FTG FEARLESS TECHNOLOGY GROUP

commercetools
Next generation commerce

Building commercetools

customizations using Custom Objects

www.commercetools.com

Table of Contents

Introduction

> ***"The most important capability of any ecommerce system for a large company is customizability. The CDOs we spoke with were largely not interested in turn-key solutions. These entities prefer to have complete creative control over the environment their customers encounter."***
>
> – The New Chiefs of Commerce in the Digital Enterprise, page 10

commercetools is a dynamically extensible, cloud-native commerce solution. It allows retailers to sculpt a solution that fits their unique needs today, and is flexible to support their evolving business strategy tomorrow.

There are many powerful extensibility features built into commercetools that handle a wide variety of use cases. For an overview of them, see Building commercetools customizations - Overview.

In this whitepaper we will do a deep dive on one powerful technique for customizing commercetools: Custom Objects.

What are Custom Objects?

Custom Objects allow you to create a new resource that you would like to store within commercetools and integrate with other aspects of the system.

> ***"Custom objects are a way to store arbitrary JSON-formatted data on the commercetools platform. It allows you to persist data that does not fit the standard data model. This frees your application completely from any third-party persistence solution and means that all your data stays on the commercetools platform"***
>
> – Custom Objects, Platform Documentation, commercetools

For instance, you may have an entity that is specific to your business domain and does not correspond to an existing commercetools resource. In this case, you can define the entity within commercetools using a Custom Object.

You can also use Custom Objects in combination with Custom Fields. For example, you could extend the `customer` resource with a Custom Field that references a complex data type stored in a custom object. For a detailed tutorial on Custom Fields, see Building commercetools customizations - Custom Fields.

Bottom line, Custom Objects provide a lot of power. In fact they're so powerful

that it's tempting to build complex solutions leveraging the feature! However, we advise some caution because in more difficult use cases, it's possible to introduce performance problems. Complex issues that require validation, richer data structures and additional business rules typically push you toward a more tailored solution using techniques like custom microservices . If you're in doubt, contact FTG and we can help you find the best path.

Let's drill down on a sample use case to see how to exploit Custom Objects.

Implementation Example

Let's use an example where we create a set of related Custom Objects that can be referenced from a Custom Field. After creating the objects, we can then update an example front end to use these new customizations. This will allow us to create a full stack example showcasing the power of this commercetools customization option.

We will use a fictional use case that does **not** have much validity in the real world. We'll choose something that we hope is fun and allows us to focus more on how to enhance commercetools and less on use case plausibility.

Use Case

A retailer wishes to store a list of all important front end features they've deployed. They then want to reference this list to track which features the customer used as part of creating and updating their cart. This use case would be better handled by a robust data analytics solution but we'll toy with it and have fun anyway!

Problem

There is no resource on the commercetools platform that allows us to store a list of deployed features.

Solution

Create Custom Objects to store the list of customer-facing features on the commercetools platform. Use a Custom Field on the cart resource to store references to feature objects as the customer uses them.

Implementation Steps

Our example implementation is thorough! We will show you how to:

A. Create a commercetools project,
B. Load sample data,
C. Model the "Features" Custom Object,
D. Create, Read, Update and Delete "Features",
E. Create a new Type to define the cart Custom Field,
F. Create a Cart,
G. Update the Cart to add Features used,
H. Test: Query to find all carts that used a specific feature,
I. Install the Sunrise SPA Front End,
J. Extend the front end to use our Custom Objects,
K. Provide you with resources if you need help.

Step A: Create a commercetools project.

If you already have a commercetools project, you can skip this step. If not, there is good news: you can easily sign up for a risk-free, fully-functional 60 day commercetools trial. The trial period does introduce a few limits, like the total number of products you can define, but the feature set is rich as we will see in this paper.

Go to https://commercetools.com/free-trial and fill out the form to get an email with instructions for creating your trial organization and initial project. The process is quite fast because commercetools automates all the work behind the scenes to provision cloud resources for you. Note the key you used for your project as it will be used in Step B. Once you have your first project in place, proceed to the next step.

Step B: Load sample data.

If you already have a commercetools project loaded with sample data, you can skip this step. If not, commercetools provides an open source project to make this easy. If you're comfortable running open source tools, you may prefer to just follow the steps in the Sunrise Data README; if not, here is what you should do:

1. Open a command line (our examples use bash) and issue these three commands to clone and initialize the commercetools-sunrise-data open source repository:

```
$ git clone
https://github.com/commercetools/commercetools-sunrise-data.git
$ cd commercetools-sunrise-data/
$ npm install
```

2. The commercetools-sunrise-data application needs some configuration so it knows what project to load the data into and has the credentials it needs to perform its work. Here are the steps:

a. Login to the Merchant Center and then navigate to Settings -> Developer Settings from the left navigation.
b. Click “Create new API Client”
c. For field **Name**, enter: `admin-client`
d. For field **Scopes**, select: `Admin client`
e. Click “Create API Client”
f. Note all the information provided by the Merchant Center as we will use them in the next step.

