
Drupal Commerce Documentation

Release 0.0.1

Isaac Horton

February 01, 2017

1	Introduction	1
1.1	Contribute to Documentation	1
2	Commerce 1.x Documentation	3
3	Commerce 2.x	5
3.1	Overview	5
3.1.1	Drupal modules	5
3.1.2	PHP libraries	5
3.1.3	Recommended Tools	5
3.2	Getting Started	6
3.2.1	Getting Started with Drupal Commerce 2	6
3.3	Libraries and dependencies	10
3.3.1	Libraries and dependencies	10
3.4	Setting up stores	20
3.4.1	Setting up stores	20
3.5	Managing products	25
3.5.1	Products	25
3.6	Catalog and product pages	38
3.6.1	Catalog and product pages	38
3.7	Product merchandising	38
3.7.1	Product merchandising	38
3.8	Working with orders	39
3.8.1	Orders	39
3.9	Configuring Checkout	42
3.9.1	Configuring your checkout	42
3.10	Payments	42
3.10.1	Setting up payments	42
3.11	Code Recipes	42
3.11.1	Code Recipes	42

Introduction

Drupal Commerce is the leading flexible eCommerce solution for Drupal, powering over 50,000 online stores of all sizes.

This documentation is rendered online at <http://drupal-commerce.readthedocs.io/en/latest/>

1.1 Contribute to Documentation

We love contributors! Please help us improve or fix the documentation by editing a document and making a pull request in Github. Our documentation is written in Restructured Text(.rst) so please only edit a document if you are familiar with .rst file formatting guidelines.

Our docs are written in Restructured Text, built locally with Sphinx, managed on Github and hosted with [ReadtheDocs](#).

If you have trouble understanding any part of the documentation, please notify those of us who work on this section by creating an issue in our [documentation repository](#) and clearly explain what you don't understand and why - we're happy to hear from you, your contribution helps everyone!

You can also contribute directly on our [documentation repository](#) by editing the files through the [GitHub](#) interface directly in your browser. Alternatively, you can clone the repository and edit the book in your favorite text editor.

Hosting

This site is hosted on [Platform.sh](#).

Commerce 1.x Documentation

Head on over to [DrupalCommerce.org](https://drupalcommerce.org). On the site you will find a nice overview of the original 1.x build and links to all the relevant user guides. Linked below for your convenience as well.

- [User Guide](#) - A site builder guide with lots of screenshots that covers installation, order management, product creation, and a myriad of other topics.
- [Developer Guide](#) - A guide built for developers that outlines architecture, building payment gateways, code workflow, and utilizing core APIs.
- [Commerce Kickstart 2](#) - A guide built for site builders that are demo'ing with Kickstart 2.x
- [Commerce Cookbook](#) - A cookbook of common site builder tasks that covers things like shipping, inventory, reporting, merchandising, etc.
- [API Documentation](#) - The doxygen output of all the code documentation that ships with all Commerce installations.

Commerce 2.x

At its core, Commerce is a set of Drupal 8 modules, which in turn depend on other best-of-breed modules and libraries.

3.1 Overview

At its core, Commerce is a set of Drupal 8 modules, which in turn depend on other best-of-breed modules and libraries.

3.1.1 Drupal modules

The following Drupal contrib modules are used:

- [Address](#) - Provides functionality for storing, validating and displaying international postal addresses.
- [Entity](#) - Extends Drupal 8's entity API with additional features.
- [State Machine](#) - Provides code-driven workflow functionality.
- [Inline Entity Form](#) - Provides a widget for inline management of referenced entities.
- [Profile](#) - Provides configurable user profiles, used for customer profiles.

3.1.2 PHP libraries

The following PHP libraries are used:

- [commerceguys/intl](#) - An internationalization library powered by CLDR data. Handles currencies, currency formatting, and more.
- [commerceguys/addressing](#) - An addressing library, powered by Google's dataset. Stores and manipulates postal addresses.
- [commerceguys/zone](#) - A zone library. Zones are territorial groupings mostly used for shipping or tax purposes.
- [commerceguys/tax](#) - A tax library with a flexible data model, predefined tax rates, powerful resolving logic.

3.1.3 Recommended Tools

The [Drupal Console](#) command-line tool.

3.2 Getting Started

3.2.1 Getting Started with Drupal Commerce 2

Drupal Commerce requires using Composer with Drupal. If you are new to Composer, or new to managing Drupal with Composer, see [Composer: the what, why, and how](#).

To get Drupal Commerce installed, see the [Installing Drupal Commerce](#) guide.

For keeping Drupal Commerce up to date, review the [Keeping a Drupal Commerce site up to date](#) guide.

Composer: the what, why, and how

@todo

- Port contents of <https://glamanate.com/blog/managing-your-drupal-project-composer>
- And <https://docs.google.com/presentation/d/1PK9q2dBkGHfyEO76bEVpqS61wTgA0LGbru2PECiwUnk/edit?usp=sharing>

Installing

Installing Commerce to contribute back? Check out our installation instructions for contributors.

Requirements

Commerce 2.x requires Drupal 8.2.0 or newer.

If you already have a web server, make sure it satisfies [Drupal 8's requirements](#).

The recommended memory limit is 256MB or more.

For local development we recommend [Drupal VM](#) (advanced users) or [Acquia Dev Desktop](#) (beginners).

You will also need [Composer](#).

Why must we use Composer?

New site

The following command will download Drupal 8 + Commerce 2.x with all dependencies to the `mystore` folder:

```
composer create-project drupalcommerce/project-base mystore --stability dev
```

Install it just like a regular Drupal site. Commerce will be automatically enabled for you.

Tips:

- The `bin` folder contains [Drupal Console](#).
- The `web` folder represents the document root.
- Composer commands are always run from the site root (`mystore` in this case).
- Downloading additional modules: `composer require "drupal/devel:1.x-dev"`

- Updating an existing module: `composer update drupal/address --with-dependencies`

See the [project-base README](#) for more details.

Existing site

Run these commands in the root of your website:

1. Add the Drupal Packagist repository

```
composer config repositories.drupal composer https://packages.drupal.org/8
```

This allows Composer to find Commerce and the other Drupal modules.

1. Download Commerce

```
composer require "drupal/commerce 2.x-dev"
```

This will also download the required libraries and modules (Address, Entity, State Machine, Inline Entity Form, Profile).

1. Enable Commerce (instructions below use [Drupal Console](#))

```
drupal module:install commerce_product commerce_checkout commerce_cart  
commerce_tax
```

Keeping a Drupal Commerce site up to date

Note: Drupal Commerce 2 has now hit beta which supports upgrades. If you have an alpha installation, you will need to implement an upgrade path manually.

To update to the newest version of Drupal Commerce, you will need to update with Composer.

```
composer update drupal/commerce --with-dependencies
```

Please note the `--with-dependencies` option. Without this option specified any needed, contributed projects and libraries will not update. Only the Drupal Commerce module will be updated.

Run your Drupal updates once all of the dependencies are updated. We recommend running them on the command line rather than the `update.php` script. See the example below.

```
drupal update:debug  
drupal update:execute
```

Getting ready for development

Preparing the local environment

Start by setting up a web server, PHP and MySQL.

We recommend [Drupal VM](#) for advanced users, [Acquia Dev Desktop](#) for beginners.

You will also need [Git](#) and [Composer](#).

Note that Drupal VM comes with Composer included.

Getting Commerce

The following command will download Drupal 8 + Commerce 2.x with all dependencies to the `mystore` folder:

```
composer create-project drupalcommerce/project-base mystore --prefer-source --stability dev
```

The `--prefer-source` option tells Composer to use Git clone as the download method.

When prompted, answer `n` to:

```
Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]?
```

This will keep the downloaded git repositories inside their parent folders (such as `web/modules/contrib/commerce`).

Tips:

- The `bin` folder contains [Drupal Console](#) and [PHPUnit](#).
- The `web` folder represents the document root.
- Composer commands are always run from the site root (`mystore` in this case).
- Downloading additional modules: `composer require "drupal/devel:1.x-dev"`
- Updating an existing module: `composer update drupal/address --with-dependencies`

See the [project-base README](#) for more details.

Preparing your fork

Note: You will need a GitHub account for contributing.

Visit the Commerce [repository on GitHub](#) and click the **fork** button.
That will create a copy of the repository on your GitHub account.

Now go to the Commerce folder and add your fork:

```
cd web/modules/contrib/commerce
git remote add fork git@github.com:YOUR_USER/commerce.git
```

Replace `YOUR_USER` with your username (the full url is shown on your fork's GitHub page).

You will now be able to push new branches to your fork and create [pull requests](#) against the main repository.

Running tests

All of the Drupal Commerce tests are based on the PHPUnit framework. In order to run the tests you will need to copy the `phpunit.xml.dist` from the core directory and modify it for your environment. An in depth article on getting ready to run the tests can be found here: <https://drupalcommerce.org/blog/45322/commerce-2x-unit-kernel-and-functional-tests-oh-my>

```
cd mystore/web
# Run PHPUnit tests
../bin/phpunit -c core/phpunit.xml modules/contrib/commerce
```

Developing

Choosing an issue

Commerce uses GitHub for code and drupal.org for tracking issues.

To choose an issue, go through [the open issues](#), pick one you like and **assign it to you**.

If you need help choosing an issue or working on one, join the Commerce 2.x office hours.

They are held every wednesday at 3PM GMT+1 on the [#drupal-commerce IRC channel](#).

****Tip:**** You can also view the issue queue as a [kanban board](#).

Creating a pull request

Start by creating a branch for your work.

The branch name should contain a brief summary of its id and the issue, e.g **2276369-fix-product-form-notice**:

```
cd web/modules/contrib/commerce
git checkout -b 2276369-fix-product-form-notice
```

Once you're done with development, push your commits to your fork:

```
git commit -a -m "Issue 2276369: Fix notice in the product form."
git push fork 2276369-fix-product-form-notice
```

You can now go to your fork's GitHub page and [create a pull request](#).

Your pull request should link to the drupal.org issue, and vice-versa.

After your code has been reviewed, you might be asked to perform some changes and then have them reviewed again:

```
# Change the desired files.
git commit -a -m "Addressed feedback."
git push fork 2276369-fix-product-form-notice
```

Updating the branch will automatically update the related pull request.

Keeping up to date

Your forked repository and the original one (called *origin*) will eventually get out of sync.

Periodically update your fork by doing:

```
# Update your local branch.
git checkout 8.x-2.x
git pull origin/8.x-2.x
# Push the update to your GitHub fork.
git push fork 8.x-2.x
```

Your pull request might also need rebasing, to re-apply your changes on top of the latest code. Once you've updated the master branch (8.x-2.x), rebase your branch:

```
git checkout 2276369-fix-product-form-notice
git rebase 8.x-2.x
git push -f fork 2276369-fix-product-form-notice
```

That's it! Happy contributing!

3.3 Libraries and dependencies

3.3.1 Libraries and dependencies

Drupal Commerce is built from many different components. Understanding these building blocks and their functionality will aid you in building your Drupal Commerce store.

- Address
- Profile
- State Machine
- Inline Entity Form
- Currency

Address Module

See Also: [Address Drupal Module](#) | [Addressing library](#) | [Address Commerce 2.x Story](#)

For the addressing needs of Commerce 1.x the [addressfield module](#) was created. It stores addresses using the xNAL standard, accommodates both name and address data, and provides per-country address forms.

It was a good first try, but we can do better.

