**What is Guided Selling?**

Guided selling is a **consistent**shopping experience that includes steps to **guide the user** through the process of **recommending, selecting, and ordering products**.

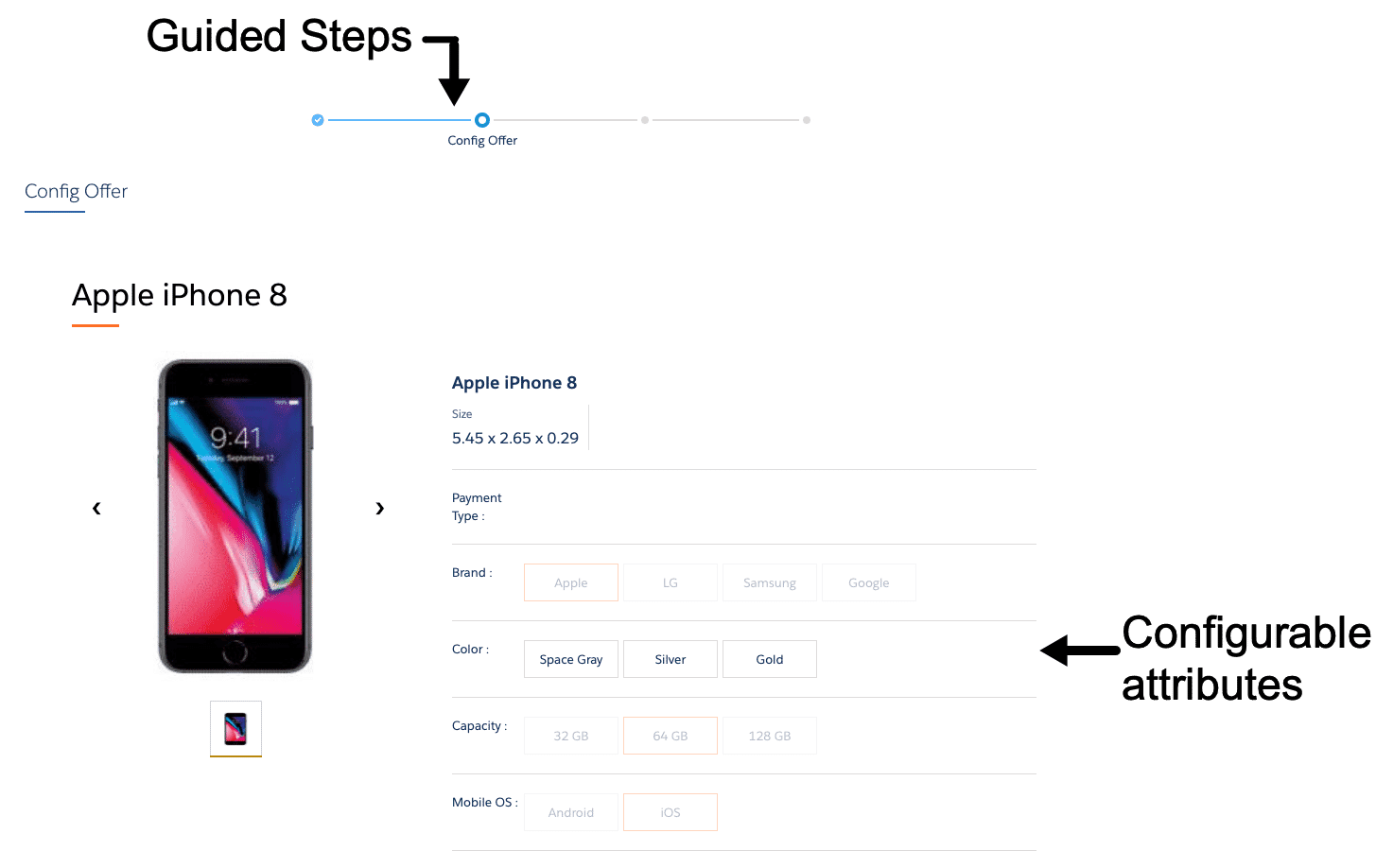
**Industries CPQ +  OmniScript = Guided Selling**

With OmniStudio, you can build guided selling experiences using Industries CPQ and OmniScript.

**Industries CPQ**is a powerful rules and pricing engine built on the Salesforce platform that ensures that quotes and orders are valid before submission, minimizing order delays, customer confusion, and customer dissatisfaction. Industries CPQ picks up where CRM ends, providing real-time product, pricing, and business rules based on natural language and declarative administration.

**OmniScript**is a declarative scripting tool, meaning you create it with clicks, not code. It allows you to build all sorts of guided interactions, including guided selling.

**Guided Selling and OmniScript**

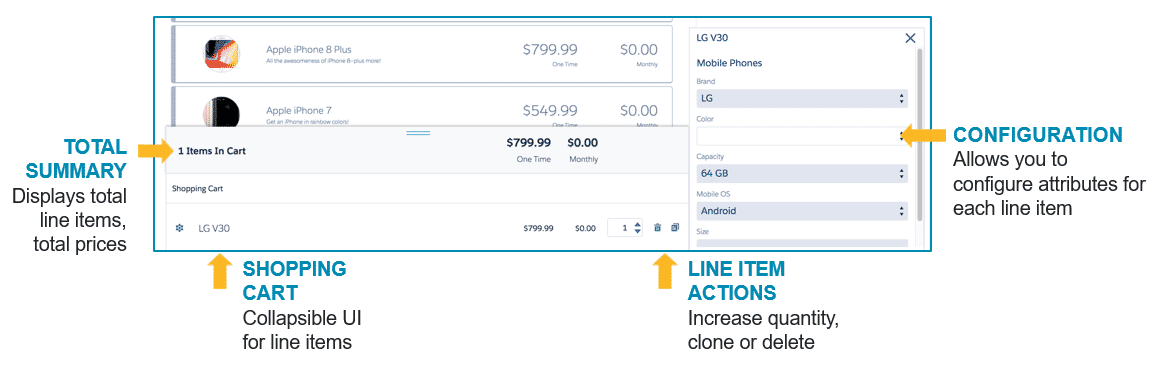


When used in a guided selling context, OmniScripts can be configured to include the following:

* A series of steps to be taken during the interaction to provide a better customer experience
* A list of products from your shared catalog that use the same pricing and rules used elsewhere in Industries CPQ
* The ability to configure items before adding them to the cart
* An order summary and running total bar at the bottom of the cart page

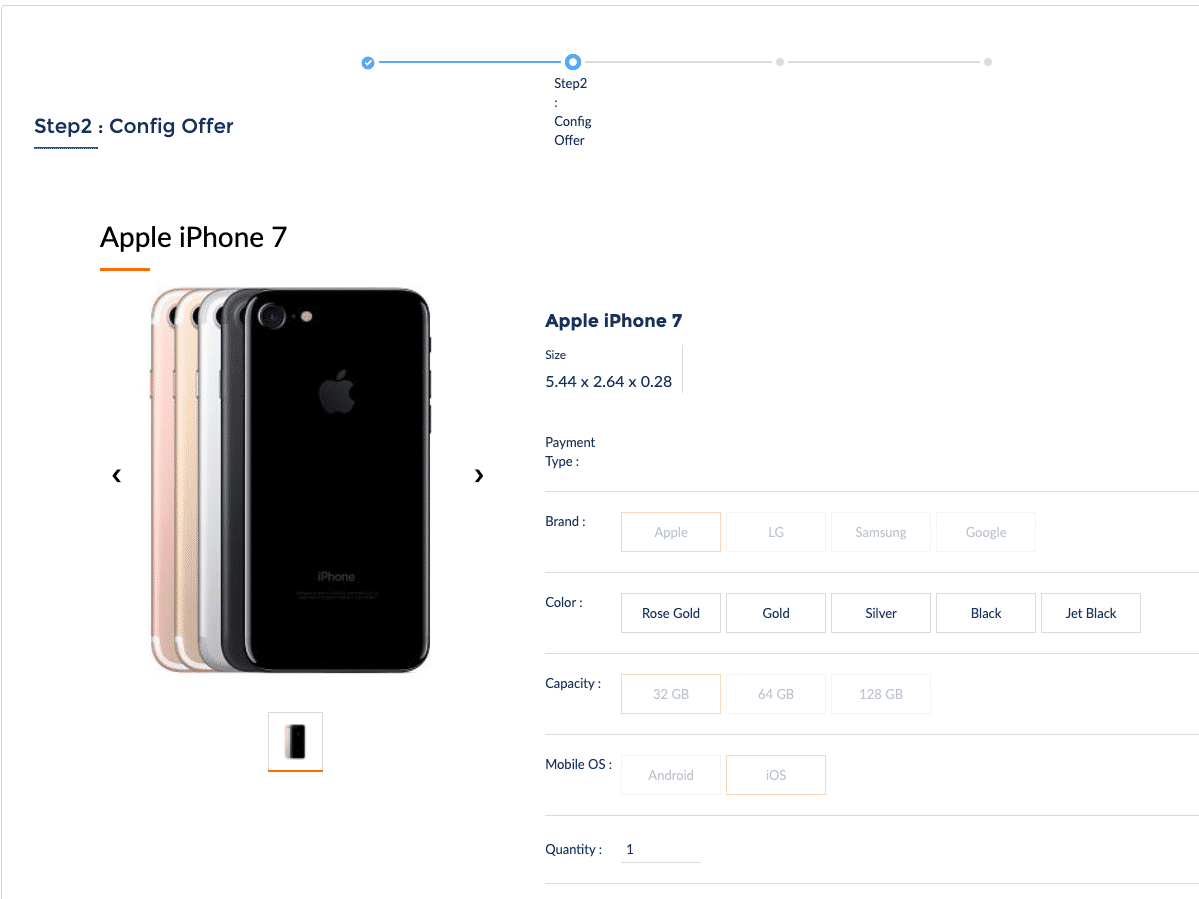
**Two Approaches to Building Guided Selling**

There are two different approaches to building a guided selling experience.  
  
**Cart-based** guided selling experiences use cart-based APIs and Angular JS templates to feature a visually persistent cart as the user navigates through the order process. You can build cart-based guided selling experiences using Industries CPQ and OmniScript very quickly while leveraging your existing catalog products, rules, and pricing. This technology works well for internal processes, with known traffic patterns and logged in users.



A cart-based guided selling experience using the OmniScript persistent cart

But most commerce companies also need to serve customers directly through their websites, which means unpredictable usage patterns. And unlike internal employees, 1 in 4 customers will abandon a process if it takes longer than 4 seconds to load, which impacts bounce rates, conversion rates and return visitors.  **Digital Commerce** (Industries Digital Commerce Gateway) guided selling experiences use cacheable APIs and Lightning web components to build a fast loading step-by-step order process.



A Digital Commerce guided selling experience using Lightning web components and OmniScript.

**Differentiating Between the Two Approaches**

Cart-based guided selling experiences are great for supporting internal business processes and microservices for **known** users. Developing a cart-based guided selling flow is straightforward with OmniScript, DataRaptors, and cart-based APIs. However, with Digital Commerce, you now have new options and ways to build guided selling experiences.

Digital Commerce allows you to create websites where **anonymous** users can browse, configure, and add products or bundles to their cart. Because users do not need to register an account, this is an ideal external-facing approach for commerce. Furthermore, you can customize your Digital Commerce solution for on or off-platform servers, where off-platform services like AWS can provide elastic scaling to meet heavy customer traffic.

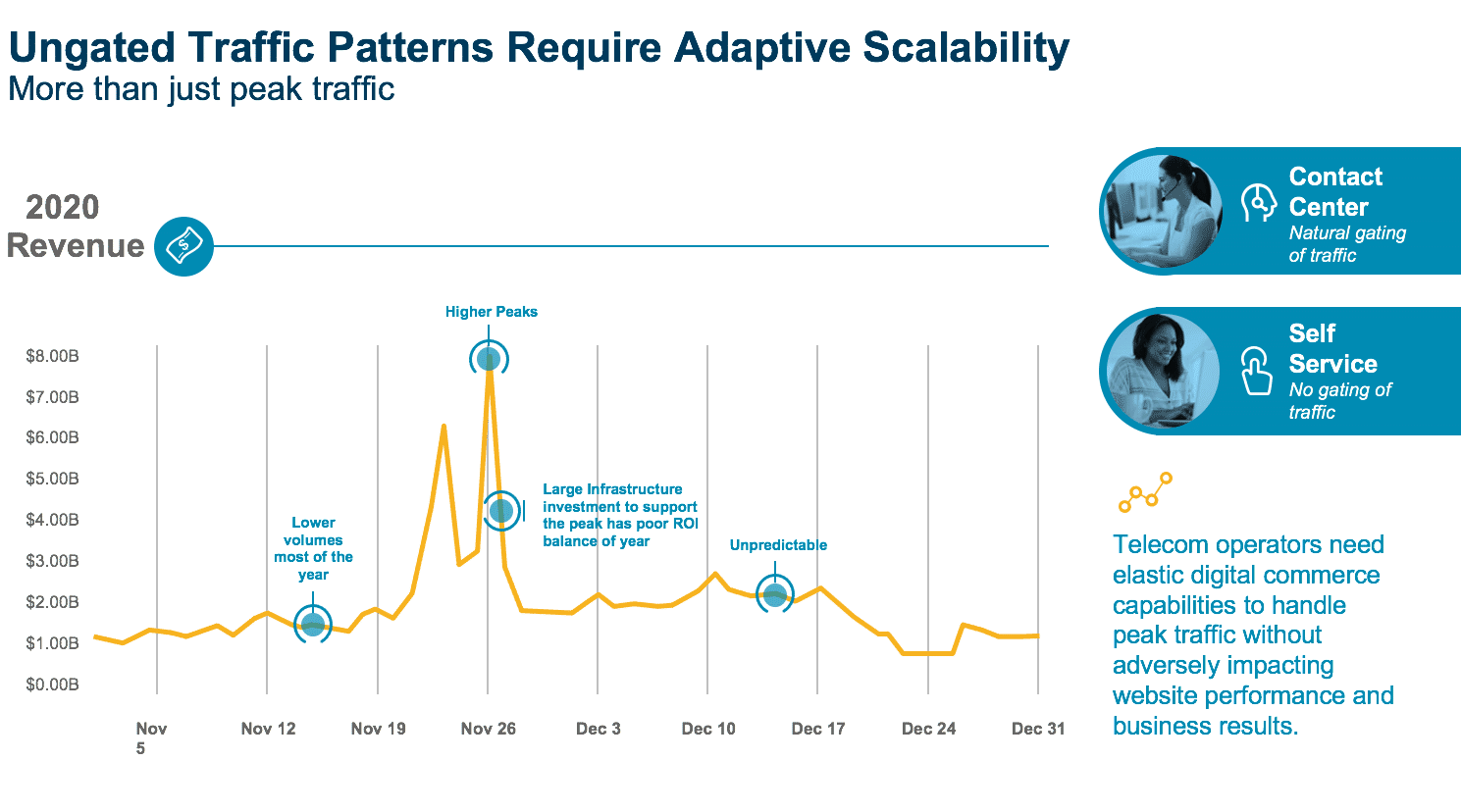
Below is a summary of the differences in guided selling approaches:

|  | **Cart-based guided selling** | **Digital Commerce guided selling** |
| --- | --- | --- |
| Ideal Use Cases | Internal users with accounts in your Salesforce org | * External OR internal customers/users * Support for logged-in AND anonymous users * Optimized performance with caching * Can be scalable for heavy customer traffic with the Digital Commerce Tier |

# Introducing Industries Digital Commerce

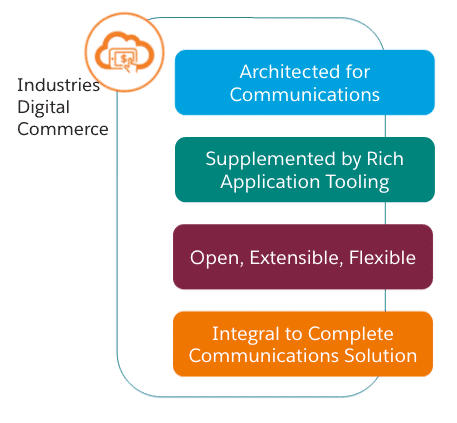
**Ungated Traffic Patterns Require Adaptive Scalability**

While contact centers naturally gate user traffic through the use of live agents, self-service digital commerce has no inherent gating of traffic. In order to enter the world of digital commerce, you have to account for more than just peak traffic, you need adaptive scalability at carrier-grade speeds.



**Introducing Digital Commerce**

Digital Commerce has been designed from the ground up to support the highest volumes of digital commerce browsing and transactions. It allows you to create websites where anonymous users can browse, configure, and add products or bundles to their cart. Because users do not need to register an account, this is an ideal external-facing approach for commerce. Furthermore, you can customize your Digital Commerce solution for on or off-platform servers, where off-platform services like AWS can provide elastic scaling to meet heavy customer traffic.

