**The Digital Commerce User Experience**

Before getting into the functionality of the Digital Commerce APIs, it's important to note that the Digital Commerce SDK simplifies the usage of the APIs by reducing the amount of data you need to handle to create straightforward commerce applications, while also being extendable. Importantly, it ensures Industries data structure rules are enforced and that data is not written incorrectly to the Cart or the shared catalog. **We recommend starting development using the Digital Commerce SDKs, unless you have an exceptional use case for directly using the APIs.**

As covered previously in the SDK and web component topics, it is best to consider the API methods by their relevancy to each of the four user experience phases.

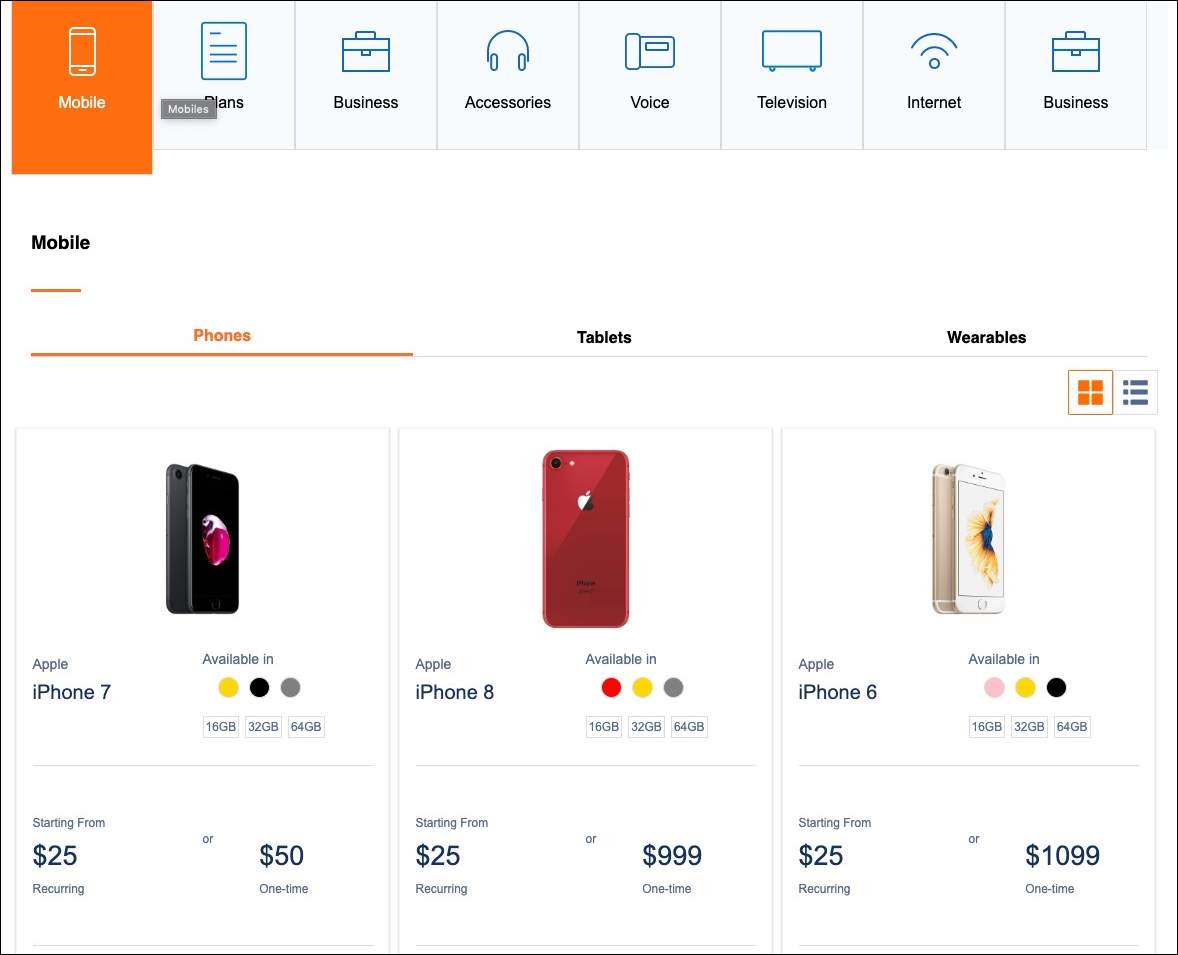
Digital Commerce APIs support the same four phases, which are **Browse**, **Configure**, **Cart**, and **Checkout**.

Examining the Sample App User Flow

Industries provides sample apps utilizing Digital Commerce APIs so that you can get a better understanding of how everything fits together.

**Step 1 - Browse Phase**In the Browse phase, the Get Offers By Catalog Digital Commerce API makes the call shown here. This is an example of **Get Offers By Catalog**for an anonymous user:

GET /catalogs/Phones/offers?context=loc:UK,customertype:new



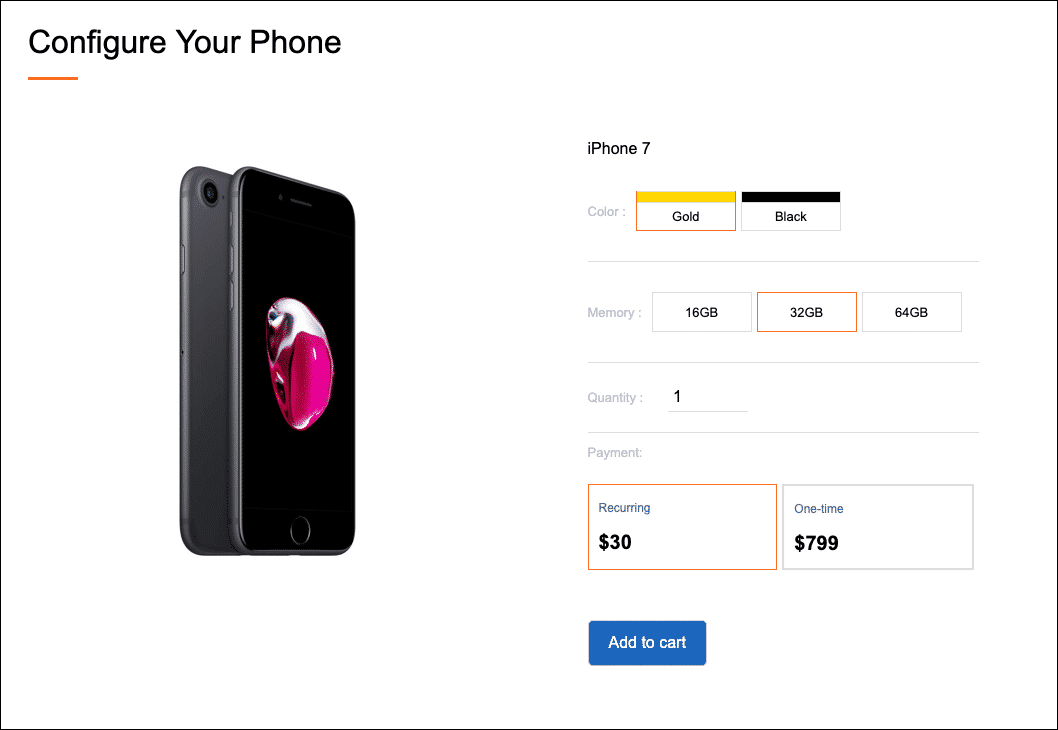
**Step 2 - Configure Phase**In the Configure phase, the user selects an offer from the previous page, which kicks off the Get Offer Details API to grab the product configuration appropriate for the user's context. This is an example of **Get Offer Details**:

GET /catalogs/phones/offers/A64D320?contextkey=6a8c4e4a5p

The user then configures the offer, which is handled by  the Post Offer Details with Configuration Digital Commerce API to make a POST call and retrieve the appropriate response from the cache. This is an example of **Post Offer Details with Configuration**:

POST /catalogs/phones/offers/A64D320?contextkey=6a8c4e4a5p

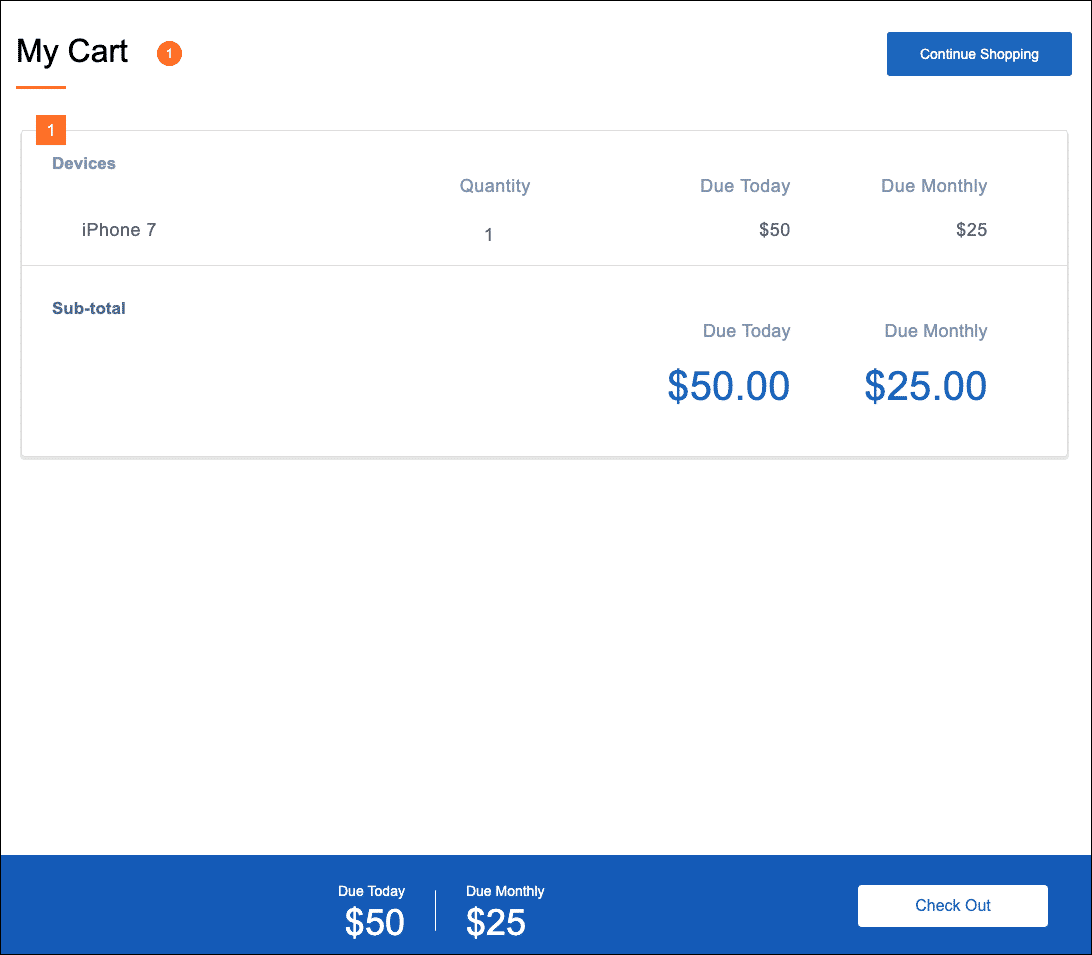
Request Body: {"offerDetails":{...}}



**Step 3 - Cart Phase**In the Cart phase, the user clicks the add to cart button. The Create Basket API call creates a "basket" to house this configured offer. This is an example of **Create Basket**:

POST /catalogs/phones/basket?contextkey=6a8c4e4a5p

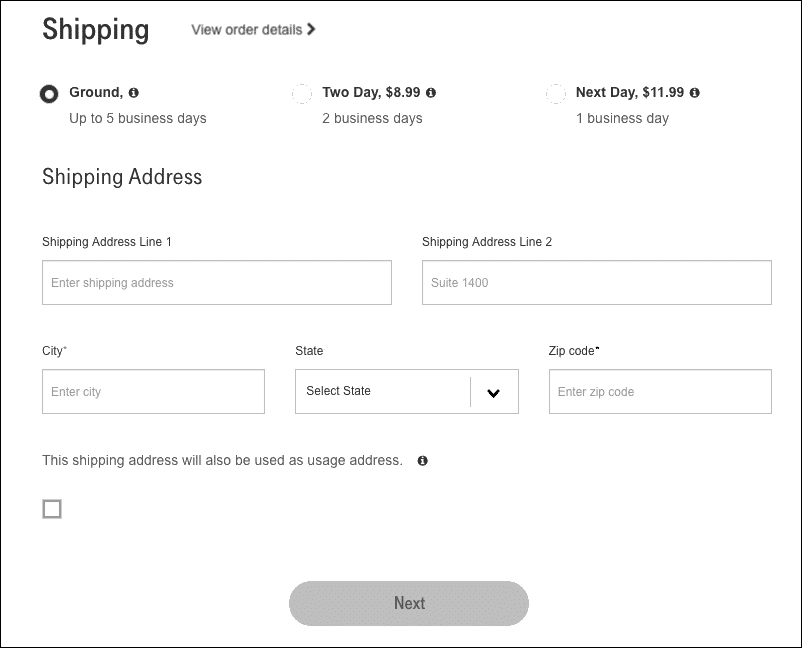
Request Body: {“basketAction”:“AddAfterConfig”...}



**Step 4 - Checkout Phase**  
The Create Cart API converts the "basket" into a Salesforce Order and submits it. This is an example of **Create Cart from Basket**:

POST /services/apexrest/vlocity\_cmt/v3/cart?createAsset=true

Request Body: {”accountId":{...}…}



This was a simplified look at how the Digital Commerce APIS can easily create a fully-featured commerce app. As you begin to build your app, we strongly encourage the use of the Digital Commerce SDK and web components in addition to the APIs.

**Cache Management**

A **cache miss**is when an application makes an API request, but the *specific data* requested is not in the cache. On the other hand, a **cache hit** is when the requested data is found and retrieved from the cache.  
  
Imagine you're making a Digital Commerce API call to retrieve a list of offers for a sales catalog named "Tablets". The API response cache is checked to see if there's a match for the request. If there's a match, it's a **cache hit**, and it will *quickly* return the cached tablets offer list. If there isn't a match, it is a **cache miss,**and Communications Cloud will process it before returning the appropriate tablets offer list and adding it to the cache to help reduce cache misses.