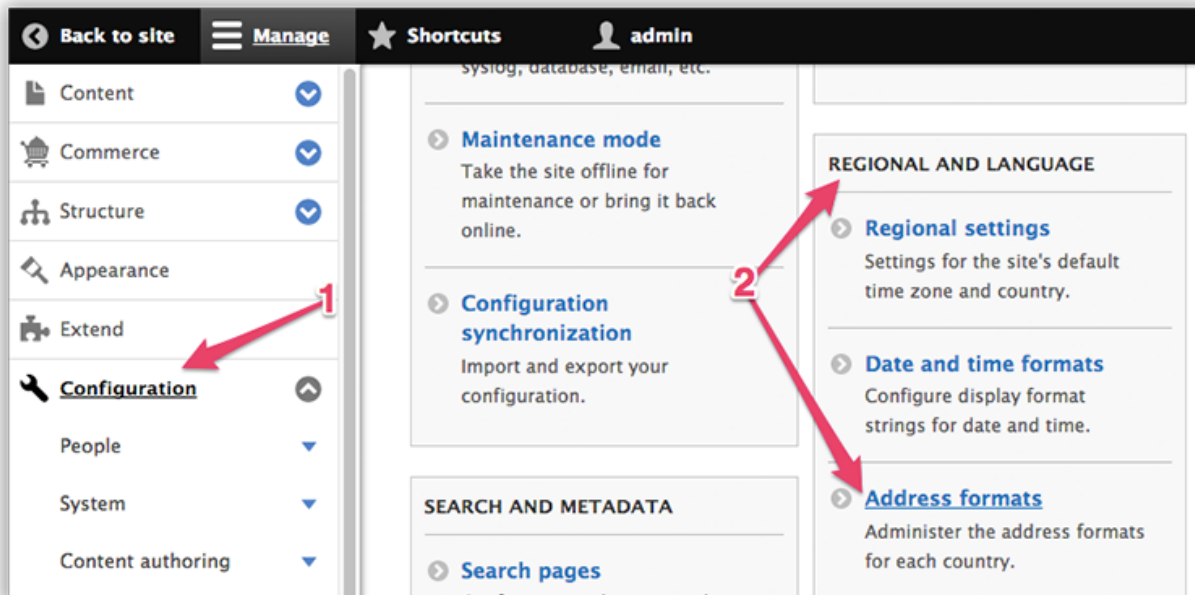
Commerce 2.x will depend on the [Address 1.x module](#), which will pull in the [commerceguys/addressing](#) library, store the address formats and subdivisions as configuration entities, and use them to generate and validate Drupal forms.

We gain a much richer dataset and greatly improved support for countries such as China, Korea, Brazil, and others. Best of all, our efforts benefit the whole wider PHP community.

Install

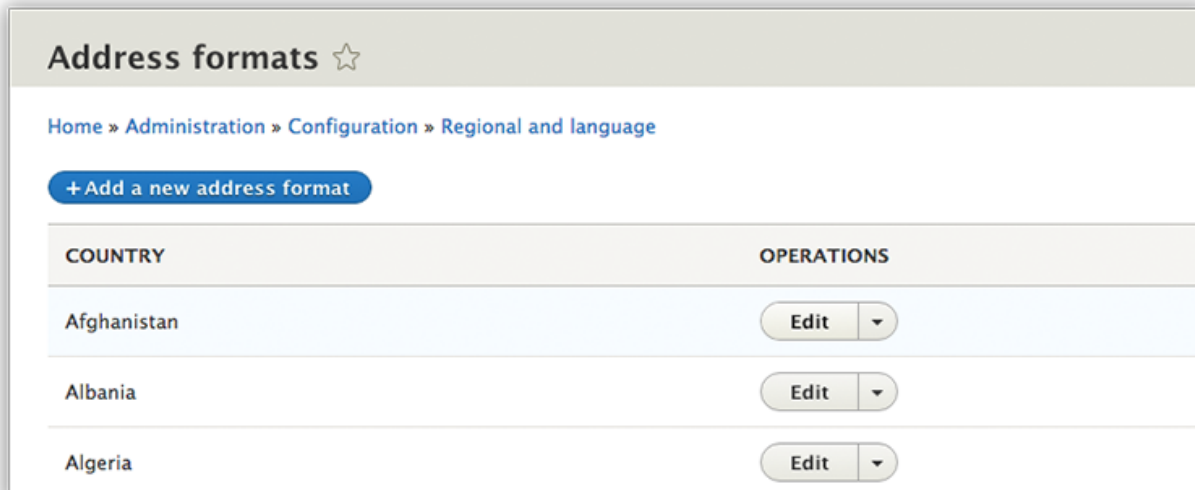
This is a dependency and once you have successfully installed commerce, you will have the address module available. See [Installation Instructions for Commerce 2.x](#).

Configure and Customize



To configure or customize address formats, navigate to the Configuration page (1) and click on (2) “Address Formats” under “Regional and Language”.

`admin/config/regional/address-formats`



The landing page for the address module shows all the default configurations by country. You can edit the postal formatting (order of fields, locality dependencies, and many many other things) just by clicking “Edit.”

Edit an address format ☆

Home » Administration » Configuration » Regional and language » Address formats

Country *
Brazil

Format *
%organization
%recipient
%addressLine1
%addressLine2
%dependentLocality

Available tokens: %administrativeArea, %locality, %dependentLocality, %postalCode, %sortingCode, %addressLine1, %addressLine2, %organization, %recipient

Required fields

- ☒ Administrative area
- ☒ Locality
- ☐ Dependent locality
- ☒ Postal code
- ☐ Sorting code
- ☒ Address line 1
- ☐ Address line 2
- ☐ Organization
- ☒ Recipient

Uppercase fields

- ☒ Administrative area
- ☒ Locality
- ☐ Dependent locality
- ☐ Postal code
- ☐ Sorting code
- ☐ Address line 1
- ☐ Address line 2
- ☐ Organization
- ☐ Recipient

Uppercased on envelopes to facilitate automatic post handling.

Postal code pattern
\d{5}[\-]?[d]{3}

Regular expression used to validate postal codes.

Postal code prefix

Added to postal codes when formatting an address for international mailing.

Postal code type
Postal code

The default values are based on an opensource 3rd party that has the best coverage of all regions in the world. The formatting of the addresses is for both the form and the display.

More information on Address formats

▼ ADDRESS

Country *
Brazil ▼

Street address *

Neighborhood

City *
 ✓ - Select -
 Abadia de Goiás
 Abadiânia
 Acreúna
 Adelândia

State *
 Goiás ▼

Each country has a different address format that tells us:

- Which fields are used in which order (Is there a state field? Does the zip code come before the city? After the state?)
- Which fields are required
- Which fields need to be uppercased for the actual mailing to facilitate automated sorting of mail
- The labels for the administrative area (state, province, parish, etc.), and the postal code (Postal code or ZIP code)
- Validation rules for postal codes, usually in the form of a regular expression.

In countries using a non-latin script (such as China, Taiwan, Korea), the order of fields varies based on the language/script used. Addresses written in latin script follow the minor-to-major order (start with the street, end with the country) while addresses written in the chinese scr

Zones

See Also: [Zone Library](#) | [Addressing Library](#) | [Address Drupal Module](#)

Overview

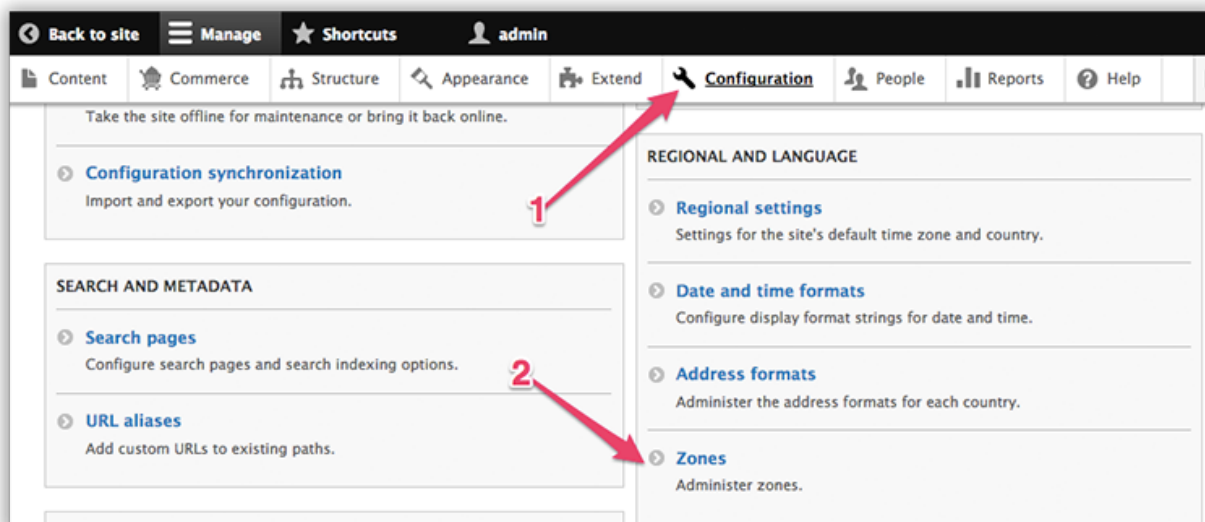
Zones are territorial groupings mostly used for shipping or tax purposes. For example, a set of shipping rates associated with a zone where the rates become available only if the customer's address matches the zone.

A zone can match other zones, countries, subdivisions (states/provinces/municipalities), postal codes. Postal codes can also be expressed using ranges or regular expressions.

Examples of zones:

- California and Nevada
- Belgium, Netherlands, Luxemburg
- European Union
- Germany and a set of Austrian postal codes (6691, 6991, 6992, 6993)
- Austria without specific postal codes (6691, 6991, 6992, 6993)

To locate Zones in your Commerce install, (1) click on Configuration and (2) click on Zones:



Each zone consists of zone members, which represent conditions that can be matched.

For example, a “France and Germany” zone would have two zone members: 1) France 2) Germany and an address would match that zone if it matches one of those two zone members.

Taxes and Zones

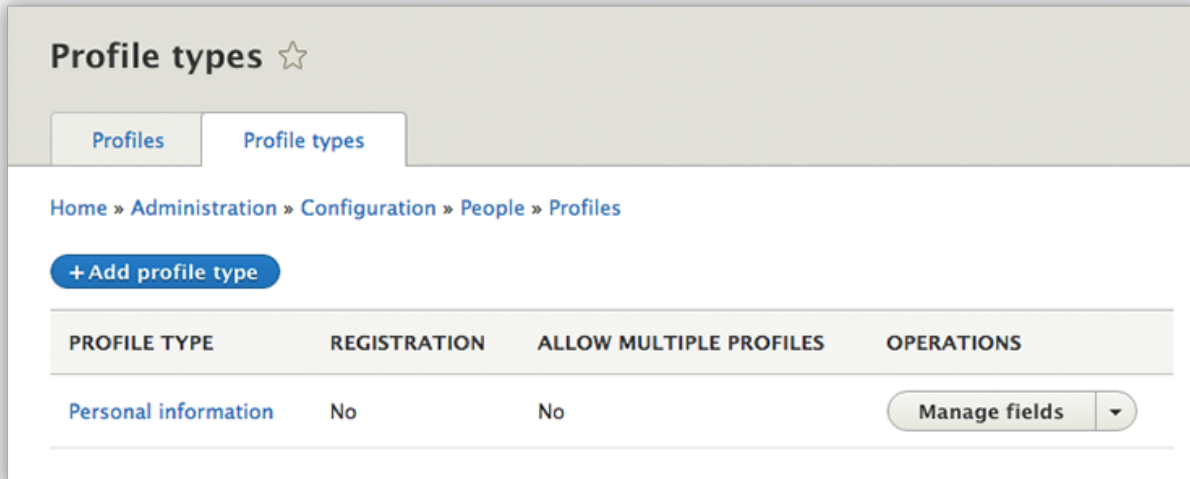
The Commerce Tax submodule creates a matching zone for each imported tax type.

For example, importing the German VAT tax type also creates a German VAT zone which contains two zone members: 1) Germany 2) Austria (postal codes 6691, 6991, 6992, 6993)

Profile

See Also: [Module on Drupal.org](#) | [Drupal 8 Issue](#)

Provides the profile entity type used to collect customer information. In Commerce 1.x, we called these entities “Customer Profiles” and for Commerce 2.x we have moved to where the community has extended user profiles to include fieldable entity bundles. Customer profiles in Commerce 2.x will be entities and orders will link to revisions, avoiding the duplication we had in Commerce 1.x.



The Profile module provides a fieldable entity, that allows administrators to define different sets of fields for user profiles, which are then displayed in the My Account section. This permits users of a site to share more information about themselves, and can help community-based sites organize users around specific information.

You can pull the latest from the repository on [Drupal.org](https://www.drupal.org).

State Machine

See Also: [module on drupal.org](https://www.drupal.org)

Provides code-driven workflow functionality.

A workflow is a set of states and transitions that an entity goes through during its lifecycle.

A transition represents a one-way link between two states and has its own label.

The current state of a workflow is stored in a state field, which provides an API for getting and applying transitions. An entity can have multiple workflows, each in its own state field.

An order might have checkout and payment workflows. A node might have legal and marketing workflows.

Workflow groups are used to group workflows used for the same purpose (e.g. payment workflows).

Architecture

[Workflow](#) and [WorkflowGroup](#) are plugins defined in YAML, similar to menu links.