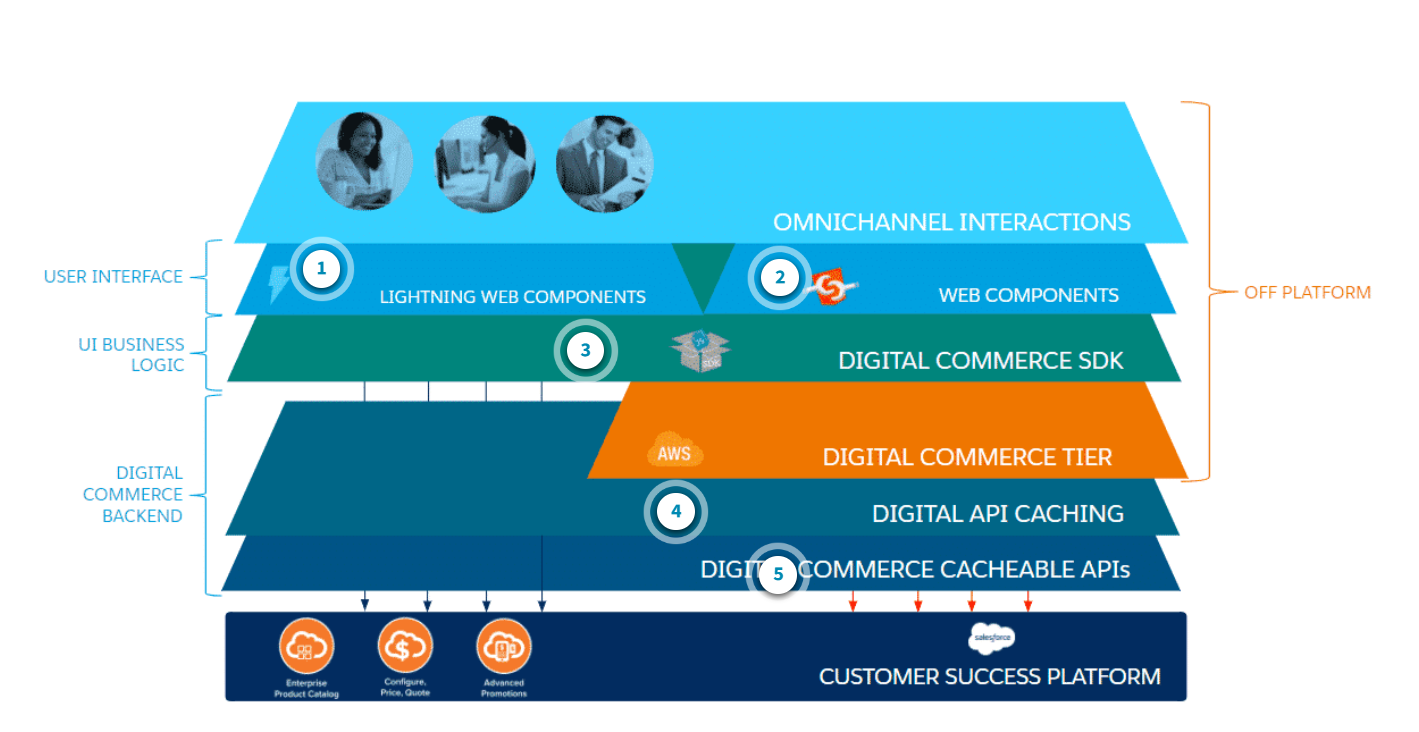


**Digital Commerce includes a digital commerce tier and on-platform caching which allows advanced order-capture and guided selling capabilities to offer best-fit products and services to customers at scale and speed.**

Digital Commerce addresses both communication service providers and subscriber needs and is built on an integrated omnichannel platform that enables design, order capture, order management, and customer management scenarios across all channels and devices. This enables you to more quickly launch new offers, and provides users a seamless experience, from purchase to delivery to support.

**Industries Digital Commerce Components**

Digital Commerce offers multiple options for its front-end UI and back-end tech stack. The illustration below breaks down each component in the tech stack



1. **Lightning web components**are web components that run inside Salesforce. They implement the web component standard but use select Salesforce libraries and packaging conventions. The Digital Commerce Lightning web components are used to build digital commerce experiences and are available within the managed package.
2. **Web components**are components and widgets built on web component standards that work across modern browsers and can be distributed with any JavaScript library or framework. Web components are typically used in off-platform digital commerce while Lightning Web Components are used on-platform. Web components can be used to create new custom, reusable, encapsulated HTML tags to use in your web pages or applications.
3. **Digital Commerce SDK**is a pure JavaScript library that abstracts and simplifies the use of Digital Commerce APIs. The SDK improves usability and reduces the effort to develop compelling user interfaces by hiding complex API semantics inside the SDK interface.
4. **Digital API Caching** pre-generates anticipated high-volume API calls. API responses are then cached on platform or off-platform, for extremely elastic scaling on Amazon Web Services (AWS).
5. **Digital Commerce cacheable APIs**extend the Shared Catalog to digital channels, allowing product browsing, selection and configuration for anonymous and logged-in users.

Digital Commerce provides the ability to retrieve products and promotions that are modeled within the Shared Catalog. The specified catalog code identifies the catalog that contains offers (products and promotions). You can optionally specify a page size to limit the number of results returned.  
  
You can also promote featured products and promotions that are modeled within the Shared Catalog. The API is a variation of Get Offers by Catalog. The specified catalog code in the request identifies the catalog that contains offers (both products and promotions) that are identified as best or featured offers.

In addition, the solution allows you to retrieve offer details that are modeled within the Shared Catalog. Within the context of a catalog, the response returns product and promotional-level details, including attribute configurations and pricing information.

Products are added to a virtual "cart" before an order is created. You can fully modify the contents of the cart before creating and submitting the order.

**Designing and Building a Digital Commerce Guided Selling Experience**

**Designing Guided Selling**

You can create your own guided selling experience simply by configuring the sample process. Before you begin, it’s important to consider the following questions.

* **Who are you guiding?** With user-focused design, you always begin with understanding your user. Is it an internal user, such as a customer service representative? Or is it a customer?
* **What products do you want to sell during the experience?**How should they be organized?
* **What should the selling experience feel like?**Should it feel like part of the Salesforce application using standard templates? Or does it require your company’s branding? Where will the guided selling experience begin? How will it end?

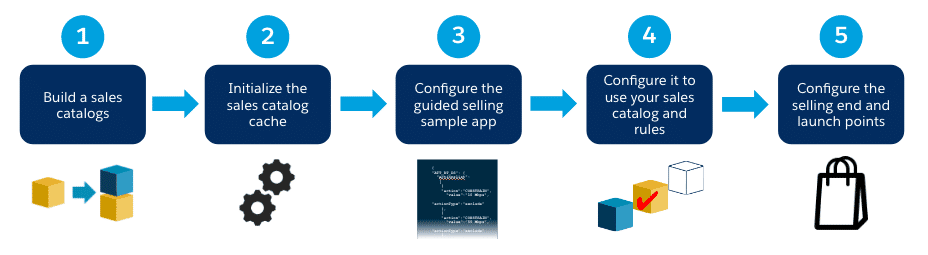


Designing a guided selling experience

**Building Digital Commerce Guided Selling**

With a design plan in hand, you can then begin configuring the guided selling experience.

1. **Build sales catalogs.**Thinking about the products you want to sell during the guided selling experience, how will you retrieve them from the shared catalog?
2. **Initialize the sales catalog cache.**Before you can configure your guided selling experience, are all of your product rules established and ready to be cached?
3. **Get data from the sales catalog.**  Have you decided which sales catalog will be featured first in your selling experience?
4. **Modify reference app parameters.**How many products should load per page upon launch of the selling experience? Do customers need to be qualified? If so, by what context dimension?
5. **Configure the script end and launch points.** You need to configure where the user goes to after completing the selling experience, and also a way to launch the guided selling script.

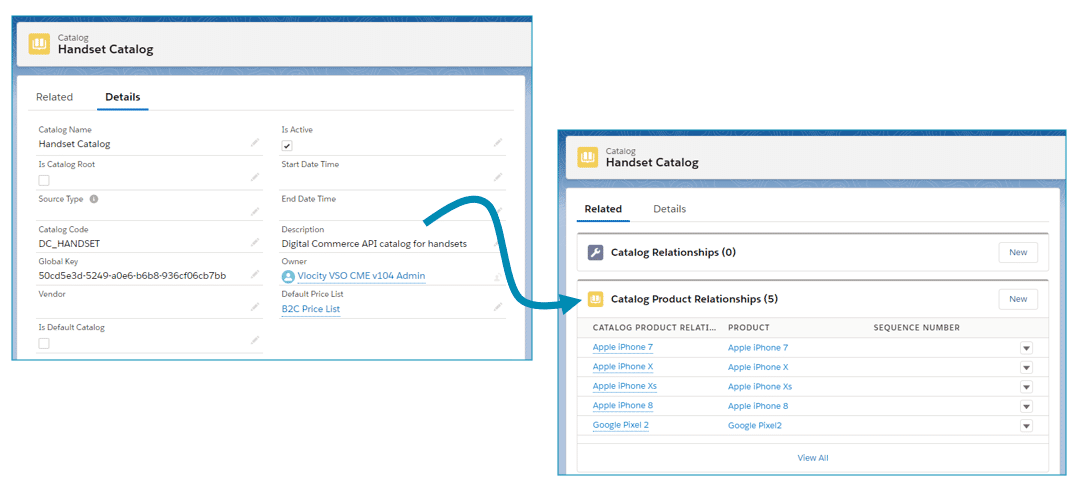


Building a Digital Commerce guided selling experience

# Using Sales Catalogs in Digital Commerce

**Sales catalogs** are used to organize groups of selected products and promotions from the Shared Catalog into digital commerce offerings. Sales catalogs are ideal for digital commerce sites because they can be curated for specific product marketing strategies.

Sales catalogs are comprised of **Catalog Product Relationships** that define the products and promotions that are included in the catalog.



**How to Build Sales Catalogs for Digital Commerce**

To create a new sales catalog, go to the Catalog tab, click New, and set the following fields:

* Catalog Name: Required. But the catalog name is not returned by any of the digital commerce queries, and will not display at run-time.
* Catalog Code: Must be unique and contain no spaces. You will use this code to request the catalog's products and promotions programmatically.
* Default Price List: The price list used for the products and promotions contained in the sales catalog. It's important to ensure that each product or promotion in the sales catalog is listed in the specified price list.
* Is Active: This flag must be checked.

When using sales catalogs in digital commerce, the catalog hierarchy (Catalog Root) is not used, nor are the Start Date or End Date times.

**Creating a Catalog Product Relationship**

Once you've created your sales catalog, create **Catalog Product Relationships** to define which products or promotions should be included in the catalog. Be sure to create separate catalog product relationships for each product or promotion.

* **Catalog Product Relationship Name:**Required. But the catalog product relationship name is not returned by any of the digital commerce queries, and thus, will not display at run-time.
* **Catalog:** This will be set for you.
* **Product**: Ensure that this product is on the price list specified in the sales catalog.

-**OR-**

* **Promotion**: Ensure that this promotion is on the price list specified in the sales catalog.
* **Is Active**: This flag must be checked.

The Effective Date and End Date fields are not used. Sequence can be used in some cases.

**Note: All products and promotions in a sales catalog must be Active. Also, any changes to product bundles require running the Product Hierarchy Maintenance batch job--just like always!**

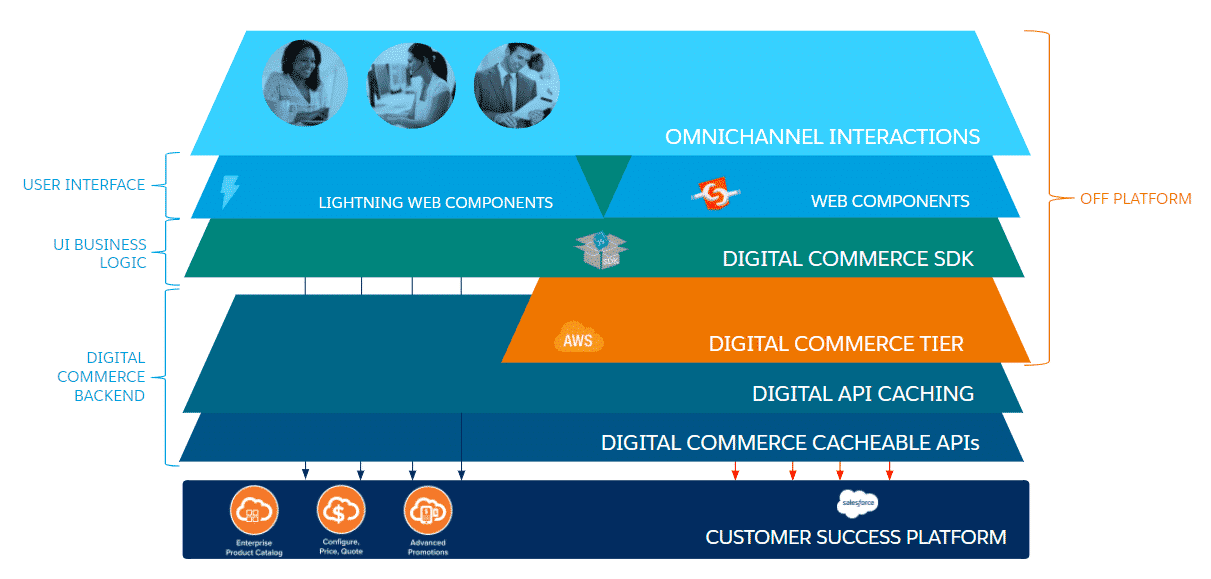
# Sales Catalogs and the Digital Commerce API Cache

After you've created a sales catalog, there's still one more step before you can begin using it with the Digital Commerce solution. You need to push the products in your catalog to the digital commerce cache, essentially prepopulating the cache. Let's take a moment to understand the cache and some of the administrative tasks that surround it.

**What's the Digital Commerce API Cache again?**

In the digital commerce world, a large number of interactions are anonymous or nearly anonymous. The early stage of browsing catalogs and products lends itself to caching since many users receive the same responses. Therefore, intelligent caching strategies and cacheable APIs are critical in any enterprise-scale digital commerce solution.

Consequently, Digital Commerce Solution includes a new caching mechanism called the Digital Commerce API Cache layer. When requests are made by users, Digital Commerce first refers to the Digital Commerce API Cache before making a new request. This cache stores pre-populated API responses inside Salesforce for on-platform solutions. In addition, the optional Digital Commerce Tier extends this caching layer to AWS for off-platform solutions. This caching design results in extremely high-performance response times and supports the high-volume anonymous and nearly-anonymous browsing traffic that is common on consumer-facing websites.



**Working with the Digital Commerce API Cache**

Catalog administrators have several cache management tasks that they must do to keep the cache up-to-date for digital commerce.

1. **Initializing the cache using the Load API MetaData job**. This job is run when first setting up Digital Commerce. It initializes the API Metadata sObject, which is a custom object that stores the metadata required by Digital Commerce APIs.
2. **Populating the cache using the Populate API Cache job**. This job populates the cache with products, product bundles, and promotions defined in the sales catalogs. In addition, context rules combinations, prices, offer details, and other data is also pre-populated and stored to ensure fast response times.
3. **Regenerating the cache.** When changes are made to any of the catalogs, products or promotions used in digital commerce, the cache must be regenerated.

**Initializing the Cache**

When you are first setting up the Digital Commerce solution, you need to initialize the cache by running the **LOAD API METADATA** job. This job will create the necessary API metadata records needed by Digital Commerce in the API Metdata sObject. This job is located in the **Vlocity CMT Administration** tab. This job only needs to be run the first time you begin using the cacheable APIs.

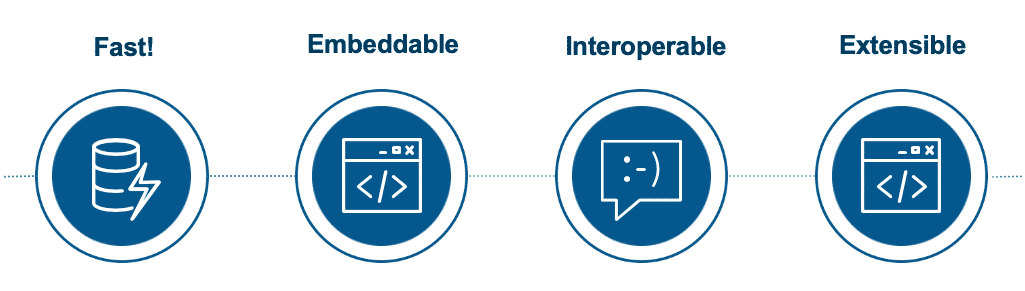
**Populating the Cache**

After you've created sales catalogs and initialized the cache, you populate the cache by running the **POPULATE API CACHE** job. This job is also located in the **Vlocity CMT Administration** tab

# Digital Commerce Lightning Web Components Overview

**What are Lightning Web Components?**

Web components utilize the latest in web standards while freeing you from Javascript framework abstractions. However, when you are building a completely on-platform Digital Commerce experience, you will want to utilize Lightning web components.  
  
**Lightning Web Components (LWC)** is the programming model for web components that run inside Salesforce.  This model implements the web component standard but uses select Salesforce libraries and packaging conventions. **Lightning** refers to a Salesforce framework that uses Javascript in the browser rather than server-side code to render the UI.



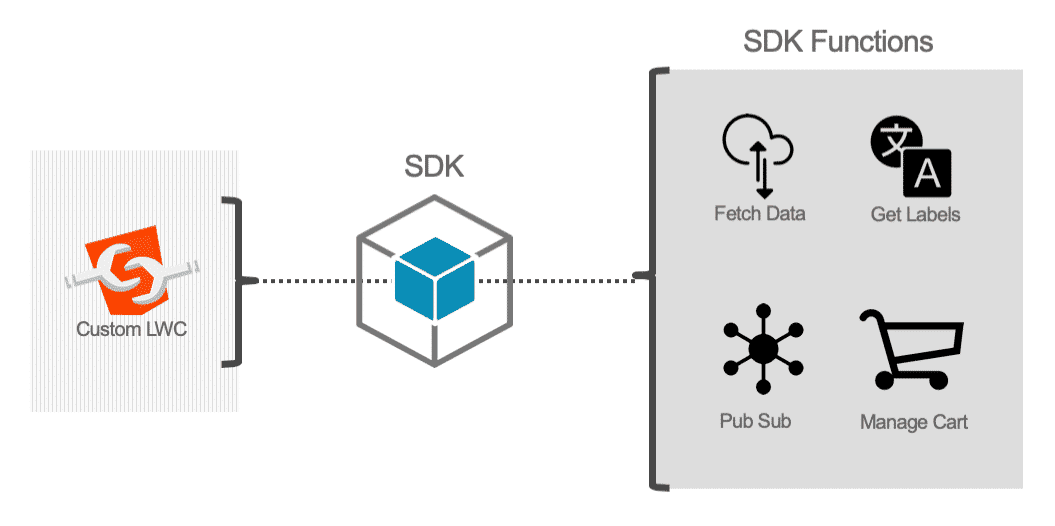
Lightning web components are the newest components available for on-platform experiences, and have many strengths—they have fast performance, are easily embedded in pages, and have deep extensibility.

Lightning web components provide only what’s necessary to perform well in browsers supported by Salesforce. Because they're built on code that runs natively in browsers, Lightning web components are lightweight and deliver exceptional performance.

**What are Digital Commerce Lightning Web Components?**

**Digital Commerce Lightning web components**refer to Lightning web components that have been specifically created to build Digital Commerce experiences (e.g., dcCatalog to render a given catalog list). Salesforce has created many Digital Commerce Lightning web components that can be used and extended, and are available within the managed package.

