commerce	56.0	Updates the guest contact information in the active checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`updateContactPointAddress(address: Address): Promise<Address>`	Commerce	56.0	Updates an existing contact point address record. This API doesn't affect the checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`updateDeliveryMethod(deliveryMethodId: string): Promise<void>`	Commerce	56.0	Updates the delivery method for the default delivery group in the active checkout session, and updates the cart summary.	Commerce Webstore Checkout
commerce/checkoutApi	`updateGuestEmail(guestEmail: string | undefined): Promise<void>`	Commerce	56.0	Updates the cached guest email value that's shared between components. This API doesn't affect the checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`updateShippingAddress(deliveryGroup: DeliveryGroup): Promise<void>`	Commerce	56.0	Updates the shipping address for the default delivery group in the active checkout session.	Commerce Webstore Checkout
commerce/checkoutApi	`waitForCheckout(): Promise<CheckoutInformation>`	Commerce	57.0	Returns a resolved promise immediately if the checkout status is complete. Otherwise, the returned promise	Commerce Webstore Checkout

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
				resolves after the checkout status becomes complete or errors.	
commerce/contextApi	`getAppContext(): Promise<AppContext>`	Commerce	53.0	Get application-context-specific data.	Commerce Webstore Application Context
commerce/contextApi	`getSessionContext(): Promise<SessionContext>`	Commerce	53.0	Get session-context-specific data.	Commerce Webstore Application Context
commerce/myAccountApi	`createMyAccountAddress( address: MyAccountAddress ): Promise<MyAccountAddress>`	Commerce	54.0	Create an account address.	Commerce Webstore Account Address
commerce/myAccountApi	`deleteMyAccountAddress( addressId: string ): Promise<void>`	Commerce	54.0	Delete an account address.	Commerce Webstore Account Address
commerce/myAccountApi	`updateMyAccountAddress( address: MyAccountAddress ): Promise<MyAccountAddress>`	Commerce	54.0	Update an account address.	Commerce Webstore Account Address
commerce/myAccountApi	`resetPassword( username: string ): Promise<Response>`			Reset a password.	Commerce Webstore Account Address
commerce/myAccountApi	`updateMyAccountProfile(profile: MyAccountProfileRequestOptions):`	Commerce	59.0	Update an existing account profile.	Commerce Webstore Account Address

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
	`Promise<MyAccountProfileResponse>`				

`commerce/orderApi`	`startReOrder(options:`
	`OrderActionAddToCartRequestOptions):`
	`Promise<OrderActionAddToCartData>`
		Commerce	57.0		Commerce Checkout Order

`experience/cmsEditorApi`	`contentBodyModify['propertyOfContentType']`
	` = 'new value';`
	`updateContent({`
	
	`contentBody:`
	`contentBodyModify`
	`}).then(() =>`
	` {`
	` });`
		CMS	54.0	Updates the property to be updated in content form. Then calls back after the form is updated.	CMS Content

`experience/effectiveAccountApi`	`effectiveAccount.update(`
	` 'Sample Account Id' ,`
	`  'Sample Account Name'`
	` ); // To set the effective account Id and account name in session storage.`
	
	`const accountId =`
	`effectiveAccount.accountId`
	` // To read the effective account Id from session storage.`
	
	`const accountName =`
	
	`effectiveAccount.accountName`
		Commerce	57	Sets the account name and id in session storage.	B2B and D2C Commerce Resources

Namespace/Bundle	Name/Interface	Product	API Version	Description	Associated Endpoint
	`// To read the effective account name from session storage. });`				

Wire Adapters in the Commerce Namespace

Namespace/Bundle	Adapter	Product	API Version	Description	Associated Endpoint
`commerce/cartApi`	`CartCouponsAdapter`	Commerce	55.0	Retrieves cart coupon information, loads cart coupons.	Commerce Webstore Cart Coupons
`commerce/cartApi`	`CartItemsAdapter`	Commerce	53.0	Fetches and loads cart items, supports sorting.	Commerce Webstore Cart Items
`commerce/cartApi`	`CartPromotionsAdapter`	Commerce	55.0	Retrieves and loads cart-level and item promotion information when eligible items are added or updated.	Commerce Webstore Cart Promotions
`commerce/cartApi`	`CartSummaryAdapter`	Commerce	53.0	Fetches current cart totals data from rollup calculators for CartBadge, CartSummary, Checkout, and other components.	Commerce Webstore Cart
`commerce/cartApi`	`CartStatusAdapter`	Commerce	58.0	Retrieves status information about a cart, including whether a cart is processing, whether a cart is ready for checkout, and whether guest cart or checkout access is enabled.	Commerce Webstore Cart

Namespace/Bundle	Adapter	Product	API Version	Description	Associated Endpoint
`commerce/checkoutApi`	`CheckoutAddressAdapter`	Commerce	56.0	Uses multiple calls to aggregate results for the get-address-list API. This adapter is a deeper get-address-list API wrapper.	Commerce Webstore Checkout
`commerce/checkoutApi`	`CheckoutGuestEmailAdapter`	Commerce	56.0	Retrieves guest email info value from the store.	Commerce Webstore Checkout
`commerce/checkoutApi`	`CheckoutInformationAdapter`	Commerce	56.0	Retrieves data from CheckoutStore, guest email info value from the store, the deliveryGroup, and the first shippingInstructions in the store. Called one time at startup and then whenever the data store updates.	Commerce Webstore Checkout
`commerce/contextApi`	`AppContextAdapter`	Commerce	53.0	Retrieves app context data from the AppContext store. If data hasn't been loaded, the adapter calls the application context API.	Application Context
`commerce/contextApi`	`SessionContextAdapter`	Commerce	53.0	Retrieves session context data from the SessionContext store. If data hasn't been loaded, the adapter calls the session context API.	Session Context
`commerce/orderApi`	`OrderAdapter`	Commerce	57.0	Called when a buyer views the order summary of an order. Retrieves an order summary, including fields,	Commerce Webstore Order Summary

Namespace/Bundle	Adapter	Product	API Version	Description	Associated Endpoint
				based on the effective account ID.	
commerce/orderApi	OrdersAdapter	Commerce	57.0	Called when a buyer views their order history page. Retrieves order summaries, including fields, for the buyer associated with the storefront.	Commerce Webstore Order Summaries
commerce/orderApi	OrdersAdjustmentsAdapter	Commerce	57.0	Called when a buyer views the order summary of an order. Retrieves adjustments for an order summary.	Commerce Webstore Order Summary, Adjustments
commerce/orderApi	OrderDeliveryGroupsAdapter	Commerce	57.0	Called when a buyer views the order summary of an order. Retrieves an order delivery group.	Commerce Webstore Order Delivery Groups
commerce/orderApi	OrderItemsAdapter	Commerce	57.0	Called when a buyer views the order summary of an order. Retrieves order items, including fields.	Commerce Webstore Order Items
commerce/orderApi	OrderItemsAdjustmentsAdapter	Commerce	57.0	Called when a buyer views the order summary of an order. Retrieves adjustments for order items.	Commerce Webstore Order Items, Adjustments
commerce/orderApi	OrderSummaryLookupAdapter	Commerce	58.0	Retrieves details about an OrderSummary for a guest shopper or a registered buyer	Commerce Webstore Order Summary Lookup
commerce/productApi	ProductAdapter	Commerce	53.0	Called when the product is loaded. Retrieves product	Commerce Webstore Product

Namespace/Bundle	Adapter	Product	API Version	Description	Associated Endpoint
				information from the store.	
commerce/productApi	ProductCategoryAdapter	Commerce	53.0	Retrieves category information, including fields and related media.	Commerce Webstore Product Category
commerce/productApi	ProductCategoryHierarchyAdapter	Commerce	53.0	Retrieves a list of product category information, including fields and related media, based on the parent product category Id.	Commerce Webstore Product Categories Children
commerce/productApi	ProductCategoryPathAdapter	Commerce	53.0	Retrieves the category path from the root category to the current category.	Commerce Webstore Product Category Path
commerce/productApi	ProductInventoryLevelsAdapter	Commerce	58.0	Retrieves the product inventory levels for a given product.	Salesforce Omnichannel Inventory Resources
commerce/productApi	ProductPricingAdapter	Commerce	53.0	Called when the product is loaded. Retrieves pricing information for a given product.	Commerce Webstore Pricing Products
commerce/productApi	ProductSearchAdapter	Commerce	57.0	Retrieves products by search parameters.	Commerce Webstore Product Search
commerce/productApi	ProductTaxAdapter	Commerce	57.0	Retrieves tax information for a given product.	Commerce Webstore Taxes
commerce/recommendationsApi	ProductRecommendationsAdapter	Commerce	55.0	Returns Einstein product recommendations. Import it from the einsteinAPI module within the commerce namespace (commerce/einsteinAPI) and then declare its	B2C Commerce Einstein Product Recommendations API

Namespace/Bundle	Adapter	Product	API Version	Description	Associated Endpoint
				required parameters with the `@wire` decorator.	

Wire Adapters in the Experience Namespace

Namespace/Bundle	Wire Adapter	Product	API Version	Description	Associated Endpoint
`experience/navigationMenuApi`	`getNavigationMenu`	Platform	54.0	Retrieves the items for a navigation menu in an Experience Cloud site.	Navigation Menu Items
`experience/cmsDeliveryApi`	`getContent`	CMS	54.0	Retrieves published content from an enhanced CMS workspace to use in a custom Lightning web component for an enhanced LWR site.	CMS Delivery Content
`experience/cmsEditorApi`	`getContext`	CMS	54.0	Retrieve metadata about the content item in the CMS content editor.	CMS Content
`experience/cmsEditorApi`	`getContent`	CMS	54.0	Retrieve the title, urlName, and contentBody of the content item based on the content type: image, news, document, or custom content.	CMS Content

Storefront Actions

Storefront Actions are centrally combined in the component bundle `commerce/actionApi`. The table shows the factory methods for this bundle.

Category	Name	Product	API Version	Description	Properties	actionApi Source
Cart	createCartItemAddAction	Commerce	58.0	Returns an Action that triggers an add-item-to-cart operation from the closest applicable Data Provider.	`productId: string` `quantity?: number`	Commerce Webstore Cart Items
Cart	createCartItemUpdateAction	Commerce	58.0	Returns an Action that triggers an update to cart contents operation from the closest applicable Data Provider.	`cartItemId: string` `quantity: number`	Commerce Webstore Cart Items
Cart	createCartItemDeleteAction	Commerce	58.0	Returns an Action that triggers a remove-item-in-cart operation from the closest applicable Data Provider.	`cartItemId: string`	Commerce Webstore Cart Items
Cart	createCartItemsAddAction	Commerce	58.0	Returns an Action that triggers an add items in cart operation from the closest applicable Data Provider.	`items: Record<string, number>`	Commerce Webstore Cart Items
Cart	createCartItemsLoadAction	Commerce	58.0	Returns an Action that triggers a load more cart items operation from the closest applicable Data Provider.		Commerce Webstore Cart Items
Cart	createCartSortUpdateAction	Commerce	58.0	Returns an Action that triggers an update cart sort order operation from the closest applicable Data Provider.	`sortOrder?: 'CreatedDateDesc' | 'CreatedDateAsc' | 'NameDesc' | 'NameAsc'`	Commerce Webstore Cart Items

Category	Name	Product	API Version	Description	Properties	actionApi Source
Cart	createCartStatusUpdateAction	Commerce	58.0	Returns an Action that triggers an update cart status operation from the closest applicable Data Provider.	`status?: { isProcessing?: boolean; isReadyForCheckout?: boolean; isGuestCartEnabled?: boolean; isGuestCheckoutEnabled?: boolean; }`	Commerce Webstore Cart Items
Cart	createCartClearAction	Commerce	58.0	Returns an Action that triggers a clear cart operation from the closest applicable Data Provider.		Commerce Webstore Cart Items
Cart	createCartInventoryReserveAction	Commerce	59.0	Returns an Action that triggers a reserve inventory for cart items operation from the closest applicable Data Provider.		Commerce Webstore Cart Items
Checkout	createCheckoutAddressesCreateAction	Commerce	58.0	Returns an Action that triggers a checkout addresses create operation from the closest applicable Data Provider.	`address: { name?: string; firstName?: string; lastName?: string; companyName?: string; street?: string; city?: string; postalCode?: string; region?: string; country?: string; addressId?:`	checkout ActionApi

Category	Name	Product	API Version	Description	Properties	actionApi Source
					`string; id?: string; fields?: unknown; geocodeAccuracy?: string; isDefault?: boolean; addressType?: string; label?: string;}`	
Checkout	createCheckoutAddressesPageChangeAction	Commerce	58.0	Returns an Action that triggers a checkout addresses page change from the closest applicable Data Provider.	`total: number`	Commerce Webstore Checkout
Checkout	createCheckoutAddressesUpdateAction	Commerce	58.0	Returns an Action that triggers a checkout addresses update from the closest applicable Data Provider.	`address: { name?: string; firstName?: string; lastName?: string; companyName?: string; street?: string; city?: string; postalCode?: string; region?: string; country?: string; addressId?: string; id?: string; fields?: unknown;`	Commerce Webstore Checkout

Category	Name	Product	API Version	Description	Properties	actionApi Source
					`geocodeAccuracy?: string; isDefault?: boolean; addressType?: string; label?: string;}`	
Checkout	createCheckoutFinalizeAction	Commerce	58.0	Returns an Action that triggers a finalize action from the closest applicable Data Provider.		Commerce Webstore Checkout
Checkout	createCheckoutPlaceOrderAction	Commerce	58.0	Returns an Action that triggers a place order action from the closest applicable Data Provider.		Commerce Webstore Checkout
Checkout	createCheckoutUpdateFormAction	Commerce	58.0	Returns an Action that triggers a checkout update from the closest applicable Data Provider.	`* applicable {@link DataProvider}.`	Commerce Webstore Checkout
Common	createCommonQuantityUpdateAction	Commerce	59.0	Returns an Action that triggers an update quantity from the closest applicable Data Provider.	`quantity: number`	Commerce Webstore Cart Item
Coupon	createCouponApplyAction	Commerce	58.0	Returns an Action that triggers an apply coupon from the closest applicable Data Provider.	`couponId: string`	Commerce Webstore Cart Promotions
Coupon	createCouponDeleteAction	Commerce	58.0	Returns an Action that triggers a delete coupon from the closest	`couponId: string`	Commerce Webstore Cart Promotions

Category	Name	Product	API Version	Description	Properties	actionApi Source
				applicable Data Provider.		
Order	createOrderAccessValidateAction	Commerce	59.0	Returns an Action that triggers an order summary access validate from the closest applicable Data Provider.	`verificationDetails: { orderSummaryIdOrRefNumber?: string; fields?: string[] | null; excludeAdjustments?: boolean | null; excludeAdjustmentAggregates?: boolean | null; excludeLineItems?: boolean | null; excludeDeliveryGroups?: boolean | null; email?: string | null; lastName?: string | null; phoneNumber?: string | null;}`	Commerce Webstore Order Summary
Product	createProductQuantityUpdateAction	Commerce	58.0	Returns an Action that triggers an update product quantity from the closest applicable Data Provider.	`productId: stringquantity: number`	Commerce Webstore Order Summary
Product	createProductVariantUpdateAction	Commerce	58.0	Returns an Action that triggers an update product variant from the closest applicable Data Provider.	`options: string[]isValid: boolean`	CMS Contents Variant

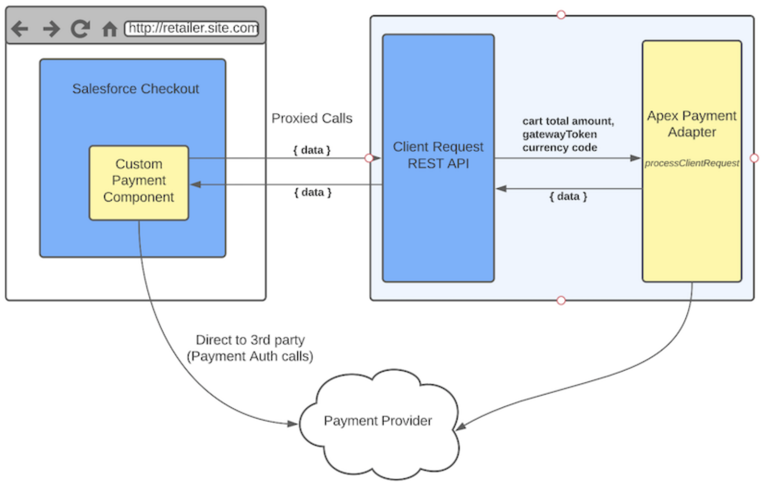
Category	Name	Product	API Version	Description	Properties	actionApi Source
Product	`createProductSubscriptionUpdateAction`	Commerce	59.0	Returns an Action that triggers an update product subscription from the closest applicable Data Provider.	`productSellingModelId: string subscriptionTerm?: number | null`	Subscription
Search	`createSearchSortUpdateAction`	Commerce	58.0	Returns an Action that triggers an update search sort order from the closest applicable Data Provider.	`sortRuleId?: string`	Commerce Webstore Product Search
Search	`createSearchFiltersUpdateAction`	Commerce	58.0	Returns an Action that triggers an update search filters from the closest applicable Data Provider.	`refinements?: ProductSearchRefinement[]; categoryId?: string; page?: number; mruFacet?: SearchFacetData;}`	Commerce Webstore Product Search
Search	`createSearchFiltersClearAction`	Commerce	58.0	Returns an Action that triggers a clear search filters from the closest applicable Data Provider.		Commerce Webstore Product Search
Wishlist	`createWishlistItemAddAction`	Commerce	58.0	Returns an Action that triggers an add items to wishliist from the closest applicable Data Provider.	`productId: string`	Commerce Webstore Wishlists
Wishlist	`createWishlistItemDeleteAction`	Commerce	58.0	Returns an Action that triggers an delete items to wishliist from the closest applicable Data Provider.	`wishlistItemId: string`	Commerce Webstore Wishlists

Custom Payment Components

You can integrate a client-side payment capability as a Lightning web component in your B2B or B2C store created with an LWR template. Registered and guest buyers interact with the component during checkout to make direct or proxied calls to process the payment with an external provider.