3. Now that we have the configuration details we need, we can create a .env file for the commercetools-sunrise-data application to leverage. Create a new file called `.env` at the root of your commercetools-sunrise-data directory. It should have the following entries; replace the generic values with information you captured in the previous step. If you lost your configuration details, you can perform the previous step again and create a new API Client without harm:

```
CTP_PROJECT_KEY = <your project key>
CTP_CLIENT_ID = <your client ID>
CTP_CLIENT_SECRET = <your client secret>
CTP_API_URL = <your apiUrl> (i.e., api.commercetools.com)
CTP_AUTH_URL = <your authUrl> (i.e., auth.commercetools.com)
```

4. You are now ready to load data. Assuming all the previous steps were successfully followed, a single command will load data for you. Note that this command will **replace all data in the project!** If you need to retain existing data, see further instructions in the README.md.

```
$ npm run start
```

Step C: Model the “Features” Custom Object.

To create a Custom Object, we need to model a CustomObjectDraft which has three required and one optional field:

- `container` - String, matching the pattern `[-_~.a-zA-Z0-9]+` A namespace to group custom objects.
- `key` - String, matching the pattern `[-_~.a-zA-Z0-9]+` A user-defined key that is unique within the given container.
- `value` - JSON types Number, String, Boolean, Array, Object
- `version` - Number - Optional

We want to create a set of related custom objects, one for each “Feature” we plan to describe and reference. We will relate each of these objects by having them all share the same `container` string: “features”. The `key` and `value` will differ for each object. The `value` will be simple, just a name and description. Let’s look at two examples:

```
{
  “container”: “features”,
  “key”: “header-keyword-search”,
  “value”: {
    “name”: “Keyword Search from Header”,
     “description”: “Customer used the keyword search feature from
the header”
  }
}
```

```
{
  “container”: “features”,
  “key”: “quick-shop”,
  “value”: {
    “name”: “Quick Shop”,
      “description”: “Customer used the quick shop feature from
products list”
  }
}
```

It’s possible to create much more complex objects in the `value` field and you can even reference other commercetools objects. If you decide to reference other objects, there are two important things to note:

- References can be expanded. This is a core commercetools feature that allows for compact query results by default (only the reference ID is returned) but allows the client to request expansion as part of a query so the result includes the full object being referenced. Awesome!
- Referential integrity is not guaranteed. If the referenced object is deleted, the custom object reference will not be deleted and will point to a non-existing object. You can work-around this issue using Subscriptions to listen for relevant delete events and asynchronously clean up your custom object references. You can learn more in Building commercetools customizations - Subscriptions.

Step D: Create, Read, Update and Delete “Features”.

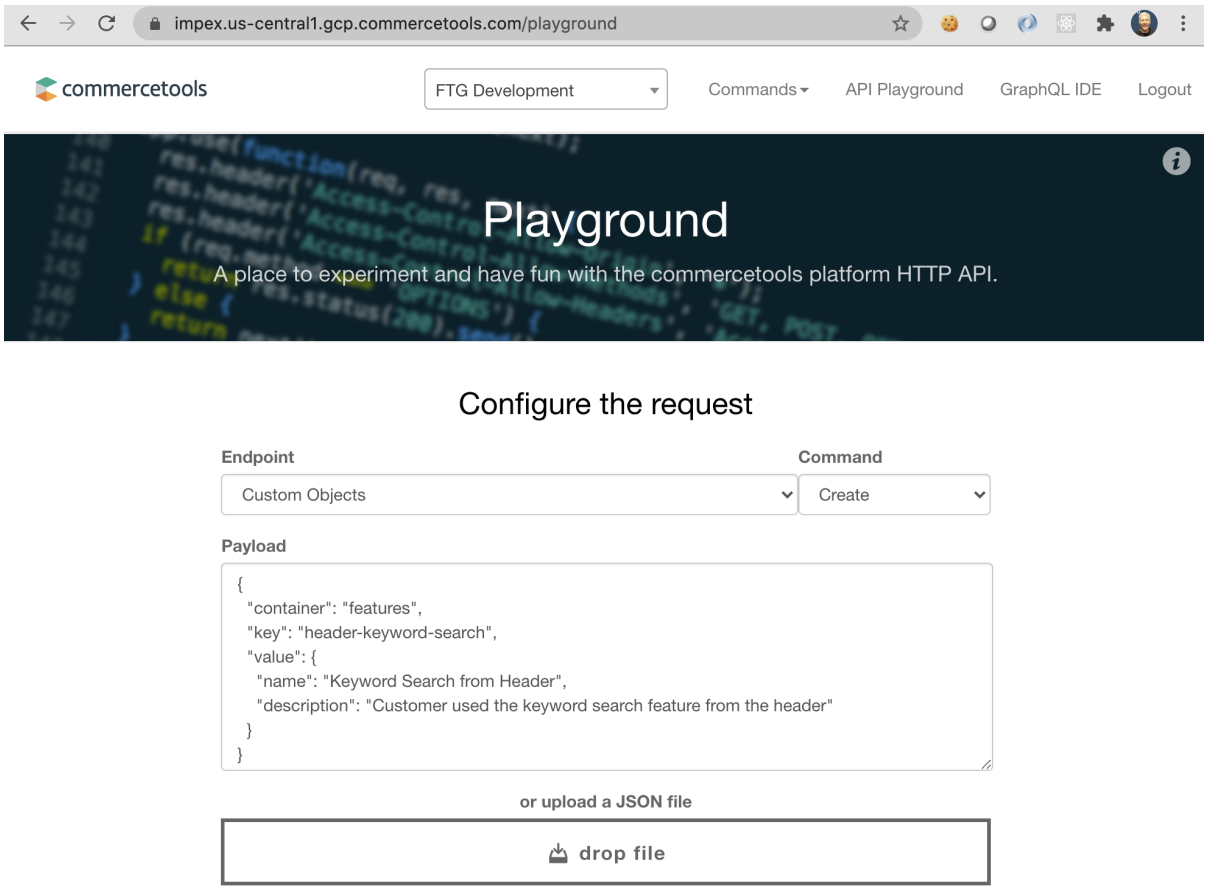
Custom Objects support all CRUD operations just as you would expect from a system that persists entities: Create, Read, Update, Delete.