**Digital Commerce Lightning Web Components and SDK**

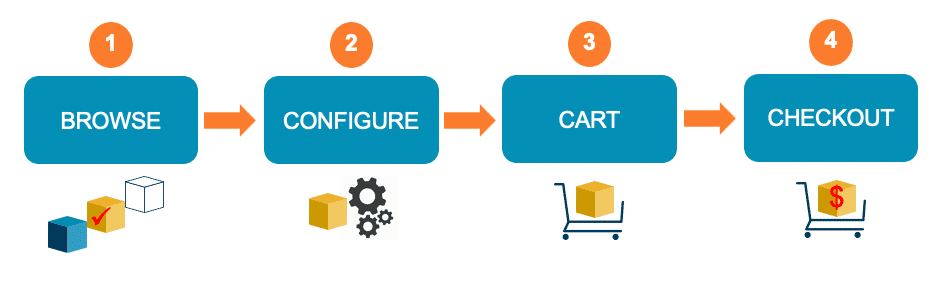


Using the Salesforce LWC standard, a library of Digital Commerce Lightning web components makes developing digital commerce applications faster and easier. Digital Commerce Lighting web components utilize the Digital Commerce SDK to perform a variety of functions. By using Digital Commerce Lightning web components and the Digital Commerce SDK together, you will be able to achieve a wealth of functionality and extensibility that is reusable, safe, and maintainable.

For example, you can put custom JavaScript functions in an extended Digital Commerce Lightning web component, and it will use the SDK to do things like deleting products from the cart.

# The Digital Commerce User Experience Model

There are four distinct phases of a digital commerce experience. Digital Commerce Lightning web components support the same four phases, which are **Browse**, **Configure**, **Cart**, and **Checkout**.



## *Browse Phase*

First, users browse the available offers in the catalog. This is the **Browse** phase.

The Digital Commerce solution supports this phase by giving you home page and catalog Lightning web components to retrieve and display catalog offers and individual offer details.

## *Configure Phase*

Second, users may select an offer and configure it, which may change the pricing and availability, we'll call this the **Configure**phase.  
Digital Commerce Lightning web components support this phase by offering ways to render offer pricing, colors, other attributes, and media. In this phase, the virtual "basket" is not shown to the user.

## *Cart Phase*

Third, users may add the configured offer to their cart (a container of saved line items), and we'll call this the **Cart** phase. The Digital Commerce Lightning web components here allows the user these functions as well as render the cart (virtual "basket") itself.  
The cart will show the total price and allow the user to proceed to checkout (i.e., initialize the checkout process by creating an order from the cart's contents).

## *Checkout Phase*

Finally, when the user is ready to review their order, enter or retrieve shipping, billing, and payment information, and finally submit their cart so that it may become an order, this is the **Checkout** phase.

The **Checkout** phase consists of the following sub-phases:

* 1. Login: identify user. (applicable if user has not already logged in)
  2. User Info: review cart, enter shipping info, and utilize payment gateway.
  3. Order Submission: create Salesforce order.

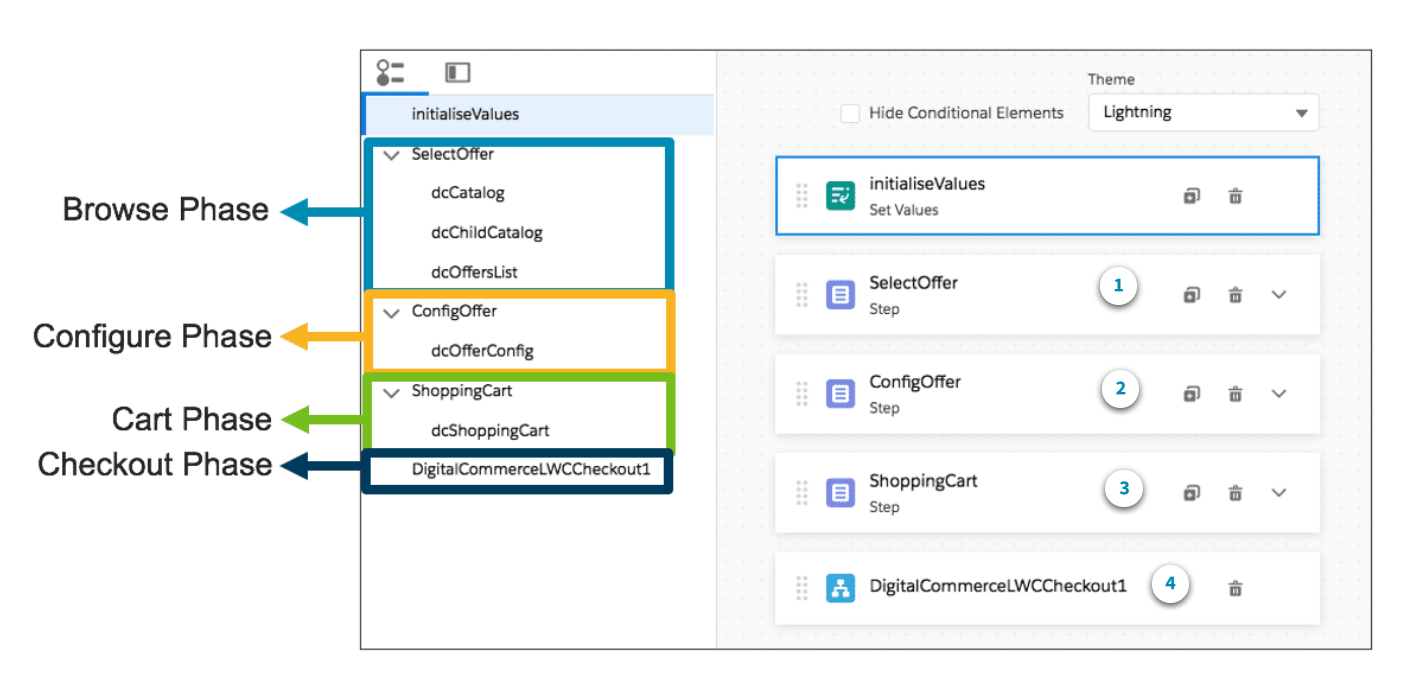
Digital Commerce Lightning web components supports OmniScript integration for easy implementation of checkout flows and Salesforce Industries Process Libraries.

# What is the Digital Commerce Lightning Web Component Sample App?

LWC OmniScripts enable you to enhance an existing OmniScript element or add custom functionality to the OmniScript through the use of custom Lightning web components.  
  
Use Digital Commerce LWC OmniScripts when your digital commerce app is on-platform and your audience is known or are guest users.

While it is possible to switch an OmniScript to either an LWC OmniScript or an Angular OmniScript at any time, Angular OmniScripts and LWC OmniScripts do not support the same components and functionalities.

**Digital Commerce LWC OmniScript Sample App Overview**



While this is a look behind the scenes of the app, let's first think about the overall user experience.  
  
First, users browse the available offers in the catalog. This is the **Browse** **phase**. The Digital Commerce solution supports this phase by giving you home page and catalog Lightning web components to retrieve and display catalog offers and individual offer details.  
  
Second, users may select an offer and configure it, which may change the pricing and availability, this is the **Configure** **phase**. Digital Commerce Lightning web components support this phase by offering ways to render offer pricing, colors, other attributes, and media. In this phase, the virtual "basket" is not shown to the user.  
  
Third, users may add the configured offer to their cart (a container of saved line items), and we'll call this the **Cart** **phase**. The Lightning web components here allow the user these functions as well as render the cart itself. The cart will show the total price and allow the user to proceed to checkout.

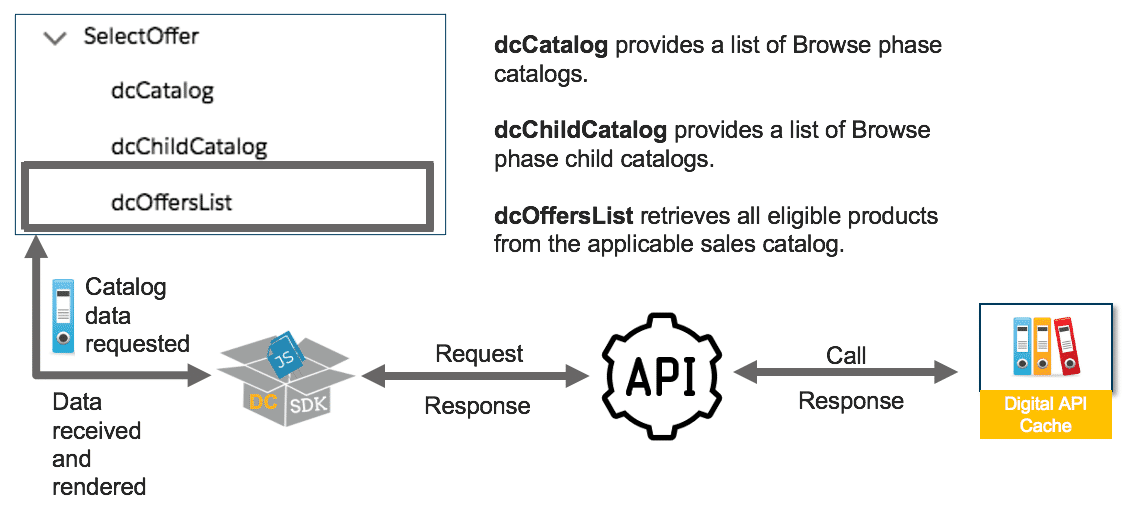
Finally, users will go through the checkout process

**Digital Commerce Sample App Lightning Web Components**

Now let's take a look at the Lightning web components that power the Browse, Configure, and Cart phases of the app.

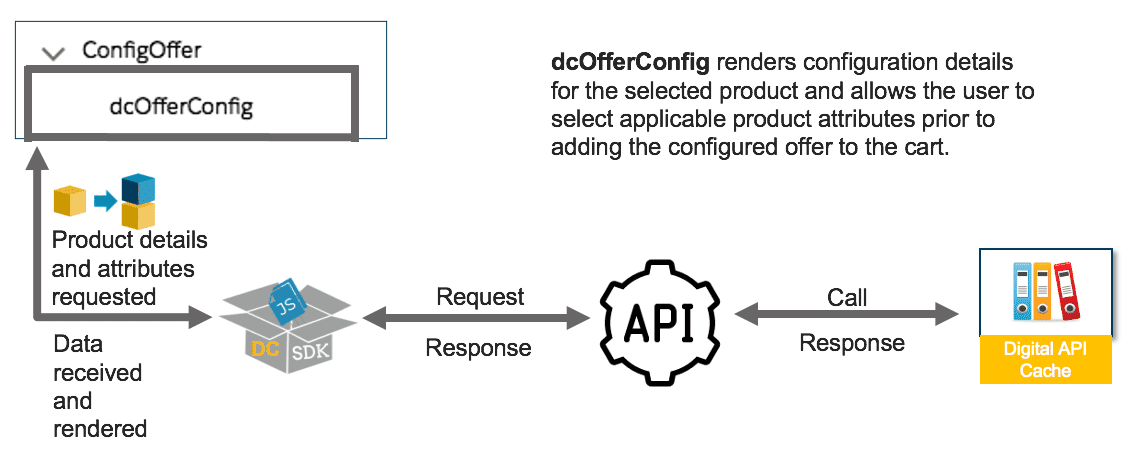
1. The Digital Commerce Lightning web components used in the Browse phase of the sample app

There are three components that make up the Browse phase. Each component renders certain parts of the user experience. **dcCatalog** and **dcChildCatalog** components render the user interface from their respective templates. **dcOffersList** is the only component in this phase to invoke the Digital Commerce SDK and APIs in order to retrieve cached sales catalog products from the Digital API Cache. The APIs then send the response back to the SDK, and the SDK parses the response and provides it back to the **dcOffersList** component, to render it for the user.

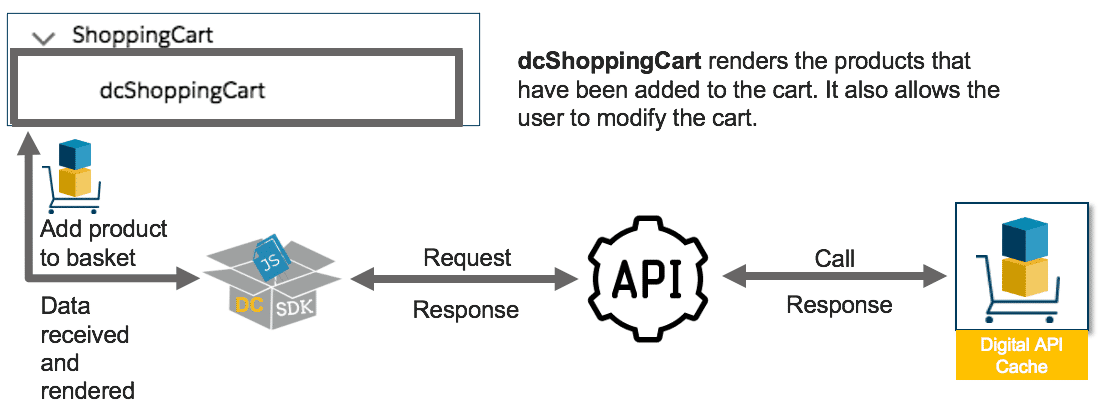


1. The Digital Commerce Lightning web components used in the Configure phase of the sample app

**dcOfferConfig** is the Lightning web component used in the Configure phase. It calls upon the Digital Commerce tech stack to retrieve product details and attributes so that the user can configure the product.



1. The Digital Commerce Lightning web components used in the Cart phase of the sample app

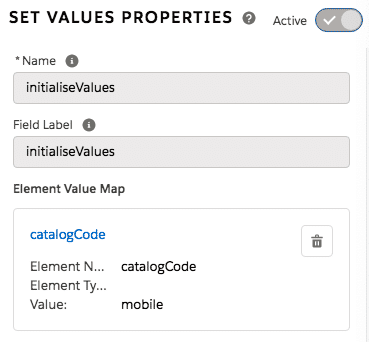
**dcShoppingCart** is the Lightning web component used in the Cart phase. It renders the products that have been added to the cart. It also allows the user to modify the cart.

# The Set Values OmniScript Component

The Set Values OmniScript component allows you to add key/value pairs to the OmniScript’s JSON, from which we read and write values for the OmniScript components.

**The initaliseValues Set Values OmniScript Component**

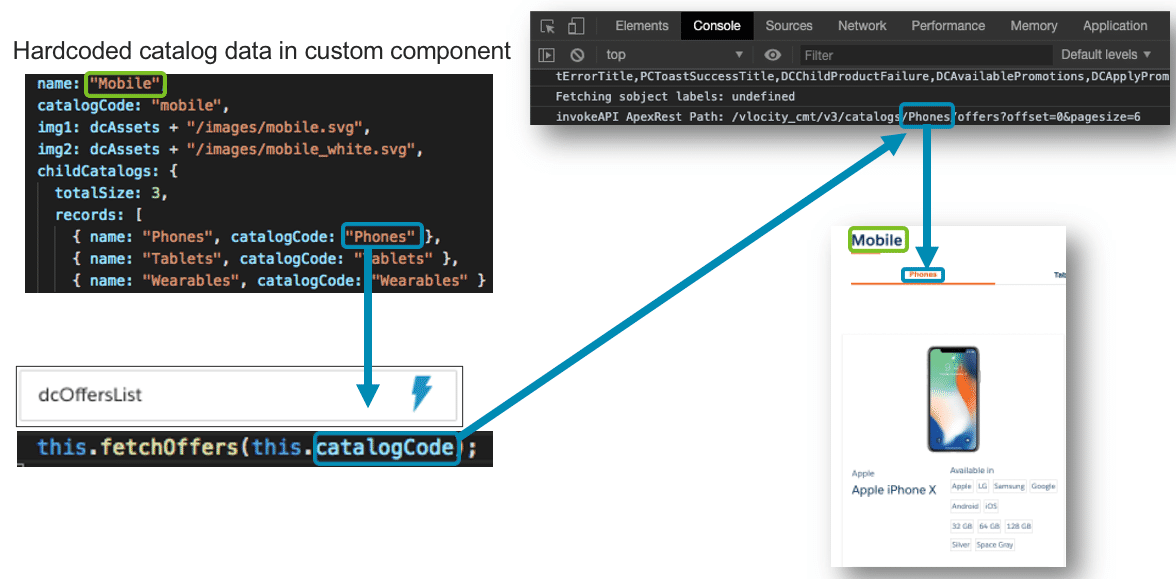
By establishing a **catalogCode** value at the beginning of the sample app, you can pass that value to your subsequent **dcCatalog** Lightning web component. The **dcCatalog** component will then render the UI. Because **dcCatalog**is expecting a catalogCode, you cannot leave this blank, otherwise, it will not render the UI.



The Set Values component designating the default selected parent catalog.

Do not confuse the parent catalogCode with the child catalogCode. The child catalogCodes are used by **dcChildCatalog** and **dcOfferList** to render the sales catalog product. The entirety of this UI is hardcoded in the various Digital Commerce sample app Lightning web components’ code and is effectively invisible to us.

During app development, a developer will create a custom Lightning web component that builds off of the out of the box **dcCatalog** to hardcode the sales catalogs that you plan on using. In this scenario, you will not be modifying catalog codes in OmniScript Designer.



How the dcOffersList Lightning web component works

In this image, you can see the hardcoded parent catalog **Mobile** outlined in green, and the child sales catalog code **Phones** outlined in blue. **dcOffersList** uses the child catalog code to call the APIs and get a response with the appropriate products. The parent catalog name is simply rendered on the page with no API calls necessary.

For the sake of seeing how **dcOffersList** works, we will temporarily override it to see how it can utilize a different sales catalog than the ones it is hardcoded for.

# Digital Commerce Cacheable API Parameters

Cacheable API calls can be modified with many different **parameters**. This allows you to customize the functionality of your guided selling experience.  
  
The accordion below shows off just a few of the possible parameters that can be used with the**Get Offers** cacheable API.

**No parameters**

This first API call shown here simply returns a list of offers for the “Phones” sales catalog. This API call has no parameters applied to it.



**Specify a context**

The second API call shown here appends a context, in this case it is specifying the context dimension of “Status” and the value of “Active”. In other words, it will retrieve all offers with a status of active.



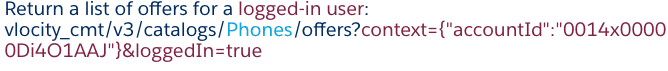
**Specify a page size**

The third API call shown here returns a list of offers with page size of 20. This means that up to 20 offers will load on the user’s initial page load. If more than 20 offers exist in the “Phones” catalog, the user will need to click a button to load the rest.



**Specify a particular logged-in user**

The fourth API call shown here retrieves a list of offers specific to a certain user.



# How are Context Rules used in Digital Commerce?

Context rules are applied to products and promotions in Digital Commerce very similarly to that of the Cart, just with a few extra context dimension fields to specify. The key addition is that **context** **rules can now be cached** in correspondence to the products and promotions available in active sales catalogs. For instance, if you wanted to create a catalog with region-specific 5G data plans and products, you could define context rules for that region so that only customers from a specific region are eligible to browse and purchase those digital commerce products.