The custom payment component incorporates the ClientSidePaymentAdapter interface, which you include in a new or existing payment adapter Apex class. The Apex class connects to the Commerce Payment Gateway adapter.

The store renders the deployed client payment component in the checkout browser. The client component works within a Named Credential, Payment Gateway, and OMS workflow that is similar to a server-side payment package but with its own corresponding metadata.



The component can make an authorized payment call directly to the payment provider without passing sensitive personal data.

Unlike with other custom LWCs, you don't drag the deployed payment component to the store canvas in Experience Builder. Instead, after you deploy the component, go to Experience Builder, select the card payment gateway, now integrated with the new payment component, and then republish the store. The store automatically presents and uses the payment component defined in the payment adapter Apex class.

Create a Custom Payment Component

Use the same credit card payment methods and order processing across stores by creating a custom payment component for your B2B or B2C store created with an LWR template.

To create and deploy the payment component, first Commerce SFDX Environment Setup.

1. In the Visual Studio Code editor, enter *SFDX*.
2. Select **SFDX: Create an Apex Class** or navigate to the project containing an existing payment adapter class.

   The Apex class implements the PaymentGatewayAdapter and ClientSidePaymentAdapter interfaces. Here's the ClientSidePaymentAdapter interface implemented with the PaymentGatewayAdapter.

```
class MyClientSideAdapter implements commerce.ClientSidePaymentAdapter,
commercepayments.PaymentGatewayAdapter {

ClientSidePaymentAdapter {
     // returns custom component name package_namespace/componentName
     String getClientComponentName();
     // returns configuration to bootstrap client component (Public API Keys, account
key, etc)
     Map<String, String> getClientConfiguration();

     // Generic method to do server side operations
     // If a gateway token is returned in the ClientResponse it will be saved
     // as the payment method for the cart.
     // Params:
             // context - Includes the total amount of the cart including tax/shipping,

             // the currency code for the transaction, the token that represents the
payment method that's associated
             // with the cart, and a key that can be used for idempotency
             // paymentsData - Arbitrary map of data passed from the custom component
on the UI

     ClientResponse processClientRequest(ClientRequestContext context, Map<String,
Object> paymentsData);
}

ClientResponse {
   global ClientResponse (String token, Map<String, Object> response)
```

3. Add a PostAuth request to the Apex class.

   The PostAuth request type ensures that the provided authentication token is valid. The token, the amount, and currency code are provided to the adapter through the PostAuth input.

   ```
   postAuthRequest.alternativePaymentMethod.gatewayToken
   ```

   The input for this request type is defined here.
   https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_connectapi_input_post_auth.htm

   The output is defined here.
   https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_connectapi_output_post_auth_output.htm

4. Create named credentials for the adapter class.

   a. In Visual Studio Code Explorer, under `force-app/main/default`, create a subdirectory called `namedCredentials`.

   b. In namedCredentials, create a file named *`payment`*`.namedCredential`.

   c. In the public repo of a reference payment package, such as the Stripe integration payments package, go to the `/namedCredentials` folder and copy the contents of the `Stripe.namedCredential` file to your *`payment`*`.namedCredential`.

5. Create the payment Lightning web component.

   a. Create an SFDX project.

   b. Authorize an org for the LWC deployment.

   c. Ensure that the LWC includes this CustomPaymentComponent interface.

```
export interface CustomPaymentComponent extends LightningElement {
    /**
    * clientConfiguration - Response from payment adapter apex getClientConfiguration
method
```

```
    *
    **/
    initialize(clientConfiguration: Record<string, string>, webStoreId: string): void;
   completePayment(address: BillingAddress): Promise<ClientSideCompletePaymentResponse>;

    focus(): void;
    reportValidity: () => boolean;
}

export interface ClientSideCompletePaymentResponse {
    responseCode?: string;
    error?: {
        message: string;
        code: string;
}
```

6. Deploy the payment component and all the code and metadata required for the integration (LWC, Apex, named credentials, trusted sites) to your org.
7. Create a payment gateway and link to the stores, as described in Set Up Payment Processing.
8. In Experience Builder, register the third-party trusted sites for scripts that are required to run the LWC, as described in Set Up Payment Processing.
9. Activate the payment gateway linked to the new payment component, as described in Set Up Payment Processing.
10. Publish the store.

Create Commerce Einstein Recommendations Components

Use Commerce Einstein Activity Tracking and Product Recommendations APIs to create custom components for your Salesforce Commerce B2C and B2B storefronts.

Commerce Einstein APIs
Use Activity Tracking and Product Recommendation APIs to call Einstein Recommendations for your Salesforce Commerce B2C and B2B stores. These APIs provide access to underlying data behind Einstein Recommendations and enable modification of the front-end experience with custom Lightning web components or Aura components.

Prepare Commerce Einstein to Use Custom Components
Before creating custom Commerce Einstein Recommendations components, deploy Commerce Einstein and enable activity tracking.

Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs
Creating a custom B2C Commerce Einstein recommendations component includes implementing the Activity Tracking API and connecting the component to the Product Recommendation API.

Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs
Creating a custom B2B Commerce Einstein recommendations component includes implementing the Activity Tracking API and connecting the component to the Commerce Einstein Webstore Recommendations Connect API resource. This process is for B2B stores on the Aura platform.

Commerce Einstein Recommendations API Reference
Modifying Einstein Recommendations interfaces in your D2C and B2B Commerce stores requires using the Activity Tracking API and a Product Recommendation API. These APIs provide access to the underlying data behind Einstein Recommendations and enable modification of the front-end experience with custom Lightning web components.

Commerce Einstein APIs

EDITIONS

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

Available in: B2B Commerce and B2B2C Commerce

Use Activity Tracking and Product Recommendation APIs to call Einstein Recommendations for your Salesforce Commerce B2C and B2B stores. These APIs provide access to underlying data behind Einstein Recommendations and enable modification of the front-end experience with custom Lightning web components or Aura components.

Einstein product recommendations require activity tracking data. For example, tracking the viewProduct activity provides insight into top-viewed products. The Activity Tracking API provides access to these activities for use in B2B and B2C Einstein Recommendations components.

Calls to the Product Recommendation API deliver Einstein recommendations. For Salesforce Commerce for B2C Einstein Recommendations components, the API is accessed using a wire adapter. The Product Recommendation API returns a set of products to populate the Einstein Recommendations component.

For Salesforce Commerce for B2B Einstein Recommendations components, use the Commerce Einstein Recommendations Connect REST API to select and deliver recommendations. Using the Connect REST API ensures appropriate product entitlement filtering before recommendations are delivered.

SEE ALSO:

- Prepare Commerce Einstein to Use Custom Components
- Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs
- Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs
- Commerce Einstein Recommendations API Reference

Prepare Commerce Einstein to Use Custom Components

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

Before creating custom Commerce Einstein Recommendations components, deploy Commerce Einstein and enable activity tracking.

1. Deploy Commerce Einstein.
2. Enable Activity Tracking.
3. Commerce SFDX Environment Setup.

Note: Make sure to install all recommended software and plug-ins.

SEE ALSO:

- Commerce Einstein APIs
- Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs
- Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs
- Experience Cloud Cookies
- Deploy Commerce Einstein
- Activity Tracking

Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs

Creating a custom B2C Commerce Einstein recommendations component includes implementing the Activity Tracking API and connecting the component to the Product Recommendation API.

For B2C storefronts, you connect the component to the Product Recommendation API through a wire adapter. Your component must incorporate calls to the Activity Tracking API. When configuring the Activity Tracking API, keep these considerations in mind.

- Implement both the viewReco and clickReco activities.
- If you build custom components that show product details without including the Product Detail Purchase Options component, make sure to implement the viewProduct activity.
- If you add custom add-to-cart functionality, also implement the addToCart activity.

EDITIONS

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

Available in: D2C Commerce

You can also introduce custom JavaScript, HTML, and CSS to make front-end modifications. For more information, see the Lightning Web Components Dev Guide.

Note: This example shows a custom similar-products Commerce Einstein recommendation component using the PRODUCT anchor type. All steps refer to files created for a sample Lightning web component. For more information, see Create a Sample Lightning Web Component.

1. In the component's JavaScript file, define a custom component class (for example, `RecommendationsSample`). Implement the ProductRecommendationsAdapter to return Einstein product recommendations.

   For example, in `force-app/main/default/<Component Name>.js`, import this wire adapter from the `recommendationsApi` module within the commerce namespace (`commerce/recommendationsApi`). Declare its required parameters with the `@wire` decorator.

```
import { LightningElement, api, track, wire } from 'lwc';
import { ProductRecommendationsAdapter, ANCHOR_TYPES } from 'commerce/recommendationsApi';
import { navigate, NavigationContext } from 'lightning/navigation';
import { trackClickReco, trackViewReco } from 'commerce/activitiesApi';

export default class RecommendationsSample extends LightningElement {

    @api products = [];
    @api productId;

    recommenderName = 'similar-products';
    anchorType = ANCHOR_TYPES.PRODUCT;
    get anchorValue() {
        return [this.productId];
    };

    @wire(ProductRecommendationsAdapter, {
        recommenderName: '$recommenderName',
        anchorType: '$anchorType',
        anchorValue: '$anchorValue',
    })
    async loadRecommendation(response) {
        let { data, error } = response;

        if (data && data.recoUUID && data.products && data.products.length > 0) {
            this.products = data.products;
            this.recoUUID = data.recoUUID;

            if (this.canDisplayRecommendations) {
                this.sendViewReco();
            }
        } else if (error) {
            // unable to load recommendation, handle accordingly
            this.products = { data: [] };
            console.error('Unable to load product recommendations');
        }
    }

    get canDisplayRecommendations() {
        return this.products.length > 0;
    }

    // NavigationContext allows us to click a link to get to new Product page
    @wire(NavigationContext)
    navContext;

    handleClickProduct(event) {
        const pid = event.currentTarget.getAttribute('pid');
        const product = this.products.filter(p => p.id === pid)[0];

        trackClickReco(this.recommenderName, this.recoUUID, {
            id: product.id,
            sku: product.sku,
        });

        // go to the Product Detail Page for the product clicked
        navigate(this.navContext, {
            type: 'standard__recordPage',
            attributes: {
                objectApiName: 'Product2',
                recordName: product.fields.Name.replace(' ', '-'),
                name: 'recordId',
                recordId: pid,
            },
        });
    }

    sendViewReco() {
        trackViewReco(
            this.recommenderName,
            this.recoUUID,
            this.products.map((product) => {
                return {
                    id: product.id,
                    sku: product.sku,
                };
            })
        );
    }
}
```

2. Update the component's HTML file (for example, `force-app/main/default/<Component Name>.html`).

```
<template>
    <section class="component-wrapper" if:true={canDisplayRecommendations}>
        <h1>Product Recommendations</h1>
        <table>
            <tr>
                <th>Image</th>
                <th>Name</th>
                <th>List Price</th>
                <th>Unit Price</th>
            </tr>
            <template for:each={products} for:item="product">
                <tr key={product.id}>
                    <td><img src={product.defaultImage.url}></img></td>
                    <td>
                        <a key={product.id} class="product-line"
onclick={handleClickProduct} pid={product.id}>
                            {product.fields.Name}
                        </a>
                    </td>
                  <td>{product.prices.listPrice} {product.prices.currencyIsoCode}</td>

                  <td>{product.prices.unitPrice} {product.prices.currencyIsoCode}</td>

                </tr>
            </template>
        </table>
    </section>
</template>
```

3. Update the metadata in the component's XML file (for example, `force-app/main/default/<Component Name>.js-meta.xml`).

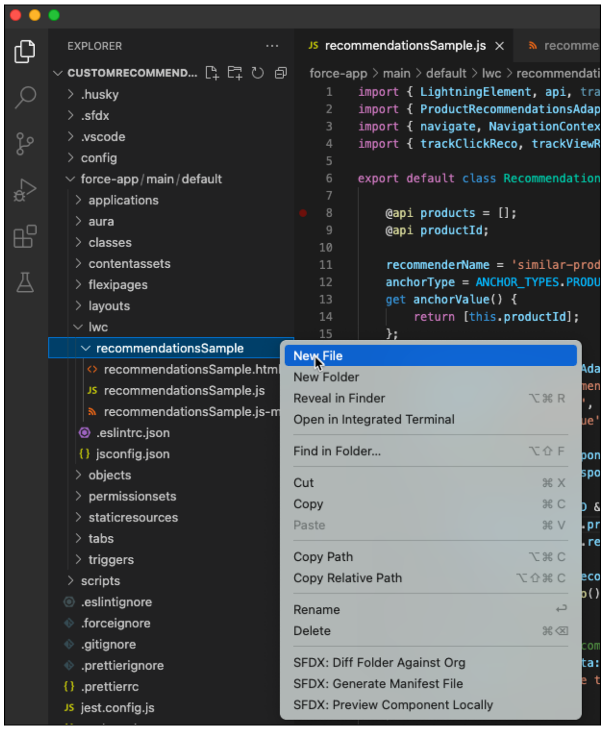
Note: Ensure that the `<apiVersion>` tag references the latest API version.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightningCommunity__Page</target>
        <target>lightningCommunity__Default</target>
    </targets>

    <targetConfigs>
        <targetConfig targets="lightningCommunity__Default">
            <property label="Product Id" name="productId" type="String"
default="{!recordId}" />
```

```
        </targetConfig>
    </targetConfigs>
</LightningComponentBundle>
```

4. In your lwc directory, create a CSS file with the same name as your project. In this example, the CSS file is named `recommendationsSample.css`.