Example: `commerce_order.workflow_groups.yml`:

```
order:
  label: Order
  entity_type: commerce_order
```

Groups can also override the default workflow class, for more advanced use cases.

Example: `commerce_order.workflows.yml`:

```
order_default_validation:
  id: order_default_validation
  group: order
```

```
label: 'Default, with validation'
states:
  draft:
    label: Draft
  validation:
    label: Validation
  completed:
    label: Completed
  canceled:
    label: Canceled
transitions:
  place:
    label: 'Place order'
    from: [draft]
    to: validation
  validate:
    label: 'Validate order'
    from: [validation]
    to: completed
  cancel:
    label: 'Cancel order'
    from: [draft, validation]
    to: canceled
```

Transitions can be further restricted by [guards](#), which are implemented as tagged services:

```
mymodule.fulfillment_guard:
  class: Drupal\mymodule\Guard\FulfillmentGuard
  tags:
    - { name: state_machine.guard, group: order }
```

The group argument allows the guard factory to only instantiate the guards relevant to a specific workflow group.

The current state is stored in a [StateItem](#) field.

A field setting specifies the used workflow, or a value callback that allows the workflow to be resolved at runtime (checkout workflow based on the used plugin, etc.).

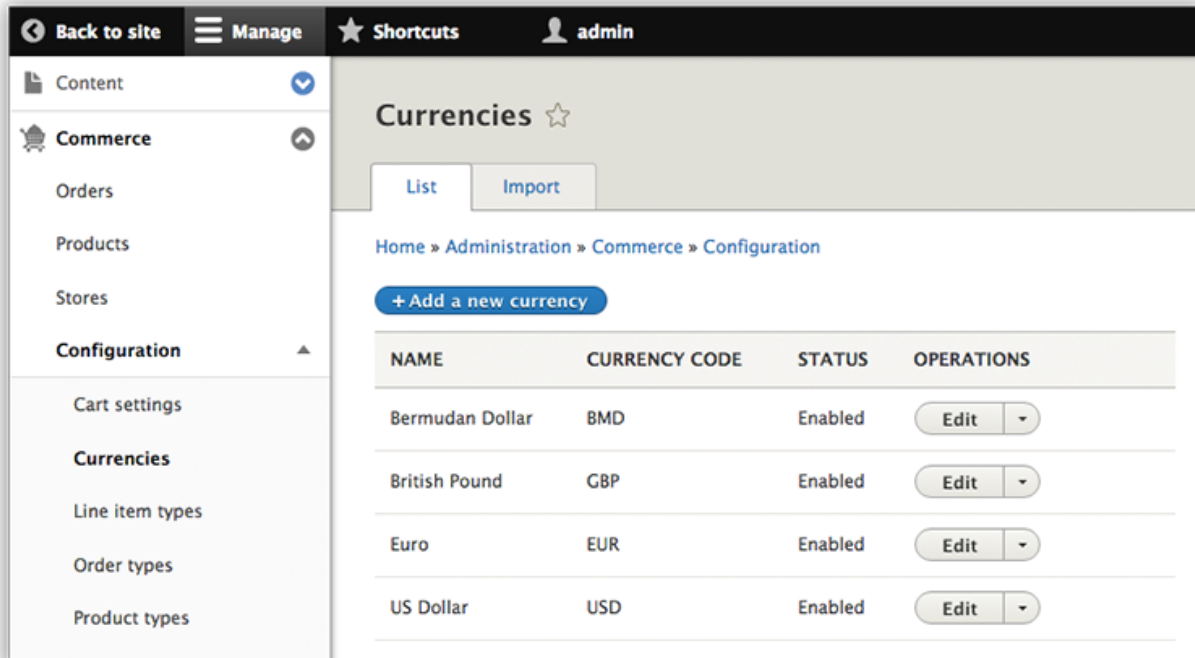
A validator is provided that ensures that the specified state is valid (exists in the workflow and is in the allowed transitions).

Inline Entity Form 8.x-2.x

Provides a widget for inline management (creation, modification, removal) of referenced entities. Commerce uses it extensively for product variations, line items, and (soon) tax rate amounts.

Currency

See Also: [Internationalization Commerce Story](#) | [Internationalization Library](#)



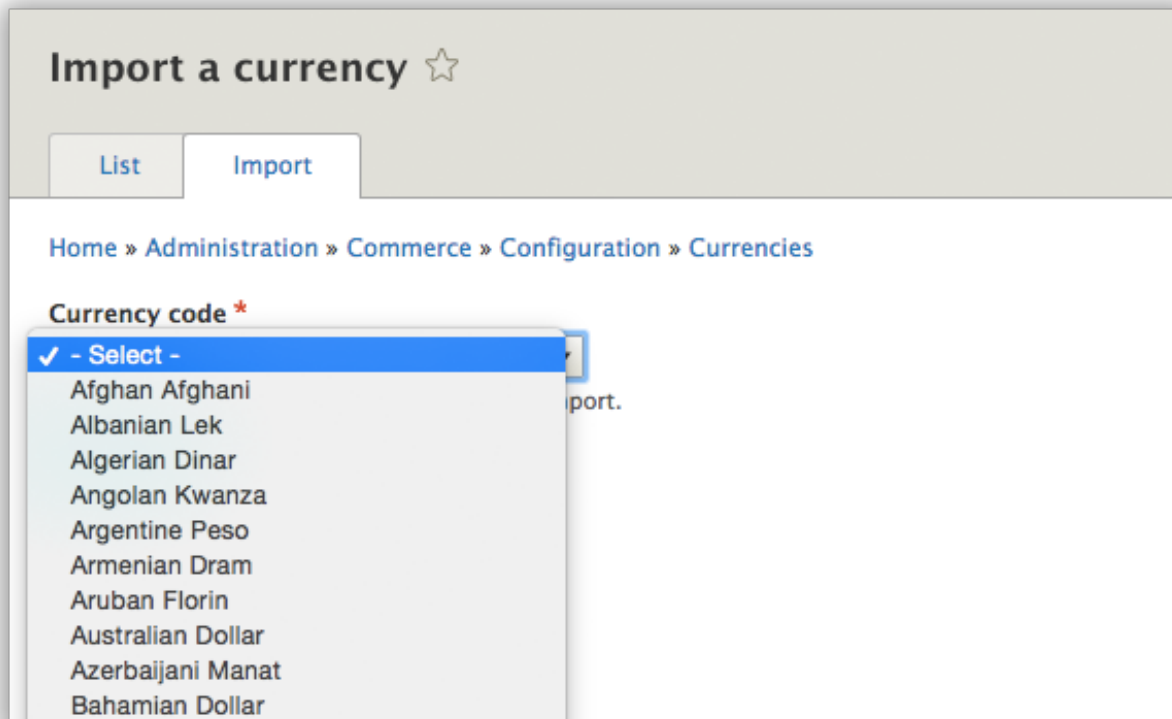
Overview

Commerce without borders means we support every language and every denomination of currency. This is a big undertaking because not only do we need to support various currencies, we need to support their regional formatting rules, what each currency is called in every other language, and many other difficult problems.

Commerce 2's currency support is built upon the [commerceguys/intl](#) library which provides a list of currencies, currency formatting, countries, and languages. This list is not something we cooked up on the back of a napkin, the intl library uses the internationally-recognized standard of [CLDR](#) data. We parse the CLDR definitions into our own more compact YAML definitions and use them to re-implement intl's NumberFormatter and provide currency, country, language data.

Importing Defined Currencies

If you navigate to `admin/commerce/config/currency` and click on the "Import" tab, you will see a simple dropdown that shows you all the supported currencies (157 active currencies).



When imported, a configuration entity called “commerce_currency” is created with all the relevant data from the CLDR definition. Once imported, the configuration entity is unique to your installation, which means you can make minor changes to formatting and not worry about an update reverting your changes.

Also, thanks to the CLDR dataset, we import all the translations of the currency you are importing for all the languages you have in your site. A small, but very practical and helpful time saver.

Creating and Editing Currencies

Edit *British Pound* ☆

[Edit](#)
[Devel](#)

[Home](#) » [Administration](#) » [Commerce](#) » [Configuration](#) » [Currencies](#)

Name *

Currency code *

Numeric code *

Symbol *

Fraction digits *

The number of digits after the decimal sign.

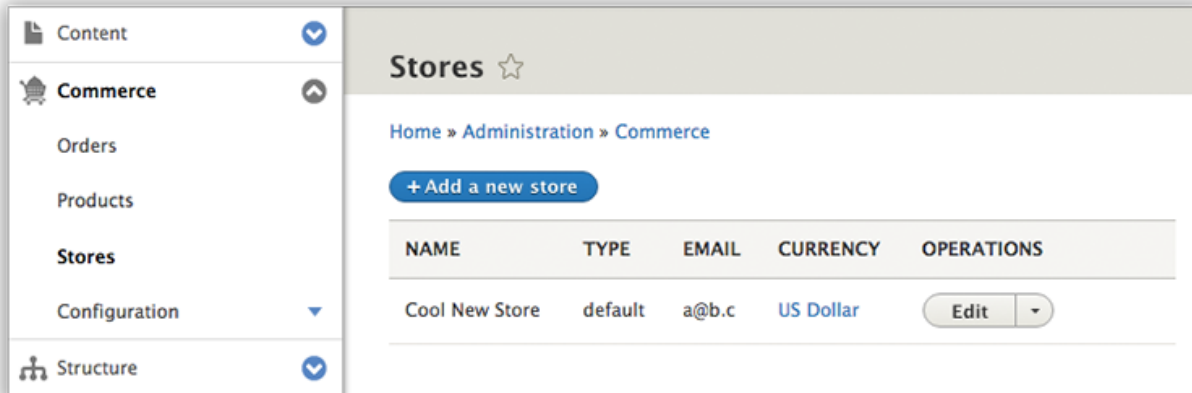
☒ Enabled

[Save](#)
[Delete](#)

Once imported (or if you click “+ Add a new currency”) you can change the name, the numeric code, the symbol and how many minor units we use in calculations and display

3.4 Setting up stores

3.4.1 Setting up stores



For Commerce 2, we have native support for stores. Stores are used for invoicing, tax types, and any other settings necessary for understanding orders. This has many applications and its important to understand what use cases are supported out of the box and how that impacts checkout and other order workflows.

Create a store

To create a store you will need to have at least one currency imported, and then you can create a store.

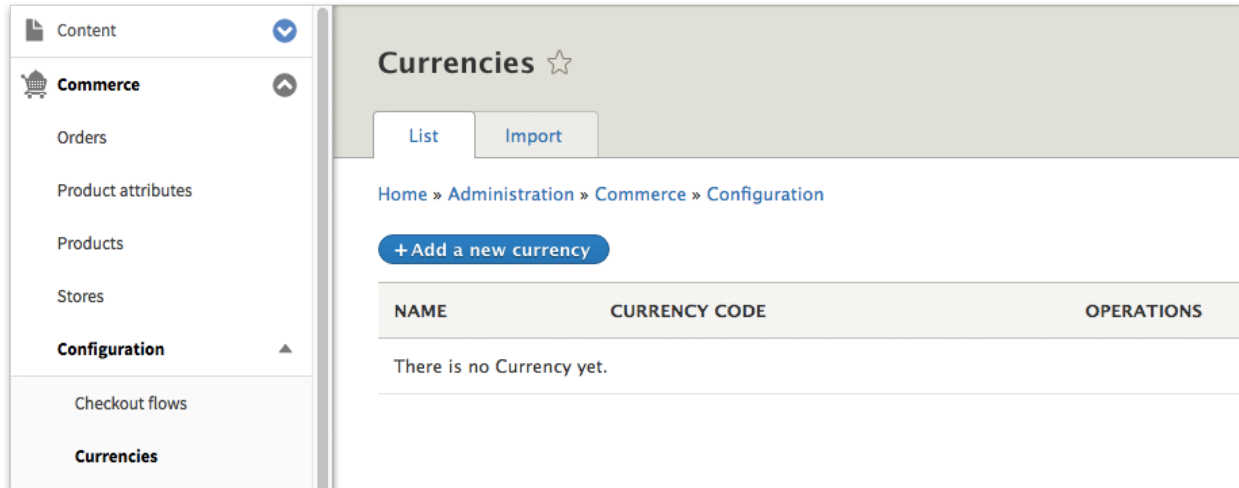
Shortcut! - The getting started process can be quickly done using Drupal Console command:

```
drupal commerce:create:store
```