**Creating Cacheable Context Dimensions**

Extra fields to specify in order to cache context dimensions

**Cacheable Mode**: when this option is checked, cacheable jobs (i.e., Populate API Cache) will utilize this context dimension. If this option is not selected, the context dimension will not be cached, and therefore not available to Digital Commerce APIs.

**Default Value**: this is the value that will be used for the context dimension (and passed to the context rules) when the input context parameters do not have a given value. The value is used in case of tightest match for price list entries. If this field is left empty, you will receive an error message when you run cacheable API jobs.

**Values for caching**: this is the list of possible values for this dimension. For each of these values, the cacheable jobs (e.g., Populate API Cache) will cache a response.

During runtime, if the input context parameter is passed a value not present in this list, the CPQ engine will evaluate the response on the fly and then cache the result.

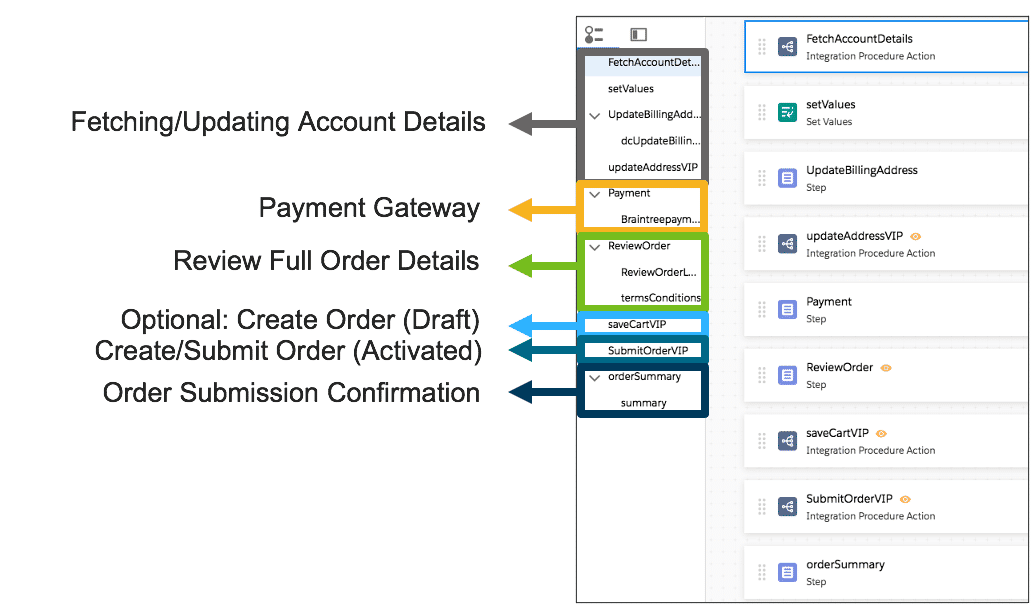
**Extending Context Rules for Digital Commerce**

When API calls are made by users, the cacheable APIs will first check the API response cache before processing the request. If a response to the user’s request is in the cache, the response is returned to the user. Because the API response cache is created for all permutations of eligible products (as defined by your context rules), you will need to make sure that you regenerate the catalog cache after you implement new rules.

**Checkout with Digital Commerce and OmniScript**

Remember the very last component of the Sample App OmniScript, **DigitalCommerceLWCCheckout1**? It is a separate OmniScript being invoked to complete the end of the guided selling experience. Let’s take a look.

**The Digital Commerce Sample App Checkout**

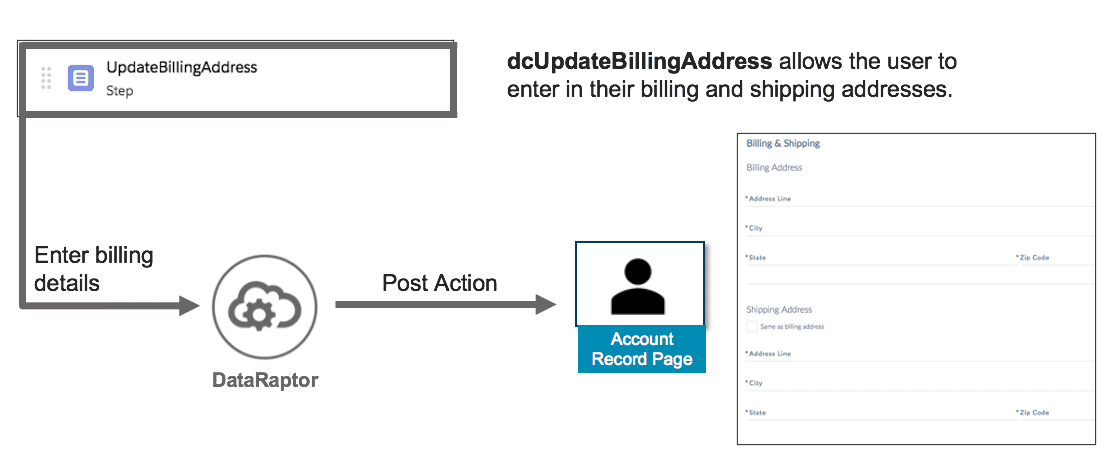


Checkout is made up of several Integration Procedures and Lightning web components.  
  
In terms of the guided selling flow, checkout is where the user will need to create an account or log in. The creation and fetching of account details are handled through Integration Procedures and Lightning web components.

Following this, the user will need to enter their payment info. For the sample app, we are using Brain Tree Payment, which is featured in the Salesforce Industries Process Library.

**The Digital Commerce Sample App Checkout Lightning Web Components**

First, we'll cover the Lightning web components (contained in the steps numbered 1, 2, and 3 in the image above) used in this OmniScript. Second, we'll take a look at the Integration Procedures, [DataRaptors](https://docs.vlocity.com/en/DataRaptor.html), and other components used in the OmniScript.

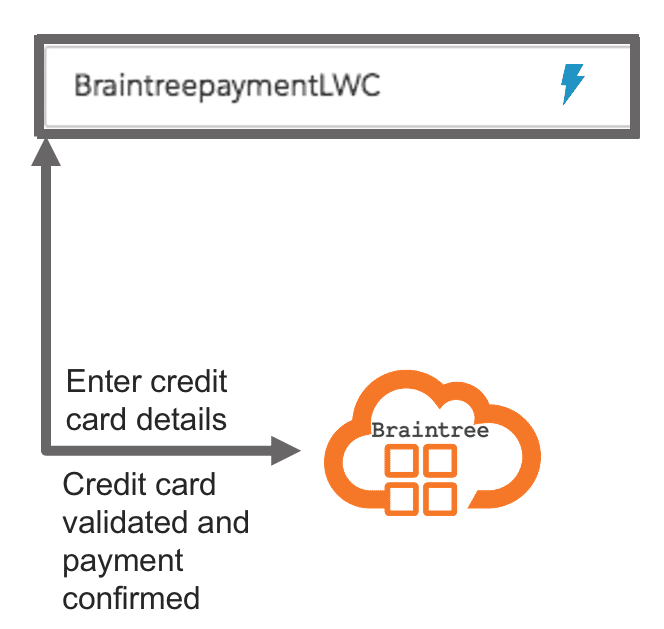


This Lightning web component allows the user to enter in their billing and shipping addresses.

**dcUpdateBillingAddress** allows the user to enter in their billing and shipping addresses. These fields are updated on the customer’s record page via Integration Procedures and DataRaptors that we’ll cover further down this page.

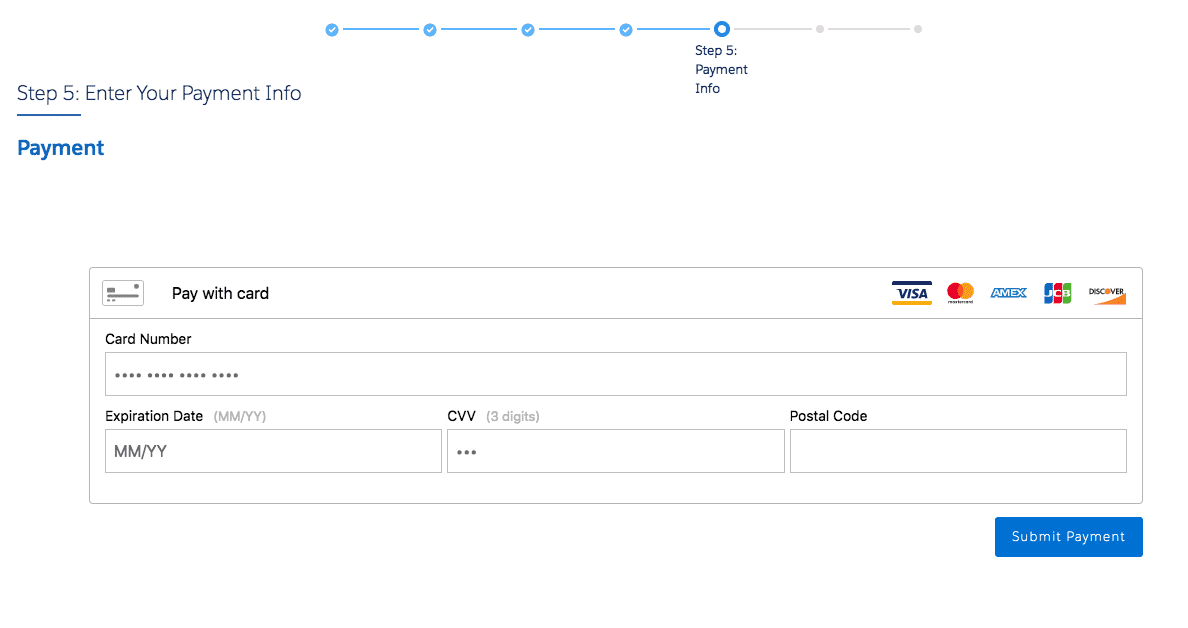


Screenshot of dcUpdateBillingAddress rendered in the sample app

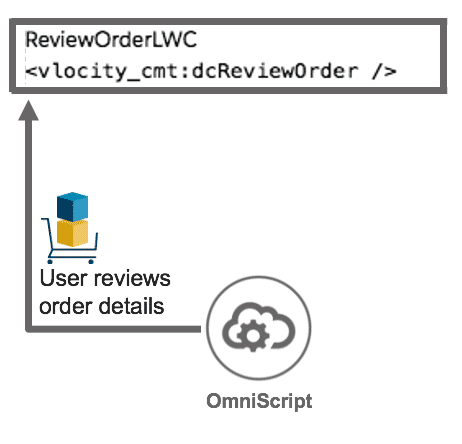


This Lightning web component allows the user to enter payment details and submit them with an off-platform payment gateway.

**dcCheckoutPayment** allows the user to enter payment details and submit them with an off-platform payment gateway. It utilizes the payment URL specified in the parent OmniScript’s Set Values component to render it in an iFrame. It uses the total due today value from dcShoppingCart.

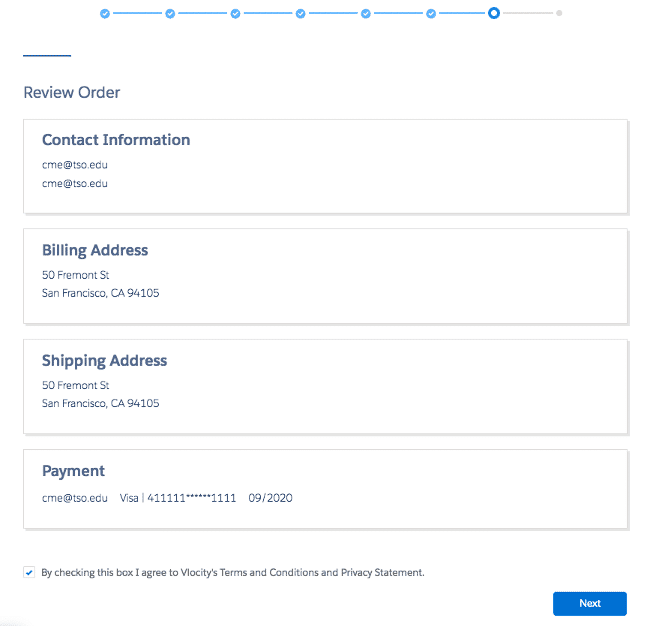


Screenshot of dcCheckoutPayment rendered in the sample app



This Lightning web component allows the user to review their contact information, billing/shipping address, and payment information, prior to submitting their order.

**dcReviewOrder** allows the user to review their contact information, billing/shipping address, and payment information, prior to submitting their order.

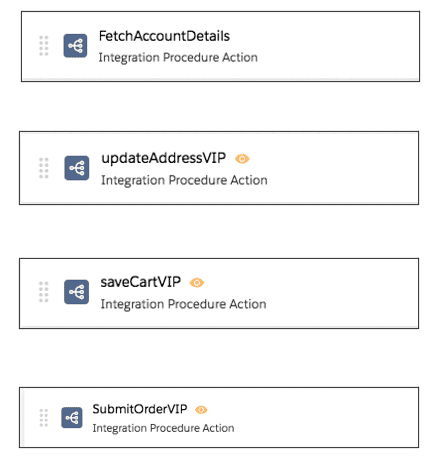


Screenshot of dcReviewOrder rendered in the sample app

# Checkout Integration Procedures and DataRaptors

Now that we've seen the Lightning web components being used in the Checkout OmniScript, let's take a look at the remaining Integration Procedures, DataRaptors, and other components that make it work.

**The Integration Procedures of the Checkout OmniScript**



The four Integration Procedures used by the Checkout OmniScript

Let’s take a high level look at the four Integration Procedures the Checkout OmniScript uses.

**FetchAccountDetails**

First in this checkout flow, we need to handle user authentication. This means either creating a new account or fetching user details for a known user. **FetchAccountDetails** accomplishes this through use of DataRaptors.  
  
Let's take a deeper look at new account creation.  
  
**CreateAccount** is a DataRaptor post action, and it is used to create accounts. You can invoke it as a DataRaptor Post Action component in the OmniScript. Similar to when you create an account manually, CreateAccount requires an account name and the type of record (billing, consumer, service, etc). We are additionally specifying the billing email, phone, and status. When given all of these things in the correct format, createAccount will return the newly created account id.

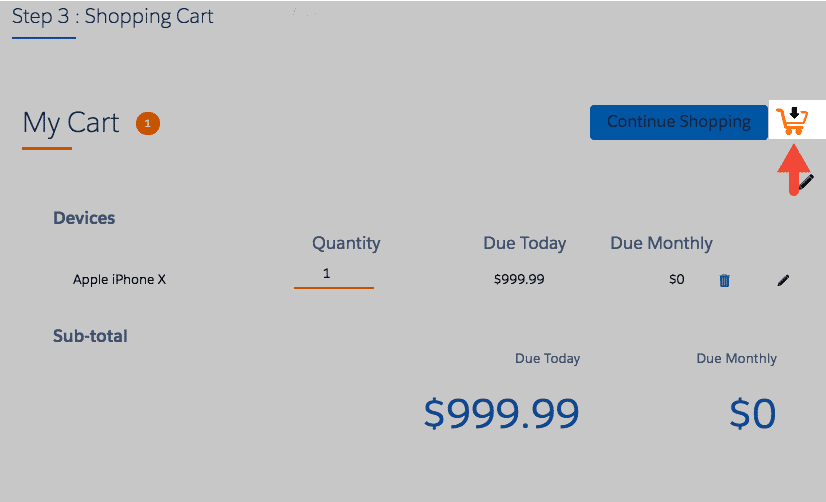


**updateAddressVIP**

Next in the flow, the user needs to enter in their billing details and we have to make sure it is updated in their account. We do this with a Lightning web component and DataRaptors in the **updateAddressVIP** Integration Procedure.

**saveCartVIP**

If the user clicks the save cart button from the cart page, the Checkout OmniScript will use the **saveCartVIP** Integration Procedure. This procedure uses an Apex remote action to create a new order for the products in the user’s cart.  
  
The image below highlights the save cart button:

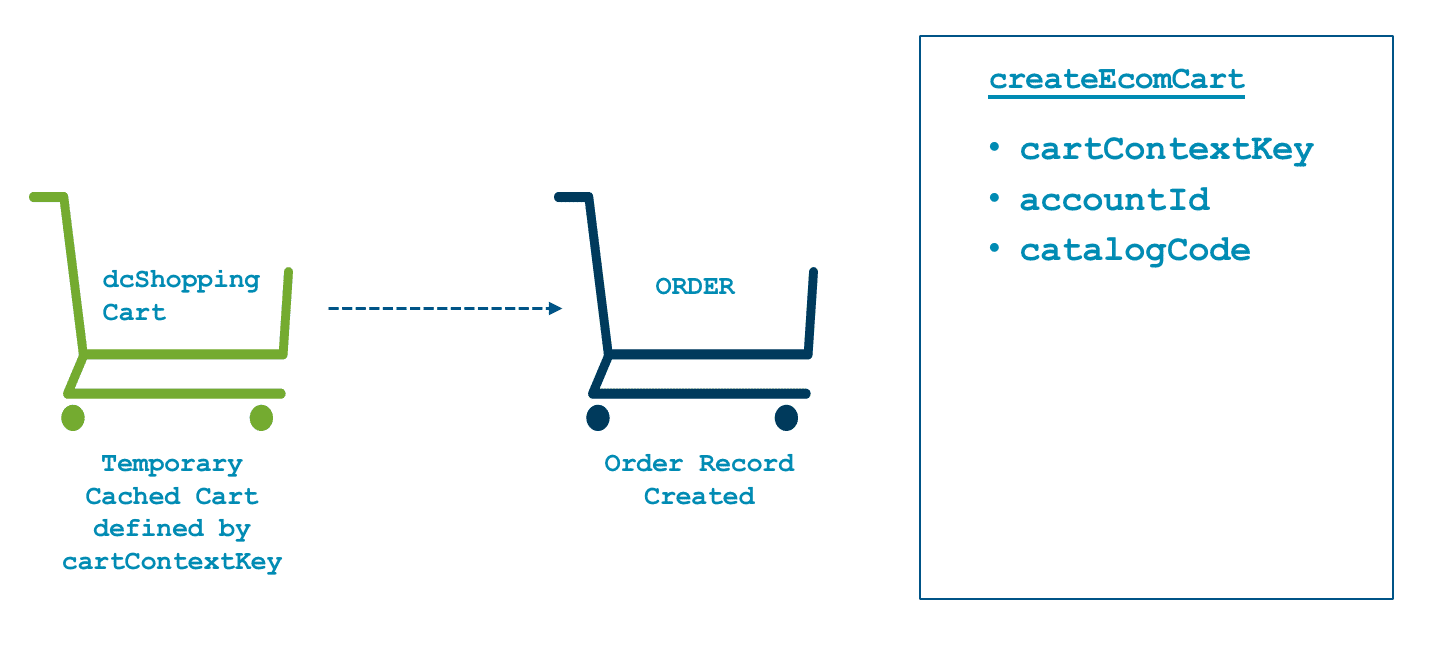


**SubmitOrderVIP**

**SubmitOrderVIP** uses the same **saveCartVIP** Integration Procedure to create an order, validates the order with the SubmitOrderService class, and then submits it with the CPQAppHandler class.