```
import { LightningElement, api, tra
import { ProductRecommendationsAdap
import { navigate, NavigationContex
import { trackClickReco, trackViewR

export default class Recommendation

    @api products = [];
    @api productId;

    recommenderName = 'similar-prod
    anchorType = ANCHOR_TYPES.PRODU
    get anchorValue() {
        return [this.productId];
    };
```

5. In the CSS file, add custom CSS to style your component. The sample code provided includes modifications to the component's image size and headings.

```
h1 {
    font-size: 1.5em;
    font-weight: 800;
}
img {
    width: 150px;
}
th {
    font-weight: 800;
}
```

6. After you create the custom Einstein Recommendations component, deploy the component from Visual Studio Code and place it in your store with Experience Builder.

7. Before publishing your site from Experience Builder, click **Preview** to see how the custom component looks in a desktop browser window or on a mobile device.

SEE ALSO:

Commerce Einstein APIs

Prepare Commerce Einstein to Use Custom Components

Commerce Einstein Activity Tracking API

B2C Commerce Einstein Product Recommendations API

Customize Sites with Experience Builder

Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs

EDITIONS

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

Available in: B2B Commerce

Creating a custom B2B Commerce Einstein recommendations component includes implementing the Activity Tracking API and connecting the component to the Commerce Einstein Webstore Recommendations Connect API resource. This process is for B2B stores on the Aura platform.

Using Connect APIs ensures appropriate product entitlement filtering before recommendations are delivered.

Make sure to reindex after you change associations between buyer groups and products, such as when changing entitlement policies or product-to-buyer group assignments.

Your component must incorporate calls to the Activity Tracking API. When configuring the API:

- Implement both the viewReco and clickReco activities.
- If you replace the Product Detail Purchase Options component with a custom component, implement the viewProduct activity.
- If you add custom add-to-cart functionality, implement the addToCart activity.

You can use custom JavaScript, HTML, and CSS to make front-end modifications. For more information, see the Lightning Aura Components Developer Guide.

1. Create a custom Apex controller.
    a. Go to **Setup** > **Developer Console**.
    b. Select **File** > **New** > **Apex Class**.
    c. Enter a name for the Apex class.

       Make sure to match the component class name for easier identification. For example, `recsController`.

       The controller file (in this case, `recsController.apxc`) opens to show an empty class.

2. Update the controller file, making sure to modify the *`orgDomain`* and *`webstoreId`* values to match your storefront domain and ID.

```
public with sharing class recsController {
     @AuraEnabled
    public static String getRecs(String recommender, String anchorValues, String cookie)
  {
          String orgDomain = 'alpinecommerce236.my.salesforce.com';
          String webstoreId = '0ZEB0000000HMNGOA4';
         String endpoint = 'https://'+orgDomain+'/services/data/v55.0/commerce/webstores/
'+webstoreId+'/ai/recommendations?language=en-US&asGuest=true&recommender='+recommender;

        if (anchorValues.length() > 0) {
            endpoint += '&anchorValues='+anchorValues;
        }

        HttpRequest req = new HttpRequest();
        req.setEndpoint(endpoint);
        req.setHeader('Cookie', cookie);
        req.setMethod('GET');
        req.setHeader('Authorization', 'OAuth ' + UserInfo.getSessionId());

        Http http = new Http();
        HTTPResponse res = http.send(req);
        return res.getBody();
    }
}
```

3. Update the controller file with the appropriate profile for buyers.
   a. Go to **Setup** > **Custom Code** > **Apex Classes**.
   b. Next to the controller file that you created, click **Security**.
   c. Add the Shopper profile to the enabled profiles column.
   d. Click **Save**.
4. Create an SFDX project in Visual Studio Code and run the *`SFDX: Create Aura Component`* command.
5. Specify a name for the component.
6. Update the component file with the attribute configuration (for example, `force-app/main/default/aura/<project_name>/<component_name>.cmp`).

   Make sure to use the controller filename that you used when creating the Apex controller, and provide the tag needed for importing activity tracking.

```
<aura:component implements="forceCommunity:availableForAllPageTypes"
controller="RecsController" access="global">

    <aura:attribute name="title" type="String" default="" required="true" />
    <aura:attribute name="recommender" type="String" default="RecentlyViewed"
required="true" />
    <aura:attribute name="anchorValues" type="String" default="" required="false" />
    <aura:attribute name="uuid" type="String" default=""/>
    <aura:attribute name="loading" type="boolean" default="false"/>
    <aura:attribute name="showProducts" type="boolean" default="false"/>
    <aura:attribute name="products" type="List" default="[]" />

    <!-- Tag needed to import Commerce Activity Tracking-->
    <commerce:activitiesApi aura:id="activitiesApi" />

    <aura:handler name="init" value="{!this}" action="{!c.onLoadComponent}"/>

    <div>
        <aura:if isTrue="{!v.loading}">LOADING...</aura:if>
        <aura:if isTrue="{!v.showProducts}">
            <div class="title">{!v.title}</div>
            <div class="products">
                <aura:iteration items="{!v.products}" var="product">
                    <div class="product">
                        <img data-pid="{!product.id}" onclick="{!c.handleClickProduct}"
 src="{!product.defaultImage.url}"/>
                        <a class="name" data-pid="{!product.id}"
onclick="{!c.handleClickProduct}">{!product.name}</a>
                    </div>
                </aura:iteration>
            </div>
        </aura:if>
    </div>

</aura:component>
```

7. Update the metadata in the component's XML file (for example, `force-app/main/default/aura/<project_name>/<component_name>.cmp-meta.xml`).

   Make sure that the *`<apiVersion>`* tag references the latest API version.

```
<?xml version="1.0" encoding="UTF-8" ?>
<AuraDefinitionBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <description>Aura Recommendations Component</description>
    <isExposed>true</isExposed>
    <targets>
        <target>lightningCommunity__Page</target>
        <target>lightningCommunity__Default</target>
    </targets>
</AuraDefinitionBundle>
```

8. Edit your component's CSS file to add custom styling (for example, `force-app/main/default/aura/<project_name>/<component_name>.css`).

   This sample code includes modifications to the component's image size and headings.

```
.THIS div.title {
    text-align: center;
    font-weight: 800;
    font-size: 2em;
}

.THIS .products {
    height: 400px;
    text-align: center;
}
.THIS .products .product  {
    width: 25%;
    height: 450px;
    padding: 10px;
    display: inline-block;
}
```

```
.THIS .products .product img  {
    cursor: pointer;
    max-height: 300px;
    width: auto;
}
.THIS .products .product .name  {
    display: block;
}
```

9. Edit your component's `Controller.js` file, making sure to modify the *`storeName`* value for the current store (for example, `force-app/main/default/aura/<project_name>/<component_name>Controller.js`).

```
({
    "onLoadComponent": function(cmp, event, helper) {
        let pageProductId = helper.getProductDetailProductId();
        if (pageProductId) {
            // Component is on a product detail page; show Similar Products recommender

            cmp.set("v.title", 'Similar Products');
            cmp.set("v.recommender", 'SimilarProducts');
            cmp.set("v.anchorValues", pageProductId);
        } else {
            // Show Recently Viewed recommender
            cmp.set("v.title", 'Recently Viewed');
            cmp.set("v.recommender", 'RecentlyViewed');
            cmp.set("v.anchorValues", '');
        }
        helper.loadProductRecommendations(cmp, event, helper);
    },

    // send a clickReco activity and navigate to the product detail page
    "handleClickProduct": function(cmp, event, helper) {
        let productId = event.currentTarget.getAttribute('data-pid');
        let trackClickReco = cmp.find('activitiesApi').trackClickReco;
        let recName = helper.recommenderNames[cmp.get("v.recommender")]
        let uuid = cmp.get('v.uuid');
        let products = cmp.get('v.products');
        let product = products.filter(p => p.id === productId)[0];
        let productToSend = {
            id: product.id,
            price: product.prices ? product.prices.listPrice : undefined,
        };
        trackClickReco(recName, uuid, productToSend);

        // navigate to the product page
        let storeName = 'AlpineB2B';
        let productName = product.name || 'detail';
        let newHref = `/${storeName}/s/product/${productName}/${productId}`;
        window.location.href = newHref;
    }
})
```

10. Edit your component's `Helper.js` file, using this example for guidance (for example, `force-app/main/default/aura/<project_name>/<component_name>Helper.js`).

```
({
    loadProductRecommendations : function(cmp,event,helper) {
        try {
            cmp.set("v.loading", true);
            var action = cmp.get("c.getRecs");
            action.setParams({
                recommender : cmp.get("v.recommender"),
                anchorValues : cmp.get("v.anchorValues"),
                cookie: document.cookie
            });
            // Create a callback that is executed after
            // the server-side action returns
            action.setCallback(this, function(response) {
                var state = response.getState();
                if (state === "SUCCESS") {
                    try {
                        let data = JSON.parse(response.getReturnValue());
                        let products = data.productPage.products;
                        // Keep it simple, only show 4 products
                        cmp.set("v.products", products.slice(0, 4));
                        cmp.set("v.uuid", data.uuid);
                        cmp.set("v.loading", false);
                        let showProducts = products.length > 0;
                        cmp.set("v.showProducts", showProducts);
                        // Only send the viewReco activity when we display the product

                        // recommendations
                        if (showProducts) {
                            helper.sendViewRecoActivity(cmp, helper);
                        }
                    } catch (err) {
                        console.error('Error fetching recommendations', err);
                        cmp.set("v.loading", false);
                    }
                }
            });
            $A.enqueueAction(action);
        } catch (error) {
            console.error('Failed to load recommendations: ', error);
            cmp.set("v.loading", false);
        }
    },
    // The recommender names we pass into the Connect API are in a different format
    // than the recommender names we pass into the activities api.
    recommenderNames: {
        "RecentlyViewed" : "recently-viewed",
        "SimilarProducts" : "similar-products",
        "MostViewedByCategory" : "most-viewed-by-category",
        "TopSelling" : "top-selling",
        "Upsell" : "upsell"
    },
    formatPrice: function(price, curr) {
        return new Intl.NumberFormat('en-US', { style: 'currency', currency:
curr}).format(price);
    },

    sendViewRecoActivity: function(cmp, helper) {
        let trackViewReco = cmp.find('activitiesApi').trackViewReco;
        let recName = helper.recommenderNames[cmp.get("v.recommender")]
        let products = cmp.get('v.products').map(p => ({id: p.id}));
        let uuid = cmp.get("v.uuid");
        trackViewReco(recName, uuid, products);
    },

    getProductDetailProductId: function() {
        let pageProductIdMatch = window.location.href.match(new
RegExp('01t[a-zA-Z0-9]{15}'));
        let pid = pageProductIdMatch ? pageProductIdMatch[0] : null;
        return pid;
    }
})
```

11. After you create the custom Einstein Recommendations component, deploy it from Visual Studio Code and place it in your store with Experience Builder.
12. Before publishing your site from Experience Builder, click **Preview** to see how the custom component looks in a desktop browser window and on a mobile device.

SEE ALSO:

Commerce Einstein APIs

Prepare Commerce Einstein to Use Custom Components

Commerce Einstein Activity Tracking API

B2B Commerce Einstein Product Recommendations API

Tracked Data

Commerce Einstein Webstore Recommendations

Commerce Einstein Recommendations API Reference

Modifying Einstein Recommendations interfaces in your D2C and B2B Commerce stores requires using the Activity Tracking API and a Product Recommendation API. These APIs provide access to the underlying data behind Einstein Recommendations and enable modification of the front-end experience with custom Lightning web components.

Commerce Einstein Activity Tracking API

Commerce Einstein product recommendations require activity tracking data. For example, tracking the viewProduct activity provides insight into top-viewed products. The Activity Tracking API provides access to these activities for use in recommendations.

B2C Commerce Einstein Product Recommendations API

Calls to the Product Recommendation API deliver Einstein recommendations. The API is accessed using a wire adapter to provide data to a Lightning web component. In this case, the Product Recommendation API returns a set of products to populate the Einstein Recommendations component.

B2B Commerce Einstein Product Recommendations API
Calls to the Product Recommendation API deliver Einstein recommendations. The API is accessed using a Connect API to provide data to a Lightning web component. The Connect API returns a set of products to populate the Commerce Einstein Recommb2b_b2c_comm_einstein_reco_api_refendations component.

SEE ALSO:

Commerce Einstein Activity Tracking API

B2C Commerce Einstein Product Recommendations API

B2B Commerce Einstein Product Recommendations API

Commerce Einstein Activity Tracking API

Commerce Einstein product recommendations require activity tracking data. For example, tracking the viewProduct activity provides insight into top-viewed products. The Activity Tracking API provides access to these activities for use in recommendations.

The Commerce Cloud Einstein Recommendation Validator extension helps with preliminary validation when integrating Commerce Einstein Activity Tracking on Salesforce production instances. This extension is available only for the Google Chrome browser. You can search for and install the Commerce Cloud Recommendation Validator extension from the Chrome Web Store.

EDITIONS

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

Available in: B2B Commerce and D2C Commerce

View Product Activity

Trigger the viewProduct activity when a shopper views a Product Detail page. Don't fire if a product is displayed via a recommendation, search result, or any other means.

Note: If you replace the Product Detail Purchase Options component with a custom component, implement the viewProduct activity to ensure that Commerce Einstein Recommendations use cases generate results based on shopper or buyer view behavior.

Parameters

- product—The product that the customer viewed. An object with an 18-character product ID.
- sku—(Optional) A unique stock keeping unit identifier for the product.

Example usage

```
trackViewProduct({ id: '01t000000000000001', sku: 'sku123' });
```

View Recommendations Activity

Trigger the viewReco activity when a recommendation is displayed to the customer. Implement this activity when building a custom Commerce Einstein recommendations component.

Note: If you calculate a recommendation but don't show it to the customer—for example, it doesn't have as many results as you like—don't fire this activity.

Parameters

- recommenderName—The name of the recommender.
- recoUUID—A string representing the unique ID for this recommendation response.
- products—The products displayed to the customer. A list of one or more 18-character product IDs.
- sku—(Optional) A unique stock keeping unit identifier for the product.

Example usage

```
trackViewReco(
 'similar-products',
 '123-456',
 [{ id: '01t000000000000001', sku: 'sku123' }, { id: '01t000000000000002', sku: 'sku456'
}]
);
```

Click Recommendations Activity

Trigger the clickReco activity when a recommended product is clicked and the customer is taken to the product detail page. Implement this activity when building a custom Commerce Einstein recommendations component.

Parameters

- recommenderName—The name of the recommender.
- recoUUID—A string representing the unique ID for this recommendation response.
- product—The product that the customer clicked. An object with an 18-character product ID.
- sku—(Optional) A unique stock keeping unit identifier for the product.

Example usage

```
trackClickReco('similar-products', '123-456', { id: '01t000000000000001', sku: 'sku123'
});
```

Add To Cart Activity

Trigger the addToCart activity when a shopper adds a product to the cart.

Note: If you replace the Product Detail Purchase Options component with a custom component, implement the addToCart activity to ensure that Commerce Einstein Recommendations use cases generate results based on shopper or buyer view behavior.

Parameters

- product—The product that the customer adds to cart. An object with an 18-character product ID.
- sku—(Optional) A unique stock keeping unit identifier for the product.
- quantity—(Optional) The total number of this item in the cart.
- price—(Optional) The price of each individual unit of this product.
- originalPrice—(Optional) The original price of each individual unit of this product.

Example usage