In terms of the Digital Commerce APIs, *specific data* not found in the cache could be related to basket configurations, uncached eligibility contexts, or logged-in user contexts. Digital Commerce is designed to reduce cache misses during the Browse, Configure, and Cart phases of the user journey, but cache misses are an inevitable occurrence that you should be aware of.

**Organic vs Inorganic Cache Creation**

**Organic cache creation**is the generation of cache responses during a real-time request. It is triggered during a cache miss scenario. Digital Commerce will organically create the cache for all Basket operations and GetOffers/GetOfferDetails requests for logged-in user contexts.

**Inorganic cache creation** is the generation of cache responses as executed by the cacheable API jobs. It increases cache hit scenarios and reduces cache misses to improve performance. Digital Commerce allows you to inorganically create the cache for GetOffers and GetOfferDetails for anonymous users by running the Populate Cache API jobs. The decision on how many cached permutations are created is determined by variations in the Product Catalog and the cached values specified as part of Context Dimensions.

**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 explains just a few of the possible parameters that can be used with the**Get Offers** cacheable API.

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

Return a list of offers from the "Phones" sales catalog:

vlocity\_cmt/v3/catalogs/Phones/offers

**Specify a context**  
The 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.

Return a list of offers and specify a context:

vlocity\_cmt/v3/catalogs/Phones/offers?context=("Status": "Active"}

**Specify a page size**  
The 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.

Return a list of offers and specify a page size of 20:

vlocity\_cmt/v3/catalogs/Phones/offers?pageSize=20

**Specify a logged-in user**  
The API call shown here retrieves a list of offers specific for a particular authenticated (logged-in) user.

Return a list of offers for a logged-In user:

vlocity\_cmt/v3/catalogs/Phones/offers?context={"accountId":"0014x00000Di401AAJ"}&isloggedin=true

In the **Specify a particular logged-in user** example above, the accountId that has been passed as input will be used to calculate a **context scope**, a map containing customer info from their Account, Contract, or rootAssetIds object, passed in the call. Take note that it has been passed as a**context**array. The **context** is converted into a **context key**, covered in the next section.

Below is an example of a **context scope** generated for the**Specify a particular logged-in user** example above:

"AccountContext":{

"contextKey":"5a5233b21ec305a99f65b85a514633cf",

"contextValue":{

"country":"US",

"status\_\_c":"Active",

"ecomm1\_\_sla\_\_c":"Gold"

}

}

Context values in the scope above are context dimensions that have been designated as cacheable

**What are Context, Basket, and Cart Context Keys?**

**Context keys**are identifiers created for anonymous users based on applicable cacheable context dimensions for your sales catalogs. Context keys are added to the cache and reused for customers with the same context scope. In essence, this is how the API will return a response very quickly for anonymous users with a generic context since it does not need to calculate a response (i.e., process rules, validation, etc.).  
For logged-in users, context keys are identifiers created based on the user's account, contracts, and assets. For getOffers and getOfferDetails calls, the catalogCode and offerCode are also used to generate a Context key.

Context keys do not need to be unique for subsequent calls (e.g., a getOffers call may have the same context key as a getOffersDetails call).

Context keys are specified in the API URI like in this example. If the context remains the same for your user, you do not need to include the context array, only the context key. This example includes both:

/services/apexrest/cacheableapis/v3/catalogs/Mobiles/offers?context={"Accountsla":"1234",

"Region": "CA"}&offset=0&pageSize=20&contextKey=d9ac6e46f20b05bd940d6f3d894e86e6

**Basket keys**are identifiers that point to the cached entry for the user's cart items. If a cached entry does not already exist for the user's cart contents, a new entry will be created in the cache with a new basket key.  
  
This basket key can be used with the **Get Basket Details API**to see and use the basket's contents.  
  
Cached baskets, similar to context keys, may be reused for multiple user responses so long as the basket contents being requested are the same.

Basket keys are specified in the API URI like in this example:

/services/apexrest/cacheableapis/v3/catalogs/Mobiles/basket/784b264ea6dc63690369e721ffd8e112

**Cart context keys**serve a similar purpose to basket keys but they are used for converting a basket to a cart (i.e., Order). Unlike basket keys, they are not used in the API URI, but rather included in the request body.  
  
This cart context key is used with the **Create Cart from Basket API**to convert the basket into a corresponding order and order line item.

Cart context keys are specified in the request body like in this example.

URI :

/services/apexrest/cacheableapis/v3/carts

Header:

{

"accountld":"0014R00002k5VPG",

"catalogCode: "Phones',

"cartContextkey": "1b42e612a401a072324c7ca026bc4e2f"

}

**Designing a User Flow**

When designing a commerce experience, it's important to begin imagining how users may use your site and map that flow to the actual tech needed.  
  
In the following scenario, we have an anonymous user visiting a commerce site. The anonymous user can view all of their products based on the catalog. No eligibility rules are being used. Here's what the user flow would look like, based on this scenario:

**Anonymous User Scenario with Industries Digital Commerce (B2C)**  
Scenario prerequisites:

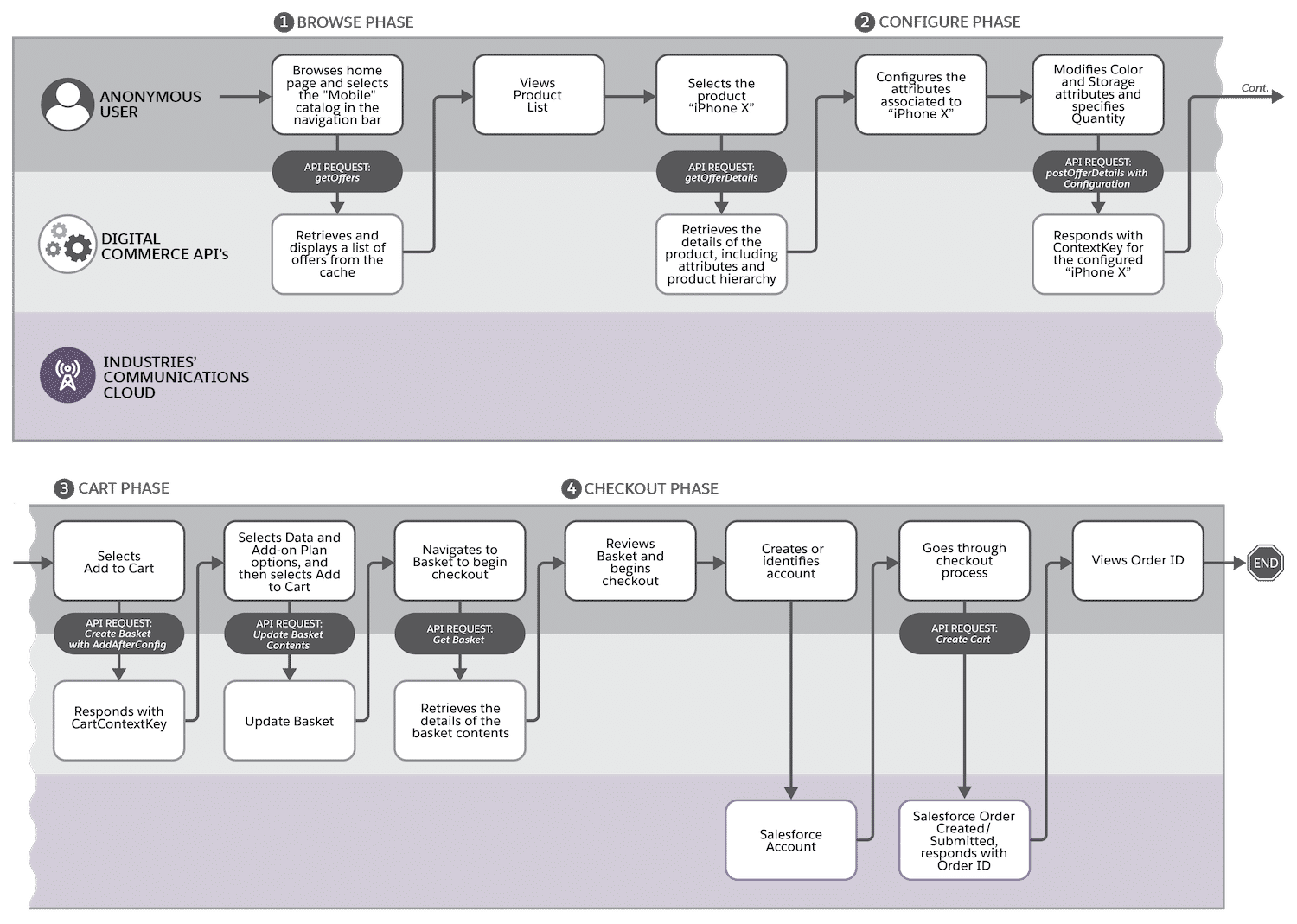
* [You have created product offerings within the product designer](https://docs.vlocity.com/en/Creating-Offerings-in-the-Product-Designer-748866.html) by specifying their name, structure, pricing, attributes, etc.
* You have [created a sales catalog](https://docs.vlocity.com/en/Creating-Product-Catalogs.html) for the product offerings and populated the API cache (i.e., [run the Cacheable API Jobs](https://docs.vlocity.com/en/Preparing-to-Use-Vlocity-Communications-API-Caching.html)).

**Browse Phase** - In this example, we are following an anonymous user browse the catalog and select a product (iPhone X).

**Configure** and **Cart Phases** - In the Cart phase, a "basket" is created. This is not a Salesforce Order and it is stored in the cache.

**Checkout** **Phase** -  The "cart" or "basket" from the previous phase is converted into a Salesforce order during this phase and becomes associated with the user's account. If the user does not have an account already, they will be required to create one at the beginning of this phase.

**Click on this image to zoom in:**



How are Guest Users used in Digital Commerce?

"Guest users" is a Salesforce term that we have not used so far in this training. **Guest users**are the same as anonymous users for our use cases. So why do we need to know about "guest users"? The answer is security!

**Platform Security**

This topic is not intended to cover security in-depth, but it's worth recapping some of the platform-level security available to you.  
  
Guest/Anonymous user profiles can be used to give unauthenticated users access to your Experience Cloud site. In general, users should only have access to the objects and fields that they need to complete their digital commerce user experience flows (i.e., browse, configure, cart, checkout). Restrict all objects and fields based on each applicable user profile. Don't forget to restrict access to sensitive pricing fields that may be fine for internal users to see (e.g., Total Contract Value) but are inappropriate for external users. You should consider documenting all of your security policies showing who has read-only or edit access to select objects and fields.  For more information, read [Object Permissions](https://help.salesforce.com/articleView?id=users_profiles_object_perms.htm&type=5) and [Field-Level Security](https://help.salesforce.com/articleView?id=admin_fls.htm&type=5). Last but not least, don't forget about [Health Check](https://trailhead.salesforce.com/en/content/learn/modules/security_basics/security_basics_healthcheck), available within Setup.

Salesforce Industries has designed and developed use-cases for guest/anonymous users that align with [overall Salesforce user security policies](https://help.salesforce.com/articleView?id=sf.networks_guest_policies_timelines.htm&type=5). Industries does **not** allow guest users to create and update records by default, **except** for certain Cart-Based and Digital Commerce APIs.



Encrypted IDs

Digital Commerce APIs are designed to be generic which is why they use basket and context keys rather than Account ID or Cart IDs. However, despite the general anonymity of context keys, a malicious user may find one and use it to view another user's basket, which in this example, may have sensitive products that others are not supposed to see. For that reason, any Digital Commerce API call that uses a context key must use encryption. Fortunately, we take care of that for you.

For all Digital Commerce API calls that go **directly into Salesforce** (i.e, Sites or Experience Cloud), **guest user IDs must be encrypted**.  
  
For all Digital Commerce API calls that go through the**Digital Commerce Gateway**, **guest and authenticated user IDs must be encrypted**.

In accordance with these policies, we encrypt those IDs and you can rest easy knowing your Digital Commerce API calls are safe.

How Do API Parameters Work with Guest Users?

Quite simply, there are no special permissions or requirements for guest/anonymous users when it comes to the Digital Commerce APIs. Of course, there is one parameter that you will not use with guest users. Take a look at the **isloggedin** parameter and examine how it's paired with Account, Contract, or rootAsset Ids. This is the key differentiator between authenticated users and guest users.  
  
The above example is for an authenticated user, passing their Account Id in the context array, and using the isloggedIn parameter:

vlocity\_cmt/v3/catalogs/Phones/offers?context=("accountId"="0014x00000Di401AAJ")&isloggedin=true

**Industries CPQ Configuration Settings**

There are a number of Industries CPQ Configuration Settings that can be modified for your Digital Commerce solution. We'll cover some of the major ones here so you can start testing Digital Commerce APIs without issue. You can read more on Industries CPQ configuration settings in the documentation.

* **CacheAPI.TimeToLiveInDays**  
  Specifies the number of days an API cache response is effective after its start time.  
    
  **The default is 30 days; you may shorten or extend this default setting.**This setting will show up when you run the Populate API Cache job.
* **CacheAPI.CreateCartFromContextKey**  
  If set to true, the Create Cart API uses the cartContextKey passed in the input to create the cart (i.e., create a Salesforce Order). If set to false or null, the JSON result from the GetOfferDetails API must be passed in the request body to create the cart.  
    
  **This setting should be set to true** so that you can use the Create Cart from Basket API. It is a fundamental Digital Commerce cacheable API. The Digital Commerce cacheable APIs will provide you with a context key as you proceed through the Cart phase, and you can use that key in the header of the Create Cart from Basket API call.
* **CacheAPI.CheckBasketEligibility**Determine whether the Digital Commerce API checks the cartContextKey parameter for the eligibility of the user’s basket. For example, imagine there are eligibility rules defined in the Shared Catalog like customers only being allowed to purchase one iPhone X. Once a customer adds the iPhone X to their basket, they will not be able to add additional iPhone X products to their basket. This is because the Get Offer Details with Configuration Digital Commerce API validates their basket.  
    
  **This setting should be set to true** so that you can properly utilize context rules.
* **MTSEnabled**When MTSEnabled is set to true, the **Multi-Transaction Service** splits Apex transactions into multiple transactions to avoid exceeding Salesforce Governor limits. This is only used with the Digital Commerce cacheable basket APIs. Generally speaking, unless you are working with a very small product model and rule set, **you should keep this set to true**, and if you are working directly with the APIs, you should also know how to handle intermediate API responses. The next section will cover handling intermediate API responses using an Integration Procedure.