Now let's take a deeper look at the Remote Methods used by the **saveCartVIP** and **SubmitOrderVIP** Integration Procedures for order creation and submission.

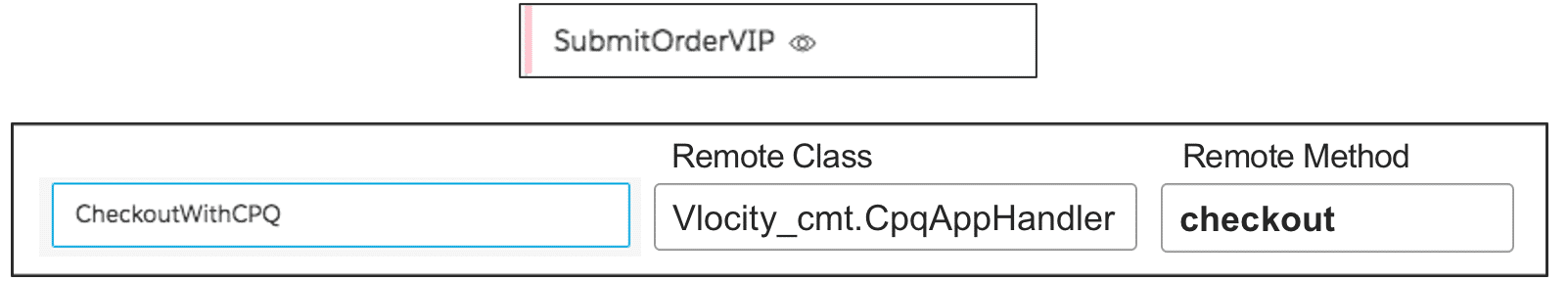
**Creating Orders**



Creating an order using the **createEcomCart** Remote Method

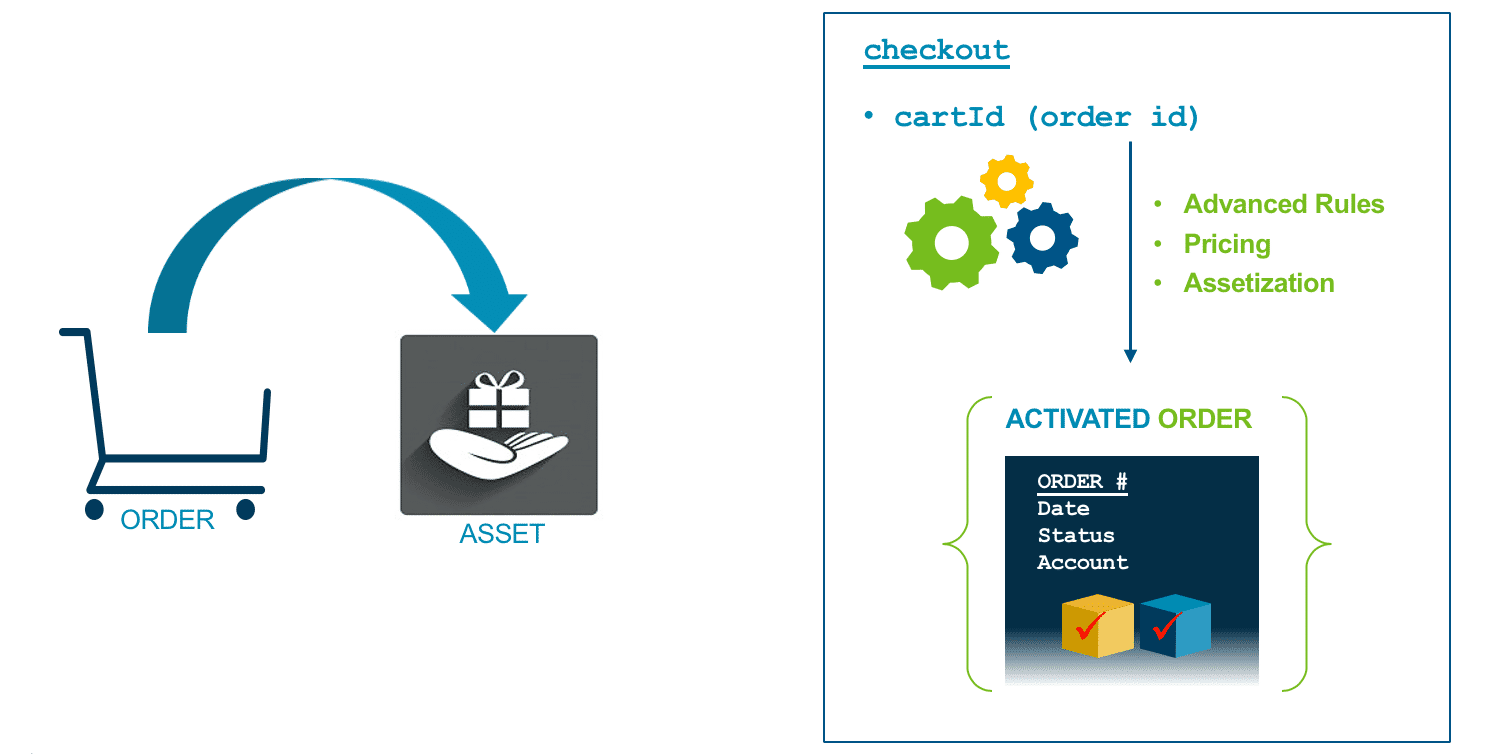
**createEcomCart** is one of CpqAppHandler’s methods. It is responsible for creating an order and it validates cart product pricing with the sales catalog pricelist.  
  
You can invoke it using a Remote Action element in the OmniScript. **createEcomCart** requires a **catalogCode**, **accountId**, and **cartContextKey**. The **cartContextKey** is a value created by **dcShoppingCart** that points to the cache representation of the user’s shopping cart. **createEcomCart** brings everything together for an accurate draft order record.

**Submitting Orders**



The checkout method as it is defined in the **SubmitOrderVIP**.

A remote action in the **SubmitOrderVIP** integrated procedure called **CheckoutwithCPQ** is used to initiate the **checkout** method. This method submits the order, moving it to the next stage.



The **checkout** remote method invokes rules and pricing before submitting and assetizing the order.

Once you’ve retrieved products, displayed them to the user, added them to the cart, there’s one final step you don’t want to forget—submitting the order! The CPQ interface, once again, comes into play and provides the checkout method to ensure every order is valid and can be fulfilled.

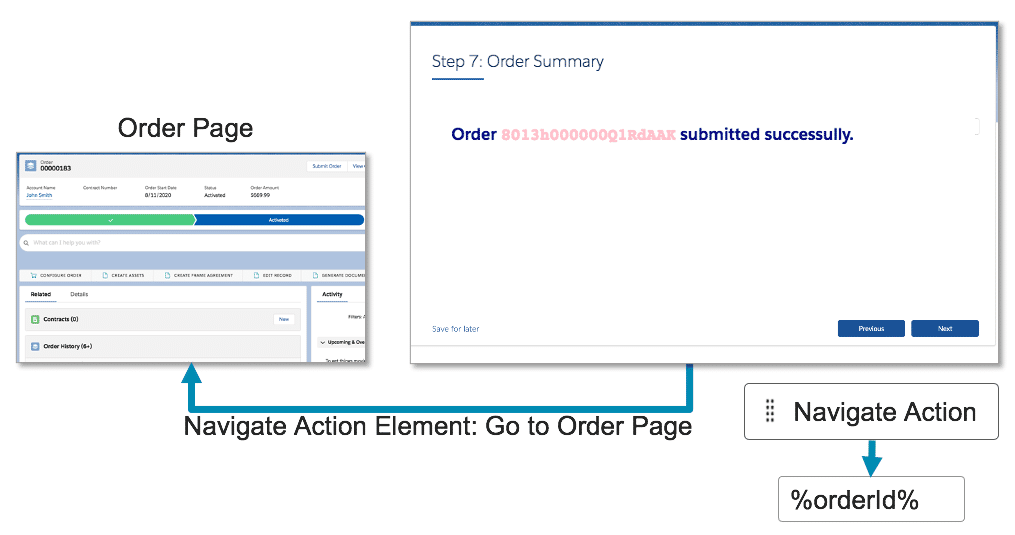
The **checkout** method requires the cartId (Order Id). Based on the header object, checkout moves the object to the next state of the ordering process. Since the header object is an **Order**, it submits the order to create assets.

When you checkout an order, rules and pricing will run one final time, and then the line items will be assetized, creating assets. The Salesforce order status will change from Draft to **Activated**.

# Guided Selling End and Launch Points

We've covered a lot so far but we're near the end! Now that we know how the sample app works, we need to create our end and launch points for it.

**The Navigate Action**



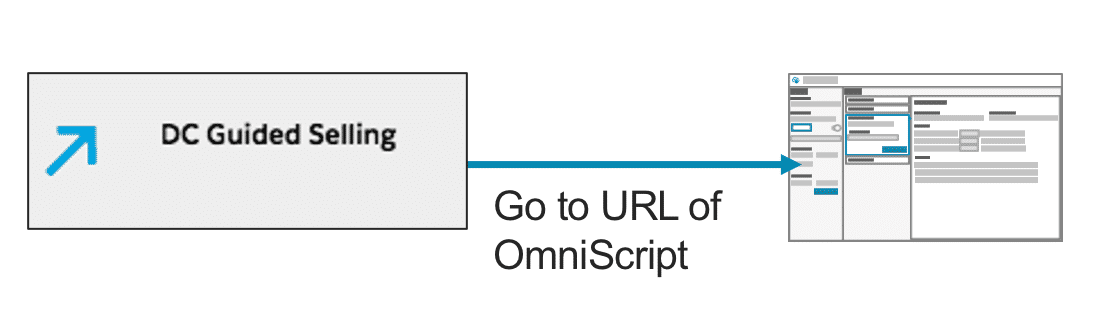
The Navigate Action Element in the OmniScript

When you’re building a guided selling experience, you need to think about where the user should logically go after completing it. For the experience we’ve been building and configuring so far, the user is going through the steps to create an order. Logically, the CSR may want to see the order after they have submitted it. This is where the Navigate action comes in.

The Navigate action can take the user to a specified app, lightning component, object page, record page, and [more](https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.reference_page_reference_type).

The Navigate action will use the %cartId% value retrieved by the **DRExtractCartId** DataRaptor Extract Action to take the user to the order page.

**OmniStudio Actions for Launch Points**

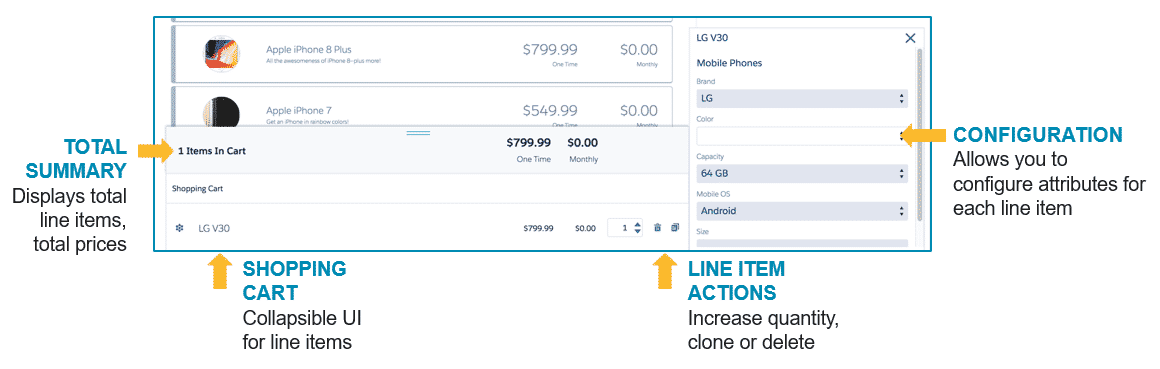


OmniStudio Actions can launch OmniScripts

Guided Selling can be launched from an OmniStudio action that can be available on an account page.  
  
OmniStudio Actions are automatically generated URLs that launch OmniScripts, OmniStudio FlexCards components, web pages, or external applications. Actions are typically specific to a given object type, such as Account, Contact, Policy, or Asset.

**Building a Cart-Based Guided Selling Experience**

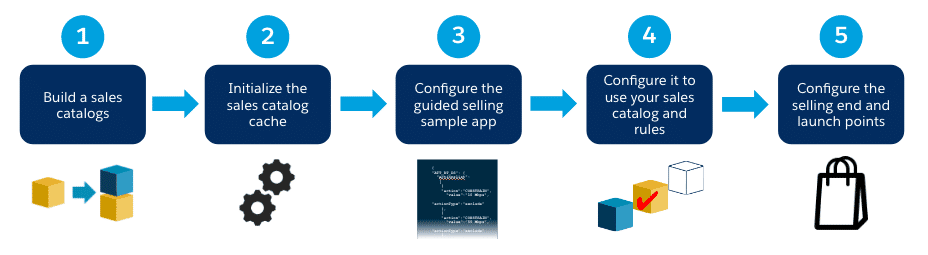
**Cart-based** guided selling experiences use cart-based APIs and Angular JS templates to feature a visually persistent cart as the user navigates through the order process. This technology works well for internal processes, with known traffic patterns and logged in users.



A cart-based guided selling experience using the OmniScript persistent cart

With a design plan in hand, you can then begin configuring the guided selling experience.

1. **Create the order header**. Before you can add products to the cart, you must create an order to hold the line items. (Or you could create a quote or an opportunity, if you’re designing a B2B guided selling experience.)
2. **Get product data from the shared catalog**. Thinking about the products you want to sell during the guided selling experience, how will you retrieve them from the shared catalog?
3. **Create product selection steps**. Once you’ve retrieved the product data from the shared catalog, how should it be organized in each step of the selling experience?
4. **Configure the script end and launch points**. You need to configure where the user goes to after completing the selling experience, and also a way to launch the guided selling script.
5. **Configure the shopping cart in the script**. What should it look like?



Building a Digital Commerce guided selling experience

**Business Logic and CpqAppHandler**

All OmniScripts require data to accomplish their purpose. Guided selling OmniScripts are no exception. They can use all the standard data components to move data in and out of the script:

* DataRaptor
* REST actions
* Remote actions

However, when you are using OmniScript in a guided selling context, you need more than just data. You could use all of the options above and retrieve a **big**pile of data. But data is not enough when it comes to guided selling.

**Guided selling OmniScripts need a big helping of business logic with a side order of data. Business logic contains the secret sauce for an effective guided selling experience.**

**Business Logic**

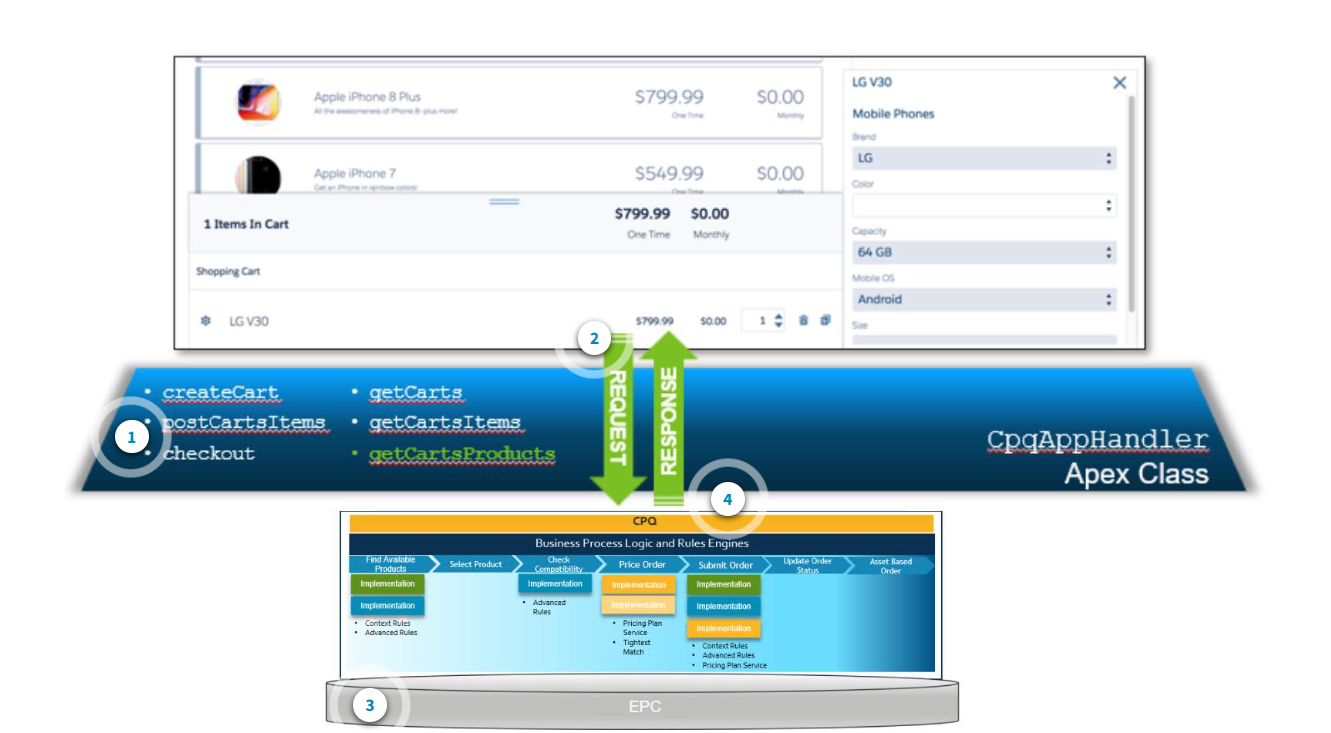
What kind of business logic? The kind that answers these important questions:

1. **What products**are available for purchase?
2. **How**should the products be configured?
3. **Which customers** can buy each product?
4. **What price** do they pay?

You need business logic and rules that govern which products are available to which customers, and at what prices. Industries CPQ defines the business logic and rules. To access product and pricing data using CPQ, you must request the data using a CPQ interface. A CPQ interface guarantees that the correct logic is applied, making sure that every order is a perfect order, whether using the Cart or a guided selling experience.