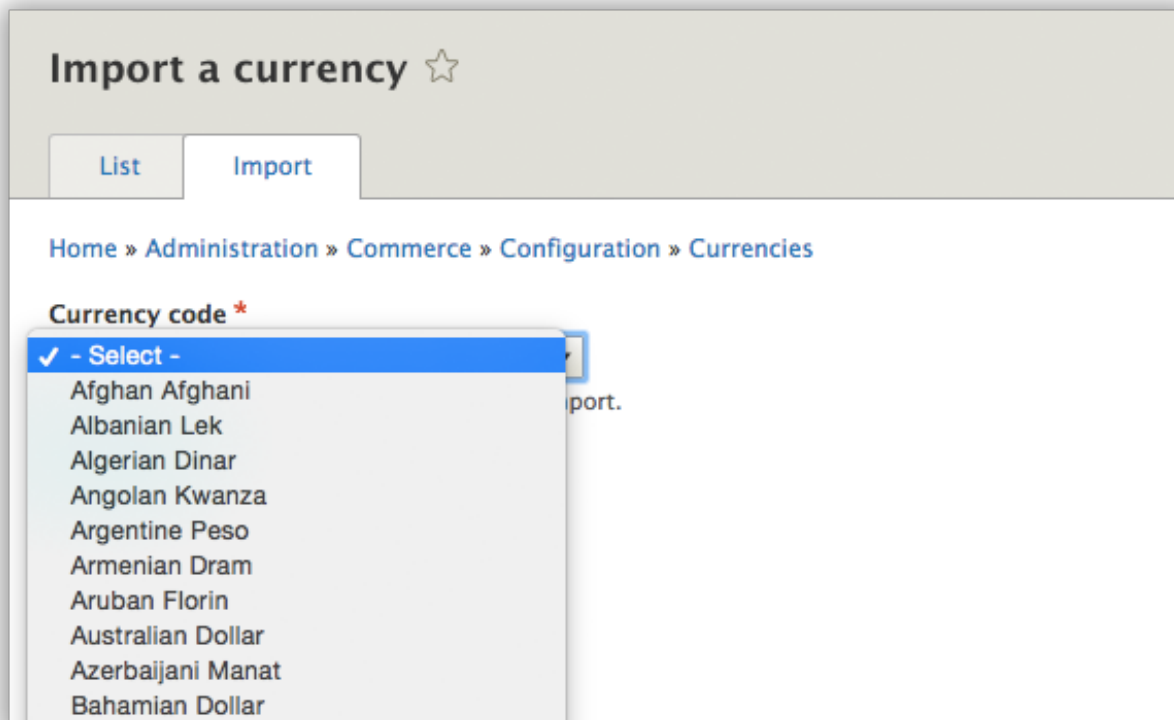
You are welcome to ignore this shortcut if you prefer the user interface.

Import the currencies your store will use.

The most basic piece of information that defines your store is the currency(s) you want to use. The vast majority of Commerce stores will simply have one currency and one store. To set this up, first you need to locate the currencies page at `admin/commerce/config/currencies`

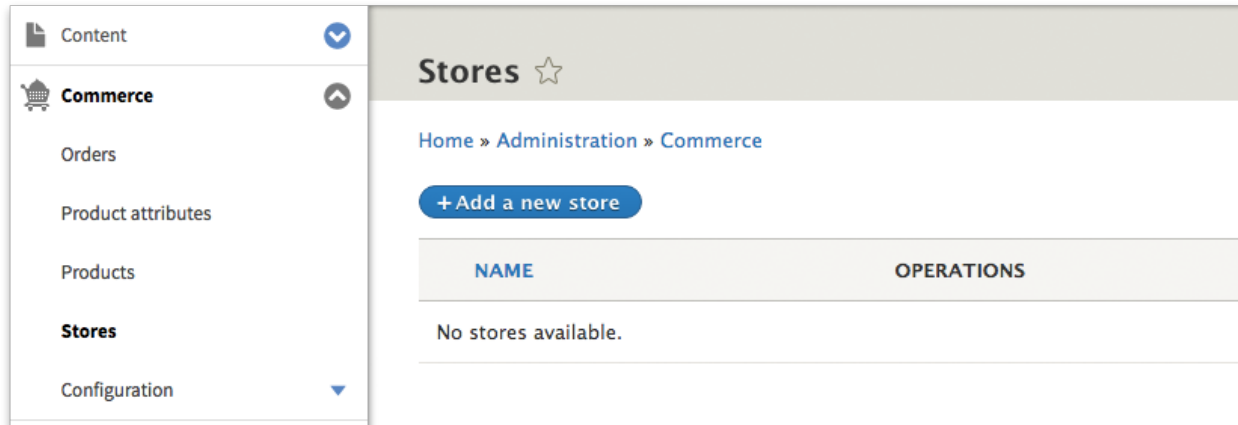


Next, click the Import tab (`admin/commerce/config/currency/import`). The reason currencies need to be imported is because we don't want to store all the world's currencies in your database if we don't have to, so we make no assumptions and let each store request specific access to specific currencies. The dataset is coming from the `intl` library which generates its dataset from an international and frequently updated standards body.



Once you've imported one or more currencies, you can move on to creating a store.

Create a store.



Next, we need to create a store. Every product requires one store. Additional details will be shared about the power of having stores baked into the core of Commerce, but for now, all you need to do is define your store's name, address, and select a few things about taxes and billing.

Add store ☆

[Home](#) » [Add store](#)

Name *

Email *

Store email notifications are sent from this address.

Default currency *

US Dollar ▼

▼ ADDRESS

Country *

United States ▼

Street address *

City * **State *** **Zip code ***

 - Select -

Supported billing countries

- All countries -
- Afghanistan
- Aland Islands
- Albania
- Algeria

Owner

admin (1) ○

☒ Default

▼ TAX SETTINGS

☐ Prices are entered with taxes included.

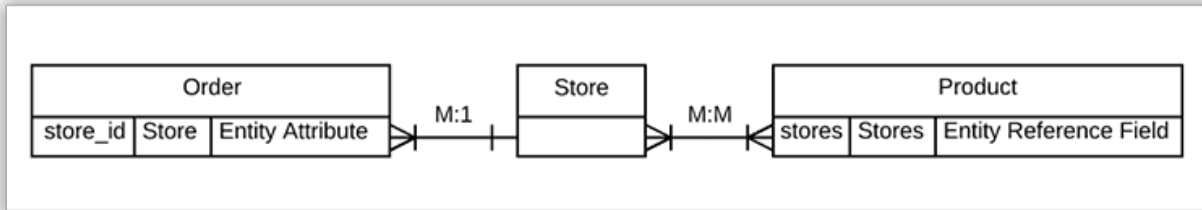
Tax registrations

- None -
- Afghanistan
- Aland Islands
- Albania

[Save](#)

Once you've got all those details filled out, click save and move on to creating products! Woohoo!

Overview & Architecture



Orders will only ever have one store, and it is stored as an entity attribute.

- Carts (which are Orders with additional functionality) can only contain products from one store.
- You can use this architecture to limit which products can be put into carts together, based on physical location or for billing/taxes purposes.

Products, by default, have an entity reference field that targets stores and allows one or more stores to be selected.

Stores are fieldable content entities (not configuration entities) that contain a lot of information about the physical location of the merchant. By default stores collect the following:

- Name
- Email Address
- Default Currency
- Address (used for determining taxes)
- Supported billing countries
- Owner
- Default status (used to select a store when one isn't given)
- Tax information

Use Cases

We optimize for the two use cases:

1. One business that has one or more locations

or

1. The marketplace model (where you have sellers)

For these use cases and possibly others, it is up to the developer to fill in the gaps that are present in the order workflow. This is different from Commerce 1.x in that we will support stores by default, allowing for community contributions to extend the functionality instead of trying to build store functionality from scratch.

1. One or more locations

This is the most common eCommerce situation where we have a single person, company, or organization that is taking payments online.

2. Marketplace model

The marketplace model is where you have many sellers who are taking payment for unique products.

Stores and Carts

A customer can have one or more Carts (which are a type of Order), if they have chosen a product from different stores.

3.5 Managing products

3.5.1 Products

Using product attributes - Before creating products, you need to create some attributes that you will use to differentiate your products. Read [here](#) to learn about how to create, edit, and making some attributes optional.

Products & Variations - Finally, you can create products! Follow the directions in this section on the most common use cases.

Purchasable Entities - When it comes to product architectures, there is no one true answer. Furthermore, different clients might have different needs. That's why it's important for Commerce 2.x to support any number of product architectures.

Setup product attributes



Fig. 3.1: Product Attribute Entity Relationships

Imagine you need to sell a DrupalCon t-shirt. This t-shirt comes in different sizes and colors. Each combination of size and color has its own SKU, so you know which color and size the customer has purchased and you can track exactly how many of each combination you have in stock.

Color and size are product attributes. Blue and small are product attribute values, belonging to the mentioned attributes. The combination of attribute values (with a SKU and a price) is called a product variation. These variations are grouped inside a product.

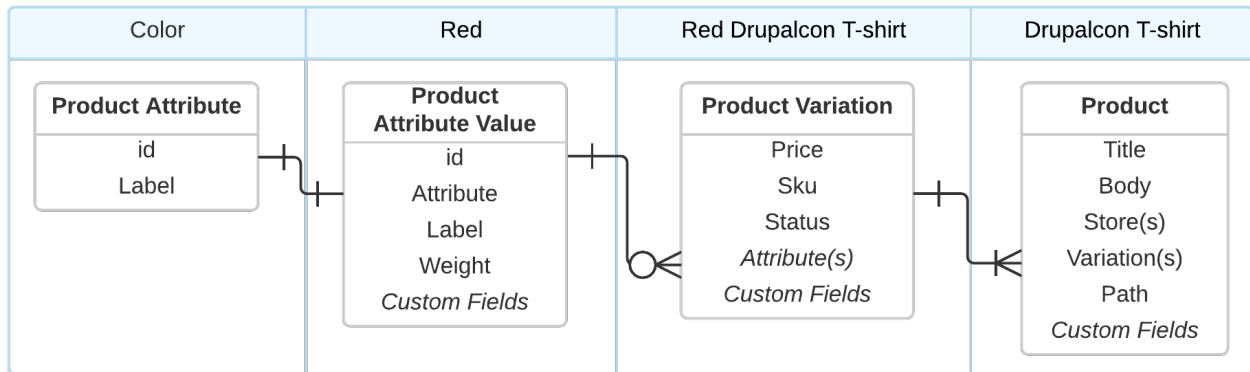


Fig. 3.2: Product Attribute Entity Relationships

Creating Attributes and their Values

For our t-shirt we need two attributes: color and size. Let's start by creating the color attribute. Go to `admin/commerce/product-attributes` and click the Add attribute link.

The screenshot shows the 'Add product attribute' form in the Drupal Commerce administration interface. The form includes the following elements:

- Title:** 'Add product attribute' with a star icon.
- Breadcrumbs:** Home » Administration » Commerce » Product attributes.
- Name:** A text field containing 'Color'. To the right, it says 'Machine name: color [Edit]'.
- Element type:** A dropdown menu set to 'Select list'.
- Help text:** 'Controls how the attribute is displayed on the add to cart form.'
- Checkbox:** 'Enable attribute value translation' (unchecked).
- Buttons:** A blue 'Save' button.

Fig. 3.3: Product Attribute Creation

After you have created the color attribute, we need to define at least one value. Normally we would simply say the color is “blue” or “red” but sometimes you might need to further define the attribute using fields. Adding fields is covered in detail later on in the documentation.

The product attribute values user interface allows creating and re-ordering multiple values at the same time and a very powerful translation capability:

Next, you will need to add the attribute to the product variation type. You can find these at `/admin/commerce/config/product-variation-types` and you just need to add/edit a product variation type that requires your new attribute.

Edit *Color* ☆

Edit
Manage fields
Manage form display
Manage display

Home » Administration » Commerce » Product attributes

Name *

Machine name: color

Element type

Controls how the attribute is displayed on the add to cart form.