Create

Let’s look at two ways to create our custom objects: IMPEX’s API Playground and direct calls to the HTTP API.

To leverage IMPEX, go to the API Playground and login. There is a drop down control in the header to allow you to select a project. Make sure you have the right one selected. In the **Endpoint** field, select “Custom Objects”.

In the **Command** field, select “Create”. In the **Payload** field, paste one of the JSON examples from the previous step. Things should look something like this:



Click the “Go!!!” button to save the object. IMPEX should give you a response similar to:

```
{
  “id”: “d289c9f5-64bd-4e76-956a-df482aa7c0ff”,
  “version”: 1,
  “createdAt”: “2020-10-05T20:30:29.177Z”,
  “lastModifiedAt”: “2020-10-05T20:30:29.177Z”,
  “lastModifiedBy”: {
    “isPlatformClient”: true
  },
  “createdBy”: {
    “isPlatformClient”: true
  },
  “container”: “features”,
  “key”: “header-keyword-search”,
  “value”: {
    “description”: “Customer used the keyword search feature from the
header”,
    “name”: “Keyword Search from Header”
  }
}
```

To alternatively use the HTTP API, we can take advantage of tools like Postman or curl. If you are familiar with Postman, commercetools provides a repository containing Postman collections for the platform. We will show examples using curl. If you're using Windows locally, this guide from Zendesk may be useful as it documents modifications you may need to make.

We can use curl in two steps. First, we need an authorization token. Run the following from the command line, substituting AUTH_HOST, CLIENT_ID, SECRET and PROJECT_KEY with data we noted in Step B:

```
curl https://AUTH_HOST/oauth/token \
--basic --user "CLIENT_ID:SECRET" \
-X POST \
-d "grant_type=client_credentials&scope=manage_project:PROJECT_KEY"
```

Second, we use the returned `access_token` to provide authorization when performing a POST to the custom-fields endpoint. Run the following from the command line, substituting ACCESS_TOKEN with the `access_token` returned in the previous step and API_HOST and PROJECT_KEY with data we noted in Step B:

```
curl -sH "Authorization: Bearer ACCESS_TOKEN" \
-H 'content-type: application/json' \
-d '{"container": "features", "key": "quick-shop", "value": {"name":
"Quick Shop", "description": "Customer used the quick shop feature
from products list"}}' \
https://API_HOST/PROJECT_KEY/custom-objects
```

Your response will be similar to the JSON response from IMPEX above.

Note that in either technique, if you include a non-zero `version` number on a Create, the response will be a statusCode 400 with a message, "version on create must be 0". So, you can just omit that field. You will use it during Updates, see below.