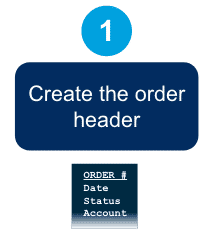
In fact, the Cart and guided selling OmniScripts both use **CpqAppHandler**. CpqAppHandler is a global Apex class that includes several methods to perform the CPQ functionality. These methods are available as a RESTful API, called the Cart-based APIs.

The diagram below shows a request and response from EPC to CPQ using the CpqAppHandler.



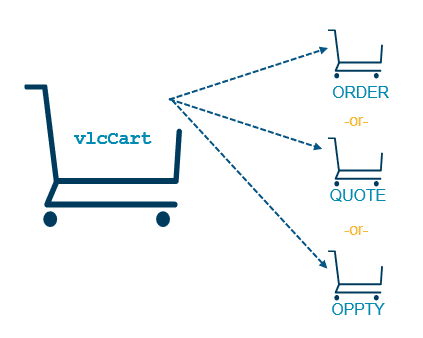
**Creating Orders with createCart Method**

**Creating the Order Header**



The first step in building a guided selling experience is to create the order header.

Before you can add products to your persistent cart (we'll explain the persistent cart later), you must create an order, quote, or opportunity to hold the line items. You can do that by calling the CPQ interface.



**Hey, CPQ, are you out there?**

You may be wondering, what does it mean to "call" an interface? It just means you are invoking some programming code to perform a procedure. Interfaces have unique names and usually require some input parameters. When you call their unique name and pass them input parameters in the format that they expect, they will happily give you a response.

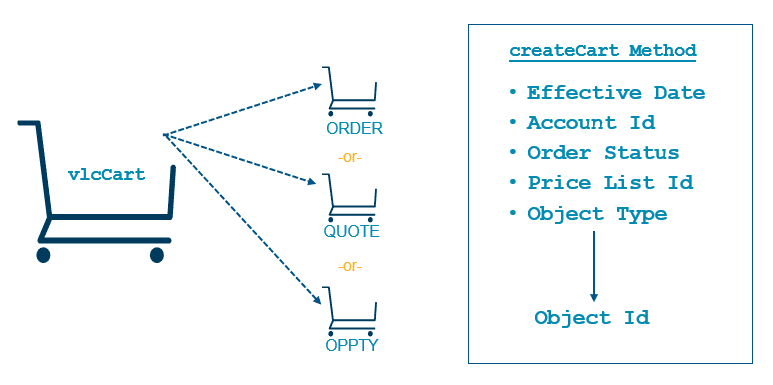


**Creating Orders with createCart Method**

createCart is one of CpqAppHandlers methods, and createCart is used to create:

* Orders
* Quotes
* Opportunities

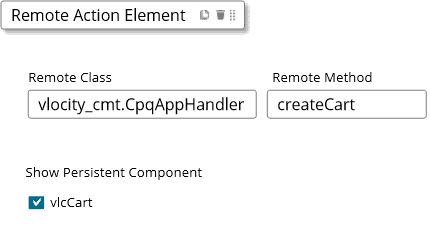
As you might expect, just as when you create an order manually, createCart requires similar information. When you invoke the createCart method, you must supply the Effective Date, Account Id, Order Status, and so forth. The method will create the order and return the Order Id to you.



**Invoking createCart using Remote Action Components**

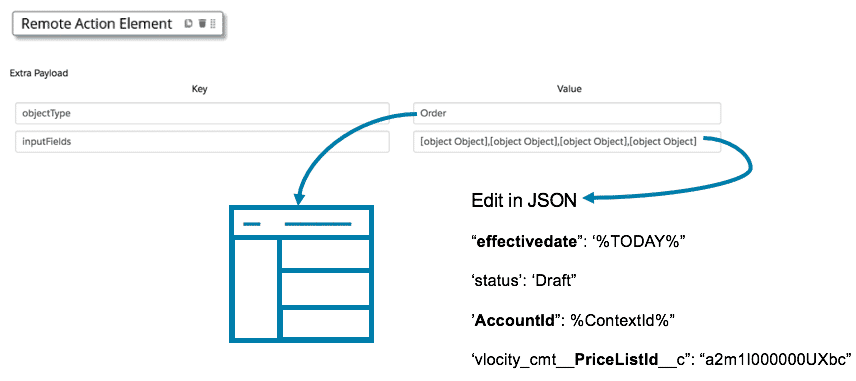
In OmniScript, you can invoke the createCart method using a remote action component.

1. Calling CreateCart
   1. Use a remote action element to invoke CpqAppHandler's remote method **createCart**. This method creates an order, quote or opportunity, depending on what input parameter it receives in the Extra Payload.



1. Input Parameters
   1. In the same remote action element, the **Extra Payload** section defines the parameters that will be passed to the **createCart** method.
   2. An order you create manually must have an account, a date, a status, and a price list. You use the inputFields parameter to pass these required fields in JSON notation, including:

* effective date
* order status
* account id
* price list id (or pricebook id)
  1. In addition, you use the objectType parameter to define what object you want to create. To create an order, you pass "Order" as the objectType parameter. If you wanted to create an opportunity or a quote as a result of the guided interaction, you would change the objectType accordingly.



**Capturing the Response**

Once createCart has been successfully invoked, it will return an object Id, either an order, quote or opportunity depending on what objectType was passed. The object Id that is returned should be shared among the other script components. In guided selling, there are several other remote actions that call CpqAppHandler, and almost all of them require a cartId parameter as an input.

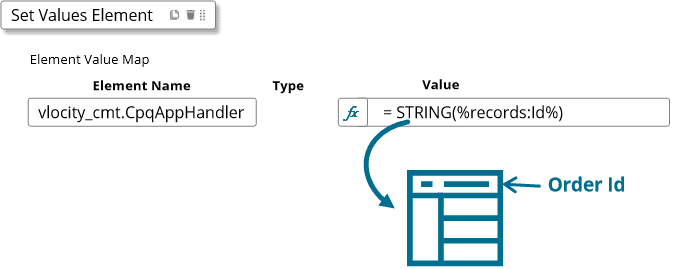
The easiest way to share something among OmniScript elements is to use a Set Values component.

SHARING THE ORDER ID

Add a Set Values component to the OmniScript to contain the Order Id. This will be a variable that can be referenced by any other script component.

All future CpqAppHandler calls need the Order Id to be able to act on the correct order. The createCart method returns this Id as a JSON array. To get the Id from the array, enter the value as a string =STRING(%records:Id%).

There is no need to check **Show Persistent Component** to display the cart because this is a setting used behind the scenes.



# Using CPQ to Get Products

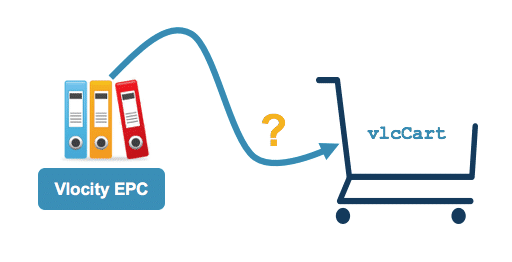


**Shared Catalog in EPC**

The second step in building a guided selling experience is to get product data from the shared catalog. The shared catalog in EPC contains a wealth of CPQ data: products, bundles, attributes, pricing, rules and more.

**Using CPQ to Get Products**

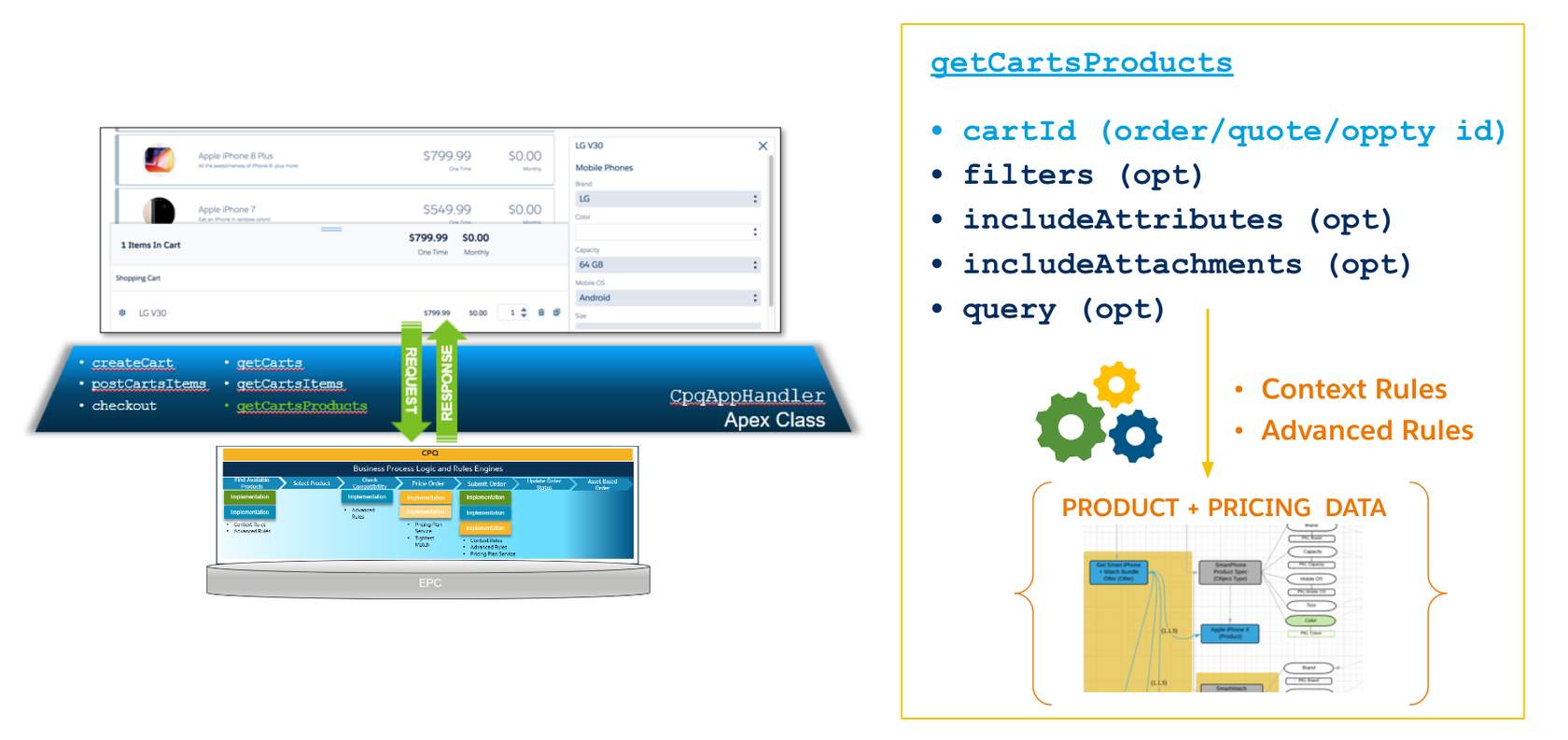
While the technology of Salesforce Industries includes many different ways to access data, the key is to ensure that only the right products are presented to the right customers at the right prices—which means that the path to success is using the CPQ application interface once again.



**Shared Catalog in EPC**

**Using getCartsProducts**

The CPQ application interface ensures the right products are shown to the right customers at the right time. CpqAppHandler knows how to retrieve products—and then some! The method getCartsProducts requires a cartId (order, quote, or opportunity id) and then returns comprehensive product information, including attributes and pricing.



**getCartsProducts** method

getCartsProducts only retrieves the products that the customer should see based upon the price list that is associated to the order (or quote or opportunity) that was passed as input. Any active context rules and advanced rules also apply. Products, of course, must also be active and within their selling period dates to be retrieved.

**Invoking getCartsProducts Using Remote Action Components**

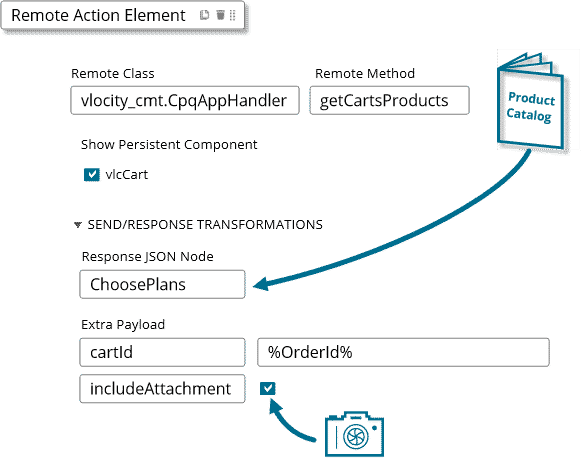
In OmniScript, you can invoke the getCartsProducts method using a remote action component.

USING GETCARTSPRODUCTS

Use a remote action element for the remote method **getCartsProducts** to retrieve product data. The cartId is a required input parameter.

In the **Send/Response Transformations**section, designate the specific JSON node where the retrieved product data should be stored in the data JSON. Later, this name will be the name of the selectable items element that will display the products. 

If you have product images that you want to display, use the includeAttachment parameter to indicate that the product images should be returned as well. 



**Getting Specific Groups of Products**

Access to tons of data can be great. But it can also be confusing and overwhelming for customers who must make a decision.



Narrowing down what is presented to the customer results in a simple, yet effective user experience. Even if you don't frequent this fast food restaurant, the straight forward menu is comforting. Order a #1, #2, or #3 meal, and that's it. You can sit back and relax.

Guided selling aims to make ordering mobile devices as easy as 1, 2, 3!

**How can you simplify the guided selling experience?**

Reduce the number of choices! Given the paradox of choice, we know that more choices means more cognitive load. So, the trick is to present just enough of the right products that you know your customer wants to buy.

Before we can discuss how you do that, you need to understand how product entities are structured and what grouping mechanisms are used in your catalog. Some deployments may use the Salesforce fields Type, SubType or Family. Others may use Salesforce Catalogs, Categories, Product Class, or other object types. So, in order to make the right choice, you must first understand how your Shared Catalog is working and what fields exist to identify the products as a group.

Once you’ve decided how to identify your product groups, the getCartsProducts method provides several parameters you can use to filter its response so the customer is only presented with the right number of products during guided selling.

**getCartsProducts Input Parameters**

**filters**

Specify a field and the values that must be matched in order to be included in the response. (This translates to an IN comparison operator in SOQL.)

The format for specifying a filter is: API\_field\_name:Value. Separate multiple values with an underscore, as follows: **filters=Product2.Name:Installation\_SmartModem**

This filter returns items with product names equal to Installation or SmartModem. You can set multiple filters using a comma, as follows: **filters=Product2.Name:Installation,Product2.Product\_Line\_\_c:DSL**

**query**

Specify a string, that if matched in the product name *or* description the product will be returned in the response. (This translates to a LIKE operator in SOQL.) **Example: query=samsung** - Return products that contain "samsung" in either the product name *or*the description.

**searchBy**

Modifies the query parameter to search fields other than Name or Description. **Example: query=samsung&searchBy=Product2.Brand\_\_c**will search for products that have "samsung" in Product Brand field.

**category**

Specify the category id. All products assigned to the category id will be returned in the response. **Example: category=01s41000005dPYQAA2** - Return all products assigned to this category id.

These parameters can be combined together to further refine the response. For example, you could have a request that combines filters with a query and a searchBy modifier like this:

**filters=Product2.Name:Installation&query=DSL&searchBy=Product2.Product\_Line\_\_c**

This would find all products with names = “Installation” and that contain the word “DSL” in the Product Line field. This is translated to the following SOQL query:

**(Product2.Name IN ('installation')) AND (Product2.Product\_Line\_\_c LIKE '%DSL%')**

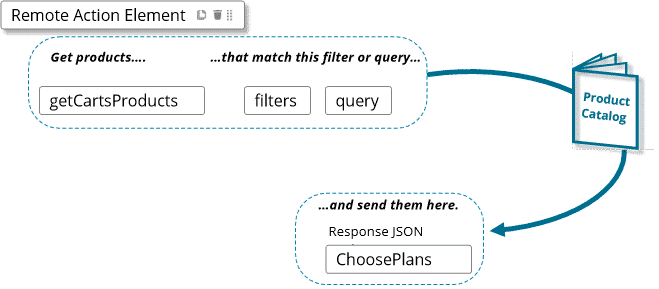
**Note:**

If you are using Context Rules you will need to do a few extra things to get it to work in the persistent cart:

* You'll need to pass the includeIneligible = true parameter to getCartsProducts. This parameter will return both qualified and disqualified products from the Context Rules engine, and importantly, it will insert a category node in the response.
* Add a modification to the selectable items template. By adding the ng-if shown below, the JavaScript in the template will filter the JSON to only display products listed under the Qualified category node. You'll learn more about the selectable items template later in the course.

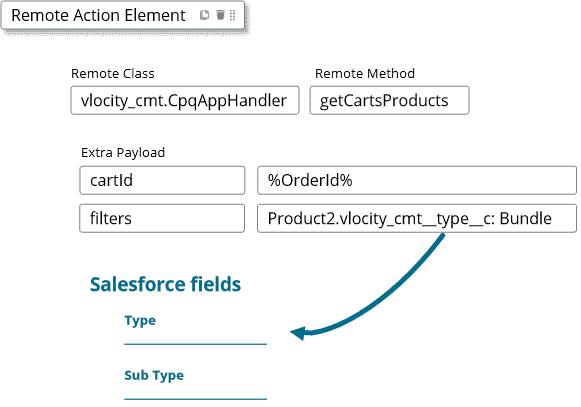


**Entering Input Parameters in the Remote Action**

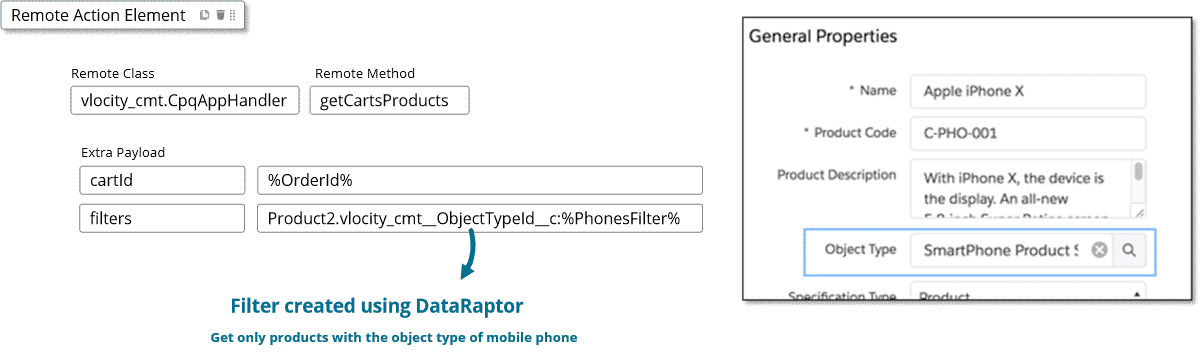


Once you've decided how you will filter getCartsProducts response, you enter the parameter and the values in the Remote Action element's **Extra Payload** section.