**Integration Procedures and Governor Limits for Digital Commerce**

Are you using Integration Procedures to prototype a Digital Commerce solution? While this is not the easiest solution to implement, it may be worth exploring as you decide on how to implement Digital Commerce with your existing team and resources.  
  
As you experiment with Digital Commerce Integration Procedure remote actions, you may find a governor limit exception in your debug logs. Salesforce limits platform resources at both a periodic and transactional level. [Governor execution limits](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm) ensure efficient code execution and processing for all tenants on the force.com platform.

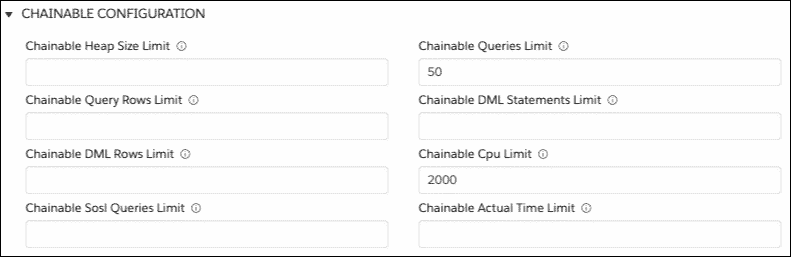
**Governor Limit Considerations**

Digital Commerce APIs like Create Cart from Basket create several SOQL queries. We recommend breaking a large Integration Procedure into separate calls to ensure you do not exceed Salesforce governor limits.  
  
No one likes hitting limits, especially when they're rapidly building a prototype! Fortunately, OmniStudio gives you some options to avoid hitting limits and intuitively develop your app.

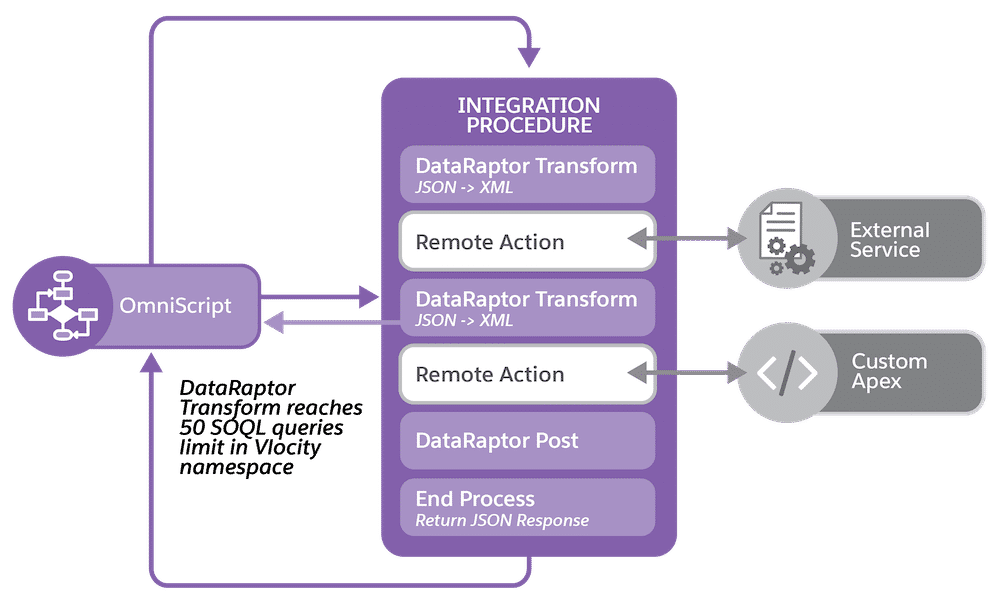
**APIs, OmniScript, and Integration Procedures**

Using an OmniScript and including Integration Procedures, we can use several Digital Commerce remote actions to perform our Digital Commerce apex transactions.  
  
First, make sure you have a solid understanding of [Integration Procedures](https://trailhead.salesforce.com/en/content/learn/modules/omnistudio-integration-procedures?trail_id=build-guided-experiences-with-omnistudio). If you have access to the Vlocity Success Community, you should complete the first topic in the [advanced Integration Procedure course](https://vlocity-university.litmos.com/course/4492069). The key is to run asynchronous apex to execute code in separate threads, which will have their own separate governor limits.

Configure limits in the Procedure Configuration section to values below Salesforce limits:



An example Integration Procedure flow where the DataRaptor Transform reaches the limit and chains a new transaction:



Using a REST API is the easiest way to see the partial responses each transaction returns before the Integration Procedure completes. So how do you invoke the Integration Procedure with REST calls? See this [documentation page](https://docs.vlocity.com/en/Invoke-a-Chainable-Integration-Procedure-with-REST-Calls.html).

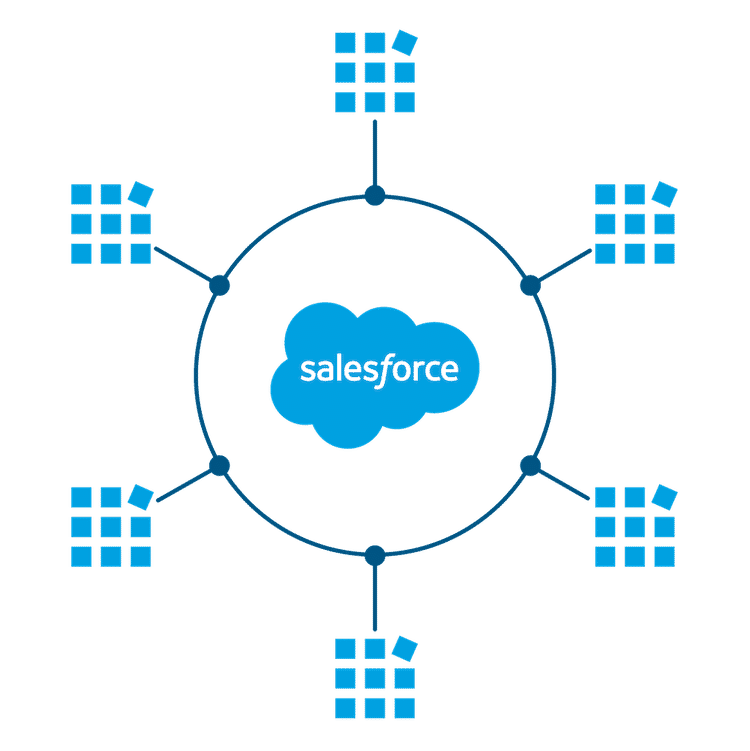
**Working with Multi-Transaction Keys and Integration Procedures**

Multi-transaction calls split one transaction into two. The intermediate response provides a multi-transaction key. This key has to be passed back in the following call to complete the transaction. The Digital Commerce SDK, web components, and Lightning web components handle the multi-transaction key for you, so you don't have to keep track of it.  
  
However, if you are developing directly with Digital Commerce APIs and using Integration Procedures, you will need to develop a way for your invoker (e.g., front-end) to keep track of each multi-transaction key and pass it back as necessary to the Integration Procedure.

**How to Test Digital Commerce APIs**

In our training, we'll cover the high-level details of the Digital Commerce APIs, and how to test them, but you should consider bookmarking our Digital Commerce API documentation. In your quest to create a complex commerce app, it will be handy to reference the documentation for a concise list of calls, methods, parameters, models, examples, and more.

[Workbench](https://workbench.developerforce.com/) is a suite of tools for interacting with your Salesforce org through the API. One of those tools is a REST Explorer, which we'll use to make Digital Commerce API calls. Because Workbench is easily accessed through the browser and integrates easily with Salesforce, it will be our primary tool to test the APIs in training.  
  
That's not to say you cannot use third-party apps to test the API, such as **Postman** and **MuleSoft**. If you use a third-party app, you will need to authorize it and configure it as a [Connected App](https://help.salesforce.com/articleView?id=connected_app_overview.htm&type=5) in the Setup area of your org.



A Connected App is a Salesforce framework that enables an external application to integrate with Salesforce using APIs and standard protocols. Connected apps use various protocols including OAuth to authenticate external apps.  
  
To call the Digital Commerce APIs from Postman, you must first deploy a connected app in your org. The connected app handles security (OAuth in this case) and permits access to org data from external APIs.

**API Considerations**

When working with Digital Commerce sales catalogs, the API response cache, and Digital Commerce APIs, it's important to note the current limitations, as they may impact your API testing. While we work to improve the product for future releases, here are a few of the major considerations:

Digital API Caching does not honor product selling period or effectivity dates. The following dates in the product catalog are ignored by Digital API caching: Selling Start Date, Selling End Date, Fulfillment Start Date, End Of Life Date, Effective Date, and Effective End.

* Starting with the **CME Winter '20**release, these product selling periods and effectivity dates are supported.
* **The Active flag** on products in the Shared Catalog and on product catalog relationships within the sales catalog **is** honored. You should see active sales catalog products in your API response.
* The off-platform Digital Commerce solution is designed to be gateway agnostic but is currently optimized for AWS (and managed by Salesforce).
  + For training purposes, all of our hands-on exercises are demonstrated **on-platform**.
* When working with a sales catalog and creating Catalog Product Relationships, do not enter both products and promotions in the same product relationship record. For example, do not create a record for a product and then add a promotion to the same record. You should instead create one record for the product and one record for the promotion.
* Products and Promotions are retrieved from the API response cache in an identical manner. Keep this in mind when designing your commerce app.
* Remote method calls are available for use, see the various calls in our [documentation](https://docs.vlocity.com/en/Digital-Commerce-Remote-Calls.html).

**Browse Phase Digital Commerce APIs**

During the **Browse** phase, the user is browsing a catalog of products or promotions.

This can be done by anonymous or authenticated users.

Typically, the user will be looking at different catalogs during this phase. For example, the user may navigate to your commerce app's homepage, which loads featured offers (**Get Featured Offers** API), and then navigate to a different page that contains all of the phones that they can upgrade to (**Get Offers By Catalog: w/Context Rules** API).

In the **Browse** phase, we'll cover how to get offers from your Digital Commerce sales catalog using several unique API calls. **For all of the APIs in this phase, the specified sales catalog code parameter identifies the catalog that contains the offers.**  
  
Additionally, offer eligibility can be determined by API parameters (i.e., context)

**Retrieving Offer List**

**Get Offers By Catalog**

Provides the ability to retrieve a list of offers (products or promotions) that are modeled within the Shared Catalog.

**Get Featured Offers**

Provides the ability to retrieve a list of offers (products or promotions) that are modeled within the Shared Catalog, which is sorted by the Sequence field for Catalog Product Relationships.

**Get Contains Offers**

Provides the ability to retrieve a list of offers (products or promotions) that are modeled within the Shared Catalog which contains the specified product.  
  
For example, if you require the user to purchase a bundle, you can have a two-step product selection process, starting with the selection of a phone, followed by only showing bundles that contain that phone.

We will look at testing the API with our e-commerce catalog to better understand exactly how it works. You will need a catalog code from a sales catalog in your org, to get started.

In order to test the APIs and follow along with the video below, use the Rest Explorer in Workbench: <https://workbench.developerforce.com/restExplorer.php>

**Retrieving Offer Details**

**Get Offer Details By Catalog**

Provides the ability to **retrieve a single offer's (products or promotions) details**that are modeled within the Shared Catalog. This API requires the catalog code and **specified offer code** parameters, the latter of which identifies the product that will be detailed in the response.

Configure Phase Digital Commerce APIs

During the **Configure** phase, the user is configuring an offer by changing quantities, adding or deleting child products in the bundle, and configuring attribute values. Once the offer is configured, it is validated to check for configuration errors.

This can be done by anonymous or authenticated users.  
  
Typically, the user will have selected a product or promotion and is on a product configuration page. On that page, they are configuring various product attributes (e.g., color, capacity, etc.).  
  
The**Configure Before Adding To Basket** API modifies the product attributes in the response from the Get Offer Details API and uses that whole modified response as the request body of the POST call. The API will validate for accurate pricing and eligibility in its response.

During this phase, the SDK can be used to return the offers that were viewed during the Browse phase from the cache. This is useful when the user is navigating through multiple pages. Note that no API calls are made to retrieve the previously cached offers.

**Passing Offer Details**

Configure Before Adding To Basket

Provides the ability to pass offer details that are modeled within the Shared Catalog in order to retrieve pricing updates. This API requires the catalog code and specified offer code parameters, the latter of which identifies the product that will be detailed in the response. Pricing eligibility can be determined by API parameters.

**Cart Phase Digital Commerce APIs**

During the **Cart** phase, the user is adding, updating, or deleting line items in the cart.  
  
When they have finished configuring the product, they will add it to their "cart". Internally, we refer to this as a "basket", with the API named **Create Basket**. The "basket" exists temporarily while the user is on the site. This API can be used for adding/deleting offers and bundle items to/from the cart. Note, no Salesforce orders are created during this phase.

How does the basket persist between user actions if it's not stored as a Salesforce order? Every call to the Create Basket API returns a **cartContextKey**, as well as the JSON Result corresponding to the basket. The cartContextKey is a hash key that uniquely identifies the contents of the basket. As long as this cartContextKey is passed as a parameter in subsequent basket API calls, there will be a persistent basket between user actions.  
  
So what makes the basket "temporary"? The UI is responsible for maintaining the cartContextKey and JSON result in a viewable format. If you were to clear browser cache, you will lose the JSON result (without which, the basket would be empty) and the cartContextKey (without which, the backend would not have a way to uniquely identify the basket).

**Creating a Basket**

Create Basket

Provides the ability to create a basket with or without offer (product or promotions) configurations that are modeled within the Shared Catalog. The specified catalog code identifies the catalog that contains the offers.

* (BasketAction: AddWithNoConfig): Add/Delete Items to/from Basket w/o Configuration. This method requires the **specified offer code.**
* (BasketAction: AddAfterConfig): Add/Delete Items to/from Basket w/Configuration. This method requires the response from the Get Offer Details API in the request body.

These are the two most basic methods for using this API. Explore further methods and calls for this API in the documentation.

Checkout Phase Digital Commerce APIs

During the **Checkout** phase, the user is ready to checkout and submit the order.

Prior to creating an order, the user must authenticate. In other words, the **Create Cart from Basket** API requires a Salesforce Account ID as a parameter. Typically, a user will create an account or log in to their existing account, just before they are ready to checkout, or when they begin the Browse phase.