☐ Enable attribute value translation

Show row weights

VALUE	OPERATIONS
<div> <div>+</div> <div> <p>Name *</p> <input type="text" value="Red"/> <p>Hex Value</p> <input type="text"/> </div> </div>	<div>Remove</div>
<div> <div>+</div> <div> <p>Name *</p> <input type="text" value="Black"/> <p>Hex Value</p> <input type="text"/> </div> </div>	<div>Remove</div>

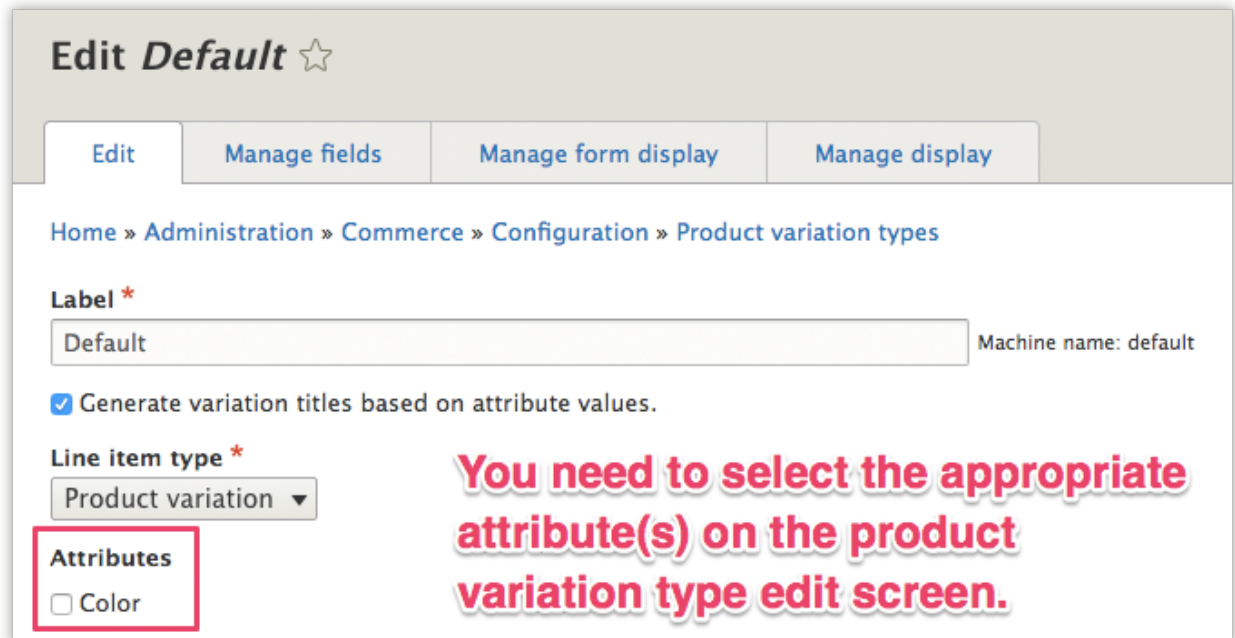
Add value

Reset to alphabetical

Save

Delete

Fig. 3.4: Product Attribute Value Creation



Edit Default ☆

Edit Manage fields Manage form display Manage display

Home » Administration » Commerce » Configuration » Product variation types

Label *

Default Machine name: default

☒ Generate variation titles based on attribute values.

Line item type *

Product variation ▼

Attributes

☐ Color

You need to select the appropriate attribute(s) on the product variation type edit screen.

Fig. 3.5: Adding Product Attribute to Product Variation

After you have added “Color” and the various colors your t-shirts are available in, the next step is to add that “color” attribute to our product. Store administrators can do this on the product variation type form, the checkbox in the last step automatically created entity referenced fields as needed:

Adding fields to Attributes

Product attributes are so much more than a word. Often times they represent a differentiation between products that is useful to call out visually for customers. The fieldable attribute value lets the information architect decide what best describes this attribute. Like any other fieldable entity, you can locate the list of attribute bundles and click edit fields:

`/admin/commerce/product-attributes`

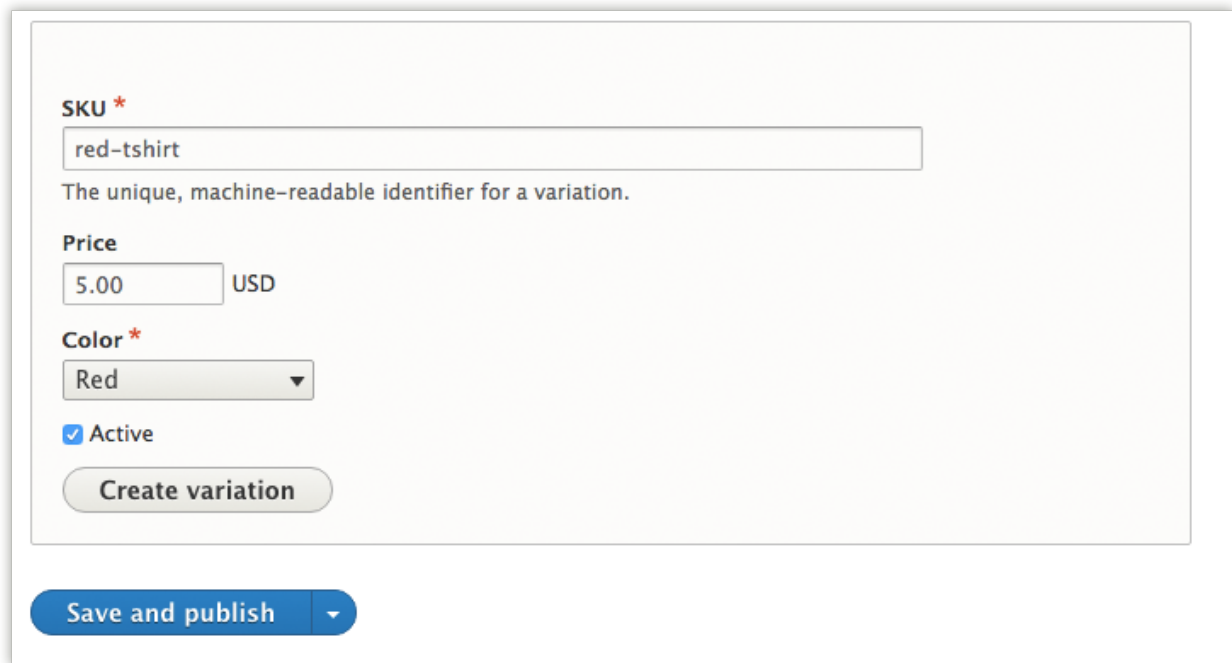
Add a field as you would expect. Most fields are supported and will automatically show up when you go to add attribute values:

Editing Attributes

Editing the attribute values is pretty easy. Simply locate the attribute type that has the values you want to edit: `/admin/commerce/product-attributes` And click “edit” and you will be taken to a screen to edit all the attributes of that type.

Optional Attributes

After creating attributes, the product variation type needs to know that it uses the attribute. The product variations are at `/admin/commerce/config/product-variation-types` and once you’ve clicked on the attribute you want...



SKU *

The unique, machine-readable identifier for a variation.

Price

 USD

Color *

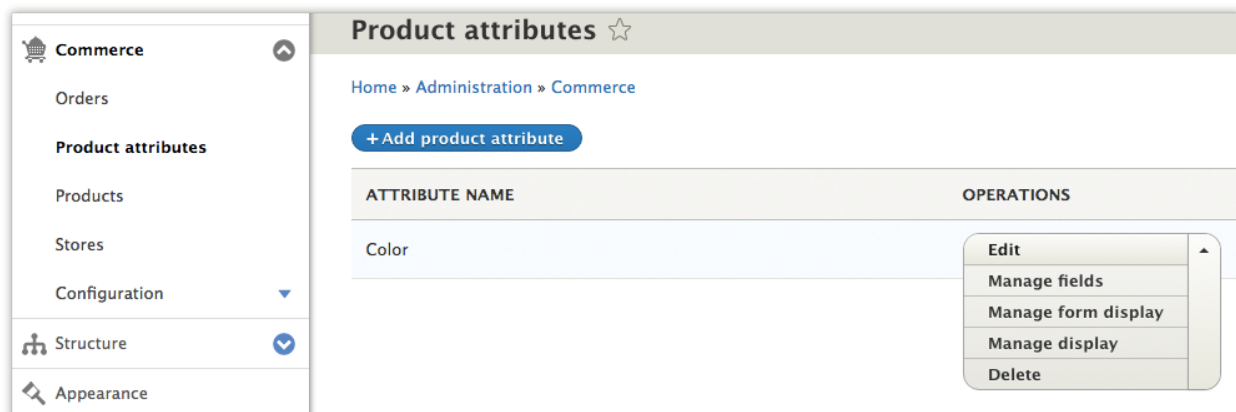
Red ▼

☒ Active

Create variation

Save and publish ▼

Fig. 3.6: Example Product variation form



Commerce

- Orders
- Product attributes**
- Products
- Stores
- Configuration ▼
- Structure ▼
- Appearance

Product attributes ☆

Home » Administration » Commerce

+ Add product attribute

ATTRIBUTE NAME	OPERATIONS
Color	<ul style="list-style-type: none"> Edit Manage fields Manage form display Manage display Delete

Fig. 3.7: Locating list of attributes

Edit *Color* ☆

Edit
Manage fields
Manage form display
Manage display

Home » Administration » Commerce » Product attributes

Name *

Machine name: color

Element type

Controls how the attribute is displayed on the add to cart form.

☐ Enable attribute value translation

Show row weights

VALUE	OPERATIONS
<div> <div>+</div> <div> <p>Name *</p> <input type="text" value="Red"/> <p>Hex Value</p> <input type="text"/> </div> </div>	<div>Remove</div>
<div> <div>+</div> <div> <p>Name *</p> <input type="text" value="Black"/> <p>Hex Value</p> <input type="text"/> </div> </div>	<div>Remove</div>

Add value

Reset to alphabetical

Save

Delete

Fig. 3.8: Example of attribute with more than one attribute

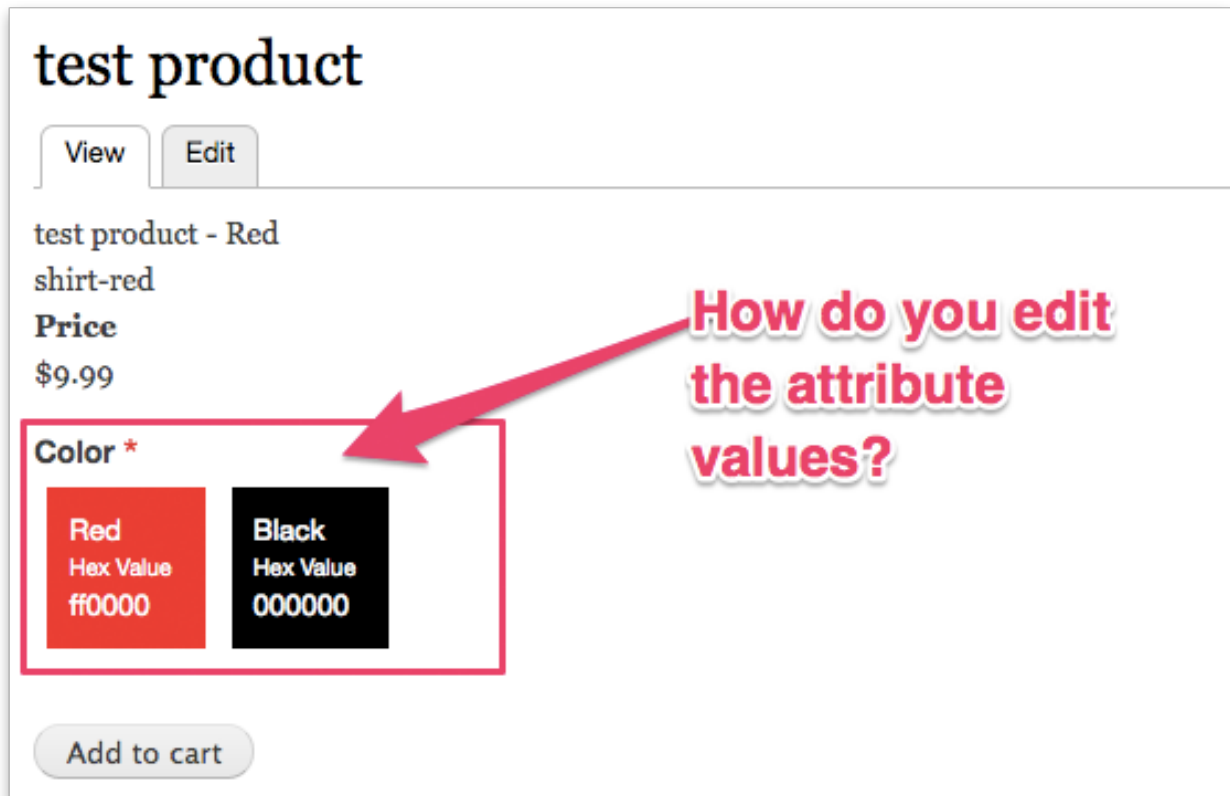


Fig. 3.9: How do you edit the attribute values?