Read

We can perform read operations by using the HTTP API Query CustomObjects endpoint at `/{projectKey}/custom-objects/{container}/`. As you can see, this endpoint allows us to include the `container` field in the path to allow us to scope the returned list to include only our "features". For example, try the following (be sure to use your ACCESS_TOKEN and PROJECT_KEY):

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
https://API_HOST/PROJECT_KEY/custom-objects/features/
```

The two custom objects we generated in the Create section will be returned. You can supply additional query constraints by optionally providing url parameters like `where`, `sort`, `limit`, `expand` and `offset`.

As you'd expect, IMPEX supports custom object queries as well. Set Endpoint to **Custom Objects**, set Command to **Query** and set Where to **key="header-keyword-search"**. This query will return one result containing the first custom object we created.

Update

You can update existing custom objects using the same technique as we used in Create above. If an object with the same `container` and `key` exists, the object will be replaced with the new value and the `version` will be incremented. If the request contains a version and an object with the given `container` and `key` exists, then the `version` must match the `version` of the existing object. Concurrent updates for the same custom object can result in a Conflict (409) even if the `version` is not provided.

Also note that fields with **null** values will not be saved.

Delete

We can delete a custom object by using the HTTP API Delete CustomObject endpoint at `/{projectKey}/custom-objects/{container}/{key}`. The unique combination of `container` and `key` identifies the object that will be deleted. For example, try the following (be sure to use your ACCESS_TOKEN and PROJECT_KEY):

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
-X DELETE \
https://API_HOST/PROJECT_KEY/custom-objects/features/quick-shop
```

If you re-execute the Read curl command you will no longer find our **quick-shop** key in the result set. Go ahead and add it back again using the curl command from section Read above.

Note that you can optionally provide parameter `dataErasure` when deleting a custom object. This parameter is common to many DELETE requests on the commercetools platform to support legal obligations surrounding Data Erasure of Personal Data required in many countries. The field defaults to **false** but if you set it to **true**, the commercetools platform guarantees that all personal data related to the object, including invisible data, is erased in compliance with the GDPR.

Step E: Create a new Type to define the cart Custom Field.

Next, we need a way to associate our new custom objects with customer carts. To do that, we need to be able to add a Custom Field to the Cart resource. Before you can use a custom field on a resource, you will need to define a new Type. We do a deep dive into Custom Fields in a separate paper, see Building commercetools customizations - Custom Fields for details.

Here is the Type we will use:

```
{
  “key”: “features-used”,
  “name”: {
    “en”: “Features used by the customer”
  },
  “resourceTypeIds”: [“order”],
  “fieldDefinitions”: [
    {
      “type”:{
        “name”: “Set”,
        “elementType”: {
          “name”: “Reference”,
          “referenceTypeId”: “key-value-document”
        }
      },
      “name”:”features”,
      “label”:{
        “en”:”Features Used”
      },
      “required”:false
    }
  ]
}
```

The `fieldDefinitions` define a **“Set”** of references to “**key-value-document”**. This creates a data type that allows us to add an array of references to our custom objects, one entry for each feature the customer uses.

You might be wondering why we set resourceTypeIds to [“**order**”] when we want to use the Custom Field on the customer’s Cart. You’ll find the reason in the list of Customizable Resources which shows us that resource Cart has a resource type id of `order`: “When a Cart is ordered, the CustomFields are copied to the Order. Therefore the Custom Type for Orders is also valid for Carts.”

We can use the HTTP API or IMPEX to create the new Type. Here is an example using curl (be sure to use your ACCESS_TOKEN and PROJECT_KEY):

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
-H ‘content-type: application/json’ \
-d ‘{“key”: “features-used”, “name”: {“en”: “Features used by the
```

```
customer"},  "resourceTypeIds":  ["order"],  "fieldDefinitions":
[{"type":{"name":  "Set",  "elementType":  {"name":  "Reference",
"referenceTypeId":  "key-value-document"}},  "name":"features",
"label":{"en":"Features Used"}, "required":false}]}' \
https://API_HOST/PROJECT_KEY/types
```

Once created, you can query for the Type by key as well:

```
curl -sH "Authorization: Bearer ACCESS_TOKEN" https://API_HOST/
PROJECT_KEY/types/key="features-used"
```

Step F: Create a Cart.

Let's create a cart so we can experiment with our custom field and custom objects. We can create an empty cart by sending a POST to the cart resource, sending two fields in the JSON body: `currency` and `country`.

```
curl -sH "Authorization: Bearer ACCESS_TOKEN" \
-H 'content-type: application/json' \
-d '{"currency": "USD", "country": "US"}' \
https://API_HOST/PROJECT_KEY/carts
```

The key thing we need from the response is the `id` field, which uniquely identifies this cart, so keep that handy. You will see that our custom field is not present in the response; we will rectify that in the next step!

It's important to note that even though `country` is not required in the CartDraft specification, subsequent cart updates may fail with errors if it's not specified. The error stems from the sunrise data set we imported. It has the "EU" as default and USD pricing is only set for country "US". So, if you create a cart with no country and USD, it can't find a matching price. By setting USD and US it can find the price. Another option is to modify the product data, then country would not be required at the cart level. If we get it wrong, here is a snippet from an error that may result:

```
{
  "statusCode": 400,
  "message": "The variant '1' of product '0d34b0cc-6dc0-4df7-943c-
321500dc492c' does not contain a price for currency 'USD' all
countries, all customer groups and all channels.",
  "errors": [
    {
      "code": "MatchingPriceNotFound",
    [...]
}
```

Step G: Update the Cart to add Features used.

Now that we have a cart, we can start taking advantage of our **"features-used"** Type to reference our custom objects. To Update a Cart we need to supply an array of Update Actions we want to perform. For our use case, we want to leverage the action Set Custom Type. Let's say the customer used both of the **"features"** we created custom objects for in Step D. We want to reference the custom objects by their unique `id`. This will allow us to use reference expansion as we will see later.

To get the `id` values of our **"features"** custom objects, run the curl command we used in section Read above:

```
curl -sH "Authorization: Bearer ACCESS_TOKEN" \
https://API_HOST/PROJECT_KEY/custom-objects/features/
```

The command will return a result like the following but note that **your** `id` **values will be different** as they were uniquely generated when you created your custom objects:

```
{
  "limit": 20,
  "offset": 0,
  "count": 2,
  "total": 2,
  "results": [
    {
      "id": "d6490f1a-dceb-4efa-8f29-46f18dc950b1",
      [...]
      "container": "features",
      "key": "header-keyword-search",
      "value": {
       "description": "Customer used the keyword search feature...",
        "name": "Keyword Search from Header"
      }
    },
    {
      "id": "0b04e62f-65a1-4daf-80e5-42aedccb4175",
      [...]
      "container": "features",
      "key": "quick-shop",
      "value": {
        "description": "Customer used the quick shop feature...,
        "name": "Quick Shop"
      }
    }
  ]
}
```

We can now reference each `id` value in the `features` array in our update to the cart we created. Replace the `id` values based on your execution of the last curl command:

```
{
  “version”: 1,
  “actions”: [
    {
      “action”: “setCustomType”,
      “type”: {“key”: “features-used”, “typeId”: “type”},
      “fields”: {
        “features”: [
          {
            “typeId”: “key-value-document”,
            “id”: “d6490f1a-dceb-4efa-8f29-46f18dc950b1”
          },
          {
            “typeId”: “key-value-document”,
            “id”: “0b04e62f-65a1-4daf-80e5-42aedccb4175”
          }
        ]
      }
    }
  ]
}
```

Here is the curl command we can use to update our cart. Be sure to substitute CART_ID with the cart’s id noted in Step F, substitute FEATURE-ID1 and FEATURE-ID2 with the feature id values from the last curl command and be sure to use your ACCESS_TOKEN and PROJECT_KEY:

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
-H ‘content-type: application/json’ \
-d ‘{“version”: 1, “actions”: [{“action”: “setCustomType”, “type”:
{“key”: “features-used”, “typeId”: “type”}, “fields”: {“features”: [{
“typeId”: “key-value-document”, “id”: “FEATURE-ID1”}, { “typeId”:
“key-value-document”, “id”: “FEATURE-ID2”}]}}]}’ \
https://API_HOST/PROJECT_KEY/carts/CART_ID
```

Cart responses are large so here is a small snippet showing our referenced custom objects:

```
{
  “type”: “Cart”,
  “id”: “CART_ID”,
  “version”: 2,
```

```
 [...]
  “custom”: {
    “type”: {
      “typeId”: “type”,
      “id”: “b671d023-df84-4190-a60f-97d032edc6d0”
    },
    “fields”: {
      “features”: [
        {
          “typeId”: “key-value-document”,
          “id”: “d6490f1a-dceb-4efa-8f29-46f18dc950b1”
        },
        {
          “typeId”: “key-value-document”,
          “id”: “0b04e62f-65a1-4daf-80e5-42aedccb4175”
        }
      ]
    }
  },
  [...]
}
```

If you end up experimenting with variations on the update, you may end up getting a `statusCode` 409 Conflict returned with a `message` similar to “Object 58ff9d4e-7aef-43d3-9613-c1a6cdd472b3 has a different version than expected.” If this happens, increment the version number and try again. You can get the current version of the resource by doing a GET:

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
https://API_HOST/PROJECT_KEY/carts/CART_ID
```

Step H: Test: Query to find all carts that used a specific feature.

Now that we have at least one cart with a custom field referencing our custom objects, let’s write some queries to see what we can find.

A basic GET on our CART_ID will only show us the `id` values of our associated custom objects. We can use Reference Expansion to get all the details. You can do this in IMPEX by setting **Endpoint** to “Carts”, set **Command** to “Query”, set **Where** to “id=CART_ID”, and set **Expand** to “custom.fields.features[*]”. Using curl, you must URL encode the expand parameter:

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
https://API_HOST/PROJECT_KEY/carts/CART_ID?expand=custom.
fields.features%5B%2A%5D
```