```
trackAddProductToCart({ id: '01t000000000000001', sku: 'sku123', quantity: 2, price: '9.99',
 originalPrice: '10.99' });
```

SEE ALSO:

Commerce Einstein APIs

Prepare Commerce Einstein to Use Custom Components

Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs

Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs

B2C Commerce Einstein Product Recommendations API

EDITIONS

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

Available in: D2C Commerce

Calls to the Product Recommendation API deliver Einstein recommendations. The API is accessed using a wire adapter to provide data to a Lightning web component. In this case, the Product Recommendation API returns a set of products to populate the Einstein Recommendations component.

ProductRecommendationsAdapter Wire Adapter

The ProductRecommendationsAdapter returns Einstein product recommendations. Implement this wire adapter by importing it from the `einsteinAPI` module within the commerce namespace (`commerce/einsteinAPI`) and then declaring its required parameters with the `@wire` decorator.

Parameters

- Recommender Name—The name of the recommender. This parameter determines what kind of product results you see. It must be one of the recommender names listed in the Recommender Names and Anchors table.
- Anchor Type—The anchor type is either PRODUCT, CATEGORY, or NO_CONTEXT. A product recommendation with an anchor type PRODUCT bases its results on the product IDs that you pass.
- Anchor Value—If the anchor type is PRODUCT, the value is a list of 18-character product IDs. If the anchor type is CATEGORY, the value is a list of 18-character category IDs. If the anchor type is NO_CONTEXT, there’s no anchor value; it can be null or undefined.

Result

- products—A list of product details.
- recoUUID—A string representing the unique ID for the recommendation response. Use this value when you trigger the viewReco and clickReco activities.

Constants

- ANCHOR_TYPE.PRODUCT
- ANCHOR_TYPE.CATEGORY
- ANCHOR_TYPE.NO_CONTEXT

Recommender Names and Anchors

Each recommender name requires a specific anchor type and anchor value. For example, if you use the recommender name “similar-products,” the anchor type must be PRODUCT, and the anchor value must be a list of product IDs. This table shows the anchor type and value associated with each recommender name.

Anchor Type	Anchor Value	Recommender Names
NO_CONTEXT	Any value that evaluates to false. For example: null or undefined.	• personalized-for-shopper • recently-viewed • top-selling
PRODUCT	A list of one or more 18-character product IDs. For example: `['01t000000000000001', '01t000000000000002', '01t000000000000003']`.	• complementary-products • customers-who-bought-also-bought • recently-viewed • similar-products • upsell

Anchor Type	Anchor Value	Recommender Names
CATEGORY	A list of one or more 18-character category IDs. For example: `['0ZG000000000000001', '0ZG000000000000002', '0ZG000000000000003']`.	• most-viewed-by-category • recently-viewed • top-selling-by-category

SEE ALSO:

Einstein Recommendations Component (LWR) for B2C Stores

Commerce Einstein APIs

Prepare Commerce Einstein to Use Custom Components

Create a Custom Recommendations Component for B2C Stores Using Commerce Einstein APIs

B2B Commerce Einstein Product Recommendations API

EDITIONS

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

Calls to the Product Recommendation API deliver Einstein recommendations. The API is accessed using a Connect API to provide data to a Lightning web component. The Connect API returns a set of products to populate the Commerce Einstein Recommb2b_b2c_comm_einstein_reco_api_refendations component.

Commerce Einstein Webstore Recommendations Resource

EDITIONS

Available in: Available in: B2B Commerce

The Commerce Einstein Recommendations Connect API resource (`/commerce/webstores/`***`webstoreId`***`/ai/recommendations`) returns Einstein product recommendations.

Recommender Names and Anchors

You can add one or more Einstein Recommendations components to any page in B2B Commerce stores. When using a product recommender, such as `SimilarProducts` or `ComplementaryProducts`, specify product IDs. When using a category recommender, such as `MostViewedByCategory` or `TopSellingByCategory`, specify category IDs. All other anchors are supported on any page, but they're intended for use on certain pages.

B2B recommenders don't require that you specify an anchor type. The Connect API automatically determines the anchor type based on the anchor values and the recommendation use case that you specify.

Anchor Type	Anchor Value	Fields Returned
No anchor	Any value that evaluates to false. For example: null or undefined.	• RecentlyViewed • TopSelling • PersonalizedForShopper
Product ID	A list of one or more 18-character product IDs. For example: `01t000000000000001, 01t000000000000002, 01t000000000000003`	• RecentlyViewed • SimilarProducts • ComplementaryProducts • CustomersWhoBoughtAlsoBought

Anchor Type	Anchor Value	Fields Returned
		• Upsell
Category ID	A list of one or more 18-character category IDs. For example: `0ZG000000000000001`, `0ZG000000000000002`, `0ZG000000000000003`	• RecentlyViewed • MostViewedByCategory • TopSellingByCategory

For example, a URL containing an anchor value for the connect API looks like this.

```
/services/data/v55.0/commerce/webstores/0ZERM00000009eR/ai/recommendations?recommender=SimilarProducts&anchorValues=01tRM000000R8DnYAK,01tRM000000R8DWYA0
```

SEE ALSO:

- Commerce Einstein APIs
- Prepare Commerce Einstein to Use Custom Components
- Create a Custom Recommendations Component for B2B Stores Using Commerce Einstein APIs
- Commerce Einstein Webstore Recommendations

Create a Custom Checkout Component for a B2B or B2C Store (LWR)

You can create custom checkout components to extend the default checkout processing for a B2B or B2C store created with an LWR template.

Checkout Component Hierarchy
The checkout page for a B2B or B2C store created with an LWR template contains a recommended hierarchy of nested checkout components. These components rely on a checkout data provider that loads and saves all of the necessary data for checkout.

UseCheckoutComponent Interface
On the checkout page for a B2B or B2C store created with an LWR template, child checkout components implement the `useCheckoutComponent` mixin interface.

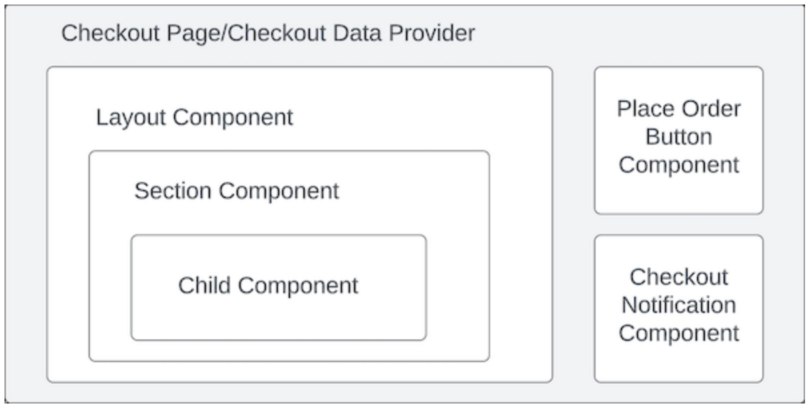
Checkout Component Communication
Checkout components communicate with each other during checkout processing. Understanding how they communicate helps you develop a custom checkout component for a B2B or B2C store created with an LWR template.

Sample Custom Checkout Component
The Terms and Conditions sample component shows how to create a custom checkout component for a B2B or B2C store created with an LWR template. You can add a Terms and Conditions component to any section of your Checkout page, but we recommend placing it after the Payment component.

Checkout Component Hierarchy

The checkout page for a B2B or B2C store created with an LWR template contains a recommended hierarchy of nested checkout components. These components rely on a checkout data provider that loads and saves all of the necessary data for checkout.

The checkout component hierarchy consists of three levels. At the top of the component hierarchy is a layout component. The layout component contains multiple section components, and it organizes the customer experience into a one-page workflow or an accordion workflow. The Layout: One Page component shows all sections at once, and the customer can complete them in any order. The Layout: Accordion layout expands one section at a time, and the customer must complete them in a specific order.


Checkout Page/Checkout Data Provider
Layout Component
Section Component
Child Component
Place Order Button Component
Checkout Notification Component

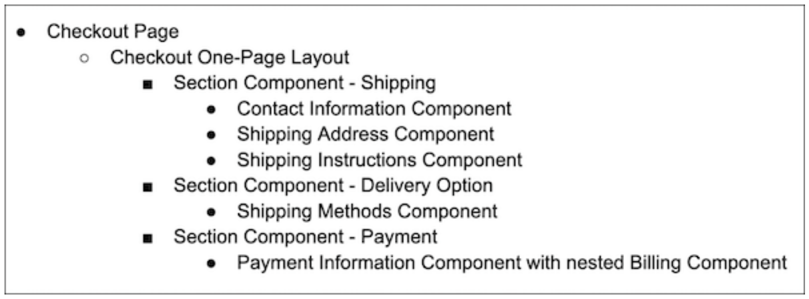
Section components group related child components. For example, a Shipping section can contain a Shipping Address component and a Shipping Instructions component. You can add, remove, reorder, or rename section components as needed.

Child components show and accept the customer details that are required to complete checkout. These components apply the mixin `extend useCheckoutComponent(LightningElement)` and implement certain checkout methods, depending on the desired behavior, to be managed by the page, layout, and section components.

Some components on the Checkout Page, such as the Place Order Button component and Checkout Notification component, are placed directly on the page and aren’t part of this hierarchy.

Default Checkout Components

When you create a new store in Experience Builder, we provide a preconfigured Checkout Page by default that follows this setup.



- Checkout Page
  - Checkout One-Page Layout
    - Section Component - Shipping
      - Contact Information Component
      - Shipping Address Component
      - Shipping Instructions Component
    - Section Component - Delivery Option
      - Shipping Methods Component
    - Section Component - Payment
      - Payment Information Component with nested Billing Component

You can add custom components anywhere in the workflow and expect the same or different behavior as the default components, depending on the implementation.

UseCheckoutComponent Interface

On the checkout page for a B2B or B2C store created with an LWR template, child checkout components implement the `useCheckoutComponent` mixin interface.

The component API is used to create custom components that integrate with the checkout process of form validation and the synchronization of external API validations.

```
/**
 * Applies mixin for base class for any checkout dedicated building block.
 *
 * example:
```

```
 * export default class MyCheckoutInput extends useCheckoutComponent(LightningElement) {
 *     setAspect(newAspect: CheckoutContainerAspect): void {
 *         console.log(`dbb newAspect`, JSON.stringify(newAspect));
 *     }
 *     private handleButton(): void {
 *         this.dispatchCommit();
 *     }
 * }
 */
 export function useCheckoutComponent(
    superclass: Constructor<LightningElement>
): Constructor<LightningElement & CheckoutComponent>;
```

This component API adds these helper methods to a LightningElement.

```

/**
 * interface implemented by CheckoutComponentMixin
 * and required to exist for CheckoutContainerMixin
 */
export interface CheckoutComponent extends CheckoutComponentHandlers {
    /**
     * notify the container the child component has modified data to put in the form store

     * the container should start 'dirty form' processing:
     * - optionally stageAction on this and other components
     * - save form store data to the server
     * - report errors
     */
    dispatchCommit(): void;
    /**
     * notify the DataProvider to update the form with the supplied changes.
     * the DataProvider catches updateForm errors so components do not need to.
     * unlike most dispatch functions this one is awaitable.
     */
    dispatchUpdateAsync(formRequest: CheckoutFormRequest): Promise<void>;
    /**
     * specialized version of dispatchUpdateAsync for setting client side errors.
     * note: this will never reject or throw, so it's safe not to await if circumstances
permit
     */
    dispatchUpdateErrorAsync(errorRequest: CheckoutException): Promise<void>;
    /**
    * notify the DataProvider to start 'final' processing (the place order button pressed)

     * returns a Promise to facilitate advanced payment integrations.
     */
    dispatchFinalizeAsync(): Promise<void>;
    /**
     * notify the DataProvider to call place order API
     * unlike most dispatch fns this one is awaitable.
     */
    dispatchPlaceOrderAsync(): Promise<OrderConfirmation>;
    /**
     * ask our container to change our display such as put us in summary mode
     */
    dispatchRequestAspect(desiredAspect: CheckoutContainerAspectRequest): void;
}
```

The component API also adds these default implementations for container-initiated actions. The component designer can override these implementations as needed.

```
/**
 * ComponentRegistration delegates CheckoutContainerSubscriptionPayload to these handlers
 in the derived component implementation
 */
export interface CheckoutComponentHandlers {
    /**
     * called on connect. derived classes should expose this key the DOM
     * if DOM ordering of container children is needed.
     * container classes combine a DOM query with sortSubscribers to break
     * through the otherwise opaque CheckoutComponentReference.
     * e.g. this.setAttribute('data-checkout-domkey', suggestedDomKey)
     */
    setDomKey(suggestedDomKey: string): string;
    /**
     * display hints from container such as summary mode, stencil, etc.
     * @param newAspect update checkout mode and stage
     */
    setAspect(newAspect: CheckoutContainerAspect): void;
    /**
     * derived class must implement REPORT_VALIDITY_SAVE calls reportValidity if defined.

     * derived class must implement CHECK_VALIDITY_UPDATE or do equivalent before
dispatchCommit.
     * unsummarize if an error encountered to ensure it is seen
     *
     * aside: this subsumes all of CheckoutSavable's reportValidity, checkValidity,
checkoutSave, placeOrder
     *
    * @param _checkoutStage used to synchronize processing and responses across components

     * @returns a Promise that resolves false if processing should be blocked
     *          should always return true if uninterested in a stage.
     */
    stageAction(checkoutStage: CheckoutStage): Promise<boolean>;
```

Checkout Stages

These are the currently defined checkout stages.

```
/**
 * where in the linear steps of checkout process, affects stageAction/reportValidity
responses
 * note: these are unordered, stageAction should only use equality checks.
 */
export enum CheckoutStage {
    // commit (user edit) stages that lead to form save (checkout API update)
    //
    // CHECK_VALIDITY_UPDATE may skip update if component updates itself on each change
before dispatchCommit
    CHECK_VALIDITY_UPDATE = 'CHECK_VALIDITY_UPDATE',
    REPORT_VALIDITY_SAVE = 'REPORT_VALIDITY_SAVE',
    // finalize stages that lead to place order
    BEFORE_PAYMENT = 'BEFORE_PAYMENT',
    PAYMENT = 'PAYMENT',
    BEFORE_PLACE_ORDER = 'BEFORE_PLACE_ORDER',
    PLACE_ORDER = 'PLACE_ORDER',
}
```

Aspect Type Definitions

These aspect type definitions are referenced by the helper methods.

```
/**
 * Used for nested checkout containers to specify how they are displayed
 */
export type CheckoutContainerAspect = {
    /**
     * truthy indicates DP is initializing, preparing for the place order step,
     * or reached an unrecoverable error.
     * input controls should render as readonly
     *
     * WARNING! when disabled or readonly lightning-input.reportValidity always
     *          returns true; therefore, defer setting controls to read-only
     *          unless they pass checkValidity.
     *          Otherwise they break stageAction(REPORT_VALIDITY_SAVE)
     */
    readOnlyIfValid: boolean;
    /**
     * truthy indicates expandable child sections should show as collapsed
     * typically this indicates the section is in a future accordion step, or in
     * a past or future subway step.
     */
    collapse: boolean;
    /**
     * truthy indicates summarizable components should render as summarized
     *
     * component may ignore summary: true requests, and if needed respond
     * with dispatchRequestAspect(false) to ask their containers to become unsummarized;
     * useful because errors are not typically rendered nor fixable in
     * summarized components.
     */
    summary: boolean;
};
/**
 * Used by components to ask their container to change how they are displayed
 */
export type CheckoutContainerAspectRequest = {
    /**
     * true container should enter summary mode, false should leave it
     * Children inform containers they can be summarized.
     */
    summarizable: boolean;
    /**
     * if truthy there are no options and summarized container can hide edit button
     */
    uneditable?: boolean;
};
```

Checkout Data Provider with Form Data

The checkout data provider is used to access checkout session API data and local, not-yet-persisted form changes. The data provider `Checkout.Details` publishes an object of type `CheckoutFormOverlay`.