By our definition, the user's "basket" becomes a "cart", when it becomes a Salesforce order. This API adds all basket offers to the cart, and then configures, validates, and prices them. This API will always create an order, not a quote or opportunity. In the response, it returns the order number of the created cart and a link to the cart-based API to retrieve the full cart contents.  
  
The sample app and cart-based checkout API should be used to complete the rest of the checkout process (i.e., shipping/billing info, payment gateway, and order submission).

**Creating and Submitting an Order**

Create Cart from Basket

Create Order Cart from Basket. Provides the ability to create a cart from a basket with offers (product or promotions) and its configurations that are modeled within the Shared Catalog.  
  
Additionally, in the request body of the call, it should include the user's accountId, specified catalog code, and the JSONResult from the Create Basket API response.

You also have the option to submit the order immediately following order creation, if you append  
?createAsset=true to the URI. For example:  
/services/apexrest/vlocity\_cmt/v3/carts?createAsset=true

Working with Context Rules and Digital Commerce APIs

Context rules can be used to control product availability and pricing eligibility for your users. When it comes to Digital Commerce APIs, this is achieved by appending a user **context** parameter to API calls. For example, if you wanted to restrict a new subscriber promotion to only new customers, you could create a context rule that qualifies that promotion for only new customers.

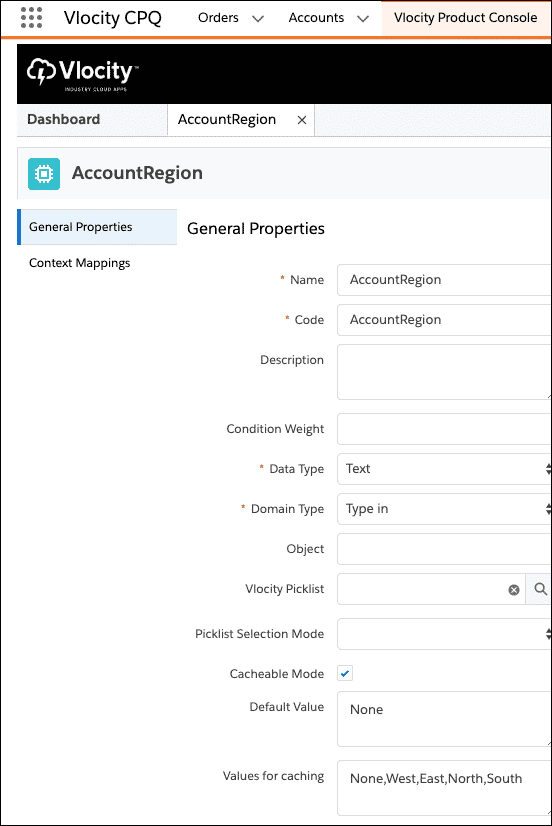
Context Rules in the Browse Phase

One prominent Digital Commerce design pattern in the Browse phase is making select offers and products available to select users, based on a variety of criteria. In order to accomplish this, you will need to use create and maintain context rules in your shared catalog, for the products and promotions that are in your sales catalog. Once you have context rules set up with some of your Digital Commerce Catalog products, you can test them using Workbench and the **Get Offers By Catalog** API.

Let's walk through the steps to test the context rules with the Get Offers By Catalog API. In our example, we have created context rules for the Apple iPhone 8. We have created rules that qualify only users with the account region of "West". This means, only users of the "West" region should be able to see the Apple iPhone 8.

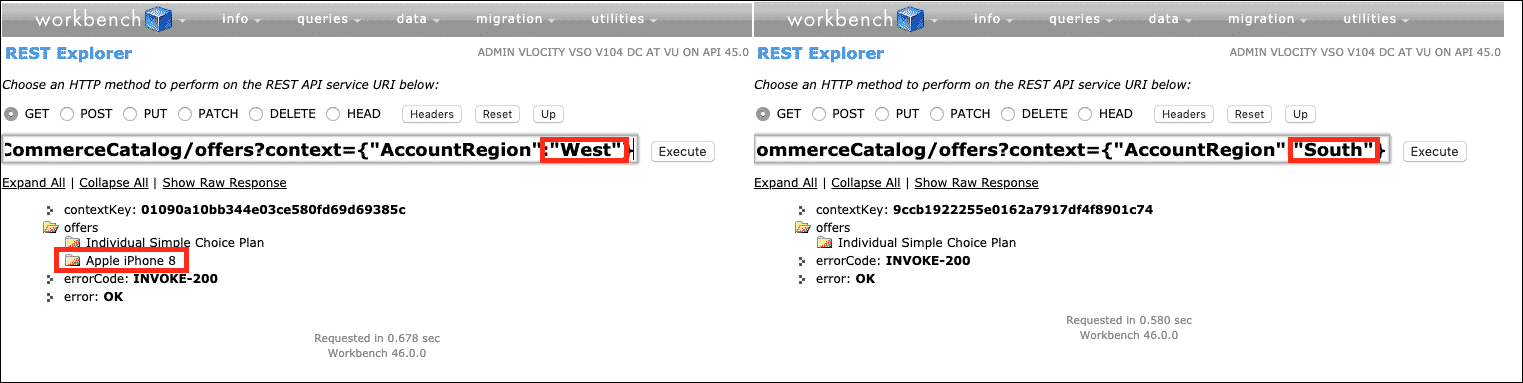
1. In [Workbench's REST explorer,](https://workbench.developerforce.com/restExplorer.php) leave **GET** as the HTTP method.
2. Enter a URL in the format of the Get Offers By Catalog API covered in the previous section, adding on the Context Dimension following this format:  
   /services/apexrest/vlocity\_cmt/v3/catalogs/[CATALOG CODE]/offers?context={"[CONTEXT DIMENSION]":"[sOBJECT FIELD VALUE]"}
3. Our example:  
   /services/apexrest/vlocity\_cmt/v3/catalogs/**CommerceCatalog**/offers?context={"**AccountRegion**":"**West**"}  
     
   In this example, you can see we are mapping the "AccountRegion" Context Dimension to the value of "West" in order to retrieve all offers that are eligible for users of the "West" region.
4. Click **Execute**.  
     
   If you see the desired result, try changing the context to something that would alter the response, and execute the call again, to see if the response changes.

The context dimension for "AccountRegion" created in the Product Console:



The context dimension must have the **Cacheable Mode**setting checked in order for it to be cached. If you make create a new context rule and apply it to products, you must regenerate the cache with the Cache Management APIs or cache jobs in Vlocity CMT Administration.

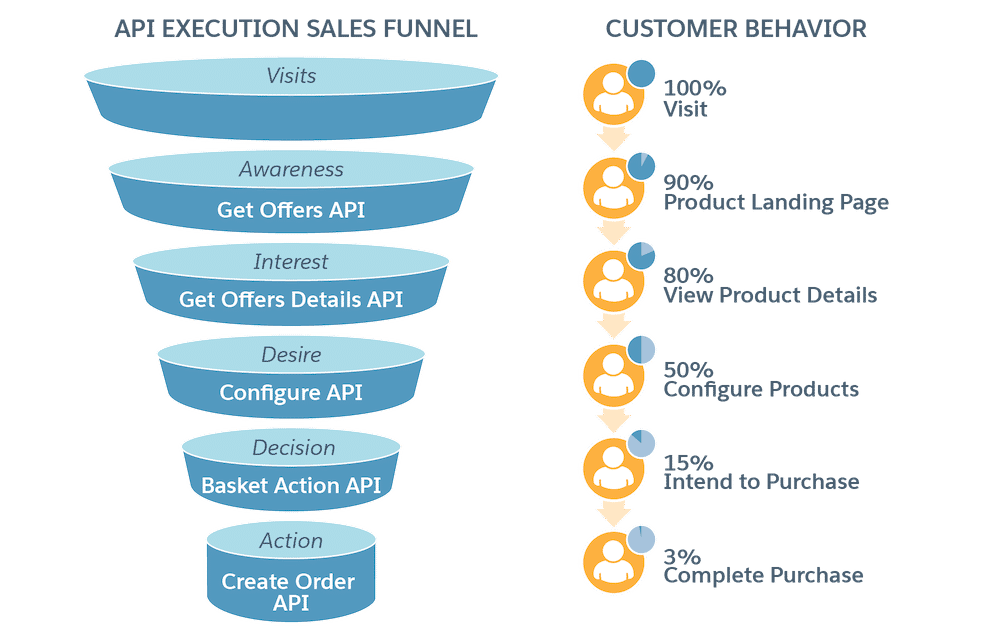
In this example, we attached a context rule set to the Apple iPhone 8 that affects its availability to customers. In this case, only customers from the "West" region can purchase it. When the catalog is retrieved by the GetOffers API, it will only get offers that qualify for the context specified:



**An Introduction to Cache Management**

Most anonymous visitors on a commerce website spend most of their browsing experience viewing products that capture their interest. Consequently, as users make their way down the commerce funnel towards purchasing the product, we organically create the cache based on the request made to our Industries CPQ engine. As time goes by, we have a robust set of cached records within our cached response object.

So what is **cache management** exactly? When Digital Commerce API calls are made, the API response cache is checked before processing the request. The handling of API requests and cache creation is cache management. Cache Management covers the Awareness, Interest, Desire, and Decision phases of a customer journey as depicted below.



This graphic illustrates an example use-case: anonymous browsing at the top of the funnel and new user checkout near the bottom of the funnel. Other industry examples may have differing percentages. For example, a media industry PPV event could indicate >90% product landing page through complete purchase with 0% traffic on intend to purchase, due to the nature of the PPV user experience.

Furthermore, you can customize and optimize the relevant Cache Management jobs by leveraging **Cache Management APIs**.   
  
But before we look at the Cache Management APIs, let's better understand the concept of cache management.

**Cache Misses vs Cache Hits**

A **cache miss**is when an application makes an API request, but the *requested data* is not in the cache. On the other hand, a **cache hit** is when the requested data is found and retrieved from the cache.  
  
Imagine you're making a Digital Commerce API call to retrieve a list of offers for a sales catalog named "Tablets". The API response cache is checked to see if there's a match for the request. If there's a match, it's a **cache hit**, and it will *quickly* return the cached tablets offer list. If there isn't a match, it is a **cache miss,**and Industries CPQ engine will process it before returning the appropriate tablets offer list and adding it to the cache to help reduce cache misses.

In terms of the Digital Commerce APIs, *specific data* not found in the cache could be related to basket configurations, uncached eligibility contexts, or logged-in user contexts. Digital Commerce is designed to reduce cache misses during the Browse, Configure, and Cart phases of the user journey, but cache misses are an inevitable occurrence that you should be aware of.

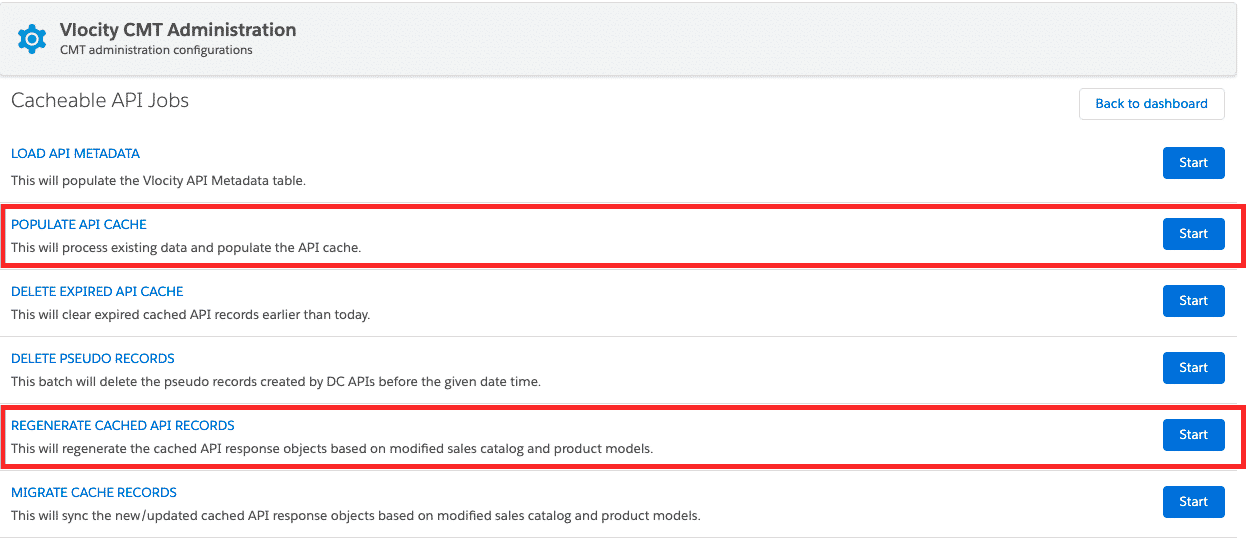
**Organic vs Inorganic Cache Creation**

**Organic cache creation**is the generation of cache responses during a real-time request. It is triggered during a cache miss scenario. Digital Commerce will organically create the cache for all Basket operations and GetOffers/GetOfferDetails requests for logged-in user contexts.

**Inorganic cache creation** is the generation of cache responses as executed by the cacheable API jobs. It increases cache hit scenarios and reduces cache misses to improve performance. Digital Commerce allows you to inorganically create the cache for GetOffers and GetOfferDetails for anonymous users by running the Populate Cache API jobs.  
  
For both organic and inorganic cache creation, the number of generated cached permutations is determined by variations in the Product Catalog and the cached values specified as part of Context Dimensions.

**Administer Cache Management**

Since anonymous customers spend most of their time browsing, we provide the ability to populate the cache in advance (i.e., inorganic cache creation) to ensure that their first visit is highly productive.  
  
**There are currently two ways to manage your cached responses.**The first is by executing Cache Management batch Apex jobs. The Populate API Cache job populates cache entries related to Get Offers and Get Offer Details APIs for anonymous users.

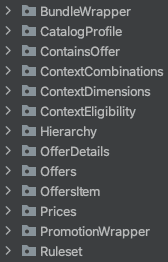


Cacheable API Jobs that execute batch Apex

There are several cacheable API jobs an administrator can perform. In the Building Sales Catalogs course, we covered the jobs for populating, which initializes the cache and regenerating the cache, which updates the existing cache based on product catalog changes. These jobs are conveniently accessed via the UI presented in **Vlocity CMT Administration Cacheable API Jobs**.