Here's what you should see in the expanded result with some content removed for brevity:

```
{
  “type”: “Cart”,
  “id”: “CART_ID”,
  “version”: 2,
  [...],
  “custom”: {
    “type”: {
      “typeId”: “type”,
      “id”: “b671d023-df84-4190-a60f-97d032edc6d0”
    },
    “fields”: {
      “features”: [
        {
          “typeId”: “key-value-document”,
          “id”: “d6490f1a-dceb-4efa-8f29-46f18dc950b1”,
          “obj”: {
            “id”: “d6490f1a-dceb-4efa-8f29-46f18dc950b1”,
            “version”: 1,
            [...],
            “container”: “features”,
            “key”: “header-keyword-search”,
            “value”: {
            “description”: “Customer used the keyword search feature
from the header”,
              “name”: “Keyword Search from Header”
            }
          }
        },
        {
          “typeId”: “key-value-document”,
          “id”: “0b04e62f-65a1-4daf-80e5-42aedccb4175”,
          “obj”: {
            “id”: “0b04e62f-65a1-4daf-80e5-42aedccb4175”,
            “version”: 1,
            [...],
            “container”: “features”,
            “key”: “quick-shop”,
            “value”: {
               “description”: “Customer used the quick shop feature
from products list”,
              “name”: “Quick Shop”
            }
          }
        }
      ]
    }
  },
  [...]
}
```

It's also possible to use our customizations in cart searches by leveraging Query Predicates. For our use case, we can find all carts that used the

"Keyword Search from Header" feature, which has an `id` of "d6490f1a-dceb-4efa-8f29-46f18dc950b1", by using this query predicate:

```
custom(fields(features(id="d6490f1a-dceb-4efa-8f29-
46f18dc950b1")))
```

This predicate can directly be leveraged in IMPEX by setting **Endpoint** to "Carts", **Command** to "Query" and **Where** to the predicate. Try it out!

Of course, we can also query via the HTTP API. Here is the command and note that the *query predicate must also be URL encoded.*

```
curl -sH "Authorization: Bearer ACCESS_TOKEN" \
"https://API_HOST/PROJECT_KEY/carts?expand=custom.fields.feat
ures%5B%2A%5D&where=custom%28fields%28features%28id%3D%22d649
0f1a-dceb-4efa-8f29-46f18dc950b1%22%29%29%29"
```

It's important to note that on a high volume ecommerce site, it's expensive to run a query like this. The commercetools platform does a great job of adding indexes dynamically but be aware it may take some time as this snippet from the docs makes clear:

> ***"If you add a custom field to your carts and start querying it, the commercetools platform will add an index to the project to improve performance if it meets criteria like query frequency. The automatic index creation needs to collect a significant amount of data to not optimize for outlier scenarios. That's why it can take up to two weeks before a new index is added."***
>
> – Query Predicates Performance Considerations, Platform Documentation

Step I: Install the Sunrise SPA Front End.

This feature we've built is not much use unless you can update your front end to take advantage of the new Custom Field and Custom Objects. The commercetools platform doesn't restrict you to an out of the box website so there is a lot of implementation variety and many Marketplace Integration solutions to choose from. For this example, we will leverage the open source Sunrise Single Page Application (SPA) front end. Here are the steps to get a local installation we can experiment with.

1. We will use Fearless Technology Group's fork of the commercetools Sunrise SPA since it includes branches containing the sample code we will review in subsequent steps. Install the application using these commands:

```
$ git clone https://github.com/FearlessTechnologyGroup/
sunrise-spa.git
$ cd sunrise-spa/
$ npm install
```

2. Create an API client we can use to configure the application. We can follow the same steps we used in Step B, number 2 but we will create credentials specifically for our front end application:

a. Login to the Merchant Center and then navigate to Settings -> Developer Settings from the left navigation.
b. Click "Create new API Client"
c. For field **Name**, enter: `spa-client`
d. For field **Scopes**, select: `Mobile & single-page application client`
e. Under the **View** subheading, click "`Products (all)`"
f. Click "Create API Client"
g. Note all the information provided as we will use them in the next step.

3. Configure the SPA application to use your commercetools project. Create a file called `.env.local` in the root of sunrise-spa. It should look like the following but with the parameters in all caps substituted with information from the previous step:

```
VUE_APP_CT_PROJECT_KEY=PROJECT_KEY
VUE_APP_CT_CLIENT_ID=CLIENT_ID
VUE_APP_CT_CLIENT_SECRET=SECRET
VUE_APP_CT_SCOPE=manage_my_orders:PROJECT_KEY view_
categories:PROJECT_KEY view_products:PROJECT_KEY manage_my_
shopping_lists:PROJECT_KEY create_anonymous_token:PROJECT_KEY
view_published_products:PROJECT_KEY manage_my_profile:PROJECT_
KEY manage_my_payments:PROJECT_KEY
VUE_APP_CT_AUTH_HOST=https://AUTH_HOST
VUE_APP_CT_API_HOST=https://API_HOST
```

4. Run the SPA locally:

```
$ npm run serve
```

Open a browser to http://localhost:8080/ to check out the site.