```
/**
 * holds current state of checkout from the checkout api overlayed with
 * with unpersisted client side changes and related meta-data.
 */
export type CheckoutFormOverlay = CheckoutInformation & {
    /**
     * captured billing details
     */
    billingInfo?: CheckoutBillingInfo;
    /**
     * captured client side exceptions
     */
    notifications?: FormNotification[];
    /**
     * computed meta information about the overlay
     */
    formStatus?: CheckoutFormStatus;
};
/**
 * billing information that is not necessarily represented
 * in the checkout session
 */
export type CheckoutBillingInfo = {
    address?: Address;
    email?: string;
};
/**
 * computed meta information about the form like is
 * there data that should be saved
 */
export type CheckoutFormStatus = CheckoutFormActivity & {
    /**
     * true if persistForm should be called
     * does not  account for incongruent data, billing info, etc.
     * cleared by persistForm (sometimes) and revertForm
     */
    dirty: boolean;
    /**
     * true if some data typically saved by persistForm is incomplete
     * for example only some of the fields required to save
     * contactInfo have been set.
     * promoted (cleared) to dirty (set) when updateform gets missing data.
     * cleared by revertForm
     */
    incongruent: boolean;
    /**
     * indicates if billingInfo.address explicitly set
     */
    useShippingAddressForBilling: boolean;
};
/**
 * used to add and remove client side notifications returned in the overlay.
 * note: overlay's server side errors are not affected by adding or removing these.
 * note: new client side notifications are appended to the overlay array
 */
export type FormNotification = {
    /**
     * unique ID used to clear all client notifications added in previous requests that
     * had the same groupId.
     */
    groupId: string;
    /**
     * pass a unique type that can control where in the UI the exception renders.
     */
    type?: RequestErrorType;
    /**
     * the l10n string to display in the UI as the notification body.
     * when detail falsey clear all client side set exceptions of this groupId.
     */
    detail?: string;
    /**
     * optional unique error class identifier which can be used to further customize
     * the notification.
     *
     * Examples are the Error.name (PaymentAuthorizationError)
     * or Error.message (CheckoutError.NO_DELIVERY_ADDRESSES)
     */
    code?: string;
};
```

Updating Checkout Form Data

The checkout data provider published data is updated using the provided component API `dispatchUpdateAsync`. It accepts an object of type `CheckoutFormRequest` with the list of fields to update.

```
/**
 * used by components to update the unpersisted client side data.
 * assumes all contents are "validated" with checkValidity
 */
export type CheckoutFormRequest = {
    /**
     * caller short hand for deliveryGroups.items[0]
     */
    defaultDeliveryGroup?: CheckoutFormRequestDeliveryGroup;
    contactInfo?: ContactInfo;
    billingInfo?: {
        /**
         * in FormRequest send null to revert to useShippingAddressForBilling
         */
        address?: Address | null;
        email?: string;
    };
    notifications?: FormNotification[];
};
/**
 * selectable fields of a DeliveryGroup
 */
export type CheckoutFormRequestDeliveryGroup = {
    /**
     * notice changing the deliveryAddress results in all existing
     * availableDeliveryMethods and any selectedDeliveryMethod (even
     * an explictly set one) to be removed from the overlay until the
     * new deliveryAddress is saved.
     */
    deliveryAddress?: Address;
    desiredDeliveryDate?: string;
    shippingInstructions?: string;
    /**
     * in FormRequest send null to revert to existing selection
     *
     * notice an invalid (or no longer valid) delivery method selection
     * is ignored (the xisting selection (if any) shows in the overlay
     *
     * notice that by default changing the deliveryAddress will not
     * clear any explicitly set selectedDeliveryMethodId.  if a caller
     * wants to reset to the default (cheapest) delivery method when
     * changing addresses they must also explicitly clear any previous
     * selectedDeliveryMethodId setting.
     */
    selectedDeliveryMethodId?: string | null;
};
```

Checkout Component Communication

Checkout components communicate with each other during checkout processing. Understanding how they communicate helps you develop a custom checkout component for a B2B or B2C store created with an LWR template.

Checkout Layout Options

There are two checkout layout options: one-page and accordion. The layout option that you choose affects how the data provider and sections interact with your custom child components.

In the one-page layout, the shopper can generally complete checkout sections in any order. All checkout components are kept in edit mode, so the shopper can edit each section. In Experience Builder, you can reorder the sections and components.

In the accordion layout, the shopper completes the sections in a prescribed order from top to bottom: Shipping Address, Shipping Method, and Payment. The default minimum set of sections and components must be kept in this order. Each section in this layout has a Proceed (or Next) button.

In both layouts, an authenticated customer must provide or confirm their shipping address, shipping method, payment, and billing information. A guest customer must also enter an email address and a phone number as part of the shipping address. To place the order, the customer clicks **Place Order**.

One-Page Layout Processing Flow

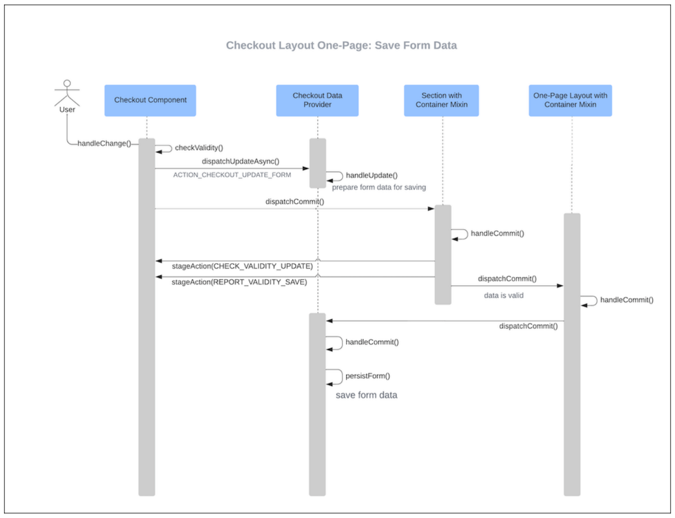
In the one-page layout, shoppers can edit sections in any order. Editing autosaves on the shipping component, which triggers the shipping methods to load. The checkout place order operation triggered by the Place Order button succeeds when all required sections and forms are completed and they report as valid. The address and shipping methods autoload for a returned authenticated shopper, and only payment information must be entered.

Form Updates and Autosave

One-page layout sections are always expanded and editable. After a customer enters valid information into a component's form fields, the component typically calls two methods, `dispatchUpdateAsync()` and `dispatchCommit()`, to notify the checkout data provider or parent section containers that they're ready to be saved.

The `dispatchUpdateAsync()` method notifies the checkout page data provider to update the internal form store with the supplied changes. To avoid updating the internal form store with invalid data, the component typically calls the `checkValidity()` method first. The `dispatchCommit()` method informs the section to attempt to save the new internal form store changes. The component `stageAction()` method handles the `CHECKOUT_VALIDITY_UPDATE` and REPORT_VALIDITY_SAVE actions. When the component `checkValidity()` and `reportValidity()` methods return true, the subsequent `dispatchCommit()` calls are made through the intermediary one-page layout container component. Finally, the checkout data provider handles the commit and saves the checkout data.

This diagram shows the sequence initiated when a customer interacts with a component form.



One-Page Place Order Sequence

In one-page checkout, after the user clicks the place order button, the button calls the helper `dispatchFinalizeAsync` to start the validation cycle. The checkout page data provider executes `reportValidity` on each child component to determine if all forms are valid and ready. This validity check is complete when components implement the `stageAction()` method with `CHECK_VALIDITY_UPDATE` and `REPORT_VALIDITY_SAVE` handlers. See Sample Custom Checkout Component on page 78.

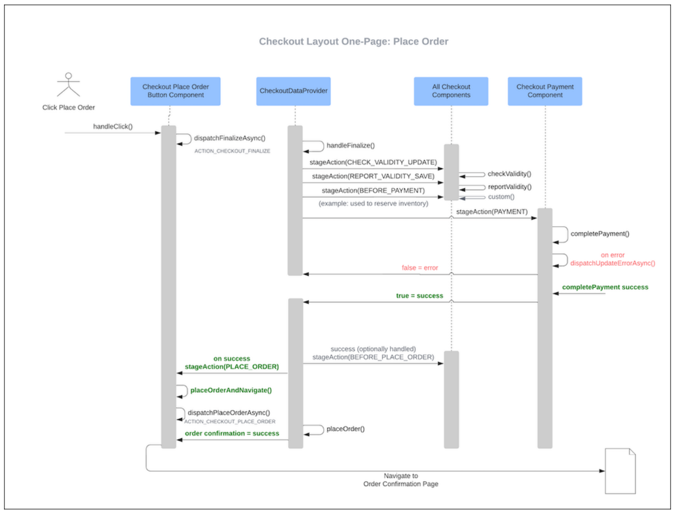
If any form fields are invalid, errors are shown and the place order operation is halted. The customer can then provide missing information or correct mistakes.

If all initial `stageAction(...)` calls report valid and return true, the `ACTION_CHECKOUT_FINALIZE` operation continues with `stageAction(PAYMENT)` and calls `completePayment()` on the Payment component.

If `completePayment()` succeeds and returns true from the `stageAction(PAYMENT)` call, the checkout data provider continues to process calls `stageAction(BEFORE_PLACE_ORDER)` and `stageAction(PLACE_ORDER)`. When `stageAction(PLACE_ORDER)` is called, the component uses `dispatchPlaceOrderAsync` to make the server API call.

Finally, the Place Order Button component handles the `` `PLACE_ORDER` `` option and dispatches the `ACTION_CHEKOUT_PLACE_ORDER` action. This component also navigates to the order confirmation summary. Custom components can also implement the `stageAction(checkoutStage:CheckoutStage)` method and handle `BEFORE_PAYMENT` and `BEFORE_PLACE_ORDER` for any custom processing needed at those stages.

This diagram shows the sequence initiated when the customer clicks Place Order.


Checkout Layout One-Page: Place Order
Click Place Order
Checkout Place Order Button Component
CheckoutDataProvider
All Checkout Components
Checkout Payment Component
handleClick()
dispatchFinalizeAsync()
ACTION_CHECKOUT_FINALIZE
handleFinalize()
stageAction(CHECK_VALIDITY_UPDATE)
stageAction(REPORT_VALIDITY_SAVE)
stageAction(BEFORE_PAYMENT)
(example: used to reserve inventory)
checkValidity()
reportValidity()
custom()
stageAction(PAYMENT)
completePayment()
on error
dispatchUpdateErrorAsync()
false = error
completePayment success
true = success
success (optionally handled)
stageAction(BEFORE_PLACE_ORDER)
on success
stageAction(PLACE_ORDER)
placeOrderAndNavigate()
dispatchPlaceOrderAsync()
ACTION_CHECKOUT_PLACE_ORDER
placeOrder()
order confirmation = success
Navigate to
Order Confirmation Page

Accordion Layout Processing Flow

The shopper clicks **Proceed** on each section to go to the next section.

By default, the accordion layout shows these sections in order: Guest Information (for unauthenticated shoppers), Shipping Address, Shipping Methods, and Payment and Billing. You can add custom sections and components, but you must keep the default sections for checkout to work.

Error Handling

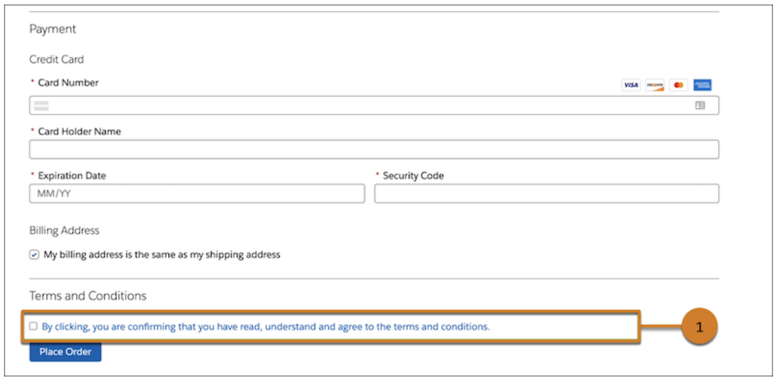
Error handling behaves similarly for both layout options. Whenever an error is caught, a `dispatchUpdateErrorAsync()` call can be made. For example, in the Place Order diagram, a `dispatchUpdateErrorAsync()` call is made on the Payment component `completePayment()` method. The checkout page data provider then updates the Checkout Notification component to display the error. The checkout process is halted until the shopper corrects any issues and clicks **Place Order** or **Proceed** again. See FormNotification details in Checkout Interfaces on page 70.

Sample Custom Checkout Component

The Terms and Conditions sample component shows how to create a custom checkout component for a B2B or B2C store created with an LWR template. You can add a Terms and Conditions component to any section of your Checkout page, but we recommend placing it after the Payment component.

The Terms and Conditions component presents a checkbox to the customer. If the customer accepts the terms and conditions by selecting the box, they can continue the checkout process. Otherwise, the component blocks the checkout process. The checkbox text links to a page that describes your store's terms and conditions, which you provide.

This example shows a Terms and Conditions component (1) placed in its own Terms and Conditions section after the Payment section.


Payment
Credit Card
* Card Number
* Card Holder Name
* Expiration Date
MM/YY
* Security Code
Billing Address
My billing address is the same as my shipping address
Terms and Conditions
By clicking, you are confirming that you have read, understand and agree to the terms and conditions.
1
Place Order

You need four files to implement the Terms and Conditions component.

The first file is the LWC template: `terms.html`

```
<template>
    <div>
        <input
            type="checkbox"
            id="termsandconditions"
            name="terms"
            checked={checked}
            onchange={handleChange}
        ></input>
        <label class="slds-p-left_x-small" for="termsandconditions">
            <lightning-formatted-rich-text
                value={disclaimerLink}
            ></lightning-formatted-rich-text>
        </label>
    </div>
    <label if:true={showError} class="slds-text-color_error">
        {error}
    </label>
</template>
```

The second file is the JavaScript component: `terms.js`:

```
import { LightningElement, api } from 'lwc';

import { useCheckoutComponent } from 'commerce/checkoutApi';

const CheckoutStage = {
    CHECK_VALIDITY_UPDATE: 'CHECK_VALIDITY_UPDATE',
    REPORT_VALIDITY_SAVE: 'REPORT_VALIDITY_SAVE',
    BEFORE_PAYMENT: 'BEFORE_PAYMENT',
    PAYMENT: 'PAYMENT',
    BEFORE_PLACE_ORDER: 'BEFORE_PLACE_ORDER',
    PLACE_ORDER: 'PLACE_ORDER'
```

```
};

/**
 * Terms and Conditions has a link to the terms and conditions for the
 * checkout user to read and a checkbox to accept the terms. Place order
 * should be blocked by this component when placed in the payment step
 * before or after the payment component.
 *
 * One page layout: this component may be placed anywhere.
 *
 * Accordion layout: this component may be placed in its own section
 * before the payment section or directly in the payment section
 * before or after the payment component.
 */
export default class CheckoutTerms extends useCheckoutComponent(LightningElement) {
    _checkedByDefault = false;
    checked = false;
    showError = false;

    // The message to show to the shopper
    @api
    disclaimer = 'I accept the [[Terms and Conditions]]';

    // The link to the page containing the terms and conditions
    @api
    link = '/s/terms-and-conditions';

    // The error message instructing the user to accept the terms
    @api
    error = 'Please click the checkbox to accept the terms and conditions';

    /**
     * The terms may be checked by default from the builder property panel.
     */
    @api
    get checkedByDefault() {
        return this._checkedByDefault;
    }

    set checkedByDefault(value) {
        this._checkedByDefault = value;
        this.checked = value;
    }

    /**
     * Embed a link directing in the disclaimer string.
     */
    get disclaimerLink() {
        if (this.disclaimer.indexOf('[[') > 0 && this.disclaimer.indexOf('[[') <<
this.disclaimer.indexOf(']]')) {
            return this.disclaimer
                .replace(
                    '[[',
                    `<a href="${this.link}"
```

```
                            target="termsandconditions">`
                )
                .replace(']]', '</a>');
        }

        return `<a href="${this.link}"
        target="termsandconditions">${this.disclaimer}</a>`;
    }

