# Commerce Services Documentation

This document contains a general feature overview of the Commerce Services resource implementation and lists the currently implemented resources. Each resource conforms to the uniform interface I defined in our other document, summarized here:

1. GET on a collection resource returns a list of items in the collection. We use query parameters to let you determine the sort order, filter the result set, page through the results, specify a response format, and more (all expanded upon below).
2. GET on an item resource returns the full representation of a resource. The media type is determined by your request headers, not a file extension (thus you would not append .xml to a URL to get an XML version of the resource; you'd use an Accept header and request application/xml).
3. POST on a collection resource creates a new item in the collection. The unique ID of the created resource is determined by the application server, not the client.
4. DELETE on an item resource deletes the item if it exists.
5. PUT currently updates a resource, because the Services module does not natively support PATCH. I believe it can be made to do so, but if not I'd use PUT as PATCH and signify it using a specific media type (e.g. application/json-patch+json)

## General features of Commerce Services

Commerce Services defines resources to be exposed through an endpoint defined by the Services module. Services lets you configure the server type, authentication method, request and response media types, and resources for each API endpoint on your site. This means you could create a Commerce REST API by defining an endpoint that:

1. Uses the REST server.
2. Supports JSON for read and write requests.
3. Uses session authentication if users log in to access the API (or some other authentication method, such as our Oauth2 implementation).
4. Enables the required resources (e.g. product, order, user, etc.).

This means you can actually create multiple different API endpoints if you so desire. One could be a read only API for schedule class data exposed to the public while authenticated read and write access to transaction data is served from a completely separate endpoint.

The core resources defined by the Services module do not necessarily conform to the best practices of REST API design. Their support of "targeted actions" for authentication are one example. Additionally, they don't necessarily use the appropriate HTTP response codes and use raw form data for write requests as though you were submitting a Drupal form.

As much as possible, we've attempted to develop our resources according to the best practices identified in the Apigee Web API Design e-book.

**1. Representations of entities are enriched to include the full entities that correspond with the ID values of entity reference fields.**

In an attempt to reduce the number of requests required to get all the data that makes up a product display or an order, we introduced the concept of *entity expansion*. Any representation of an entity requested via GET to a collection or item resource can return any number of referenced entity data in the same request using this query parameter:

- *expand_entities* specifies the depth to which referenced entities should be expanded and included in the return value of the request.

When a Commerce entity is being prepared to be returned by a GET request, Commerce Services iterates over all of the fields on the entity looking for product reference, line item reference, customer profile reference, and taxonomy term reference fields. If any of these are found, the module adds an additional property to the entity named via the pattern [reference_field_name]_entities that includes an associative array of referenced entities keyed by entity ID.

The *expand_entities* parameter defaults to a depth of 1, but you can specify any depth to which you'd like referenced entities to be expanded. To disable entity expansion on any given GET request, you would include the parameter with a value of 0:

- `GET /product-displays?expand_entities=0`

With a value of 1, fetching a product display would also fetch the full representation of the referenced products on the display. If those referenced products in turn had entity reference fields on them, their referenced entities would not be expanded and included in their representations.

With a value of 2, fetching an order would also fetch the full representation of its referenced customer profiles and line items. Since product line items have product reference fields on them, the expanded line item entities found in the order's *commerce_line_items_entities* array would also include a *commerce_product_entities* array that contained full representations of the referenced products.

This process takes place in the `commerce_services_expand_entities()` function.

**2. Field value arrays are flattened to just include the current language's value and to use scalar values for single column fields.**

Drupal field value arrays are multi-level associative arrays. The first level is keyed using language codes even on a single language website, often just using "und" (represented via the `LANGUAGE_NONE` constant in Drupal code). The second level is keyed using the delta value for the current field item, starting at 0 even for single value fields. Finally the third level is the associative array keyed according to the field's schema.