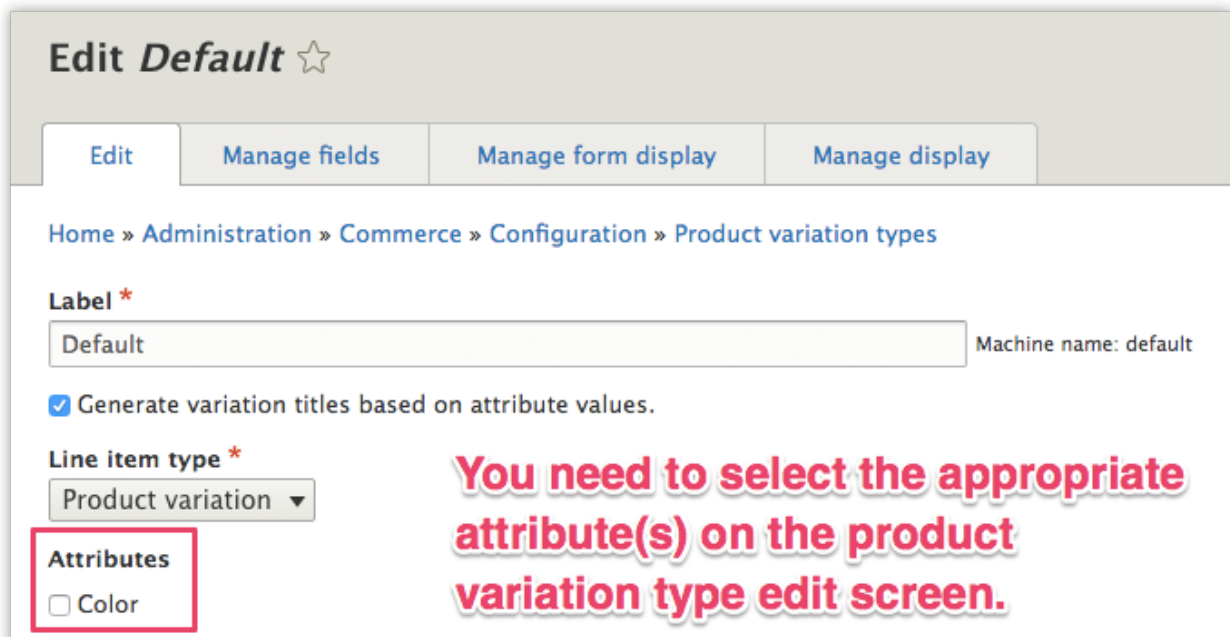


Fig. 3.10: Adding Product Attribute to Product Variation

Fields are added to the variation type that can then be modified. By default, all attribute fields are required. If your attribute is optional (perhaps some of the drupalcon t-shirts only come in blue), then you can locate the manage fields of your particular product variation type and make the `color` attribute optional by following these steps:

1. Go to `/admin/commerce/config/product-variation-types`
2. Click the drop down next to the variation type you want and click “manage fields”
3. Un-select the “required” checkbox to make the attribute optional.

Color settings for Default ☆

Edit Field settings

Home » Administration » Commerce » Configuration » Product variation types » Edit Default » Manage fields

Label *

Color

Help text

Instructions to present to the user below this field on the editing form.

Allowed HTML tags: <a> <big> <code> <i> <ins> <pre> <q> <small> <sub> <sup>

This field supports tokens

☒ Required field

Remove the required check to make the attribute optional

Fig. 3.11: Un-select the required checkbox

Make a product

Every product has one or more variations. In the event that a product has more than one variation, each variation is differentiated by some aspect of the product, whether it’s the product’s color, size, fabric, etc.

For example, you sell t-shirts (Product Type) and you have a new shipment of a particular Drupalcon t-shirt (Product). This Drupalcon t-shirt comes in different sizes and colors. Each combination of size and color (Small Red, Large Blue) represents a physical version of the t-shirt (Product Variation).

NOTE: In order to create your first product, you will need to have a store and a currency already set up. If you don’t have this, there’s a Getting Started section that will walk you through the steps.

Managing Products and their Variations

By default, variations are only manageable from the parent product, using Inline Entity Form. Variations do not have labels or titles. Labels, by default, are dynamically constructed from the attribute labels. To create or update a product variation, you must go to the product screen and either choose an existing product or create a new one.

You can simply go to `admin/commerce/products` and click “Add Product.”

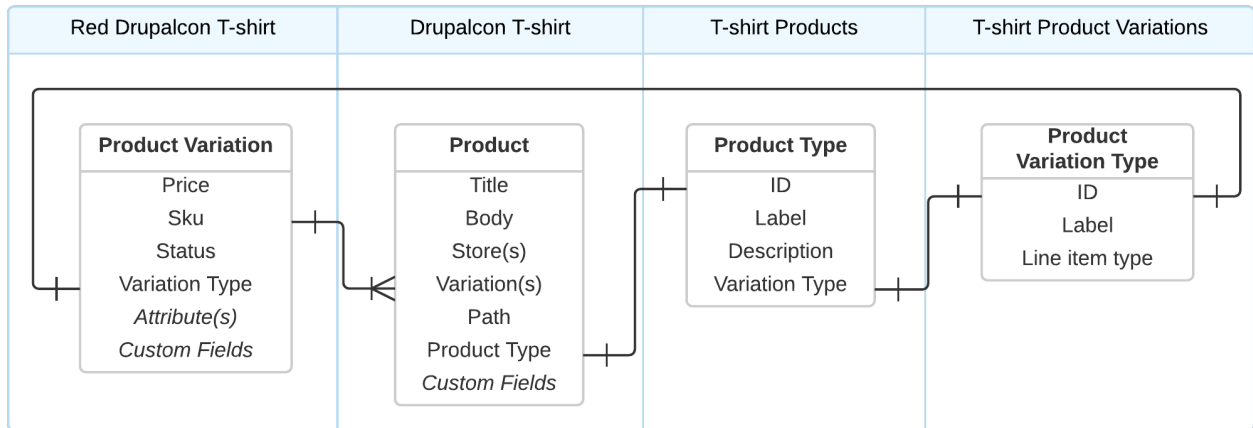


Fig. 3.12: Product Entity Relationships

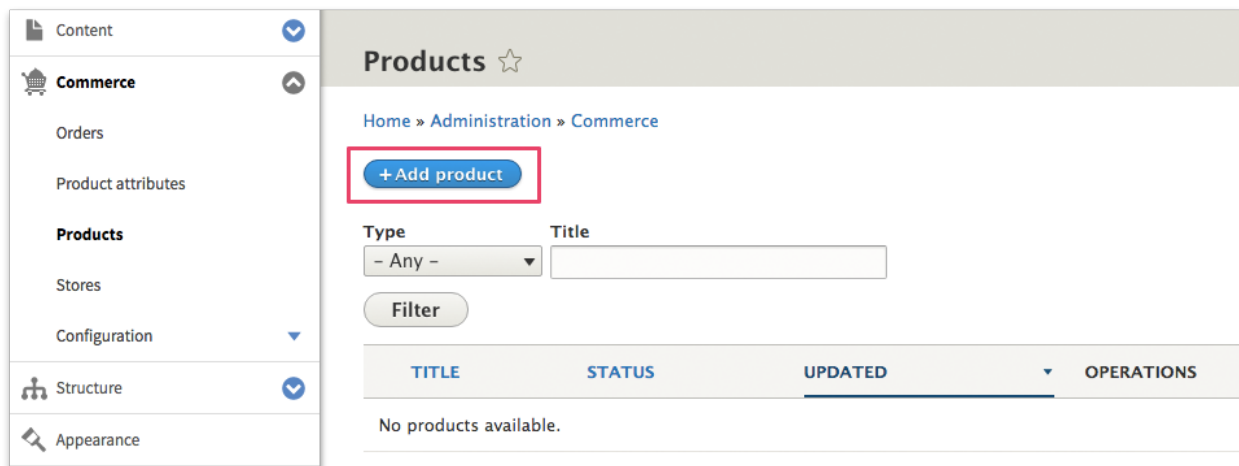


Fig. 3.13: Product select

Once you have selected an existing product or added a new one, you will be presented with a form that looks similar to the following. It will have “product details” like title, description, and path. And a widget for creating an unlimited number of variations that have prices, skus, and any available attributes.

Home » Add product **Product**

Title *

Body

URL PATH SETTINGS

URL alias

The alternative URL for this product. Use a relative path. For example, "/my-product".

AUTHORING INFORMATION

Author

admin (1)

Created

06/10/2016 03:28:45 PM

SKU *

The unique, machine-readable identifier for a variation.

Price

9.99 USD

Color *

- Select a value -

☒ Active

Create variation

Save and publish

Product Variations

Fig. 3.14: Product edit screen

Deleting a product deletes its variations. Adding a variation to a product automatically creates a backreference on the variation, accessed via `$variation->getProduct()`.

Product Fields

Products can have all kinds of fields. Often Commerce products will have a very media-rich set of content that is used to describe and present the product. These fields will remain the same and be available no matter which product variation is selected on the product page. Perhaps all of our t-shirt products have videos that show off Drupalers sprinting while wearing each of the t-shirts. We will need a field that accepts video urls and can render them for the page.

Adding a Product Field Product types, for example, our tshirt product type, can be found at `admin/commerce/config/product-types` (under the configuration menu option) and clicking on the arrow next to the Edit button will reveal all the management tasks for product variation types. Click on the Manage Fields option.

Once on the manage fields screen for our product type, you can add as many types of fields as you like by clicking the + Add Field button.

Variation Fields

Products variations can have attributes and other kinds of fields. Going back to our t-shirt analogy from above, if our t-shirts come in sizes and colors, perhaps the product variation should have an image field so you can upload a picture of a small red shirt. These kinds of non-attribute fields are loaded dynamically when variations are chosen.

Adding a Product Variation Field Product variation types can be found at `admin/commerce/config/product-variation-types` and clicking on the arrow next to the Edit button (1) will reveal all the management tasks for product variation types. Click on the Manage Fields option (2).

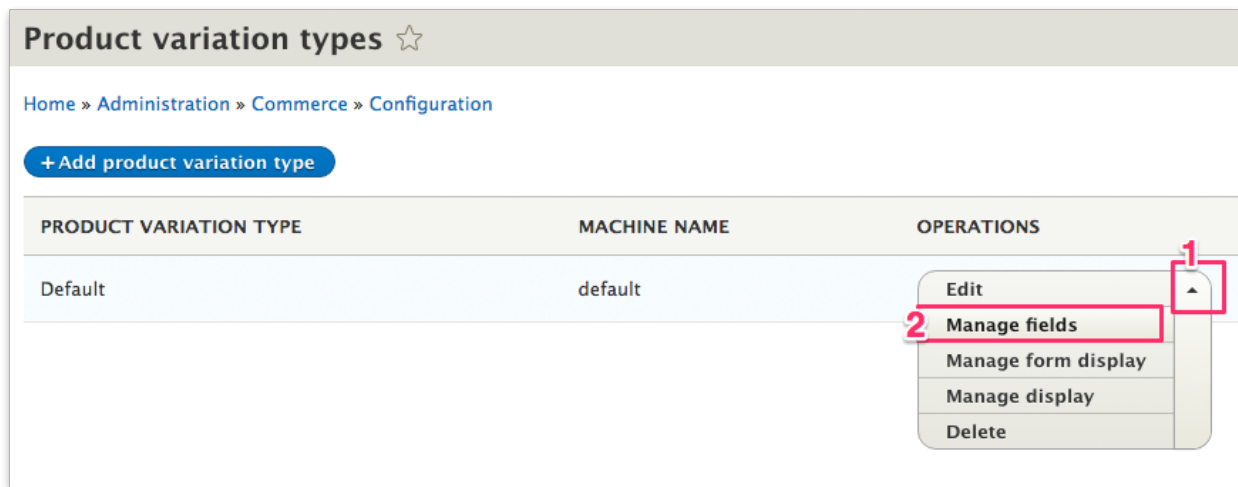


Fig. 3.15: Manage Fields

Once there, you can add as many types of fields as you like. Note that attributes that you have added in the past will show up here as entity reference fields. For our example, we will be adding an image field.