    /**
     * update form when our container asks us to
     */
    stageAction(checkoutStage /*CheckoutStage*/) {
        switch (checkoutStage) {
            case CheckoutStage.CHECK_VALIDITY_UPDATE:
                return Promise.resolve(this.checkValidity());
            case CheckoutStage.REPORT_VALIDITY_SAVE:
                return Promise.resolve(this.reportValidity());
            default:
                return Promise.resolve(true);
        }
    }

    /**
     * Return true when terms checkoutbox is checked
     */
    checkValidity() {
        return !this.checked;
    }

/**
     * Return true when terms checkbox is checked
     */
    reportValidity() {
        this.showError = !this.checked;

        if (this.showError) {
            this.dispatchUpdateErrorAsync({
                groupId: 'TermsAndConditions',
                type: '/commerce/errors/checkout-failure',
                exception: 'Terms and Conditions must be accepted first by clicking the
checkbox',
            });
        }

        return this.checked;
    }

    /**
     * Check and uncheck the checkbox. Show error unless checked.
     * @param {*} event
     */
    handleChange(event) {
        this.checked = event.target.checked || false;
        this.showError = !this.checked;
    }
}
```

The third file is the component's metadata: `terms.js-meta.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>56.0</apiVersion>
  <isExposed>true</isExposed>
  <masterLabel>Terms and Conditions</masterLabel>
  <targets>
    <target>lightningCommunity__Page</target>
    <target>lightningCommunity__Default</target>
  </targets>
  <targetConfigs>
    <targetConfig targets="lightningCommunity__Default">
      <property
        label="Checked by Default"
        name="checkedByDefault"
        type="Boolean"
        default="false"/>
      <property
        label="Disclaimer"
        name="disclaimer"
        type="String"
       default="By clicking, you are confirming that your have read, understand and agree
 to the [[Terms and Conditions]]"/>
      <property
        label="Link URL"
        name="link"
        type="String"
        default="/s/terms-and-conditions"/>
      <property
        label="Error Message"
        name="error"
        type="String"
        default="Please click the checkbox to accept the terms and conditions"/>
    </targetConfig>
 </targetConfigs>
</LightningComponentBundle>
```

And the fourth file is a checkout icon: `terms.svg`:

```

<svg viewBox="0 0 52 52"
    width="52"
    height="52"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:bx="https://boxy-svg.com">
    <rect width="100%" height="100%" fill="#225f8c"/>
    <path d="M 34.996 27.173
             C 41.499 27.169 45.571 33.914 42.324 39.316
             C 39.077 44.718 30.947 44.724 27.689 39.328
             C 26.944 38.093 26.552 36.692 26.553 35.265
```

```
              C 26.562 30.8 30.336 27.181 34.996 27.173
              Z
              M 21.34 34.522
              C 23.637 34.522 25.072 36.904 23.923 38.81
              C 22.776 40.716 19.906 40.716 18.757 38.81
              C 18.496 38.375 18.357 37.883 18.357 37.381
              C 18.357 35.802 19.693 34.522 21.34 34.522
              Z
              M 38.075 31.976 L 33.556 36.83 L 31.513 34.85
              C 31.283 34.634 30.914 34.634 30.685 34.85
              L 29.857 35.615
              C 29.634 35.798 29.61 36.12 29.804 36.33
              L 29.857 36.387
              L 32.706 39.074
              C 32.931 39.292 33.236 39.416 33.556 39.418
              C 33.875 39.427 34.183 39.302 34.399 39.074
              L 39.76 33.449
              C 39.926 33.239 39.926 32.946 39.76 32.734
              L 38.925 31.984 C 38.695 31.768 38.327 31.768 38.098 31.984
              L 38.075 31.976
              Z
              M 12.272 9.054
              C 13.046 9.072 13.72 9.561 13.95 10.269
              L 14.412 11.812
              L 40.73 11.812 C 41.345 11.797 41.859 12.26 41.879 12.849
              C 41.886 12.964 41.871 13.077 41.834 13.185
              L 38.717 23.657
              C 38.663 23.89 38.525 24.098 38.329 24.243
              C 37.245 23.944 36.122 23.791 34.996 23.785
              C 33.602 23.794 32.221 24.036 30.916 24.5
              L 20.595 24.5
              C 19.689 24.492 18.947 25.19 18.939 26.058
              C 18.938 26.241 18.97 26.423 19.036 26.595
              L 19.036 26.651
              C 19.247 27.34 19.906 27.811 20.654 27.81
              L 25.949 27.81
              C 25.071 28.789 24.373 29.903 23.883 31.105
              L 18.089 31.105
              C 17.307 31.11 16.62 30.612 16.411 29.889 L 11.004 12.356
              L 9.386 12.356
              C 8.412 12.341 7.635 11.574 7.648 10.641
              L 7.648 10.597
              C 7.762 9.706 8.56 9.04 9.497 9.054
              L 12.272 9.054 Z"
            style="paint-order: fill; fill: rgb(255, 255, 255);" bx:origin="0.507 0.496"/>
</svg><svg viewBox="0 0 52 52" width="52" height="52" xmlns="http://www.w3.org/2000/svg"
xmlns:bx="https://boxy-svg.com">
    <rect width="100%" height="100%" fill="#225f8c"/>
    <path d="M 34.996 27.173 C 41.499 27.169 45.571 33.914 42.324 39.316 C 39.077 44.718
30.947 44.724 27.689 39.328 C 26.944 38.093 26.552 36.692 26.553 35.265 C 26.562 30.8
30.336 27.181 34.996 27.173 Z M 21.34 34.522 C 23.637 34.522 25.072 36.904 23.923 38.81 C
 22.776 40.716 19.906 40.716 18.757 38.81 C 18.496 38.375 18.357 37.883 18.357 37.381 C
18.357 35.802 19.693 34.522 21.34 34.522 Z M 38.075 31.976 L 33.556 36.83 L 31.513 34.85
C 31.283 34.634 30.914 34.634 30.685 34.85 L 29.857 35.615 C 29.634 35.798 29.61 36.12
```

```
29.804 36.33 L 29.857 36.387 L 32.706 39.074 C 32.931 39.292 33.236 39.416 33.556 39.418
C 33.875 39.427 34.183 39.302 34.399 39.074 L 39.76 33.449 C 39.926 33.239 39.926 32.946
39.76 32.734 L 38.925 31.984 C 38.695 31.768 38.327 31.768 38.098 31.984 L 38.075 31.976
Z M 12.272 9.054 C 13.046 9.072 13.72 9.561 13.95 10.269 L 14.412 11.812 L 40.73 11.812 C
 41.345 11.797 41.859 12.26 41.879 12.849 C 41.886 12.964 41.871 13.077 41.834 13.185 L
38.717 23.657 C 38.663 23.89 38.525 24.098 38.329 24.243 C 37.245 23.944 36.122 23.791
34.996 23.785 C 33.602 23.794 32.221 24.036 30.916 24.5 L 20.595 24.5 C 19.689 24.492
18.947 25.19 18.939 26.058 C 18.938 26.241 18.97 26.423 19.036 26.595 L 19.036 26.651 C
19.247 27.34 19.906 27.811 20.654 27.81 L 25.949 27.81 C 25.071 28.789 24.373 29.903 23.883
 31.105 L 18.089 31.105 C 17.307 31.11 16.62 30.612 16.411 29.889 L 11.004 12.356 L 9.386
 12.356 C 8.412 12.341 7.635 11.574 7.648 10.641 L 7.648 10.597 C 7.762 9.706 8.56 9.04
9.497 9.054 L 12.272 9.054 Z" style="paint-order: fill; fill: rgb(255, 255, 255);"
bx:origin="0.507 0.496"/>
</svg>
```

Custom Rules for Product Readiness

Merchandisers can run Product Readiness on their catalogs to ensure that all products in a catalog are storefront ready. By default, Product Readiness uses a rule set that contains criteria based on SKUs, images, categories, and descriptions. If the default Product Readiness rule set doesn't fit your organization's needs, create a custom rule set to determine what criteria a product must fit to be considered ready.

Use an APEX class to create a rule set, and then activate the Readiness.ProductEvaluator. After the rule set is enabled, an admin must rebuild the index.

A custom rule can't be added to the default rule set. If a custom rule set is enabled, it replaces the default rule set.

ProductScore records are associated with productIds as a measure of Product Readiness. If there are no score details returned for a given product, the score record is deleted. This rule allows for obsolete product scores to be deleted if the rules no longer return scores for these cases.

Example:

```
public class ProductReadinessDescriptionEvaluator implements Readiness.ProductEvaluator
 {

    // return true for the rule to run
    public boolean isActive() {
        return true;
    }

    // return a list of Readiness.ProductScoreDetail which will become the product
readiness score
    public List<Readiness.ProductScoreDetail>
evaluateReadiness(Readiness.ProductEvaluationContext productContext) {
        // example implmentation which checks for a description and a price
        Set<ID> productIds = productContext.ProductIds;
        List<Readiness.ProductScoreDetail> scores = new
List<Readiness.ProductScoreDetail>();

        List<Product2> products = [SELECT
              id, description
              FROM  Product2
              WHERE
              id IN : productIds];
        for (Product2 product : products) {
            scores.add(new Readiness.ProductScoreDetail(
                product.Id,
                'Description Length',
                String.isBlank(product.description) ? 0 : 100,
                'Product does not have a Description.'));

            List<PricebookEntry> entries = [SELECT id FROM PricebookEntry WHERE
Product2Id = :product.Id];
            Integer pricebookScore = (entries.size() == 0) ? 0 : 100;
            scores.add(new Readiness.ProductScoreDetail(
                product.Id,
                'Price',
                pricebookScore,
                'Product must have a price.'));
        }

        return scores;
    }
}
```

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 and B2C stores created with an LWR template use the Salesforce Commerce Checkout that's installed when you create the store. For B2B and B2C 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.

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.

B2B Checkout Flow Tasks

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

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.

Configure a B2B Checkout Flow

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

Configure B2B Checkout Flows to Create Managed Order Summaries

Configure your B2B checkout flow to integrate Salesforce Order Management.

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.

B2B Legacy Checkout Reference

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

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.

Add a Promotions Subflow to a Checkout Flow

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

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.

Create a Promotion Subflow for an Existing Checkout Flow

EDITIONS

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

EDITIONS

Available in: B2B Commerce

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.

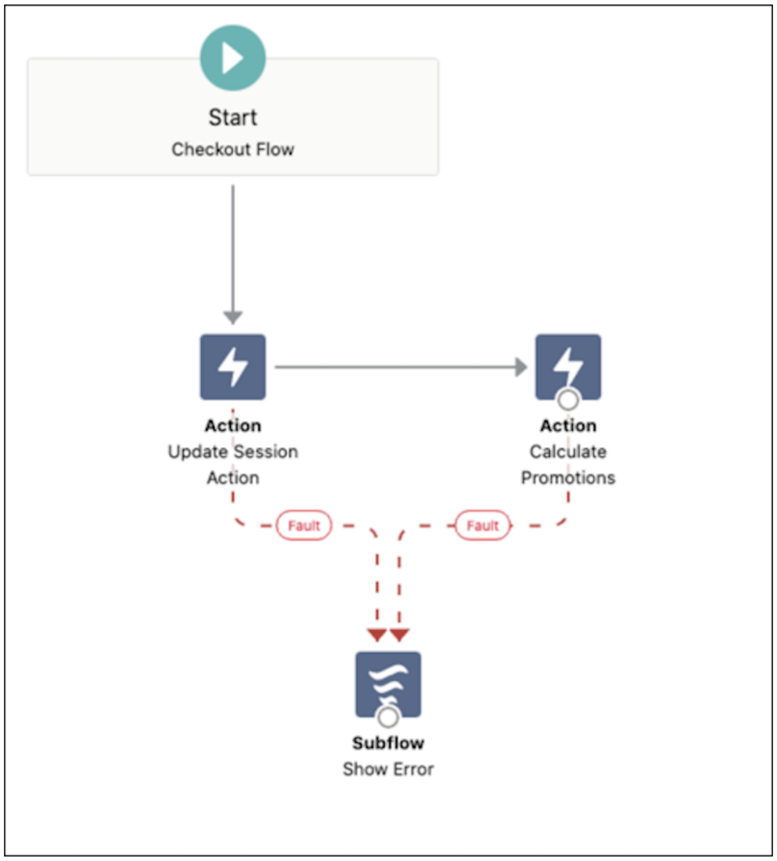
1. From Setup, in the Quick Find box, enter `Flows`, and then select **Flows**.
2. Click **New Flow**.
3. Click **All + Templates**.
4. Click **Checkout Flow**, select **Checkout Flow**, and then click **Next**.
5. Click **Freeform**.
6. Drag the **Action** element onto the canvas.
7. In the New Action screen, locate Promotions, and select **Calculate Cart Promotions**.
8. 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**.
9. Drag another **Action** element onto the canvas.
10. In the New Action screen, search for "Session" and select **Update Checkout Session Action**.
11. 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.**
   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.

12. Drag the **Subflow** element onto the canvas.
13. In the New Subflow screen, search for "Error" and select **Subflow - Error**.
14. 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.
15. Drag a connector line from the Start node to the Update Checkout Session action.
16. Drag a connector line from the Update Checkout Session to the Calculate Cart Promotions action.
17. Drag a connector line from the Update Checkout Session action to the Show Error subflow.
18. 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.



19. Click **Save**.
20. On the Save the flow screen, enter a label (for example, `Calculate Promotions`) and click **Save**.
21. 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

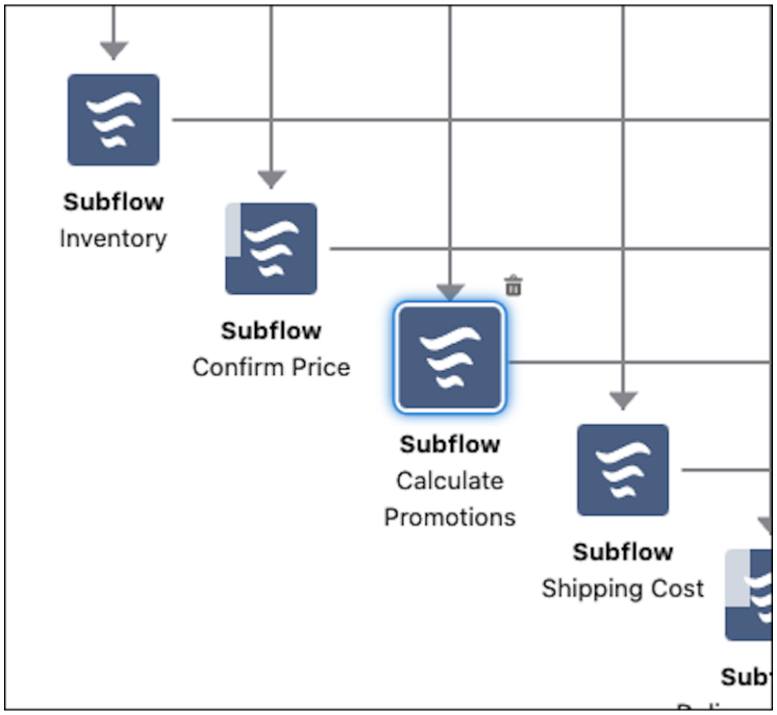
Available in: B2B Commerce

1. From Setup, in the Quick Find box, enter `Flows`, and then select **Flows**.
2. In Flow Builder, open your main checkout flow.
3. 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`.