This means that even for a single value field on a single language website, like most base price fields on Commerce products in American usage, you end up with an array that looks like the following:

```
[commerce_price] => Array (
    [und] => Array (
        [0] => Array (
            [amount] => 1000
            [currency_code] => USD
            [data] => Array (
                [components] => Array ()
            )
        )
    )
)
```

This isn't the simplest of data structures to work with, so we introduced the concept of *field flattening* to Commerce Services. When used, the language key is removed so the field value array only includes the current language's value for the field. For single value fields, the delta value level is also removed. Finally, single-column fields are reduced to simple scalar values instead of arrays.

The same price field listed above would appear like this with field flattening:

```
[commerce_price] => Array (
    [amount] => 1000
    [currency_code] => USD
    [data] => Array (
        [components] => Array ()
    )
)
```

A customer profile reference field value that normally looks like:

```
[commerce_customer_billing] => Array (
    [und] => Array (
        [0] => Array (
            [profile_id] => 3
        )
    )
)
```

Would be flattened to:

```
[commerce_customer_billing] => 3
```

(And combined with *expand_entities*, the full value of the referenced customer profile would be found in the commerce_customer_billing_entities array.)

By default, every collection and item resource assumes a depth of 1 for entity expansion, but you can specify any depth you want:

- `GET /product-displays?expand_entities=0`
- `GET /orders?expand_entities=2`

You can also alter these default depths or reuse the same code to expand entities in your custom resources.

Because the field flattening process destroys the integrity of the entity object as far as Drupal is concerned, this is always the last step of processing a response prior to returning the data. We clone the entity object, flatten it, and return it to Services, which in turn converts the entity object to an array and returns it to the client.

This process takes places in the `commerce_services_flatten_fields()` function.

**3. Prior to field flattening, entity representations are decorated with additional helper properties that make it easier to work with their data.**

The steps taken during entity decoration depend on the entity type and fields present on the entity. Currently they include:

- Commerce line items do not have title properties. Instead they have title callbacks that are used to fetch a title based on whatever type of line item it is. For example, the title of a product line item would be its referenced product's title.

  During the entity decoration step, a `line_item_title` property is added to the entity whose value is the line item title returned by the line item type's title callback.

- Commerce products use Drupal fields to identify product attributes that must be chosen by a customer during the Add to Cart process. Accordingly, every product representation includes an `attribute_fields` property that is an array of product attribute field names.

  However, since the product attribute field system is defined by the Cart module, this array will be empty unless the Cart module is enabled.

- Any entity that has a price field on it will have a corresponding property named according to the pattern [price_field_name]_formatted whose value is a string representing the formatted value of the price field according to its currency. For multi-value price fields, this property would include an array of formatted prices.

  The default product type contains a price field called `commerce_price`, so representations of products include a `commerce_price_formatted` property that turn `Array('amount' => 1000, 'currency_code' => 'USD', 'data' => array())` into `$10.00`. This holds true for any price field on any Commerce entity.

- Any entity that has a file field on it, including image fields, will have a corresponding property named according to the pattern [file_field_name]_url whose value is a string representing the full URL at which the file may be found. For multi-value file fields, this property would include an array of URLs.

  If your product type included an image field named `field_product_image`, representations of those products would include a `field_product_image_url` property with a full URL to the image file.

This process takes place in the `commerce_services_decorate_entity()` function. Note that these properties may be included in any shortened response format defined by the *fields* query parameter in GET requests (see feature number 3 below.)

## Commerce Services collection resource features

The following features are specific to collection resources in Commerce Services (what Services calls "indexes" for resources). Any resource should technically be able to take advantage of the same code we use to power these features, but it should probably be confirmed on a case-by-case basis.

**1. Commerce collection resources support filtering by one or more property names, single-column field names, or multi-column field column names.**

**A property** is inherent to an entity - examples include the nid and title of a node or its created timestamp.

**A field** is anything you attach to an entity bundle through the UI or a module - examples include price and reference fields in Commerce or file fields.