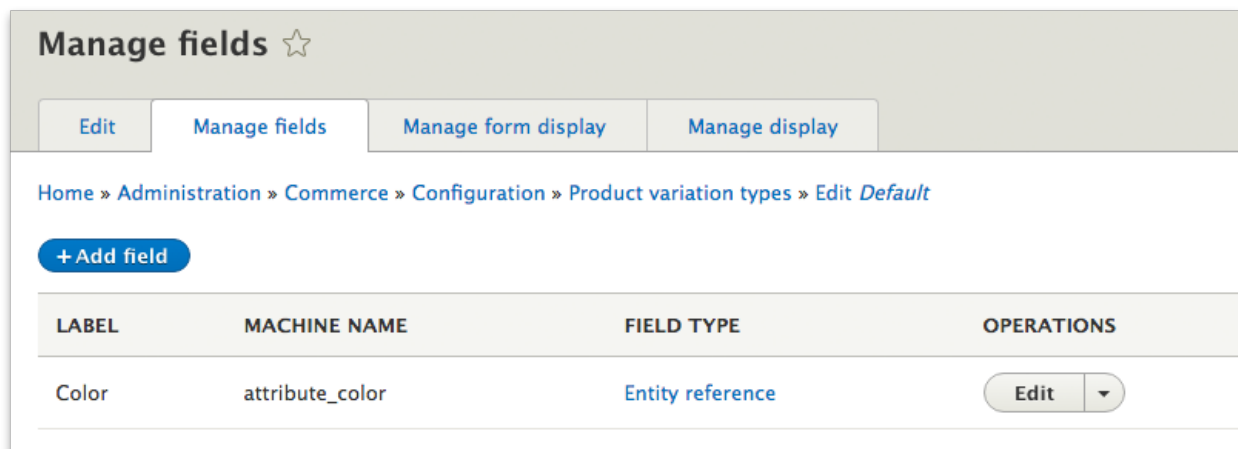


Fig. 3.16: Add a field

Choose the kind of field you would like to add and setup any of the settings as you need.

Fig. 3.17: Add an image field

Finally, you should have your new field showing up in your product add form located at `product/add`

Managing the display of the product

Once the tshirt has important content fields and the t-shirt variation fields have differentiating fields figured out, the product page may not look as clean the designer envisioned. It's likely that there are a number of labels for fields (like price, product image, SKU, etc) that you would rather not display. There are two different `Manage Display` locations you will need to manage in order to get the desired output on your product page.

NOTE: It's recommended that if you are using display modes to effect the product pages, that you use the "show weights" check box. The reason for this is that when a product is rendered, all fields, from the variation to the actual product get sorted based on weight. So if you just use the drag and drop methods, you will not get the granular control you might expect.

To fully control the display of all the fields it's helpful to think of the fields as being a part of one big group.

Above, our T-shirt Product fields (body, variations) are rendered with our T-shirt Product Variation fields (Price, Image). In order to achieve this order, the field weights must be manually set to go in order, as if they were in a large group.

Product field weight can be managed here: `admin/commerce/config/product-types`

Product Variation field weight can be managed here: `admin/commerce/config/product-variation-types`

FANCY FEATURE ALERT: You may have noticed that product variation fields can be displayed **INDEPENDENTLY** of the variations field. Lots of work has gone in to making sure these fields get replaced easily and consistently when a new product is selected on the add-to-cart form. This was developed specifically to allow fine-tuned control of how a store would want to present different pieces of information. Perhaps you really need the picture of the selected t-shirt to appear before the body field of the product. Just change the weight :)

Create a product type

@todo

SKU *

 The unique, machine-readable identifier for a variation.

Price
 USD

Color *

☒ Active

Product Image

No file chosen

One file only.
8 MB limit.
Allowed types: png gif jpg jpeg.

Fig. 3.18: New Field Available


Drupalcon T-shirt

0

This is some body text for the Product. Imagine all the cool details you can share about thread-count, t-shirt sponsorship opportunities, and interesting discounts, like the fact that if you buy 200 drupalcon tshirts, we will send you a free box of bendy straws. That's right, free.

1

2



3

Color *

'Manage Display'
Field Weights

Product Fields	Product Variation Fields
Body 0	
	1 Price
	2 Image
Variations 3	

Fig. 3.19: Manage Display field weight graphic

Purchasable Entities

When it comes to product architectures, there is no one true answer. Furthermore, different clients might have different needs. That's why it's important for Commerce 2.x to support any number of product architectures.

The ProductVariation entity class implements the PurchasableEntityInterface:

... needs screenshot of interface code ...

Any content entity type that implements this interface can be purchased. The order module doesn't depend on the product module, the product module just provides the default (and most common) product architecture. A product bundle module will probably want to define its own product architecture, etc.

Line items have a purchased_entity reference field. The target_type of that reference field is different for each line item type.

... needs screenshot of line item type edit page ...

Here the line item type points to the product variation entity type, indicating that the "Product variation" line item type is used to purchase product variations.

Early in the Commerce 2.x cycle we explored the idea of hierarchical products, but after initial exploration found out that the idea required several months of extra effort (having to rewrite the Tree module, reinvent an IEF like widget, UX and performance considerations). We removed it from the roadmap with a heavy heart, but now that Commerce 2.x supports custom product architectures, we can easily explore the idea in contrib at a later date.

3.6 Catalog and product pages

3.6.1 Catalog and product pages

@todo * Setup a catalog using Search API + Views * Customize add to cart form using order item form display * Templating a product page

Create a product catalog

@todo install search_api @todo create a server/index @todo create view

Customize the add to cart form

@todo

Theme a product page

@todo

3.7 Product merchandising

3.7.1 Product merchandising

@todo discuss features

Create a promotion

Create a promotion

@todo

3.8 Working with orders

3.8.1 Orders

Orders contain a list of order items and customer information. Orders have states that are controlled through State Machine.

Orders and order items - Orders contain order items, which represent purchased items.

Understanding order types - You can have different order types. Order types have their own settings when it comes to cart, checkout, and its processing.

Order Processing - Allows you to process an order, when the system recalculates order item prices and availability.

Order Types

Order types allow you to control how an order interacts with the other components of Drupal Commerce, and the how the order moves through the system.



Label *

Default Machine name: default

Label for the order type.

Workflow

Default ▼

Used by all orders of this type.

Fig. 3.20: Order workflow settings

Orders have a specific workflow that defines what states and transitions the order can move in. Each order type can have its own workflow.

This means your default order type, which has shippable products, can use the *Fulfillment* workflow. Meanwhile, your digital goods order type can have the more simplistic *Default* workflow.

Each order type can control its refresh settings to control how often order draft's are processed. This controls the order refresh process.

The cart module allows each order type to control the default view used when rendering carts in the cart block or cart form.

You can use a different checkout flow for each order type. In this case you would have a physical order use a multiple step checkout flow that requires shipping information. A digital order could have a more simplified checkout flow that has one step (i.e.: payment.)

▼ ORDER REFRESH

These settings let you control how draft orders are refreshed, the process during which order item prices are recalculated.

Order refresh mode

- ☒ Refresh a draft order when it is loaded regardless of who it belongs to.
- ☐ Only refresh a draft order when it is loaded if it belongs to the current user.

Order refresh frequency *

seconds

Draft orders will only be refreshed if more than the specified number of seconds have passed since they were last refreshed.

Fig. 3.21: Order refresh settings

▼ SHOPPING CART SETTINGS

Shopping cart form view

▼

Shopping cart block view

▼

Fig. 3.22: Order type cart settings

▼ CHECKOUT SETTINGS

Checkout flow *

▼

Fig. 3.23: Order type checkout settings

Order Items

An order item represents a purchasable entity inside of an order. It contains a reference to the purchasable entity, a quantity, a unit price and a total price.

Note: In Drupal Commerce 1.x, these were called line items.

The order total is based off the unit price of order items multiplied by their quantity and the sum of all order item totals.

Order items have their unit price calculated during the order refresh process. This synchronizes the price with the current purchasable entity's price while the order is still in a draft state.

The add to cart form is actually the create form for an order item entity. It is a specific form display. Selecting attributes on the add to cart form identifies the proper reference purchased entity to reference.

Manage form display ☆

Edit Manage fields **Manage form display** Manage display Devel

Default **Add to cart**

Home » Administration » Commerce » Configuration » Order item types » Edit *Product variation* » Manage form display [Show row weights](#)

FIELD	WIDGET
✚ Purchased entity	Product variation attributes ▼
Disabled	
✚ Created	- Hidden - ▼
✚ Unit price	- Hidden - ▼
✚ Quantity	Number field ✓ - Hidden -

Save

Fig. 3.24: Order item add to cart form

Advanced topics

Order Processing

Order processing is part of the order refresh process. This is run when on draft orders to ensure that it has up to date adjustments and that its order items are up to date.

3.9 Configuring Checkout

3.9.1 Configuring your checkout

Allowing guest checkout, or account login

Customizing your checkout

Creating a checkout pane plugin

Creating a checkout flow plugin

3.10 Payments

3.10.1 Setting up payments

@todo discuss onsite, offsite ability @todo call out major payment gateways

Install a payment gateway

@todo how to create a payment gateway @todo maybe two examples: like US and CA

Managing order payments

@todo void, capture, refund

3.11 Code Recipes

3.11.1 Code Recipes

A list of code samples/examples outlining how to create and load commerce entities entirely in code.

These recipes are all designed to work off of each other, so try it out - you can run all the code in a single shot from top to bottom.

Stores - Stores and types.

Variations - Product variations and types.

Attributes - Product attributes and values.

Products - Products and types.

Orders - Orders, order items, and their types.

Store recipes

Everything starts with a store. Products can belong to many stores, and orders belong to a single store.

Creating a store type

```
/**
 * id [String]
 *   The primary key for this store type.
 *
 * label [String]
 *   The label for this store type.
 *
 * description [String]
 *   The description for this store type.
 */
$store_type = \Drupal\commerce_store\Entity\StoreType::create([
  'id' => 'custom_store_type',
  'label' => 'My custom store type',
  'description' => 'This is my first custom store type!',
]);
$store_type->save();
```

Loading a store type

```
// Loading is based off of the primary key [String] that was defined when creating it.
$store_type = \Drupal\commerce_store\Entity\StoreType::load('custom_store_type');
```

Creating a store

```
/**
 * type [String] - [DEFAULT = 'online']
 *   Foreign key for the store type to use.
 *
 * uid [Integer]
 *   The user id that created the store.
 *
 * name [String]
 *   The store's name.
 *
 * mail [String]
 *   The store's email address.
 *
 * address [\Drupal\address\AddressInterface]
 *   The store's address.
 *
 * default_currency [String]
 *   The currency the store uses.
 *
 * billing_countries [Array(String)]
 *   Array of country codes selected for the store.
 */

// The store's address.
$address = [
  'country_code' => 'US',
  'address_line1' => '123 Street Drive',
  'locality' => 'Beverly Hills',
```

```
'administrative_area' => 'CA',
'postal_code' => '90210',
];

// The currency code.
$currency = 'USD';

// If needed, this will import the currency.
$currency_importer = \Drupal::service('commerce_price.currency_importer');
$currency_importer->import($currency);

$store = \Drupal\commerce_store\Entity\Store::create([
  'type' => 'custom_store_type',
  'uid' => 1,
  'name' => 'My Store',
  'mail' => 'admin@example.com',
  'address' => $address,
  'default_currency' => $currency,
  'billing_countries' => ['US'],
]);
$store->save();

// If needed, this sets the store as the default store.
$store_storage = \Drupal::service('entity_type.manager')->getStorage('commerce_store');
$store_storage->markAsDefault($store);
```

Loading a store

```
// Loading is based off of the primary key [Integer]
// 1 would be the first one saved, 2 the next, etc.
$store = \Drupal\commerce_store\Entity\Store::load(1);
```

Product Variations and types

Product variations are the purchasable parts of products, thus products need at least one variation.