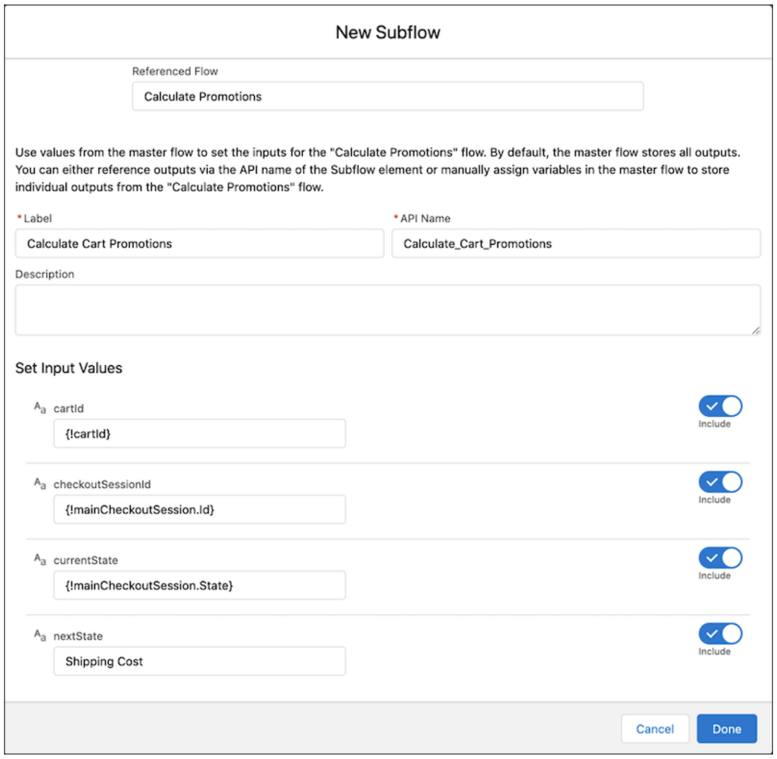
h. Click **Done**.

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



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

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



Referenced Flow: Calculate Promotions

Use values from the master flow to set the inputs for the "Calculate Promotions" flow. By default, the master flow stores all outputs. You can either reference outputs via the API name of the Subflow element or manually assign variables in the master flow to store individual outputs from the "Calculate Promotions" flow.

*Label: Calculate Cart Promotions

*API Name: Calculate_Cart_Promotions

Description:

Set Input Values

cartId: {!cartId} — Include

checkoutSessionId: {!mainCheckoutSession.Id} — Include

currentState: {!mainCheckoutSession.State} — Include

nextState: Shipping Cost — Include

7. Drag a connector line from the Main Decision Hub to your promotions subflow.
8. On the Select outcome for decision connector screen, select **Promotions**.
9. Drag a connector line from your promotions subflow to the Assignment node.
10. Double-click the **Confirm Price** subflow.
11. On the Edit "Subflow - Confirm Price" Subflow screen, enter `Promotions` as the nextState.
12. Click **Save As** and select **A New Version**.
13. 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. From Setup, in the Quick Find box, enter `Flows`, and then select **Flows**.
2. Click to select a checkout subflow that runs asynchronously .
3. 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.)
4. Enter a label and API name (such as Check Inventory) for the New Action element.
5. Under cartId, add `{!cartId}`. This variable represents the ID value of the current cart.
6. Click **Done**.
7. Delete the subflow action you want to replace.
8. Drag and connect the new action.
9. Click **Save as** and click **New Flow**.
10. Enter `Asynch Check Inventory` in the Flow Label and Flow API Name fields.
11. To make the new action available, reload the checkout flow.
12. Drag a new subflow onto the canvas.
13. Enter `Inventory` in the Referenced flow pane and choose the new asynchronous subflow.
14. Enter `Asynch Check Inventory` in the label and API Name fields for this subflow.
15. In the Set Input Values, choose `{!cartId}` and also `checkoutsessionId`.
16. Click **Done**.
17. Delete the old Apex Action Check Inventory (the synchronous node).

18. Connect the new Async Check Inventory subflow.

19. Click **Save**.

20. 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.

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.

Configure B2B Checkout Time to Live with Developer Console

EDITIONS

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

EDITIONS

Available in: B2B Commerce

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

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

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

EDITIONS

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

EDITIONS

Available in: B2B Commerce

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.

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.

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

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.

Create Third-Party Integrations with Platform Events
To trigger notifications using an external, or third-party, system, subscribe to Salesforce platform events.

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.

App, Push, and Bell Order Confirmations

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

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.
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.

Create a Notification Flow

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

Invoke a Notification Flow

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

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. From Setup, in the Quick Find Box, enter `Notification Builder`, and select **Custom Notifications**.
2. Click **New**.
3. 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**.
4. 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. From Setup, in the Quick Find Box, enter `Process Automation`, and select **Flows**.

2. Create a flow.
3. 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.

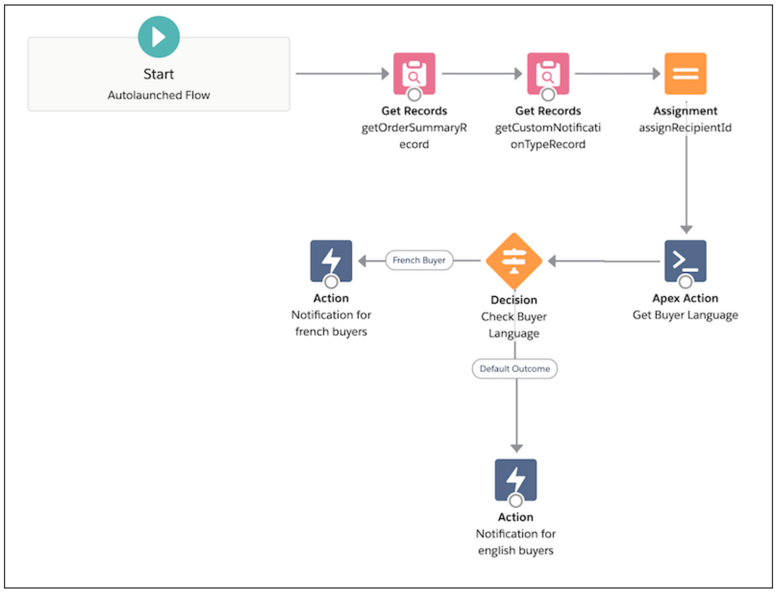
Field	Element One	Element Two
Label	getOrderSummaryRecord	getCustomNotificationTypeRecord
API Name	getOrderSummaryRecord	getCustomNotificationTypeRecord
Object	Order Summary	Custom Notification Type
Conditions	• Condition requirements: Conditions are met • Field: ID • Operator: Equals	• Condition requirements: Conditions are met • Field: DeveloperName • Operator: Equals

Field	Element One	Element Two
	• Value: {!OrderSummaryRecord.Id} Note: OrderSummaryRecord refers to the API name of the first variable that you created in this flow.	• 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. From Setup, in the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
2. Select **Notify On Order Summary Created Process**.
3. Select **Clone**, choose **Save Close As**, and select **Version of current process**.
4. Click **Save**.
5. Select **+ Add Action**.
6. 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**.
7. 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. From Setup, in the Quick Find Box, enter `Platform Events`, and select **Platform Events**.
2. Create a platform event.
3. 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**.
4. 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**.

5. 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. From Setup, in the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
2. Create a process.
3. 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**.
4. Save your process.
5. 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**.
6. 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**.
7. 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**.

8. 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. From Setup, in the Quick Find Box, enter *`Process Automation`*, and select **Flows**.
2. Create a flow.
3. Select **Auto launched Flow**, and click **Create**.
4. In Flow Builder, select the **Manager** tab, and click **New Resource**.
5. 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**.
6. Click **Done**.
7. Back in Flow Builder, select the Elements tab. Drag the Action element to the canvas.

8. 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}`.
9. Click **Done**.
10. Link the two flow actions by dragging a connection between them.
11. 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. From Setup, in the Quick Find Box, enter `Process Automation`, and select **Process Builder**.
2. Click **New**.
3. 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**.
4. Click **Save**.
5. 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**.

6. 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

7. For Conditions, select **Customize the Logic**.
8. For Logic, select (1 OR 2 OR 3 OR 4) AND 5.
9. Click **Save**.
10. Select **+ Add Action**.
11. 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.
12. 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.

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.

Add Optional Notifications to an Email Flow
To let users choose whether to receive email notification, modify your email order notification flow.

Let Users Opt Out of Checkout Notifications

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

1. From Setup, in the Quick Find Box, enter `Object Manager`, and select **User**.
2. Select **Fields and Relationships**, and click **New**.
3. For Data Type, select **Checkbox**.
4. Under Field Label, enter `Checkout Notifications`.
5. 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.
6. For Field Name, enter `Checkout_Notifications`, and click **Next**.
7. Choose which profiles see the field, and click **Next**.

   We suggest choosing the buyer user profiles.
8. Add the field to the User Profile Layout.
9. 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. From Setup, in the Quick Find Box, enter `Custom Code`, and select **Apex Classes**.
2. Click **New**.
3. 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.

Create a Flow for Localized Notifications
Create a flow that uses your Apex action to localize notifications.

Localize a Notification

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

1. From Setup, in the Quick Find Box, enter `Custom Code`, and select **Apex Classes**.
2. Click **New**.
3. 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. From Setup, in the Quick Find Box, enter `Process Automation`, and select **Flows**.
2. Create a flow.
3. Select **Autolaunched Flow**, and click **Create**.
4. On the Manager tab, click **New Resource**.
5. 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`.
6. On the Element tab, drag an **Action** element to the canvas.
7. 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**.

8. On the Element tab, drag a **Decision** element to the canvas.
9. 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**.

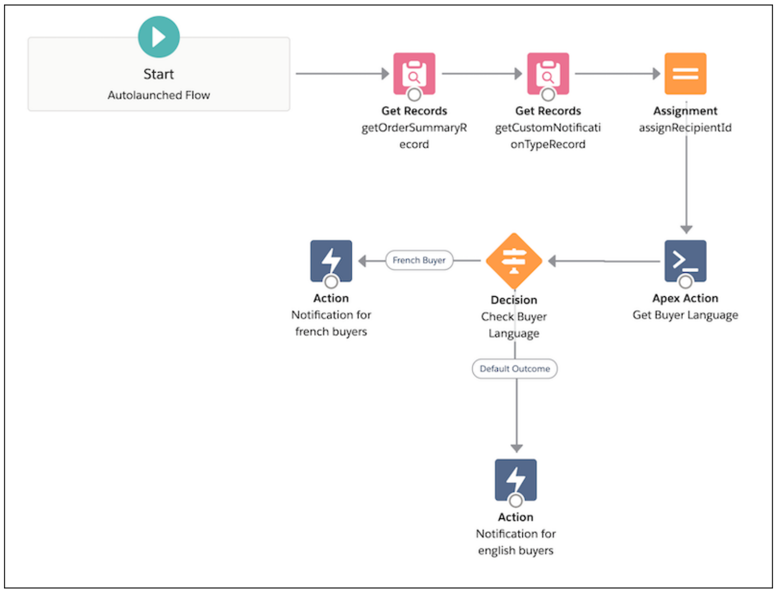
10. On the Element tab, drag an **Action** element to the canvas.
11. Fill in the Action element form.
    a. For Search All Actions, select **Send Custom Notifications**.
    b. For **Label**, enter `Notifications for French Buyers`.
    c. For **API Name**, enter `Notifications_for_French_Buyers`.
    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**.

12. On the Element tab, drag another **Action** element to the canvas.
13. 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**.

14. 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.

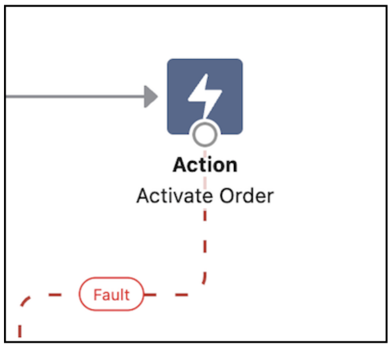
- **Order confirmation**—When an OrderSummary sObject is created successfully, Salesforce triggers the OrderSummaryCreatedEvent. Applications can subscribe to this platform event and send notifications from external systems to Salesforce.
- **Intermediate checkout stages**—Create a custom platform event, 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

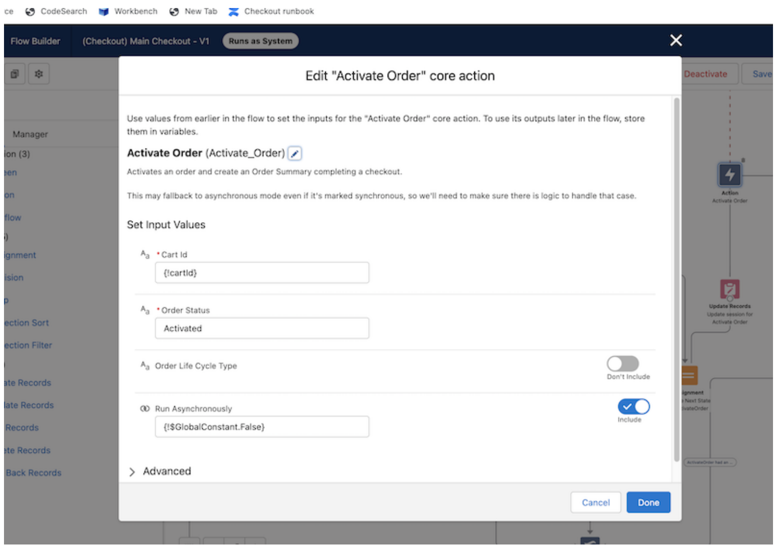
Configure your B2B checkout flow to integrate Salesforce Order Management.

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

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



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



Edit "Activate Order" core action

Use values from earlier in the flow to set the inputs for the "Activate Order" core action. To use its outputs later in the flow, store them in variables.

Activate Order (Activate_Order)

Activates an order and create an Order Summary completing a checkout.

This may fallback to asynchronous mode even if it's marked synchronous, so we'll need to make sure there is logic to handle that case.

Set Input Values

* Cart Id: {!cartId}

* Order Status: Activated

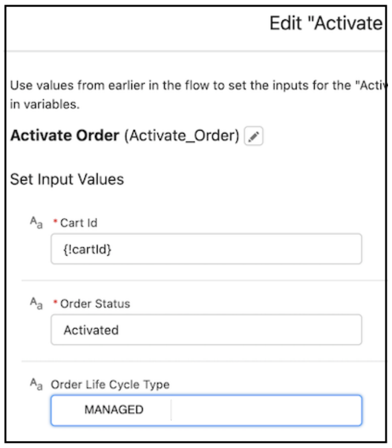
Order Life Cycle Type — Don't Include

Run Asynchronously: {!$GlobalConstant.False} — Include

Advanced

Cancel | Done

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



Edit "Activate

Use values from earlier in the flow to set the inputs for the "Activ
in variables.

Activate Order (Activate_Order)

Set Input Values

* Cart Id: {!cartId}

* Order Status: Activated

Order Life Cycle Type: MANAGED

6. Click **Done**.
7. 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.

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.

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.

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.

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.

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.

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.

Exportable and Updatable Summary Objects

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

What Happens When a Customer Places an Order

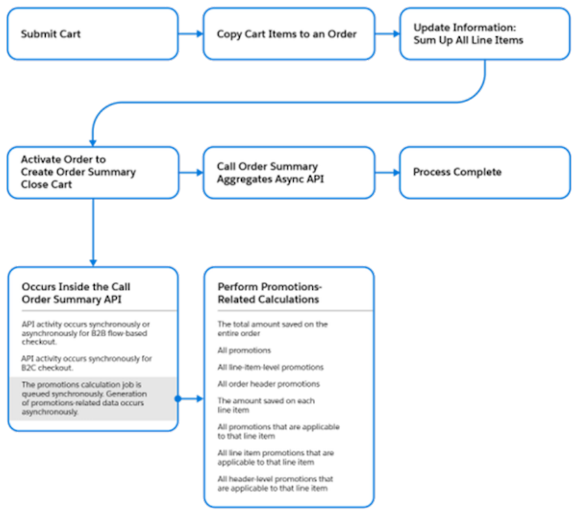
EDITIONS

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

EDITIONS

Available in: B2B and D2C Commerce

When a customer places an order, several events occur.