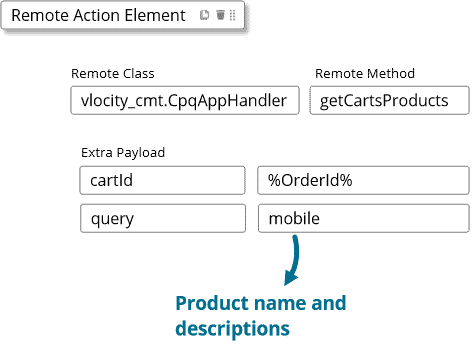
1. Filters for values
   1. You can use the filters parameter to query for products using specific values, such as in the example shown here, in which we query for any products with a type of "Bundle". This value will stay constant, even if the script is imported into another org.



1. Filters for Record IDs
   1. Sometimes you may need to search for specific records ids. For example, you may want to retrieve all smartphones with an object type of "SmartPhone Product Spec." However, because the object type is stored in the product table as an id rather than by its name, you must first get the object types record id. You can do this using DataRaptor, and then pass the object type id via a merge code as the filter parameter. Using this technique, the script can still be ported to other orgs easily.



1. Querying for Text
   1. You can also use query for products with a text string, which looks for the specific text in the product name and description. However, if you try to use the query with the text “mobile”, your results could include any product or product description with the word "mobile" in it, including mobile accessories and mobile plans.
   2. Use the searchBy modifier to search other fields than product name or description.



**Ensuring Script Portability**

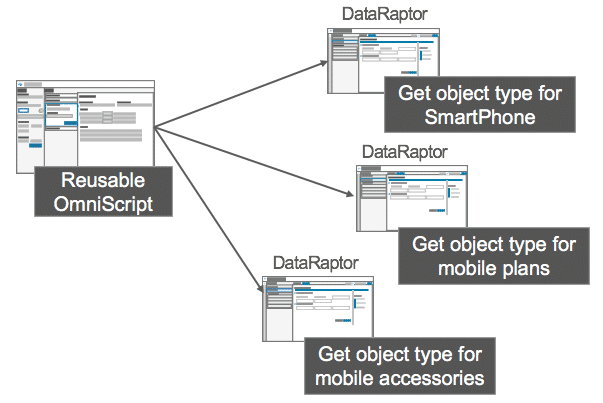
As you've seen in the previous lessons, many of the input parameters for CpqAppHandler's methods require records ids. It's one thing to pass a record id that the script just created, like an order id, but it's another thing to pass a record id for a price list or an object type.

For example, the filters parameter is formatted like this:

filters=Product2.vlocity\_cmt\_\_ObjectTypeId\_\_c: **a1y1U00000036R5QAI**

But hard coding Salesforce record ids is a horrible idea. It might work for one org, but it won't for any others!  (The record ids change between Salesforce orgs.) Telling your manager "It worked great in my test environment!" won't go over very well. Thud. Your solution needs to work in all environments (dev and test, QA, production, etc.)

OmniStudio DataRaptors and OmniScripts to the rescue!

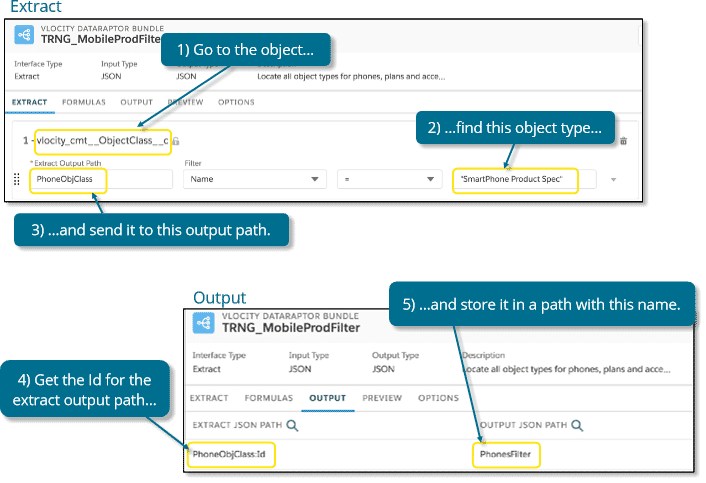


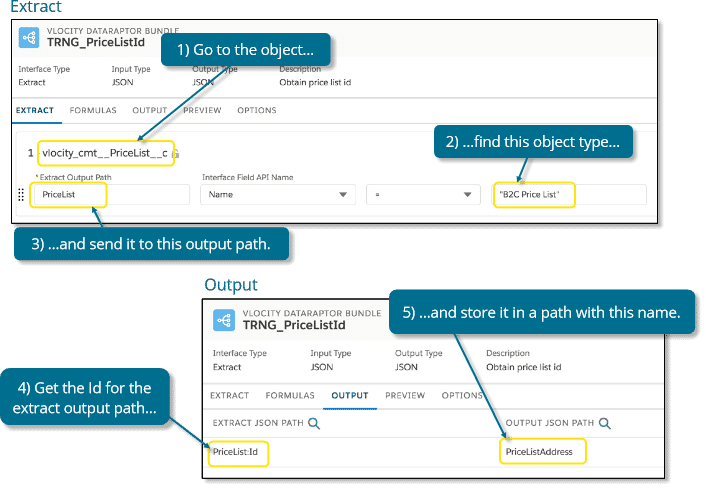
**Ensuring OmniScript Portability**

When developing your guided selling OmniScript, you can ensure that your OmniScript can be easily ported from one environment to another, such as from sandbox to production, by refraining from using record Ids in the OmniScript elements. Instead, as a best practice, use DataRaptors to retrieve specific record Ids for each environment as described below.

**Getting Object Type Ids**

You can create a DataRaptor to extract the object type Ids for use in filtering products for guided selling.





**Get Price List Ids**

A DataRaptor can extract the price list Id so that it remains persistent even when you move the org, such as from development to production.

**Using a Reusable OmniScript as a Container**

To use the filters and persistent price list Id you create with DataRaptors:

1. Create another OmniScript and designate it as a reusable subtype.
2. Configure the reusable OmniScript to extract the filters created by the DataRaptor.
3. Embed the reusable OmniScript into the guided selling OmniScript.

**Displaying Products Using Selectable Items**

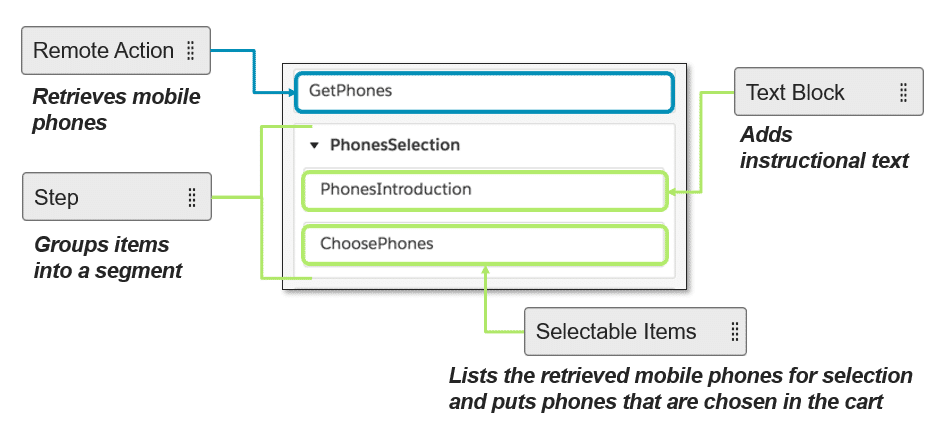


The third step of building a guided selling experience is to create product selection steps.

Once you’ve retrieved your products from the shared catalog, you need to display them to the user. In OmniScript, you do this using selectable items components.

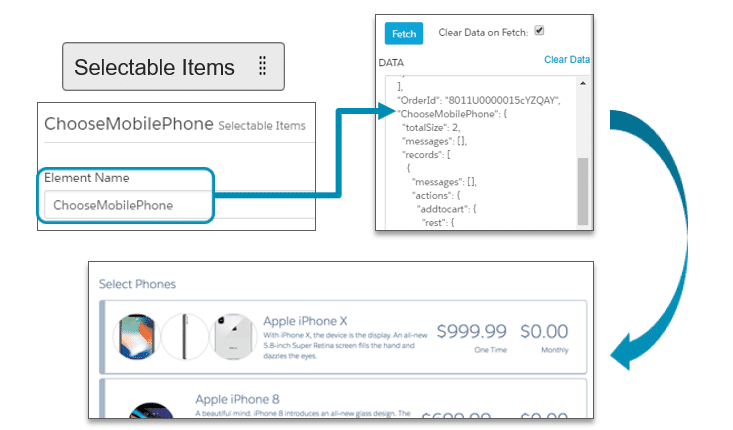
**What are Selectable Items?**

Selectable items components are OmniScript components that display a list to the user and allow them to select something from it. They must be contained within a step element.



**Using Selectable Items**

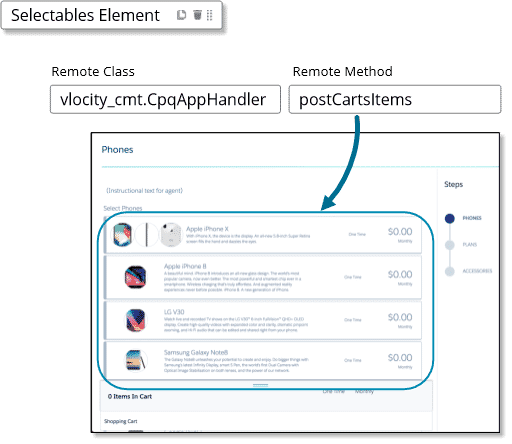
The selectable items element reads the script’s data JSON looking for a JSON node that has its same name, and then it uses the data in that node to display a list of items to the user. At runtime or during Preview, once the OmniScript engine finds the matched JSON node, the OmniScript pulls the entire section of data out of the data JSON and uses the extracted data to populate the list. This can be confusing sometimes when your entire product list seems to disappear from the data JSON!



USING SELECTABLE ITEMS

Use a selectable items element to:

* Display the products retrieved by getCartsProducts
* Enable users to add a product to the cart using the remote method postCartsItems



When the user clicks an item in the selectable items list, they add it to the persistent cart, where they can configure the item. Behind the scenes, another CpqAppHandler method is called, the postCartsItems method.

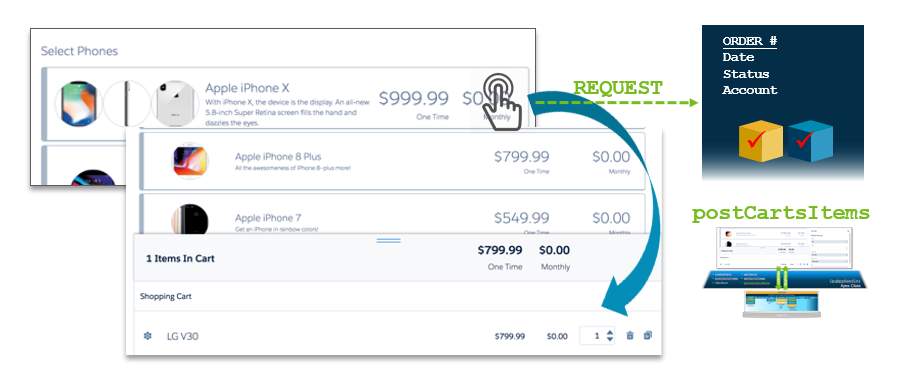
# Adding Products to the Persistent Cart



When the user clicks an item in the selectable items list, it adds it to the persistent cart, and then the user can configure it. Simple, right? There's more to that story behind the scenes, especially with the CPQ application interface. Specifically, products are added to the cart using CpqAppHandler’s postCartsItems method.

From the user's perspective, the user clicks an item in the list. It's added to the persistent cart. Next!

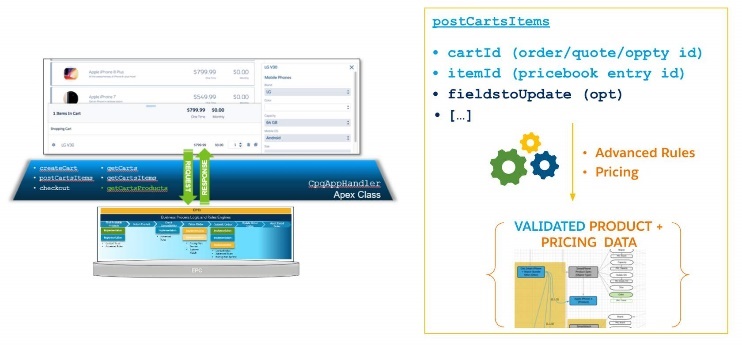
Under the covers, the selectable items element has been configured to invoke CpqAppHandler’s postCartsItems method. So, when the user clicks the item in the persistent cart, the method actually adds a line item to the order (or quote or opportunity). This way, the persistent cart stays in sync with the underlying order.



**Adding Products using postCartsItems Method**

postCartsItems method requires the cartId (order, quote and opportunity id) and the itemId, which is the pricebook entry id. If you’re wondering where and how to get all of this information, don’t worry—it is all there in the response that came back from getCartsProducts.

In postCartsItems response, you will get validated product and pricing information. This means that the data has been evaluated by Advanced Rules for compatibility errors and the pricing service has run to determine the right price, usually using pricing plans and tightest match.



**Submitting Orders in Guided Selling**



The fourth step of building a guided selling experience is to configure the end of the selling experience and then create a way to launch the script.

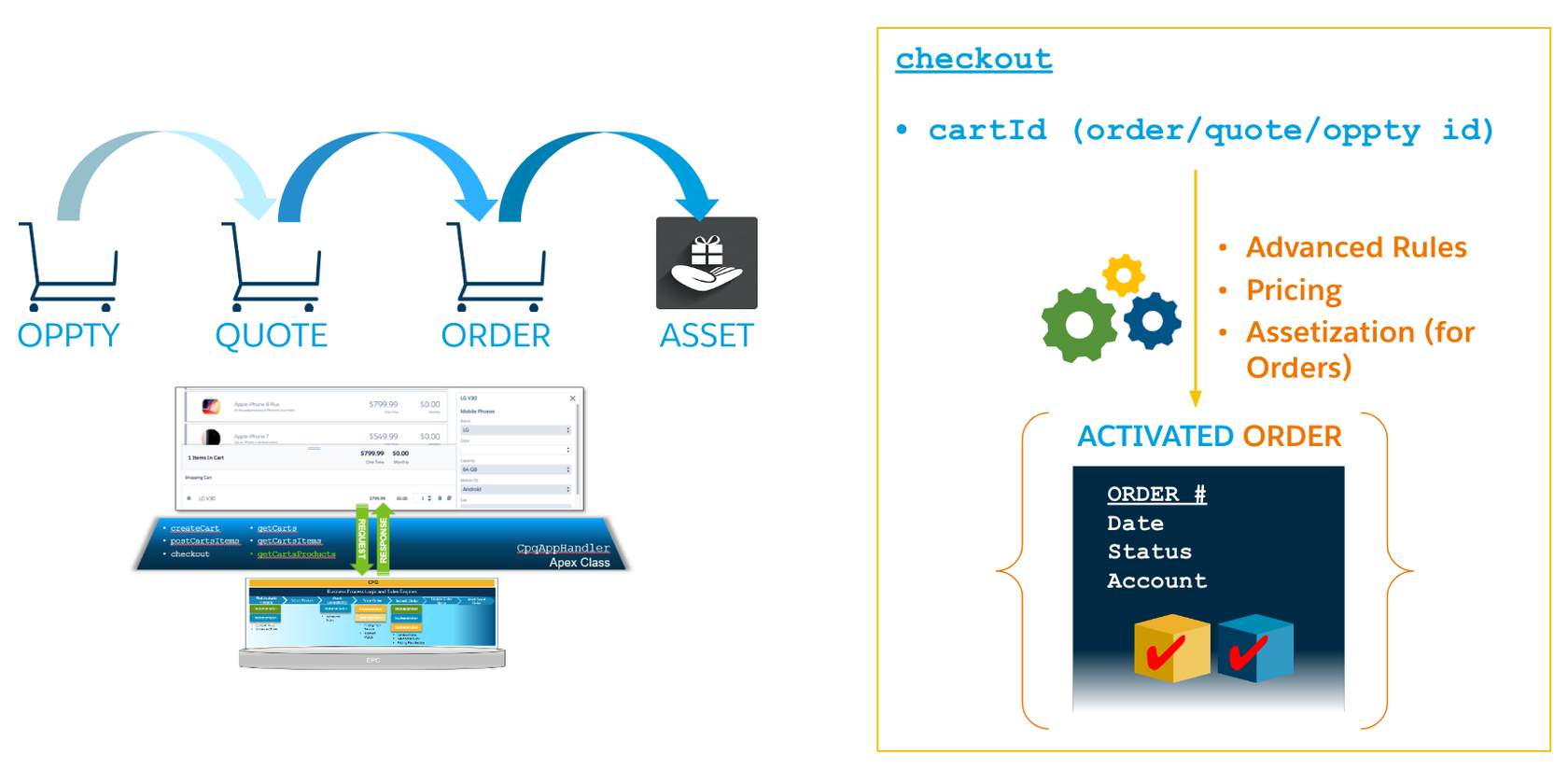
Once you’ve retrieved products, displayed them to the user, and added them to the cart, there’s one final step you don’t want to forget—submitting the order! The CPQ interface, once again, comes into play and provides the checkout method to ensure every order is valid and can be fulfilled.

**Submitting Orders Using the Checkout Method**

The checkout method requires the cartId (order, quote or opportunity id). Based on the header object, checkout moves the object to the next state of the ordering process.

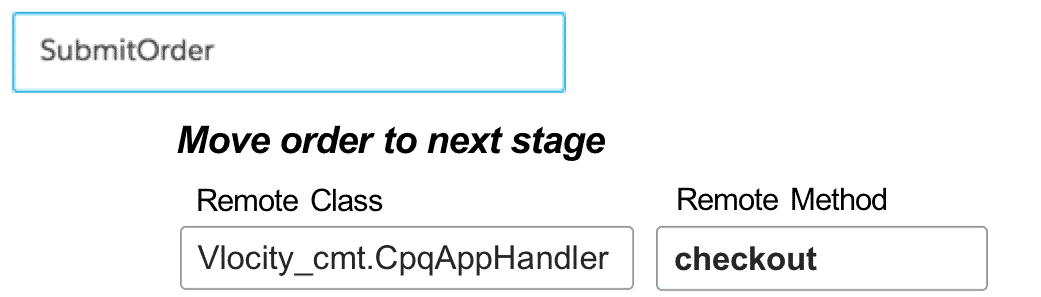
* If the header object is Opportunity, checkout submits the opportunity to create a quote.
* If the header object is Quote, checkout submits the quote to create an order.
* If the header object is Order, checkout submits the order to create assets.