Step J: Extend the front end to use our Custom Objects.

Let's make edits to the Sunrise SPA to take advantage of our customizations. To pick up all the changes we will make in this step, stop the application you started in Step I (likely control + c), check out branch "custom-objects" and start the app again:

```
$ git checkout custom-objects
$ npm run serve
```

In Step D, we created two custom objects for two features with `key`s "header-keyword-search" and "quick-shop". In the SPA, those two features are expressed in two JavaScript files: TheHeader.js and ProductQuickView.js. In each, we add a mixin named `featuresMixin` that exposes an `updateFeaturesUsed` method we can call to add a reference to our custom object by passing in the feature's `key` as highlighted below.

1. TheHeader.js

```
[...]
  import featuresMixin from '../../../mixins/featuresMixin';
  [...]
  mixins: [cartMixin, featuresMixin],
  computed: {
    totalCartItems() {
      return this.$store.state.cartItems;
    },
  },
  methods: {
    toggleSearch() {
      this.searchOpen = !this.searchOpen;
      if (this.searchOpen) {
        this.updateFeaturesUsed('header-keyword-search');
      }
    },
  [...]
```

2. ProductQuickView.js

```
 [...]
  import featuresMixin from '../../../mixins/featuresMixin';
  [...]
  mixins: [productMixin, cartMixin, featuresMixin],
  [...]
  watch: {
    showModal() {
      if (this.showModal === true) {
        this.$modal.show('quickView');
        this.updateFeaturesUsed('quick-shop');
      }
    },
  },
  [...]
```

The `featuresMixin` does the work of querying for our custom objects and referencing them in our cart resource as features are used. You can find the full implementation at featuresMixin.js but let's highlight a few things.
First, we get our custom objects using the following query which leverages vue-apollo. Note that we are constraining the results to custom objects from our 'features' container.

```
customObjects: {
  query: gql`
    query CustomObject($container: String!) {
      customObjects(container: $container) {
        results {
          container
          key
          value
          id
        }
      }
    }`,
  variables() {
    return {
      container: 'features',
    };
  },
},
```

Second, we provide computed values to consumers of the mixin. For example, we use `featuresUsed` to provide a list of features used in the mini cart (silly, I know, but it ties all this together.)

```
computed: {
  features() {
    return this.customObjects?.results;
  },

  featuresUsedIds() {
    if (this.cartExists) {
      const { customFieldsRaw = [] } = this.me?.activeCart ||
{};
      if (Array.isArray(customFieldsRaw)) {
        const features = customFieldsRaw
          .find((cf) => cf.name === 'features');
        const { value: featuresUsed = [] } = features || {};
         return featuresUsed.reduce((acc, v) => acc.concat(v.
id), []);
      }
    }
    return [];
  },
```

```
 featuresUsed() {
    if (this.featuresUsedIds && this.features) {
      return this.features.reduce((acc, feature) => {
        const { id, value } = feature || {};
        if (this.featuresUsedIds.includes(id)) {
          const { name, description } = value || {};
          return acc.concat({ id, name, description });
        }
        return acc;
      }, []);
    }
    return [];
  },
},
```

We have a private method, `getFeaturesUsedUpdate`, that we use to check if a given feature is valid (i.e., we have an entry for it in our list of custom objects) and that we've not already applied it to the current cart. If it's valid and unapplied, we need to have the update's `value` field formatted using escaped JSON. If you want to drill down on the GraphQL Schema driving this formatting, see lines 6749 to 6787 of sunrise-spa/graphql.schema.json.

```
const getFeaturesUsedUpdate = (features, featuresUsedIds,
featureToAdd) => {
  // find featureToAdd in our list of features
  if (Array.isArray(features)) {
    const featureId = features.reduce((acc, feature) => {
      const { id, key } = feature || {};
      return key === featureToAdd ? id : acc;
    }, null);

    // report this feature as being used if we've not done so
yet
    if (!featuresUsedIds.includes(featureId)) {
      const featuresUsed = featuresUsedIds.concat(featureId);
    const featuresFormatted = featuresUsed.map((featureUsedId)
=>
          (`{ \"typeId\": \"key-value-document\", \"id\":
\"${featureUsedId}\"}`)); //eslint-disable-line
      return `[${featuresFormatted.join(',')}]`;
    }
  }
  return null; // no need to update the cart
};
```