Submit Cart
Copy Cart Items to an Order
Update Information: Sum Up All Line Items
Activate Order to Create Order Summary Close Cart
Call Order Summary Aggregates Async API
Process Complete
Occurs Inside the Call Order Summary API
API activity occurs synchronously or asynchronously for B2B flow-based checkout.
API activity occurs synchronously for B2C checkout.
The promotions calculation job is queued synchronously. Generation of promotions-related data occurs asynchronously.
Perform Promotions-Related Calculations
The total amount saved on the entire order
All promotions
All line-item-level promotions
All order header promotions
The amount saved on each line item
All promotions that are applicable to that line item
All line item promotions that are applicable to that line item
All header-level promotions that are applicable to that line item

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

Order Field	Required for Order Summaries	Required for Orders	Field Type	Value
`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`*

OrderItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`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`*

OrderDeliveryGroup Field	Required for Order Summaries	Required for Orders	Field Type	Value
`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*

OrderItemAdjustmentLineItem Field	Required for Order Summaries	Required for Orders	Field Type	Value
`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`*

SEE ALSO:

Disable Cart to Order Mappings for Custom Fields

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.

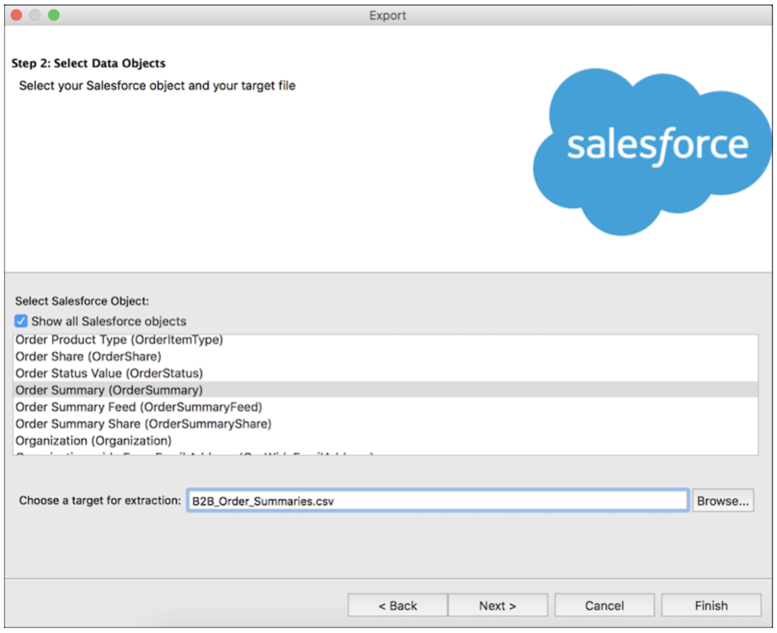
Export Unmanaged Order Summaries with Data Loader
Export your B2B Commerce order summary data out of Salesforce using Data Loader.

Update Unmanaged Order Summaries with Data Loader
Use Data Loader to make bulk updates to your B2B Commerce order summary data.

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**.

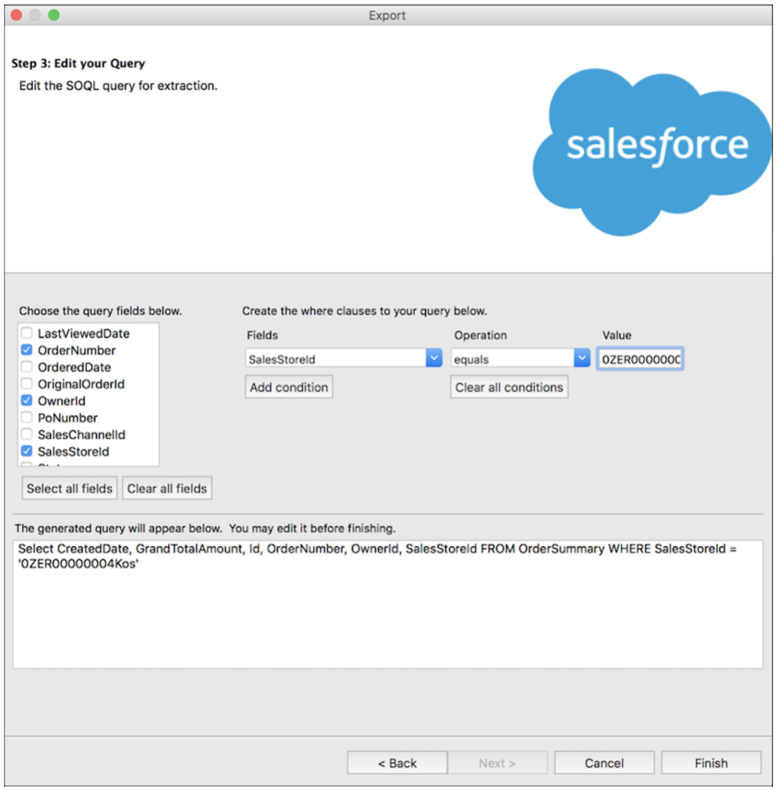


Step 2: Select Data Objects
Select your Salesforce object and your target file

Select Salesforce Object:
☑ Show all Salesforce objects
Order Product Type (OrderItemType)
Order Share (OrderShare)
Order Status Value (OrderStatus)
Order Summary (OrderSummary)
Order Summary Feed (OrderSummaryFeed)
Order Summary Share (OrderSummaryShare)
Organization (Organization)

Choose a target for extraction: B2B_Order_Summaries.csv

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.



Select CreatedDate, GrandTotalAmount, Id, OrderNumber, OwnerId, SalesStoreId FROM OrderSummary WHERE SalesStoreId = '0ZER00000004Kos'

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')**

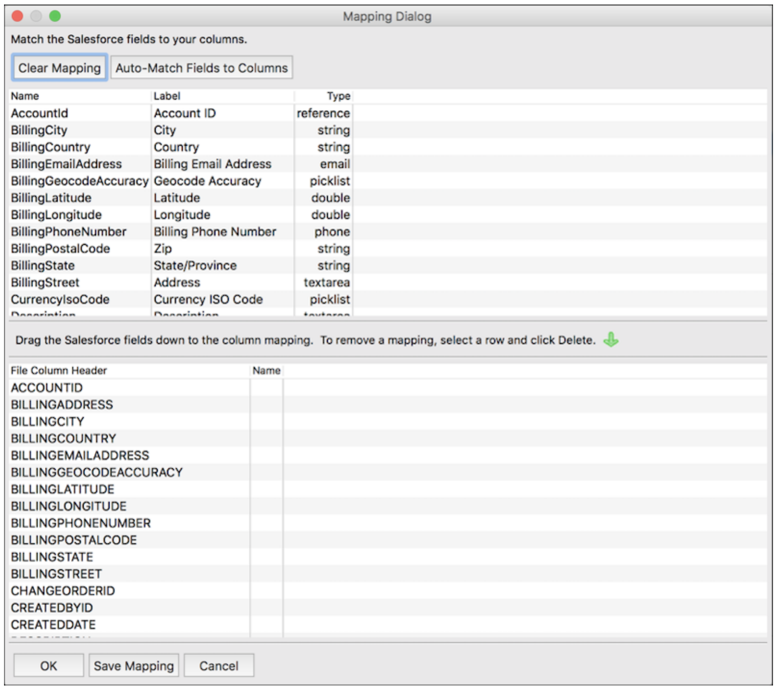
Object	Example Query
OrderItemTaxLineItemSummary	Select Id, Amount, CreatedDate, Rate, Type FROM OrderItemTaxLineItemSummary **WHERE OrderSummaryId IN ('1OsR000000002K7KAI')**
OrderAdjustmentGroupSummary	Select Id, CreatedDate, TotalAmount, TotalTaxAmount, Type FROM OrderAdjustmentGroupSummary **WHERE 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.



Match the Salesforce fields to your columns.

Name	Label	Type
AccountId	Account ID	reference
BillingCity	City	string
BillingCountry	Country	string
BillingEmailAddress	Billing Email Address	email
BillingGeocodeAccuracy	Geocode Accuracy	picklist
BillingLatitude	Latitude	double
BillingLongitude	Longitude	double
BillingPhoneNumber	Billing Phone Number	phone
BillingPostalCode	Zip	string
BillingState	State/Province	string
BillingStreet	Address	textarea
CurrencyIsoCode	Currency ISO Code	picklist

Drag the Salesforce fields down to the column mapping. To remove a mapping, select a row and click Delete.

File Column Header	Name
ACCOUNTID	
BILLINGADDRESS	
BILLINGCITY	
BILLINGCOUNTRY	
BILLINGEMAILADDRESS	
BILLINGGEOCODEACCURACY	
BILLINGLATITUDE	
BILLINGLONGITUDE	
BILLINGPHONENUMBER	
BILLINGPOSTALCODE	
BILLINGSTATE	
BILLINGSTREET	
CHANGEORDERID	
CREATEDBYID	
CREATEDDATE	

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.

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.

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.

B2B Checkout Flow Elements

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

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.

B2B Checkout States

The CartCheckoutSession object manages checkout progress.

B2B Checkout Task Modes

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

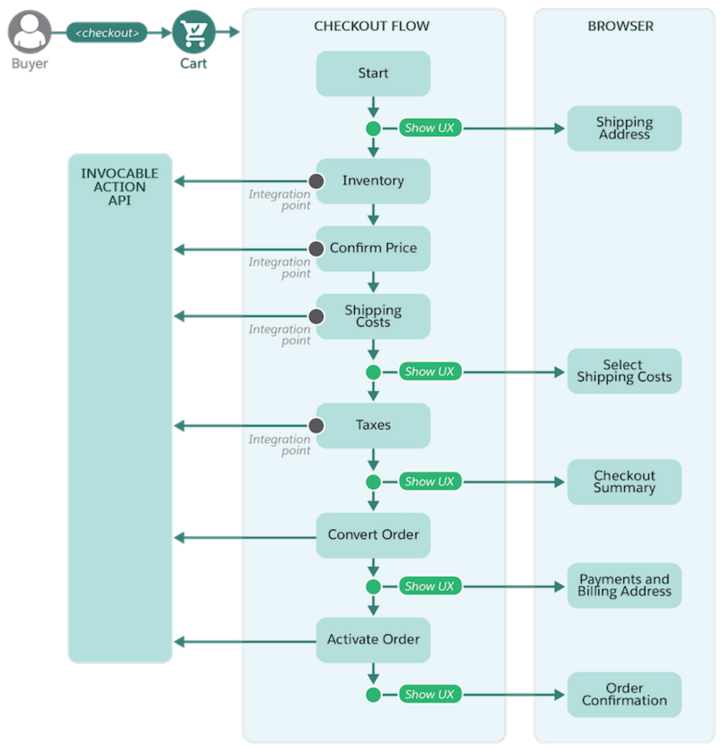
B2B Checkout Error Handling

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.

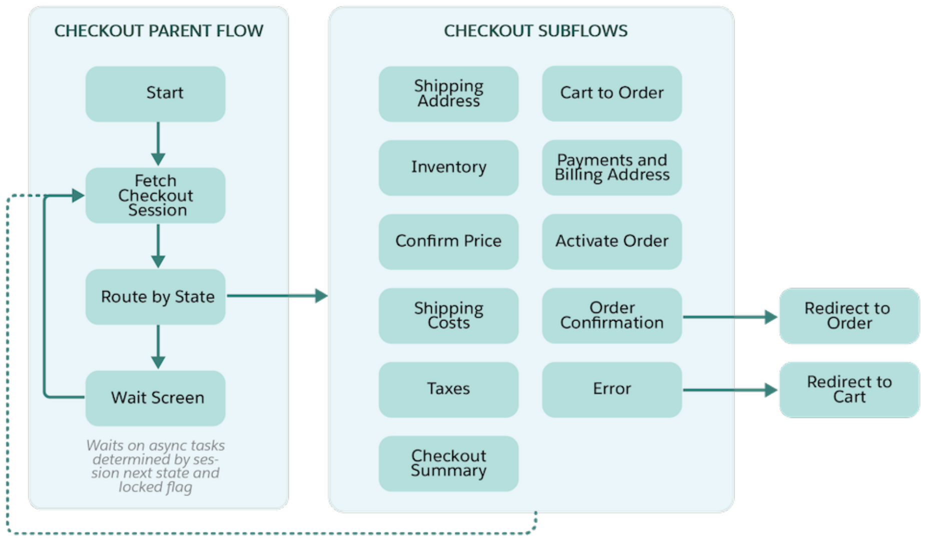
B2B Checkout Flow Design

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

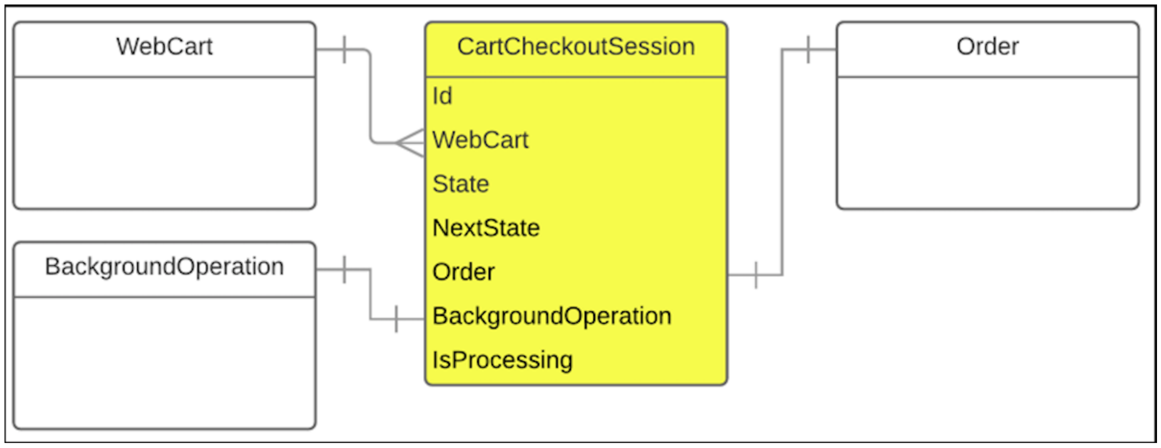
Note: The Cart to Order and Activate Order subflows run synchronously by default. To run these actions asynchronously, change the `Run Asynchronously` parameter from within the flow.

B2B Checkout States

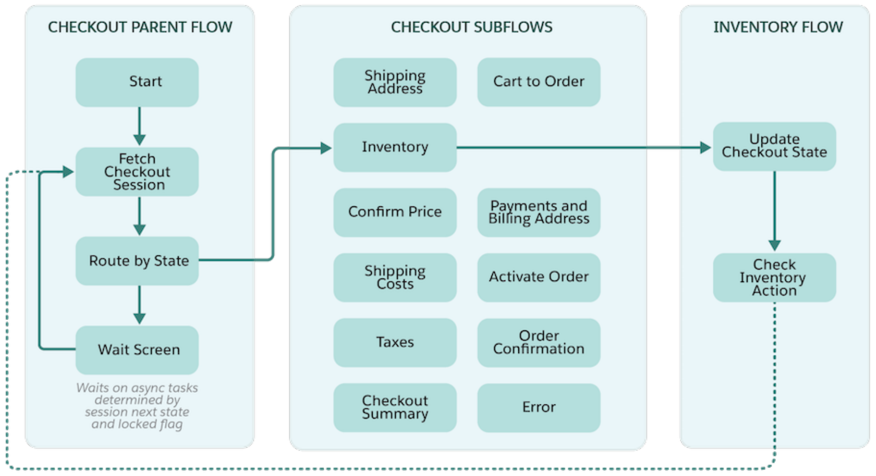
The CartCheckoutSession object manages checkout progress.

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.

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 {
}
```

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');
```