**Field types define a schema** that is one or more database columns used to save field data - examples include the product reference field using a single product_id column and the price field using three columns, amount, currency_code, and data. To reference a column in a multi-column field, you must use the pattern [field_name]_[column_name] - for example, commerce_price_amount to identify the amount column of a price field.

Filtering works by specifying the name or names of one of the above as query parameters along with the matching value. For example, to filter products by type or to find unpublished product displays of a given node type, you would use:

- `GET /products?type=ticket`
- `GET /product-displays?type=event&status=0`

Note that individual collection resources define their own default values. The product display filters to only show published nodes, so every GET request to the product display collection does not need to specify `status=1`. These default values can be altered on a resource by resource basis as need be.

When the Commerce Services module detects a request to one of its collection resources, these filter property and field names are combined into a single $filter array during request preprocessing. You don't have to use array notation for filter parameters in the GET request, but you may use it if desired. The following requests would both return a list of product displays created by user 1:

- `GET /product-displays?uid=1`
- `GET /product-display?filter[uid]=1`

Additionally, you may specify an operator for each filter field using array notation with the filter_op query parameter. During request preprocessing Commerce Services populates the array for each filter parameter to a simple equality (=) for any property or field that does not have an explicit operator specified in the request.

For example, the following request filters products to those that cost less than $10:

- `GET /products?commerce_price_amount=1000&filter_op[commerce_price_amount]=<`

When the query is created to fetch the filtered list of items, we translate these *filter* and *filter_op* values into propertyConditions and fieldConditions on an EntityFieldQuery object. Refer to their documentation in `includes/entity.inc` to find appropriate filter operators.

Any collection supporting the *filter* query parameter includes in its definition of that parameter a `commerce_services_field_populate` value that tells Commerce Services where to find the relevant field and property names. This is how the module knows to translate those field and property names found in the query parameters into values in the actual filter array used to filter the query. In the event that a field or property name matches another query parameter, you must use array notation to filter the collection.

Additionally, Commerce Services resource argument definitions support a property called `commerce_services_match_keys` that ensures one argument is an array with keys matching some other argument. The value is an associative array with an `arg` key whose value identifies the argument whose keys must be matched and a `default value` key that specifies the default value to use when filling in missing keys. All of our *filter_op* parameters use this property to set the default values to correspond with each *filter* key.

**2. Commerce collection resources support sorting by one or more property names, single-column field names, or multi-column field column names.**

When it comes to sorting the items in a collection, you can choose to sort by either a property name / single column field name or by the specific column of a multi-value field using the following query parameters:

- *sort_by* specifies one or more sort parameters in a comma separated list; as with a database query, these should be listed in order of sort priority.
- *sort_order* specifies the order of each sort applied to the query, either ASC or DESC, in a comma separated list; default values will not be supplied, so you must supply an equal number of *sort_order* values as *sort_by*.

For example, when you're fetching a list of product displays for display on the front page of an eCommerce mobile application, you might request them sorted so that the most recently created nodes are first with sticky nodes at the top of the list:

- `GET /product-displays?sort_by=sticky,created&sort_order=DESC,DESC`

Alternately, you might want to page through the full list of product displays starting with the first, using an ascending sort on the nod ID:

- `GET /product-displays?sort_by=nid&sort_order=ASC`

The Commerce Services collection resource definitions specify default values for these parameters that can be modified via an alter hook. For any resources that you define using the same paradigm, you should specify your own reasonable defaults.

**3. Commerce collection resources support limiting the result set from full entity representations to smaller arrays of specified properties and fields.**

A collection resource returns a full representation of the items in its collection by default. However, you can specify exactly what you want included in the response to reduce its size using the following query parameter:

- *fields* specifies the properties and fields to include in the result set via a comma separated list.

For example, the following request will return a collection of product display metadata:

- `GET /product-displays?fields=nid,title,body,field_product`

This is useful for cutting down on network traffic when paging through collections, but it doesn't reduce the performance impact of a GET request on the server side. The module still builds the full representation and then just filters the properties on the entity to just include the specified properties on the base entity object. That means you can only specify field names, not field column names.