When you checkout an order, rules and pricing run a final time, and then the line items are assetized, creating assets. The Salesforce order status changes from Draft to Activated.



CHECKOUT USING REMOTE ACTIONS

Use a Remote Action element to invoke the remote method **checkout**. This remote method moves the order to the next stage by submitting it. The purchased items become assets on the customer's account.  
  
Note this remote action requires the cartId and ContextId in the Extra Payload. The ContextId and cartId should both have the same value, that value being the cartId.



**Setting the Selling End and Launch Points**

As your final task, you need to create a way for the user to navigate somewhere after the script ends, and create a way to launch the script.

**Using Done Actions to Navigate**

USING DONE ACTIONS

If the guided selling experience begins in one place, it's usually okay to return the user there after the script completes. You can use Done action components to do this in OmniScript.



**Launching the OmniScript with a OmniStudio Action**

After you customize the guided selling OmniScript, you must create a way to launch it. In this course, you launch the OmniScript from an account using the following steps.

* Copy the guided selling OmniScript URL.
* Create an action button for the account.
* Link the guided selling URL to the action button.

It is a little bit more involved than that, but those are the high-level steps. You should be familiar with this process from doing it with the Digital Commerce OmniScript sample app. To not repeat ourselves, we created it for you for the next exercise.

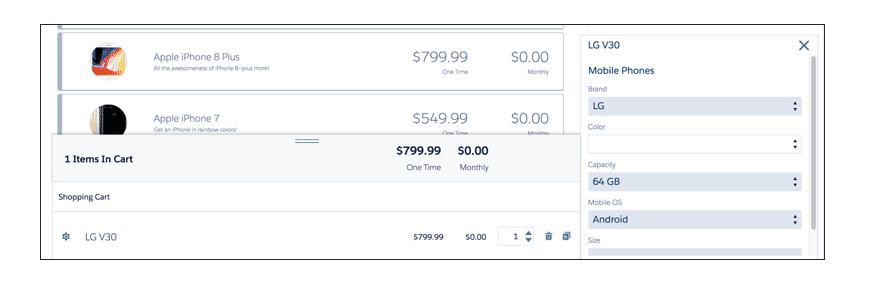
**The Persistent Cart**



The final step of building a guided selling experience is to configure the shopping cart, or the **persistent cart**. Let's take some time to get acquainted with it!

**What is the persistent cart?**

The persistent cart is an OmniScript component that allows users to configure and buy products during a guided interaction. The persistent cart provides a lightweight user interface, while still using the same pricing, rules and products defined in the shared catalog.



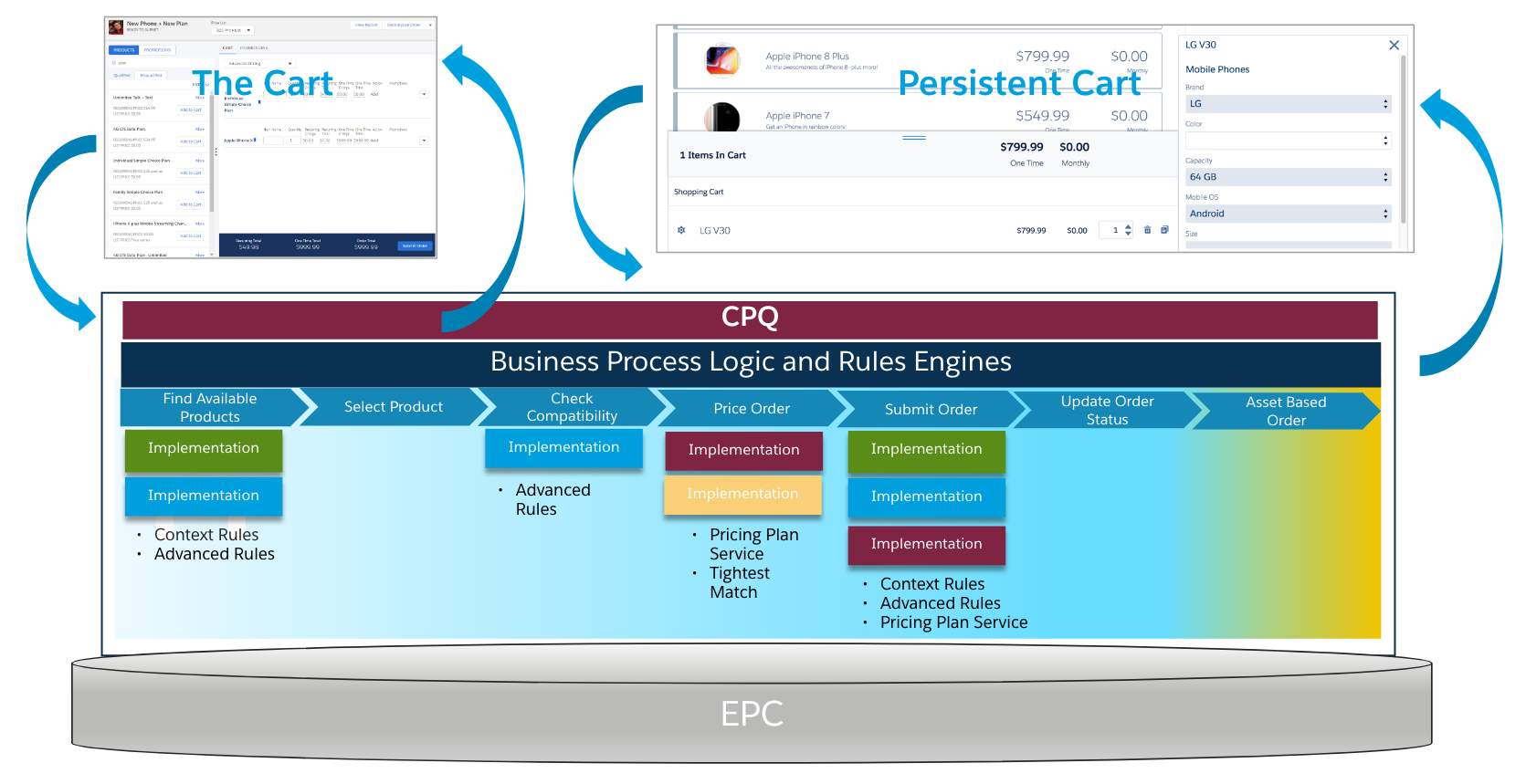
**The Cart and the Persistent Cart**

You may be wondering, what's the difference between the Cart and the persistent cart?

The**Cart**is intended for internal users, like customer service reps or sales agents, who are familiar with the products that you sell and who are comfortable creating and submitting orders in the application. The Cart supports power users with enhanced features that allow you to access additional product information, make and review pricing adjustments, and perform advanced line item configuration.

The **persistent cart**, on the other hand, is a lightweight component that provides basic shopping cart functionality within an OmniScript page. The persistent cart is tuned for the guided selling experience and simplifies the user interface for even the most novice user.

The Cart and the persistent cart both benefit from Industries' CPQ’s common business process logic and rules engines and the products, rules, and pricing defined in EPC’s Shared Catalog.

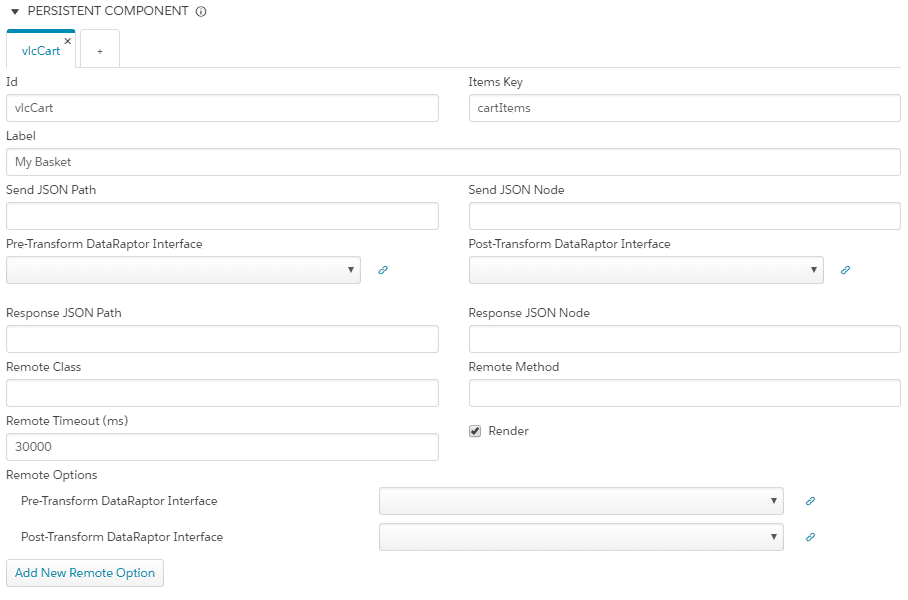


**Configuring the Persistent Cart**

**The vlcCart Persistent Component**

Remember that the persistent cart is an official OmniScript component? It's actually a specialized type of **persistent component**. Persistent components are used for anything that should persist throughout the script. Persistent components can be used for lots of things in OmniScript, such as displaying helpful knowledge articles for a troubleshooting script.

You define the cart in the Persistent Component section of Script Configuration. The Id of the component must be vlcCart in order for the cart to function.



The vlcCart Persistent Component

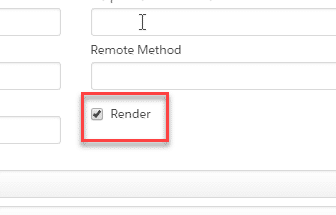
**What can you configure?**

After you've defined the vlcCart persistent component, you can configure it. The good news is that you can change almost anything! Some of the most common elements that can be configured are:

* **Label:**What should the cart be called in the UI?
* **Rendering:** Should the cart display all the time? Or should it run in the background sometimes?
* **Template:** What should the cart look like?
* **Mode:** Should it display horizontally or vertically?

The Label is set in the Persistent Component definition screen. As for the other topics, a bit more explanation is needed.

**Rendering the Persistent Cart**



When you set up the vlcCart persistent component, you will notice a property called **Render**. This property initializes the persistent cart, so it must stay checked for the cart to work properly.

However, especially when building advanced guided selling scripts, there may be times that you don't want the cart to appear. You can turn the display on or off using the Show Persistent Cart vlcCart property for each element of your script. This property indicates whether or not the cart appears in the user interface and can be individually set for each element.

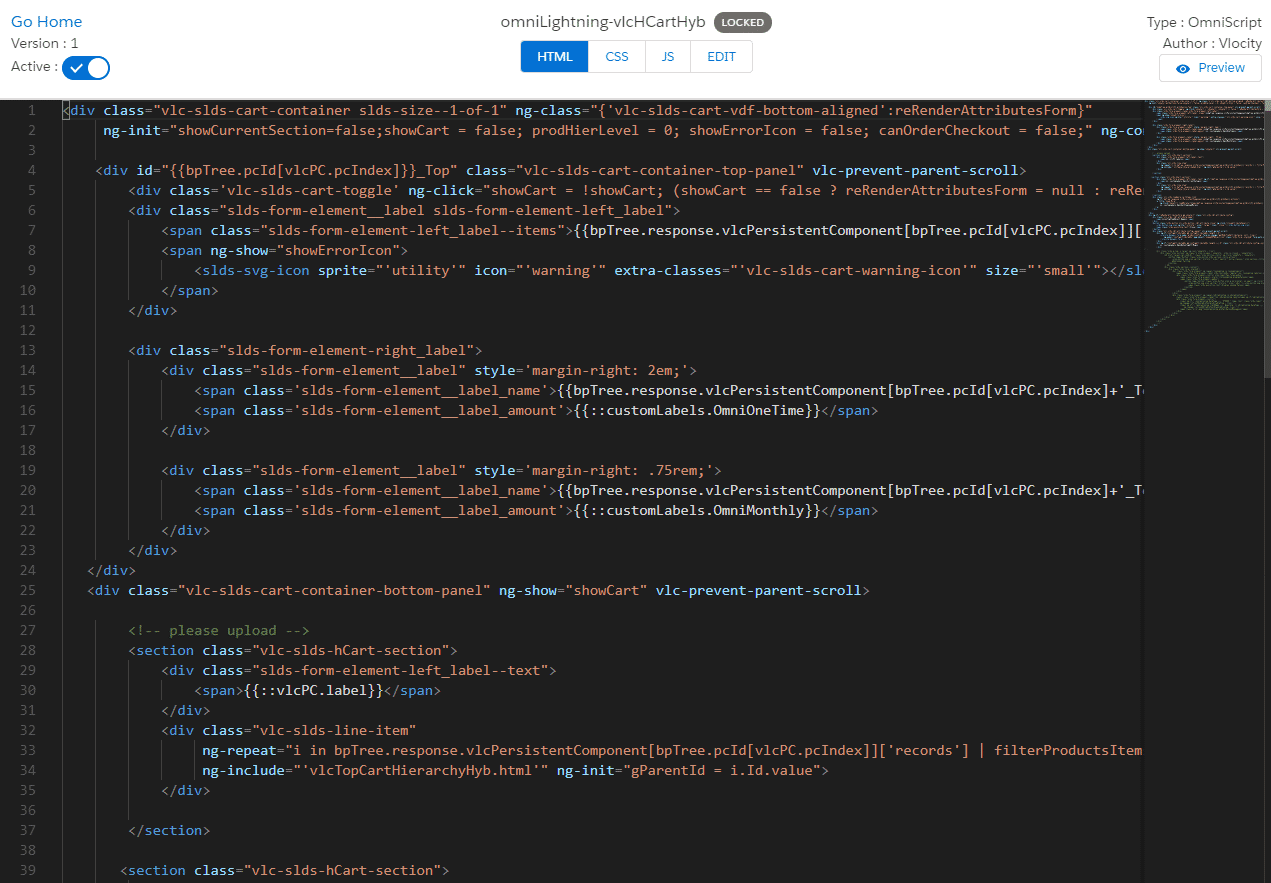
**Configuring the Cart's Template**

**Mapping the persistent cart's template**

As we discussed, the persistent cart is actually a specialized type of **persistent component**. In guided selling, you are going to use the persistent component element as a shopping cart, and not to display articles or any of the other uses for persistent components.

This means you need a specialized template that displays products and pricing and allows users to perform all of the various shopping functions. In order to use the persistent component as a shopping cart, you must override the standard element type’s template to use the persistent cart template. You do that by specifying an element type and a template name in the **Element Type to HTML Template Mapping**in Script Configuration. This lets OmniScript know that you want to use a shopping cart as a persistent component in your script.





**Templates for the Persistent Cart**

Salesforce Industries provides templates for the Horizontal and Vertical modes of the persistent cart. You install them in your org using the **Install OmniScript Lightning**option in the install dropdown menu in Vlocity Templates. Then, you can customize them as needed. There are multiple templates that are embedded within one another, and depending on the types of changes you wish to make, you may need to customize some or all of them. The primary horizontal mode template is **omniLightning-vlcHCartHyb**and the vertical mode template is **omniLightning-vlcVCart**.

**Configuring the Cart's Mode**

**Horizontal or Vertical Modes**

The persistent cart can display in horizontal or vertical modes. When it's horizontal, it will display along the bottom of the browser window. When it's vertical, it will display vertically along the side.

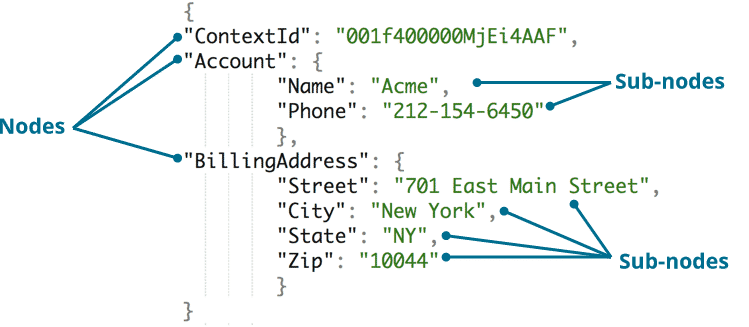
To configure the Horizontal Mode

1. In Script Configuration in Element Type to HTML Mapping, set the element type to **Horizontal Mode Persistent Component**.
2. Set the template to **omniLightning-vlcHCartHyb** or **vlcHCartHyb.html**.

To configure the Vertical Mode

1. In Script Configuration in Element Type to HTML Mapping, set the element type to **Vertical Mode Persistent Component**.
2. Set the template to **omniLightning-vlcVCart** or **vlcVCart.html**.

**Understanding the Cart's Data JSON**



Standard JSON Notation

Just like all OmniScripts, a guided selling OmniScript depends on a good, well-formatted data JSON. And when you use the persistent cart, there are two JSON nodes that are important for the cart to operate correctly.

* **vlcCart**
* **vlcCart\_Top**

**vlcCart JSON Node**



vlcCart Node

**vlcCart**

This node contains the bulk of the data used by the persistent cart and its operations, including the total number of items that have been added to the cart.

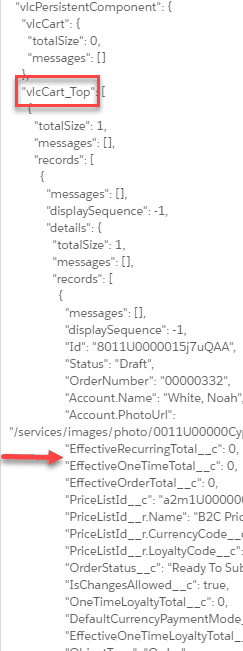
You create this node using a remote action and setting the **Response JSON Node** to **vlcCart**.

It **must**exist within the parent node, vlcPersistentComponent, of the data JSON in order for the cart to function and for products to be added to the cart.

**How do you ensure that vlcCart is in the correct parent node?** If **Show Persistent Component** is checked for the remote action that creates it, OmniScript will create the vlcCart node under the vlcPersistentComponent parent node.

You'll see all of this in action in the lab exercise that follows.

**vlcCart\_Top JSON Node**



vlcCart\_Top JSON Node

**vlcCart\_Top**

This node contains the one time and monthly pricing totals that appear in the Total Summary of the persistent cart.

You create this node just like you create vlcCart, by using a Response JSON Node in a remote action. However, if it does not exist when the cart is launched, the cart can still function. The one time and monthly totals will just be blank until some products have been added to the cart.