Each job works by executing several different Apex jobs. Because the Apex jobs are executed sequentially, these jobs can take a **significant** amount of time to complete if you have a large catalog and/or have many cacheable context dimensions. This cache management method is relatively simple and non-technical. **However, developers have a second cache management method option, with the Cache Management APIs.**

Introducing the Cache Management APIs



A preview of the Cache Management APIs

The underlying functionalities of the cacheable API jobs are now available to you as a set of 18 individual APIs (13 APIs for populating the cache and 5 APIs for regenerating the cache). Moving from the previous Batch APEX jobs in Vlocity Administration to the REST API Cache Management APIs gives you the **flexibility** to manage the cache with **precision** and **performance**. This can speed up new catalog releases with faster catalog update rollouts. The Cache Management APIs are intended to work with your CI/CD implementation.  
  
With the Cache Management APIs, you can make API calls in **parallel**, which can dramatically speed up the process of populating and regenerating your cache. Additionally, you can perform API calls on a **catalog** and **context dimension-specific level** to maximize performance.

To get started, first, you should understand the APIs available to you. Next, you will need to orchestrate the APIs to make parallel calls for the products/promotions in your catalog. Finally, you should verify that your cache has been properly populated by testing Digital Commerce API calls (e.g., Get Offers by Catalog).

# How to Populate the Cache

A set of 13 REST APIs are available to populate the API cache. Before you begin using them, you should be aware of their prerequisites.

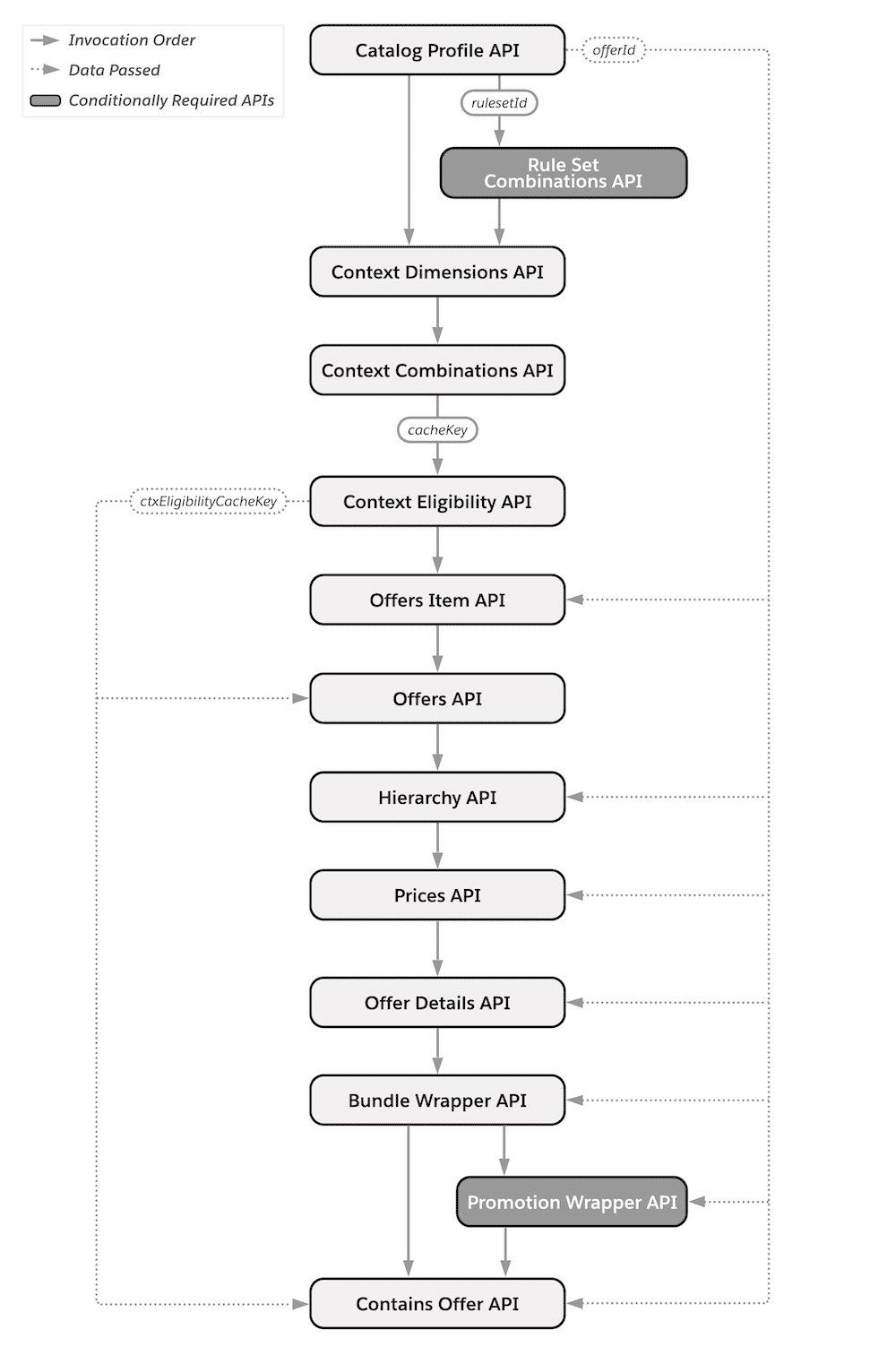
**Populate Cache API Prerequisites**

Here are the prerequisites for using the populate cache APIs:

* [You have created product offerings within the product designer](https://docs.vlocity.com/en/Creating-Offerings-in-the-Product-Designer-748866.html) by specifying their name, structure, pricing, attributes, etc.
* You have [created a sales catalog](https://docs.vlocity.com/en/Creating-Product-Catalogs.html) for the product offerings.
* **You must**execute each API in the chronological order presented in the next section.
* **You must**execute all populate cache APIs, with a few exceptions described in the next section.
* **You must** **not** modify your catalog or product offerings until you have executed **all** of the populate cache APIs listed in the next section. For example, **do not**make a Catalog Profile API call, remove products from your sales catalog, and then use the Rule Set Combinations API.

**Populate Cache APIs**

In this section, we will briefly describe each API and demonstrate each of the populate cache APIs.



This is a diagram of the Populate Cache APIs in their invocation order. The dotted line represents data passed between API calls. The shaded boxes represent APIs that are only required under certain circumstances, described below.

**1. Catalog Profile API**

**Required**

GET /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/catalogprofile  
  
The Catalog Profile API uses the **catalogCode** to generate its data profile and cache it. You define a **catalogCode** when you initially create a sales catalog.

There are two parameters that we recommend using with this API. One is **effectiveStartTime** and the other is **expirationTime**.  
  
**effectiveStartTime** is the time which you want your cached catalog entries to be effective. It should be in the UTC date-time format. Your effectiveStartTime **must occur** **after** **the current time** for which you are executing the Populate Cache APIs or you will receive an error message.  
  
**expirationTime** is the time which you want your cached catalog entries to expire. It should also be in the UTC date-time format. Your expirationTime **must occur** **after your effectiveStartTime**or you will receive an error message.

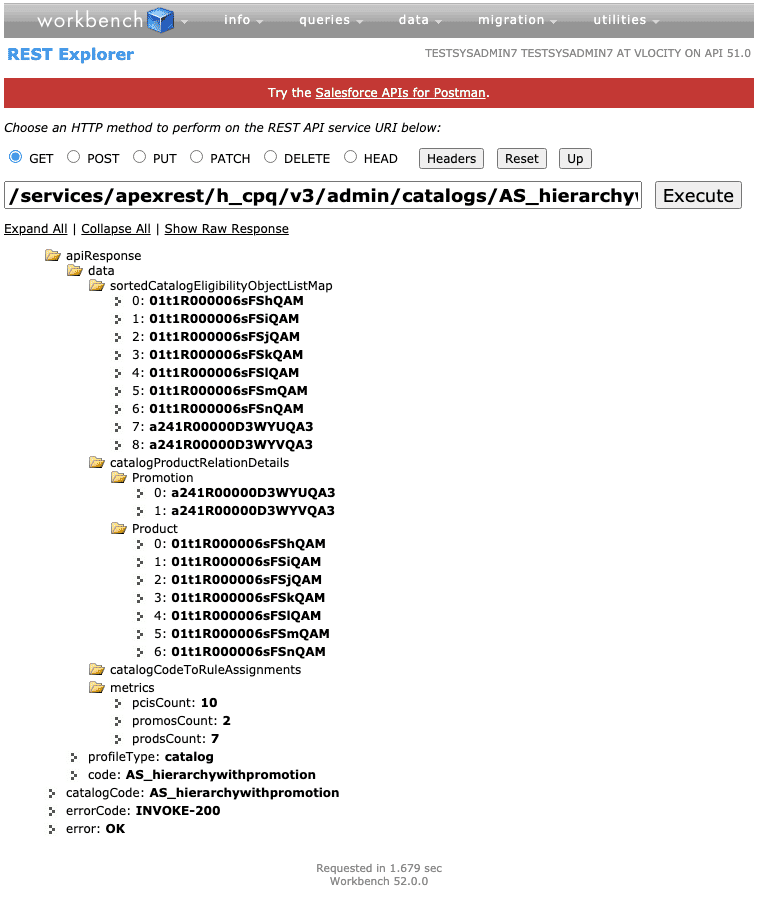
If you do not use the parameters, your cached catalog entries will be effective from the current time for which you are executing the Populate Cache APIs and they will expire as per the value defined by **TimeToLive** in CPQ Configuration Setup (available via Vlocity CMT Administration). By default, **TimeToLive**has a default value of 30 days. If you specify just one of the two parameters, it will follow the behavior described above and accept your defined time for the parameter you specified.

If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

Here is an example of passing these parameters:

GET /services/apexrest/**vlocity\_cmt**/v3/admin/catalogs/**AS-Test-Catalog**/catalogprofile?**effectiveStartTime**=2021-07-04T12:08:56&**expirationTime**=2021-08-04T12:08:56

In the example below, you can see an example response for this API call:



**2. Rule Set Combinations API**

**Required if you have rulesets associated with your catalog. If you do not have any rulesets linked to your catalog, you do not need to perform this API call. If this is the case, the CatalogCodeToRuleAssignments node from the 1. Catalog Profile API's response will be empty.**

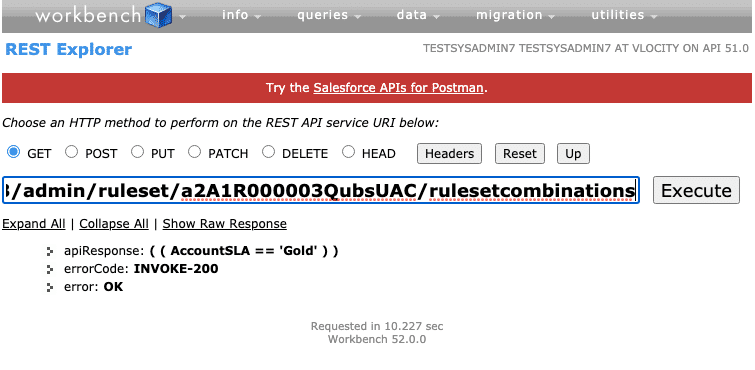
Note that this has a different URI path than other similar calls:

GET /services/apexrest/**{namespace}**/v3/admin/ruleset/**{rulesetId}**/rulesetcombinations

This API generates a string representation of boolean expression created by entity filters involved in the passed ruleset. The **rulesetId**can be retrieved from the response of the previous API call in the **catalogCodeToRuleAssignments**node.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

In the image below, you can see the context dimension relevant to our sales catalog:



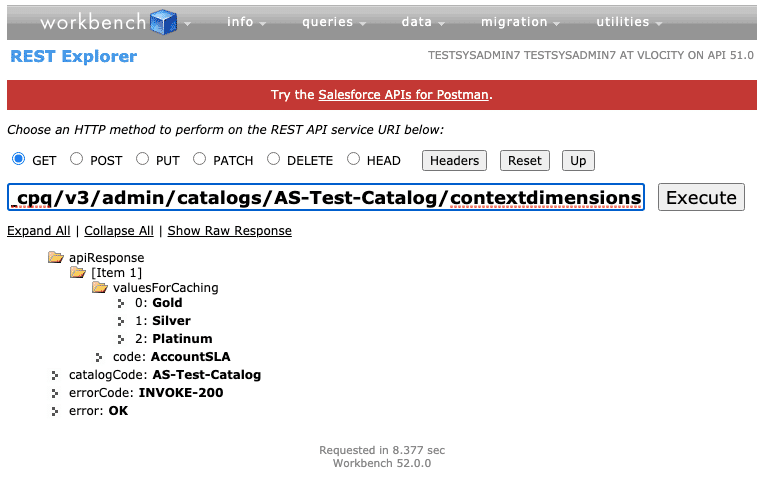
**3. Context Dimensions API**

**Required regardless if you do or do not have any rulesets linked to your catalog.**This is because this API's output is internally required by subsequent APIs, unlike 2. Rule Set Combinations API.