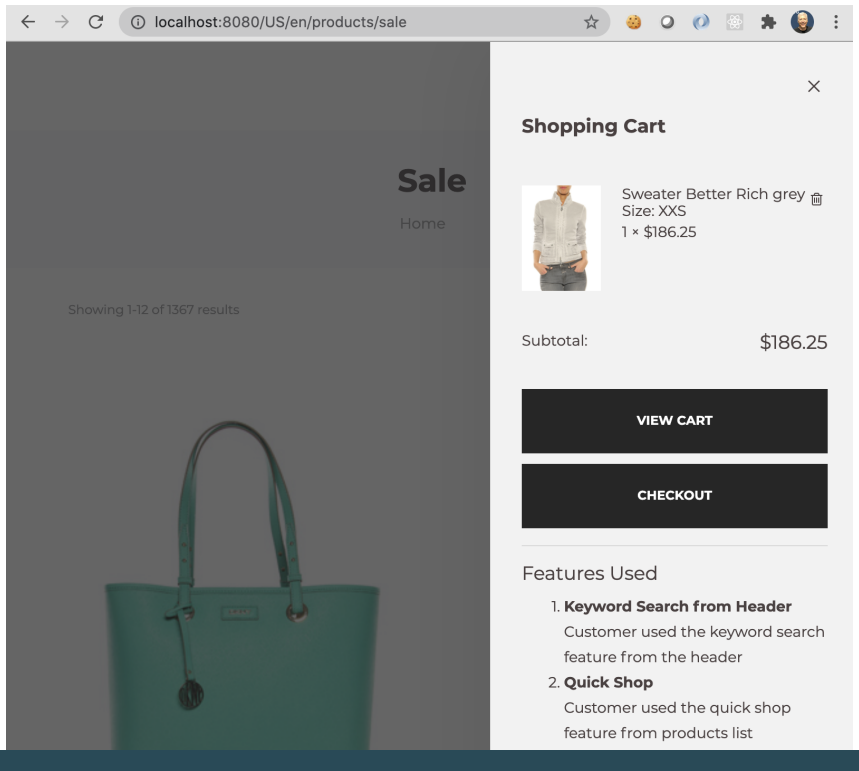
Finally, we expose a method consumers can call to update the cart with a feature that's been used.

```
async updateFeaturesUsed(featureToAdd) {
  // check to see if we've already set this feature and
  // if not, get a string we can use for the update
  const featuresUsed = getFeaturesUsedUpdate(
    this.features,
    this.featuresUsedIds,
    featureToAdd,
  );

  if (this.cartExists && featuresUsed) {
    return this.updateMyCart({
      setCustomType: {
        type: { key: 'features-used', typeId: 'type' },
        fields: {
          name: 'features',
          value: featuresUsed,
        },
      },
    });
  }

  // if we don't have a cart yet or the feature is already set,
  // just resolve the promise
  return Promise.resolve();
},
```

To try it out, open a browser to http://localhost:8080/ and add something to your cart. Then, click the search icon in the header and open the quick shop from one of the product thumbnails. If you monitor XHR requests in the browser's dev tools while you perform these actions, you will see graphql requests that use the setCustomType action. Open the mini cart to see a (silly) sample message:



The MiniCart.vue uses the mixin's computed value `featuresUsed` to render the messages:

```
<div v-if="featuresUsed.length > 0">
  <hr />
  <h4>Features Used</h4>
  <ol>
    <li
      v-for="feature in featuresUsed"
      :key="feature.id"
    >
      <div>
        <span><strong>{{ feature.name }}</strong></span>
        <br />
        <span>{{ feature.description }}</span>
      </div>
    </li>
  </ol>
</div>
```

Whew! We hope this extensive round trip was useful to you.

Step K: Provide you with resources if you need help.

We travelled quite a bit of ground covering Custom Objects showing you the power they provide. If you have questions or need additional help, Fearless Technology Group (FTG) is available to assist you. Shoot us an email at contactus@fearlesstg.com so we can lend a hand.

Additional Help

As you consider all your customization options, FTG and commercetools are here to assist you.

About Fearless Technology Group

Fearless Technology Group (FTG) helps retailers modernize their technology architecture, solve critical business problems, and capitalize on business opportunities in an evolving landscape. FTG is both a commercetools and Google Cloud Platform Partner. Contact us at contactus@fearlesstg.com or 720-432-9068.

About commercetools

commercetools is a next-generation software technology company that offers a true cloud commerce platform, providing the building blocks for the new digital commerce age. Our leading-edge API approach helps retailers create brand value by empowering commerce teams to design unique and engaging digital commerce experiences everywhere – today and in the future. Our agile, componentized architecture improves profitability by significantly reducing development time and resources required to migrate to modern commerce technology and meet new customer demands. It is the perfect starting point for customized microservices.

Europe - HQ
commercetools GmbH
Adams-Lehmann-Str. 44
80797 Munich, Germany
Tel. +49 (89) 99 82 996-0
info@commercetools.com

Americas
commercetools, Inc.
324 Blackwell, Suite 120
Durham, NC 27701
Tel. +1 212-220-3809
mail@commercetools.com

www.commercetools.com
Munich - Berlin - Jena - Amsterdam - London - Durham NC - Singapore - Melbourne