Additionally, it means any properties Commerce Services adds to entity objects during its "decoration" process can be specified for inclusion ( e.g. formatted prices, file URLs, and expanded entities).

Each collection resource defines the delimiter used to separate property and field names in this parameter's value via the `commerce_services_explode` property. This can be altered on a resource-by-resource basis, or you can define resources that use different delimiters if necessary.

Each collection resource also defines the required fields that must be present in any response even via the `commerce_services_required_fields` property. It is an array of property and field names that will always be added to the response even if not specified in the *fields* query parameter. For example, for product displays this is the nid of the node, and for shopping carts it is both the cart order ID and the related user ID.

**4. Commerce collection resources support paging.**

Per Apigee's recommendation, we use two separate query parameters to control paging:

- *limit* specifies the number of items to include in the response; defaults to 10.
- *offset* specifies how many items to offset the result set by; defaults to 0.

Thus paging through the product display collection would look like:

- `GET /product-displays?limit=10&offset=0`
- `GET /product-displays?limit=10&offset=10`
- `GET /product-displays?limit=10&offset=20`

To page through a collection, we recommend using an explicit sort as a best practice. The default sort order may be reasonable, but coding your clients to use explicit sort values protects them against future changes in default sort values at the server.

Ideally your collection media type would support a links section for collections resources that contains links to the first, last, next, and previous pages as available.

## Commerce Services resource table

The following table identifies all the resources and methods available in Commerce Services out of the box at present.

| URI | Method | Description |
|---|---|---|
| /product-displays | GET | Returns a collection of product display nodes. |
| /product-displays/# | GET | Returns a single product display. |
| /products | GET | Returns a collection of products; in Drupal Commerce, this is an entity including at least a SKU, a title, and a base price. |
| /products | POST | Creates a new product; returns the newly created product in the body of the response. |
| /products/# | GET | Returns a single product. |
| /products/# | PUT | Updates a single product; returns the updated product in the body of the response. |
| /products/# | DELETE | Deletes a single product. |
| /carts | GET | Returns a collection of shopping cart orders with a single item in it: the current API user's current shopping cart order. |
| /carts | POST | Creates a shopping cart order for the current API user; does not permit you to specify a uid value. For that, you'd just create an order of the appropriate status via a POST to /orders. Returns the newly created shopping cart order in the body of the response. |
| /orders | GET | Returns a collection of orders. |
| /orders | POST | Creates a new order; returns the newly created order in the body of the response. |
| /orders/# | GET | Returns a single order. |
| /orders/# | PUT | Updates a single order; returns the updated order in the body of the response. |
| /orders/# | DELETE | Deletes a single order. |

| | | |
|---|---|---|
| /orders/#/line-items | GET | Returns a collection of line items on the given order. This is a convenience resource that is equivalent to the response of a GET request to /line-items?order_id=#. |
| /line-items | GET | Returns a collection of line items; in Drupal Commerce we use a bi-directional reference between orders and line items, so you can identify the order any line item belongs to via its `order_id` property. |
| /line-items | POST | Creates a new line item; returns the newly created line item in the body of the response. |
| /line-items/# | GET | Gets a single line item. |
| /line-items/# | PUT | Updates a single line item; returns the updated line item in the body of the response. |
| /line-items/# | DELETE | Deletes a single line item. |

Note: we aren't using OPTIONS anywhere, preferring instead to identify available query parameters via hypermedia in the response. This requires using an appropriate media type for responses like Collection+JSON, but that is out of scope for the Commerce Services module at present. I've begun a separate module to add Collection+JSON support to Services, but it isn't on drupal.org at present.

Additionally, I'd like to explore using PUT (or PATCH) and DELETE requests on collection resources in the future. For example, you might want to update the quantity of all line items on an order at once or remove all line items from an order to "empty" a cart. Technically speaking, REST would support a DELETE to /orders/#/line-items that deleted every line item in the collection.