GET /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/contextdimensions

This API generates a list of cacheable and active context dimension codes and their values for caching. This API does not require anything from the previous API response.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**  
  
In the image below, you can see the context dimension values relevant to our catalog:



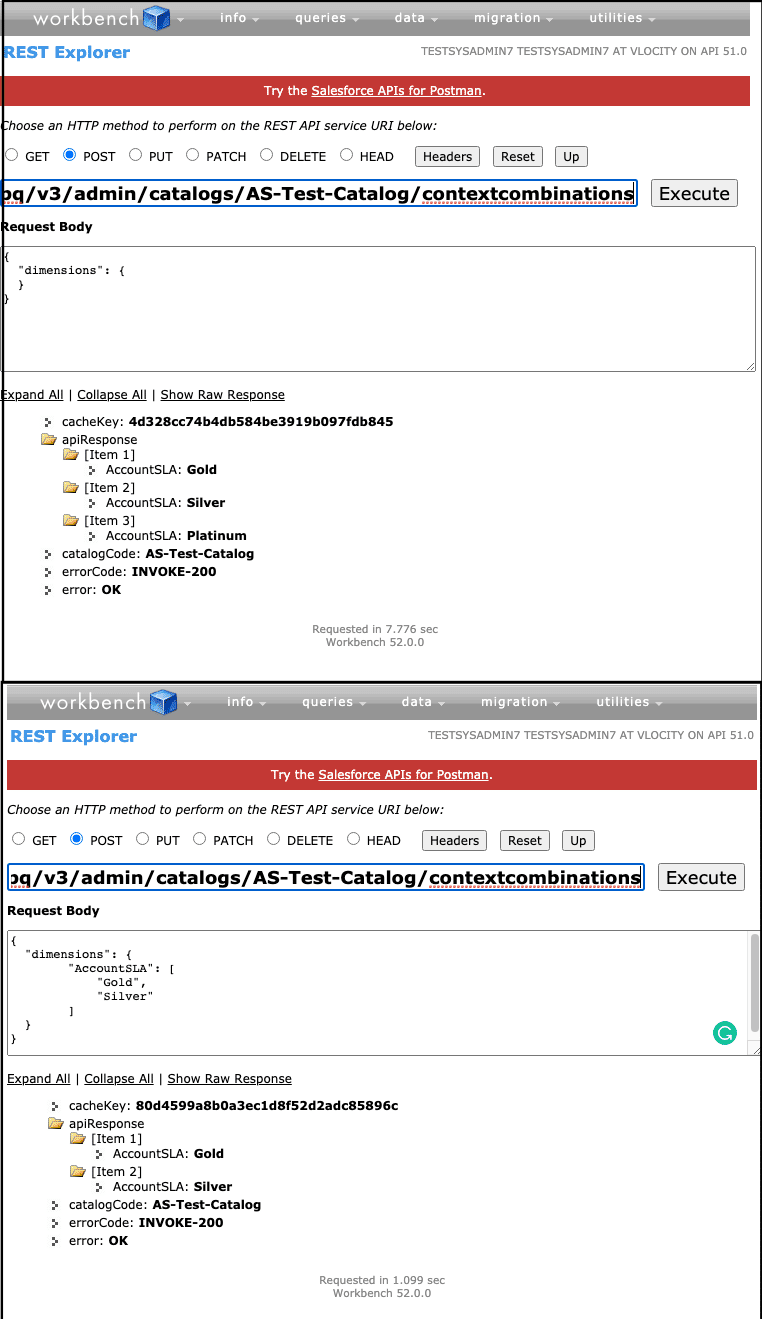
**4. Context Combinations API**

**Required**

POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/contextcombinations

This API generates a cartesian product of all the sets of dimension combination values passed in the **request body**. The **request body**requires the **dimensions**parameter. This parameter can be left blank if you want to perform all the context combinations relevant to your sales catalog or are not using any context dimensions with your catalog. This is demonstrated in the top image below.  
  
If you only want to generate cached entries for specific context dimensions and values, you have that capability with this API. In the bottom image below, you can see that we're only calculating combinations for the Gold and Silver values for the Account SLA dimension. Please see the documentation for more detail on formatting the request body parameters for this API call.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

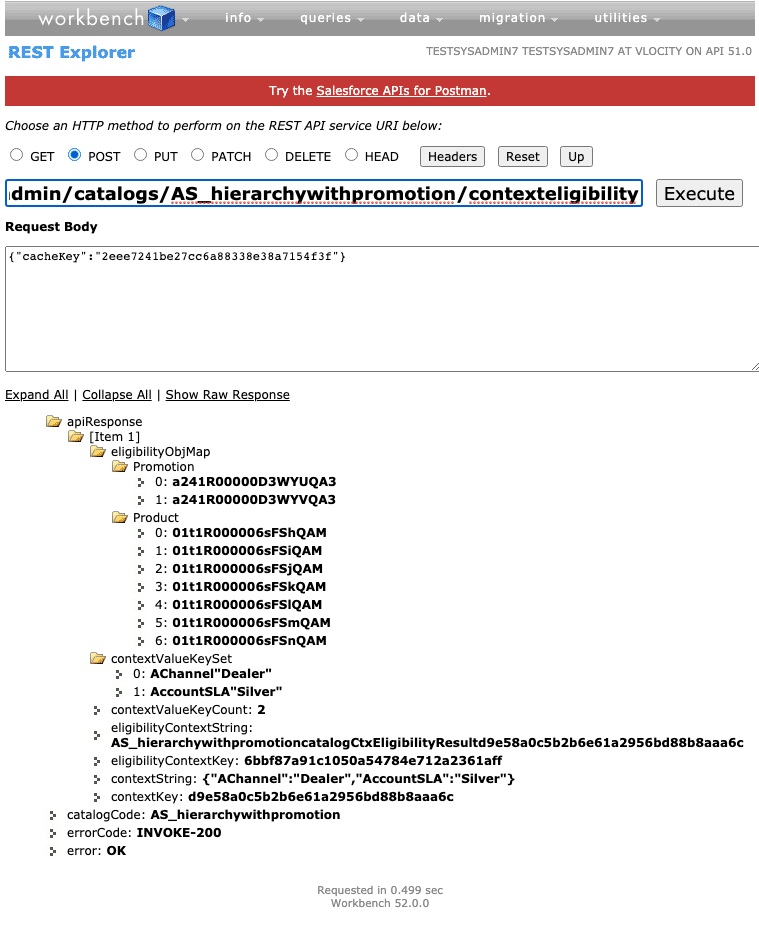


**5. Context Eligibility API**

**Required**  
  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/contexteligibility

The API identifies products and promotions that are eligible for each context combination. It requires a **cacheKey** in the **request body**which is retrieved from **4. Context Combinations API**'s response.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

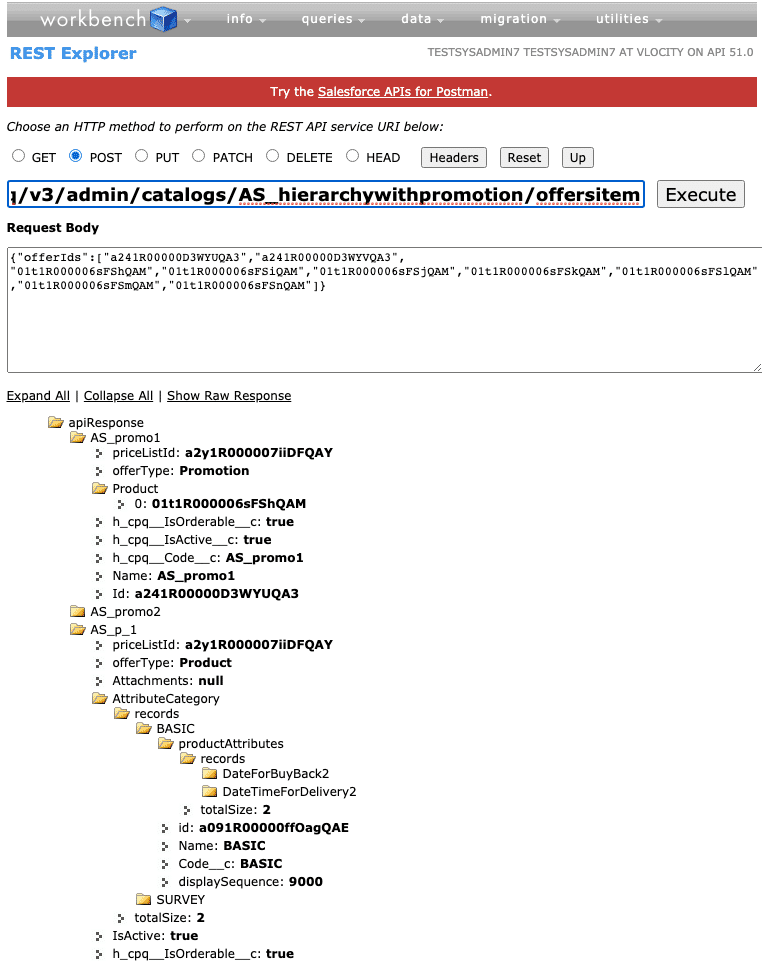


**6. Offers Item API**

**Required**

POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offersitem  
  
This API generates detail for the **offerIds** passed in the**request body**. The offerIds are retrieved from **1. Catalog Profile API**'s response's **Product**and **Promotion**nodes.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

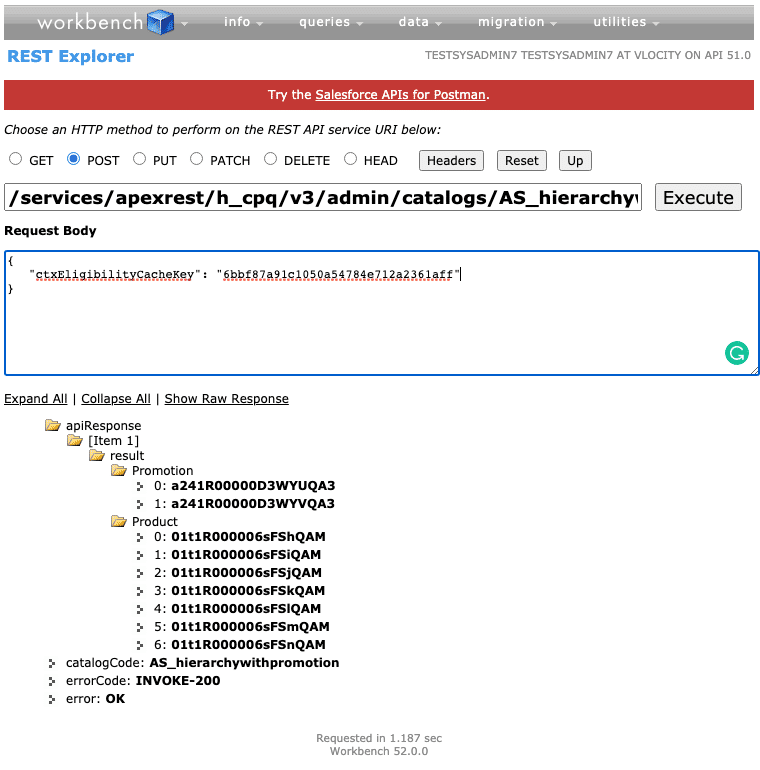


**7. Offers List API**

**Required**  
  
POST /services/apexrest/**{namespace**}/v3/admin/catalogs/**{catalogCode}**/offers

This API paginates the list of eligible offers and ensures the proper sequencing. **ctxEligibilityCacheKey**must be passed in the **request body**and is retrieved from the response of **5. Context Eligibility API**named **eligibilityContextKey**. Note to avoid confusion: ctxEligibilityCacheKeyand eligibilityContextKeyhave similar but different names—copy the value for eligibilityContextKey and paste it into the request body for this call as shown in the image at the bottom.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**



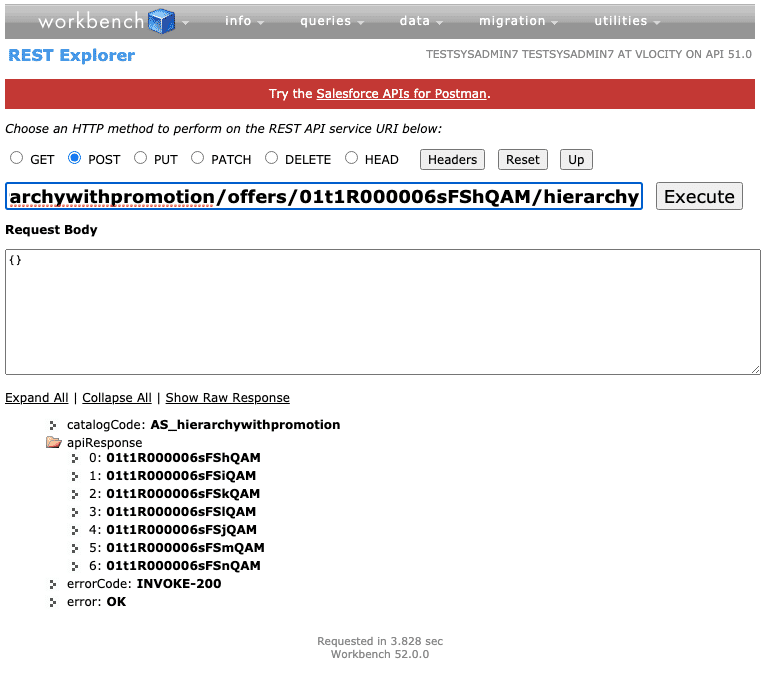
**8. Hierarchy API**

**Required**

POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offers/**{offerId}**/hierarchy

This API is for product Ids only, do not use promotion Ids with this API. Promotions will be handled by **12. Promotion Wrapper API**.  
  
This API takes an individual product's Id (**offerId**) as input in the URI and populates its flat hierarchy in the cache. This API must be executed for each **Product** **offerId** that you received in the**1. Catalog Profile API**'s response.  
  
You must specify {} in the **request body** for this API.

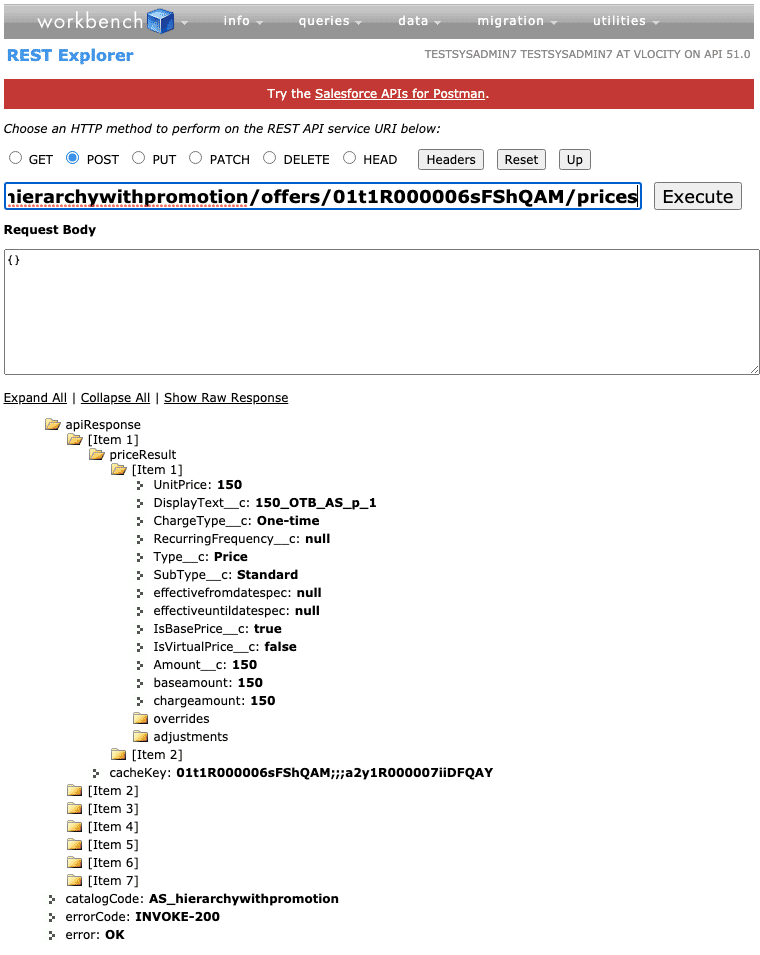
This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**  
  
An example response is shown below:



**9. Prices API**

**Required**  
  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offers/**{offerId}**/prices  
  
This API takes an individual product's Id (**offerId**) as input in the URI and populates its prices in the cache. This API must be executed for each **Product** **offerId** that you received in the**1. Catalog Profile API's** response.  
  
You must specify {} in the **request body** for this API.

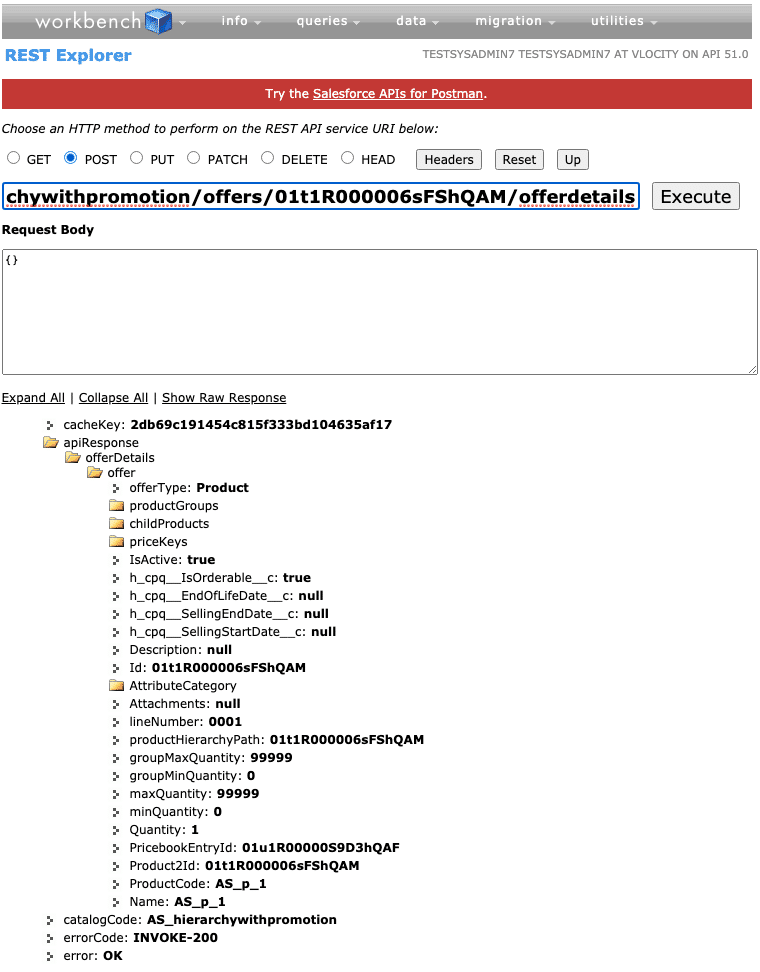
This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**  
  
An example response is shown below:



**10. Offer Details API**

**Required**  
  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offers/**{offerId}**/offerdetails  
  
This API takes an individual product's Id (**offerId**) as input in the URI and populates its details in the cache. This API must be executed for each **Product** **offerId** that you received in the**1. Catalog Profile API**'s response.  
  
You must specify {} in the **request body** for this API.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**  
  
An example response is shown below:

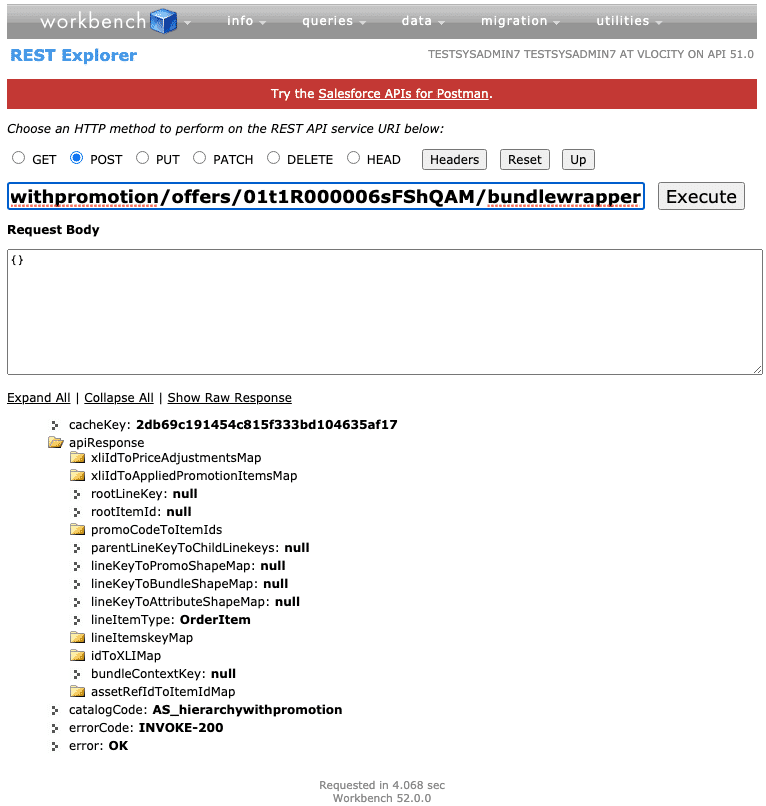


**11. Bundle Wrapper API**

**Required**  
  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offers/**{offerId}**/bundlewrapper  
  
This API takes an individual product's Id (**offerId**) as input in the URI and populates the bundle wrapper in the cache, which are the in-memory XLI objects used during basket operations. This API must be executed for each **Product** **offerId** that you received in the**1. Catalog Profile API's** response.  
  
You must specify {} in the **request body** for this API.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**

An example response is shown below:

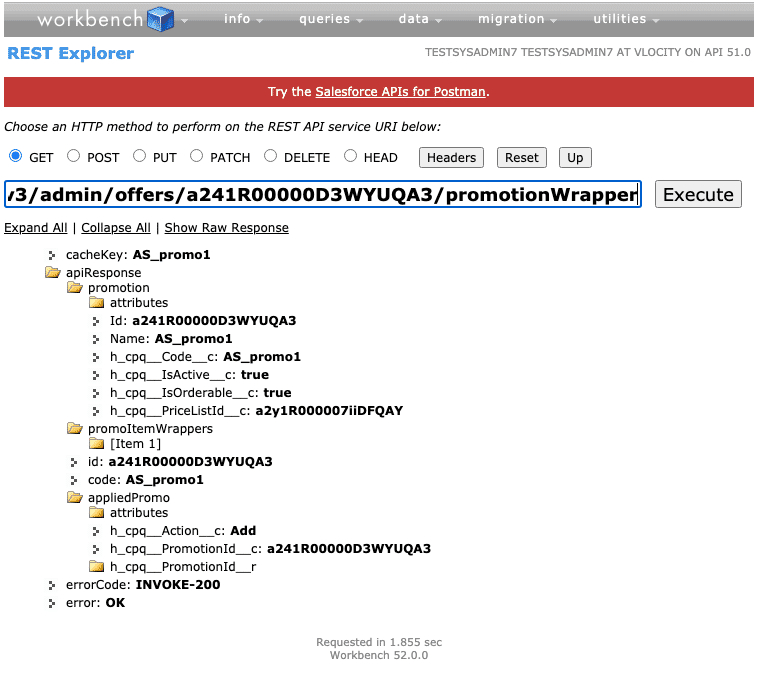


**12. Promotion Wrapper API**

**Required if you have promotions. If you are not using any promotions in your catalog, you do not need to perform this API call.**

Note that this has a different URI path than other similar calls:  
GET /services/apexrest/**{namespace}**/v3/admin/offers/**{offerId}**/promotionWrapper  
  
This API takes an individual promotion's Id (**offerId**) as input in the URI and populates the promotion wrapper in the cache. This API must be executed for each **Promotion** **offerId** that you received in the **1. Catalog Profile API's** response. An example response is shown in the image at the bottom.

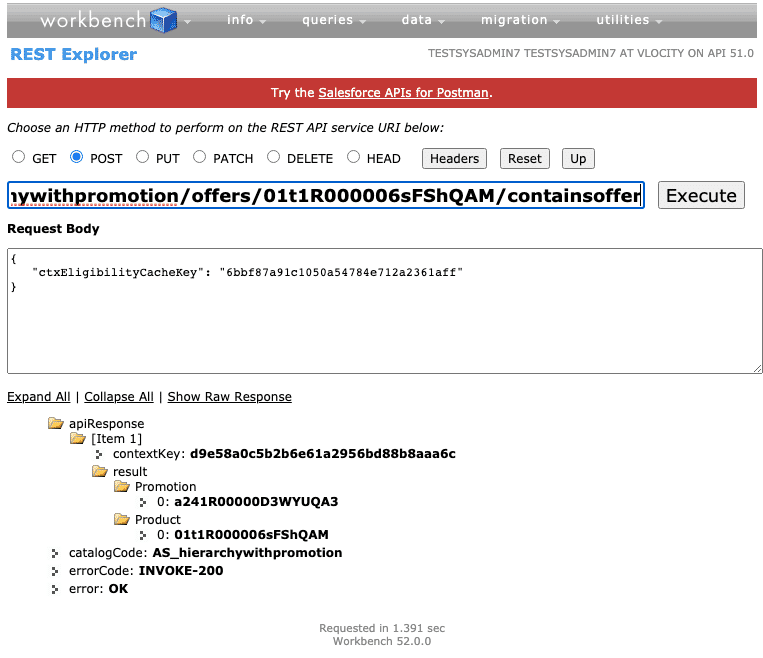
This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**



**13. Contains Offer API**

**Optional**  
  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/**{catalogCode}**/offers/**{offerId}**/containsoffer  
  
This API finds all the products and promotions that contain the given offer in their hierarchy and qualify under the given context combination. It also handles pagination and related action nodes.  
  
**ctxEligibilityCacheKey**must be passed in the **request body**and is retrieved from the response of **5. Context Eligibility API**named **eligibilityContextKey**. Note to avoid confusion: ctxEligibilityCacheKeyand eligibilityContextKeyhave similar but different names—copy the value for eligibilityContextKey and paste it into the request body for this call as shown in the image at the bottom.

This API uses the **effectiveStartTime**and **expirationTime**parameters. Please read the description of how these parameters work and how to use them within the description for **1. Catalog Profile API**. If you are using one or both parameters, **you must specify them with the same values for all of the Populate Cache APIs in this sequence.**



# How to Regenerate the Cache

A set of five REST APIs are available to regenerate the API cache. Before you begin using them, you should be aware of their prerequisites.

**Regenerate the Cache APIs Prerequisites**

Please note, you should have step 1-3 completed **before** you make any catalog changes.

## *Step 1*

### Enable Cache Management in your Custom Settings

1. From **Setup**, in the **Quick Find** box, enter Custom Settings.
2. Click **Custom Settings**.
3. Go to**Trigger Setup**.
4. Next to Trigger Setup, click **Manage**. The Trigger Setup page appears.
5. Click **Edit** beside **CacheAPI.CacheManagement**, and turn the trigger **on**.
6. If you do not see a trigger for **CacheAPI.CacheManagement**:
   1. Click **New**.
   2. Enter CacheAPI.CacheManagement in the **Name** field.
   3. Click**Trigger On**.
   4. Click **Save**.

## *Step 2*

### Set the following CPQ Configuration Custom Settings:

Go to the **Vlocity CMT Administration** tab, and under **Custom Settings** select **CPQ Configuration Setup**. Create the following setting:  
  
**CacheAPI.RegenerateBatchCacheProcessor** = True

This setting only affects the Regenerate the Cache APIs.

## *Step 3*

### Add Picklist Values

Ensure the following picklist values are added to their appropriate object fields specified on this [page](https://docs.vlocity.com/en/CME-Picklist-Values.html):

* CachedAPIChange: vlocity\_cmt\_\_ChangeType\_\_c
* CachedAPIChange: vlocity\_cmt\_\_CodeType\_\_c

## *Step 4*

### Run the following Maintenance Jobs from Vlocity CMT Administration :

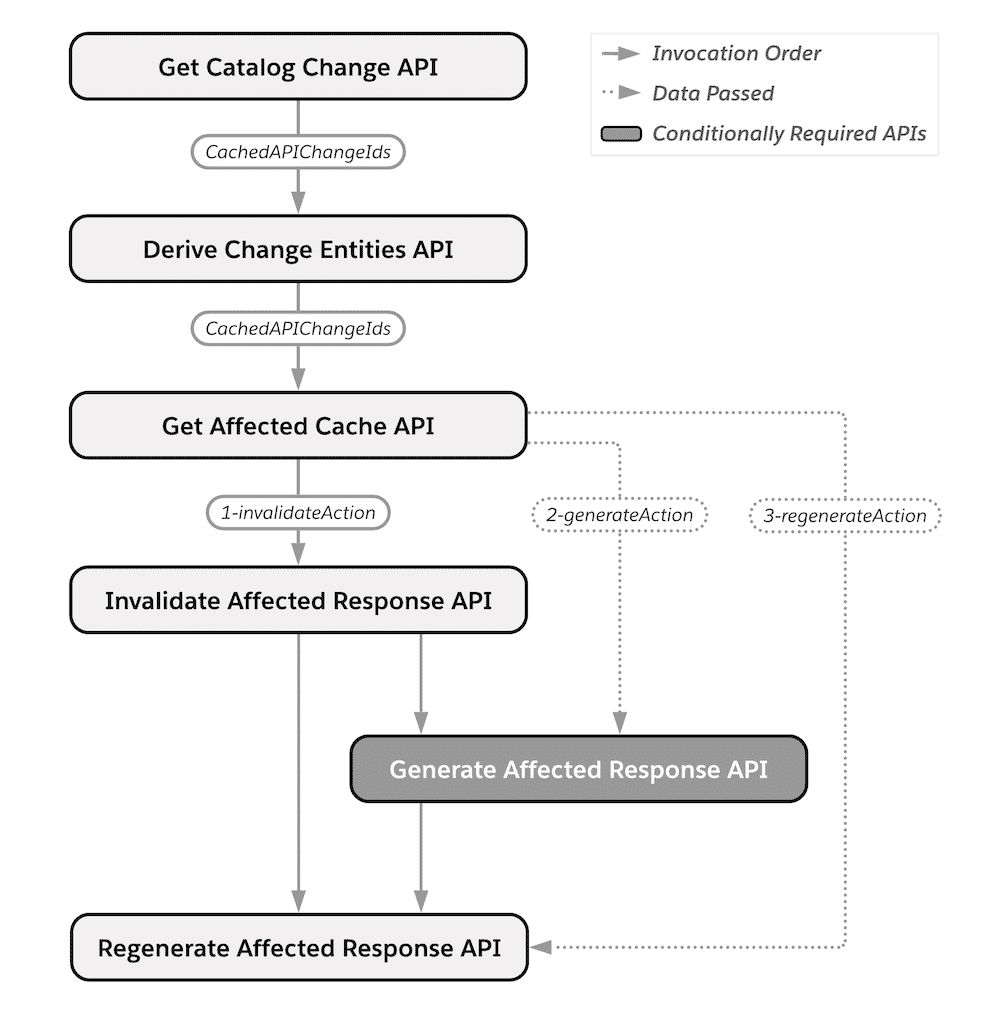
* Run the [Product Hierarchy Maintenance](https://docs.vlocity.com/en/Product-Hierarchy-Maintenance-410596-1.html) Job
* Run the [Clear Managed Platform Cache](https://docs.vlocity.com/en/Clear-Managed-Platform-Cache-410588-1.html) Job

Here are some additional requirements for using the Regenerate the Cache APIs:

* **You must**run the maintenance jobs listed in step 4 above any time you need to execute the Regenerate the Cache APIs,
* **You must**have populated the API cache using the **populate cache APIs**. We do not recommend using the Vlocity CMT Administration Cacheable API Jobs (batch APEX) while using the cache management APIs. Use either the batch APEX jobs **or** the cache management APIs, not both.
* **You must**execute the APIs in the chronological order presented in the next section.
* **You must** execute all of the regenerate cache APIs.
* **You must not** modify your catalog or product offerings until you have executed all of the regenerate cache APIs listed in the next section.

**Regenerate the Cache APIs**

In this section, we will briefly describe each API and demonstrate each of the regenerate cache APIs. Please read the [documentation](https://docs.vlocity.com/en/DC-Regenerate-Cache-APIs.html) for the full description of these APIs.



This is a diagram of the Regenerate Cache APIs in their invocation order. The dotted line represents data passed between API calls. The shaded box represents an API that is only required under certain circumstances, described below.

**1. Get Catalog Change API**

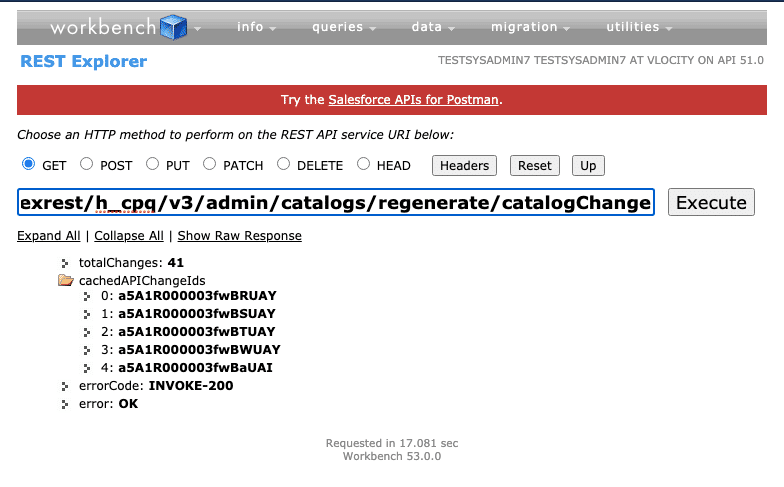
**Required**  
GET /services/apexrest/**{namespace}**/v3/admin/catalogs/regenerate/catalogChange

This API defines all offer and availability changes applicable to your catalogs. Some examples of changes include offer or bundle modifications such as changes in description, attachments, and cardinality changes. Additionally, this API will also define any changes that impact the eligibility/effectivity of an offer.  
  
The custom setting **CacheAPI.RegenerateBatchCacheProcessor** must be set to True for this API to function properly, as covered in the previous section.

You may specify **pageSize** and **offset** parameters for this API if you wish to customize how many and/or which catalog changes it returns. Here is an example of how to use them:

GET /services/apexrest/**vlocity\_cmt**/v3/admin/catalogs/regenerate/catalogChange?**pageSize**=10&**offset**=0

In the example below, we made a Price List Entry modification to a single product within a catalog:



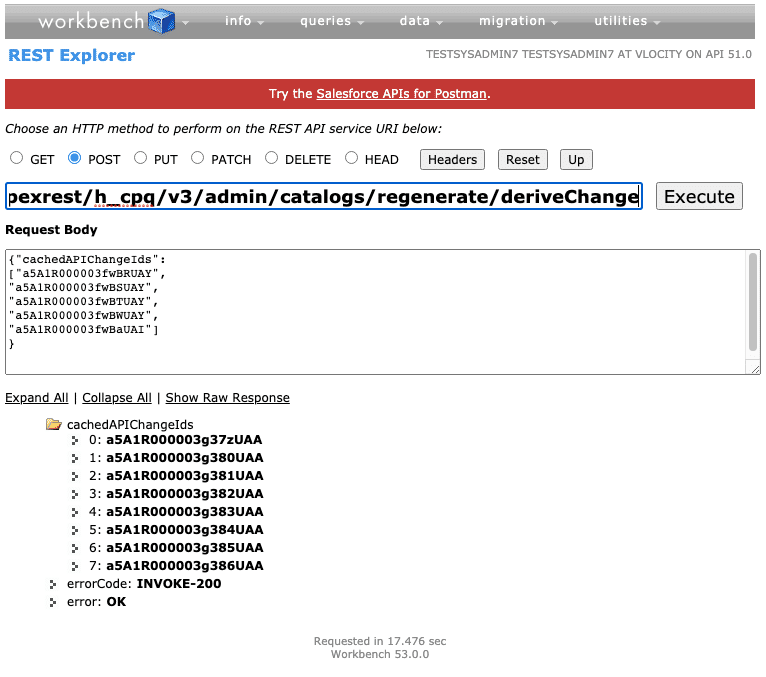
**2. Derive Change Entities API**

**Required**  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/regenerate/deriveChange  
  
The CachedAPIChangeIds received from the previous API should be provided in the request body for this API. This API derives the scope of the impacted cache records from those unprocessed CachedAPIChange Ids.  
  
For example, for a Price List Entry modification, the impacted scope entity is the cached pricing of the offer and cached baskets containing that offer.

**If you are executing APIs in parallel using an orchestrator** (such as the sample regenerate cache orchestrator covered in the next lesson), **you must use the** **uniqueDeriveIds** parameter. This parameter ensures that duplicate records are not created and processed.

Here is an example of how to use it:

POST /services/apexrest/**vlocity\_cmt**/v3/admin/catalogs/regenerate/deriveChange?**uniqueDeriveIds**=true

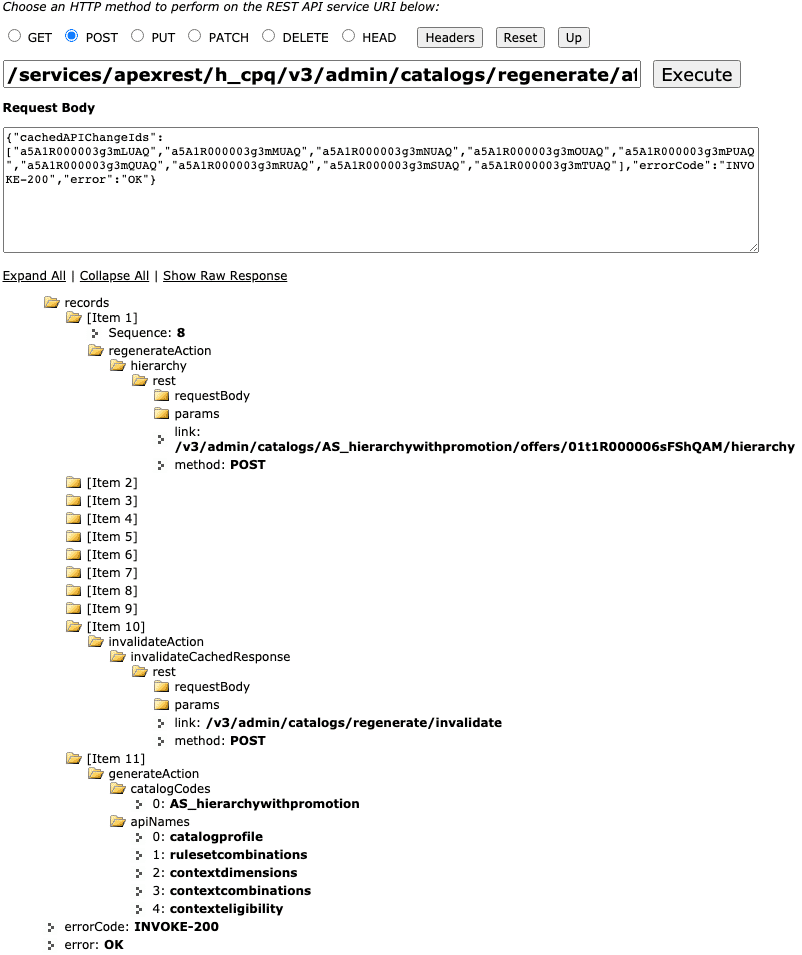


**3. Get Affected Cache API**

**Required**  
POST /services/apexrest/**{namespace}**/v3/admin/catalogs/regenerate/affectedCache  
  
The **cachedAPIChangeIds** received from the previous API should be provided in the request body for this API. This API gets the cached records that will be affected by the derived cachedAPIChange objects.  
  
There are up to 3 different API calls provided by the action nodes of this API's response.They are **invalidateAction**, **generateAction**, and**regenerateAction**.  
  
**They must be executed with the invalidateAction calls first, followed by the generateAction calls, and finally the regenerateAction calls.**In some cases, you will not need to perform generateAction calls, and you will not receive a corresponding node.

If you wish to have this cache effective from a future date, you may use the **cacheEffectiveStartTime** parameter. Here is an example of how to use it:

POST /services/apexrest/**vlocity\_cmt**/v3/admin/catalogs/regenerate/affectedCache?**cacheEffectiveStartTime**=2021-08-24T12:08:56  
  
In the example below, you can see we received multiple regenerateAction nodes and a single invalidateAction and generateAction node. We will cover each of these in 3a, 3b, and 3c.

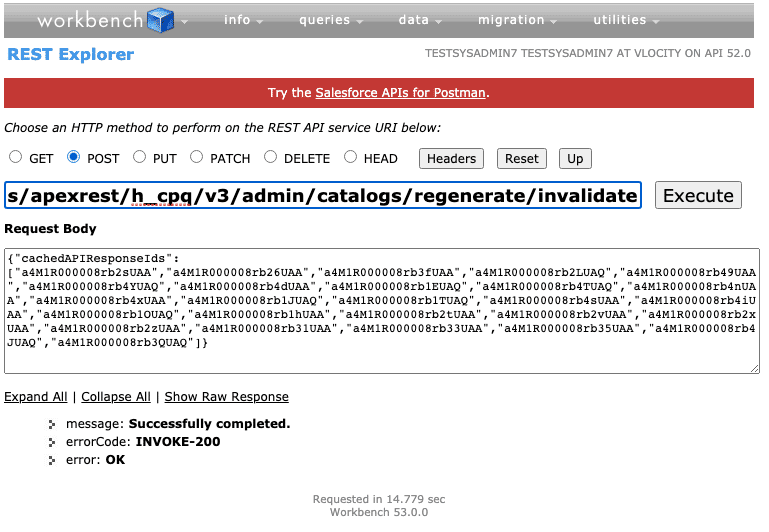


**3a. Invalidate Affected Response API**

**Required**  
POST /services/apexrest/vlocity\_cmt/v3/admin/catalogs/regenerate/invalidate  
  
This API invalidates the cache entries which have become inaccurate due to catalog offer changes. You must pass the cachedAPIResponseIds provided in the invalidateAction node received in **3. Get Affected Cache**'s response.

If you wish to have this cache expire from a certain date, you may use the **expirationTime** parameter. Here is an example of how to use it:

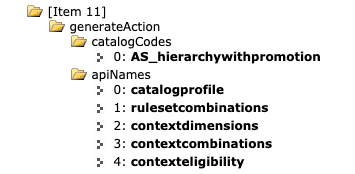
POST /services/apexrest/**vlocity\_cmt**/v3/admin/catalogs/regenerate/invalidate?**expirationTime**=2021-08-24T12:08:56



**3b. Generate Affected Response APIs**

**Required, if the generateAction node is present in 3. Get Affected Cache's response**

Based on the response you receive in **3. Get Affected Cache**, if you have at least one offer availability change, you will need to execute certain populate cache APIs again, using the catalog codes provided in the response. For example, we received the following generateAction (image below). It is asking us to perform 5 of the populate cache APIs with the catalog code AS\_hierarchywithpromotion. Please see the previous lesson for how to perform each of these API calls.



**3c. Regenerate Affected Response API**

**Required**

POST /v3/admin/catalogs/**{value}**/offers/**{value}**/**{APIname}**?ChangeEntryId=**{value}**

You will likely receive several regenerateAction nodes in your **3. Get Affected Cache**'s response.  
  
You must execute an API call for each node in the chronological **Sequence** provided (e.g., perform the node with Sequence: 8, followed by the node with Sequence: 9, etc.) For more information on the sequencing, please see the documentation. You must append the parameter **ChangeEntryId**and its provided valueto the URI.