Creating variation types

```
/**
 * id [String]
 *   The primary key for this variation type.
 *
 * label [String]
 *   The label for this variation type.
 *
 * status [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   [AVAILABLE = FALSE, TRUE]
 *   Whether or not it's enabled or disabled. 1 for enabled.
 *
 * orderItemType [String] - [DEFAULT = default]
 *   Foreign key for the order item type to use.
 *
 * generateTitle [Bool] - [DEFAULT = TRUE]
 *   Whether or not it should generate the title based off of product label and attributes.
```



```
*/
$variation_type = \Drupal\commerce_product\Entity\ProductVariationType::create([
  'id' => 'my_custom_variation_type',
  'label' => 'Variation Type With Color',
  'status' => TRUE,
  'orderItemType' => 'default',
  'generateTitle' => TRUE,
]);
$variation_type->save();
```

Loading a variation type

```
// Loading is based off of the primary key [String] that was defined when creating it.
$variation_type = \Drupal\commerce_product\Entity\ProductVariationType::load('my_custom_variation_type');
```

Creating variations

```
/**
 * type [String] - [DEFAULT = default]
 *   Foreign key of the variation type to use.
 *
 * sku [String]
 *   The sku for this variation.
 *
 * status [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   [AVAILABLE = FALSE, TRUE]
 *   Whether or not it's enabled or disabled. 1 for enabled.
 *
 * price [\Drupal\commerce_price\Price] - [OPTIONAL]
 *   The price for this variation.
 *
 * title [String] - [POTENTIALLY NOT REQUIRED]
 *   The title for the product variation.
 *   If the variation type is set to generate a title, this is not used.
 *   Otherwise, a title must be given.
 */
$variation = \Drupal\commerce_product\Entity\ProductVariation::create([
  'type' => 'my_custom_variation_type',
  'sku' => 'test-product-01',
  'status' => TRUE,
  'price' => new \Drupal\commerce_price\Price('24.99', 'USD'),
]);
$variation->save();
```

Loading a variation

```
// Loading is based off of the primary key [Integer]
//   1 would be the first one saved, 2 the next, etc.
$variation = \Drupal\commerce_product\Entity\ProductVariation::load(1);
```

Product Attributes and Values

Product variation types can have certain attributes (ex. color) and those attributes have values (ex red, blue). In this example, we will create two attributes (color and size) and add them to the variation type we made previously.

Creating attributes

```
/**
 * id [String]
 *   The primary key for this attribute.
 *
 * label [String]
 *   The label for this attribute.
 */
$color_attribute = \Drupal\commerce_product\Entity\ProductAttribute::create([
  'id' => 'color',
  'label' => 'Color',
]);
$color_attribute->save();

$size_attribute = \Drupal\commerce_product\Entity\ProductAttribute::create([
  'id' => 'size',
  'label' => 'Size',
]);
$size_attribute->save();

// We load a service that adds the attributes to the variation type we made previously.
$attribute_field_manager = \Drupal::service('commerce_product.attribute_field_manager');

$attribute_field_manager->createField($color_attribute, 'my_custom_variation_type');
$attribute_field_manager->createField($size_attribute, 'my_custom_variation_type');
```

Loading an attribute

```
// Loading is based off of the primary key [String] that was defined when creating it.
$size_attribute = \Drupal\commerce_product\Entity\ProductAttribute::load('size');
```

Creating values for an attribute

```
/**
 * attribute [String]
 *   Foreign key of the attribute we want.
 *
 * name [String]
 *   The name of this value.
 */
$red = \Drupal\commerce_product\Entity\ProductAttributeValue::create([
  'attribute' => 'color',
  'name' => 'Red',
]);
$red->save();

$blue = \Drupal\commerce_product\Entity\ProductAttributeValue::create([
```

```
'attribute' => 'color',
'name' => 'Blue',
});
$blue->save();

$medium = \Drupal\commerce_product\Entity\ProductAttributeValue::create([
  'attribute' => 'size',
  'name' => 'Medium',
]);
$medium->save();

$large = \Drupal\commerce_product\Entity\ProductAttributeValue::create([
  'attribute' => 'size',
  'name' => 'Large',
]);
$large->save();
```

Loading an attribute value

```
// Loading is based off of the primary key [Integer]
// 1 would be the first one saved, 2 the next, etc.
$red = \Drupal\commerce_product\Entity\ProductAttributeValue::load(1);
```

Assigning attributes to a variation

Let's say we want our hypothetical product to have two variations. One will be the color red and size medium, and the other will be the color blue and size large. // [IMPORTANT] - If a Product Variation Type has fields for attributes (as we added above), then variations of that type **MUST** have those attributes.

```
/**
 * attribute_<ATTRIBUTE_ID> [\Drupal\commerce_product\Entity\ProductAttributeValueInterface]
 * The attribute value entity to use for the attribute type.
 */
$variation_red_medium = \Drupal\commerce_product\Entity\ProductVariation::create([
  'type' => 'my_custom_variation_type',
  'sku' => 'product-red-medium',
  'price' => new \Drupal\commerce_price\Price('10.00', 'USD'),
  'attribute_color' => $red,
  'attribute_size' => $medium,
]);
$variation_red_medium->save();

$variation_blue_large = \Drupal\commerce_product\Entity\ProductVariation::create([
  'type' => 'my_custom_variation_type',
  'sku' => 'product-blue-large',
  'price' => new \Drupal\commerce_price\Price('10.00', 'USD'),
  'attribute_color' => $blue,
  'attribute_size' => $large,
]);
$variation_blue_large->save();
```

Products and types

Creating product types

```
/**
 * id [String]
 *   Primary key for this product type.
 *
 * label [String]
 *   Label for this product type
 *
 * status [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   [AVAILABLE = FALSE, TRUE]
 *   Whether or not it's enabled or disabled. 1 for enabled.
 *
 * description [String]
 *   Description for this product.
 *
 * variationType [String] - [DEFAULT = default]
 *   Foreign key for the variation type used.
 *
 * injectVariationFields [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   Whether or not to inject the variation fields.
 */

// Create the product type.
$product_type = \Drupal\commerce_product\Entity\ProductType::create([
  'id' => 'my_custom_product_type',
  'label' => 'My custom product type',
  'description' => '',
  'variationType' => 'my_custom_variation_type',
  'injectVariationFields' => TRUE,
]);
$product_type->save();

// These three functions must be called to add the appropriate fields to the type
commerce_product_add_variations_field($product_type);
commerce_product_add_stores_field($product_type);
commerce_product_add_body_field($product_type);
```

Loading a product type

```
// Loading is based off of the primary key [String] that was defined when creating it.
$product_type = \Drupal\commerce_product\Entity\ProductType::load('my_custom_product_type');
```

Creating products

```
/**
 * uid [Integer]
 *   Foreign key of the user that created the product.
 *
 * type [String] - [DEFAULT = default]
 *   Foreign key of the product type being used.
 *
```

```

* title [String]
*   The product title.
*
* stores [Array(\Drupal\commerce_store\Entity\StoreInterface)]
*   Array of stores this product belongs to.
*
* variations [Array(\Drupal\commerce_product\Entity\ProductVariationInterface)]
*   Array of variations that belong to this product.
*/

// The variations that belong to this product.
$variations = [
    $variation_blue_large,
];

$product = \Drupal\commerce_product\Entity\Product::create([
    'uid' => 1,
    'type' => 'my_custom_product_type',
    'title' => 'My Custom Product',
    'stores' => [$store],
    'variations' => $variations,
]);
$product->save();

// You can also add a variation to a product using the addVariation() method.
$product->addVariation($variation_red_medium);
$product->save();

```

Loading a product

```

// Loading is based off of the primary key [Integer]
//   1 would be the first one saved, 2 the next, etc.
$product = \Drupal\commerce_product\Entity\Product::load(1);

```

Orders and order items

Creating order types

```

/**
 * id [String]
 *   The primary key for this order type.
 *
 * label [String]
 *   The label for this order type.
 *
 * status [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   [AVAILABLE = FALSE, TRUE]
 *   Whether or not it's enabled or disabled. 1 for enabled.
 *
 * workflow [String] - [DEFAULT = order_default]
 *   [AVAILABLE = order_default, order_default_validation, order_fulfillment, order_fulfillment_validation]
 *   The workflow id to use as the workflow.
 *
 * refresh_mode [String] - [DEFAULT = always]

```

```
*   [AVAILABLE = always, customer]
*   The refresh mode to use as the refresh mode.
*
* refresh_frequency [Integer] - [DEFAULT = 30]
*   The refresh frequency in seconds.
*/
$order_type = \Drupal\commerce_order\Entity\OrderType::create([
  'status' => TRUE,
  'id' => 'custom_order_type',
  'label' => 'My custom order type',
  'workflow' => 'order_default',
  'refresh_mode' => 'always',
  'refresh_frequency' => 30,
]);
$order_type->save();

// This must be called after saving.
commerce_order_add_order_items_field($order_type);
```

Loading an order type

```
// Loading is based off of the primary key [String] that was defined when creating it.
$order_type = \Drupal\commerce_order\Entity\OrderType::load('custom_order_type');
```

Creating order item types

```
/**
 * id [String]
 *   The primary key for this order item type.
 *
 * label [String]
 *   The label for this order item type.
 *
 * status [Bool] - [OPTIONAL, DEFAULTS TO TRUE]
 *   [AVAILABLE = FALSE, TRUE]
 *   Whether or not it's enabled or disabled. 1 for enabled.
 *
 * purchasableEntityType [String] - [DEFAULT = commerce_product_variation]
 *   Foreign key to use for the purchasable entity type.
 *
 * orderType [String] - [DEFAULT = default]
 *   Foreign key to use for the order type.
 */
$order_item_type = \Drupal\commerce_order\Entity\OrderItemType::create([
  'id' => 'custom_order_item_type',
  'label' => 'My custom order item type',
  'status' => TRUE,
  'purchasableEntityType' => 'commerce_product_variation',
  'orderType' => 'custom_order_type',
]);
$order_item_type->save();
```

Loading an order item type

```
// Loading is based off of the primary key [String] that was defined when creating it.
$order_item_type = \Drupal\commerce_order\Entity\OrderItemType::load('custom_order_item_type');
```

Creating order items

```
/**
 * type [String] - [DEFAULT = product_variation]
 *   Foreign key to use for the order item type.
 *
 * purchased_entity [Integer | \Drupal\commerce\PurchasableEntityInterface]
 *   Foreign key to use for the purchased entity. Either the id, or object implementing the interface.
 *
 * quantity [Integer]
 *   How many of the purchased items.
 *
 * unit_price [\Drupal\commerce_price\Price]
 *   The price per each item, not the total.
 *
 * adjustments [OPTIONAL] - [Array(Drupal\commerce_order\Adjustment)]
 *   Array of any price adjustments.
 */
$order_item = \Drupal\commerce_order\Entity\OrderItem::create([
  'type' => 'custom_order_item_type',
  'purchased_entity' => $variation_red_medium,
  'quantity' => 2,
  'unit_price' => $variation_red_medium->getPrice(),
]);
$order_item->save();

// You can set the quantity with setQuantity.
$order_item->setQuantity('1');
$order_item->save();

// You can also set the price with setUnitPrice.
$unit_price = new \Drupal\commerce_price\Price('9.99', 'USD');
$order_item->setUnitPrice($unit_price);
$order_item->save();
```

Loading an order item

```
// Loading is based off of the primary key [Integer]
//   1 would be the first one saved, 2 the next, etc.
$order_item = \Drupal\commerce_order\Entity\OrderItem::load(1);
```

Creating orders

```
/**
 * type [String] - [DEFAULT = default]
 *   Foreign key to use for the order type.
 *
 * state [String] - [DEFAULT = draft]
```

```

*   [AVAILABLE = draft, completed, canceled]
*   The state the order is in.
*
* mail [String]
*   The email address the order belongs to.
*
* uid [Integer]
*   The user id the order belongs to.
*
* ip_address [String]
*   The ip address the order was created from.
*
* order_number [Integer | String] - [OPTIONAL, DEFAULTS TO id]
*   The order number for the order. If left out, defaults to the order's id.
*
* billing_profile [\Drupal\profile\Entity\ProfileInterface]
*   The billing profile for the order.
*
* store_id [Integer]
*   The foreign key for the store that this order belongs to.
*
* order_items [Array(\Drupal\commerce_order\Entity\OrderItemInterface)]
*   Array of all the order items that belong to this order.
*
* adjustments [OPTIONAL] - [Array(Drupal\commerce_order\Adjustment)]
*   Array of any price adjustments.
*
* placed [Timestamp]
*   The time the order was placed.
*
* completed [OPTIONAL] - [Timestamp]
*   The time the order was completed.
*/

// Create the billing profile.
$profile = \Drupal\profile\Entity\Profile::create([
  'type' => 'customer',
  'uid' => 1,
]);
$profile->save();

// Next, we create the order.
$order = \Drupal\commerce_order\Entity\Order::create([
  'type' => 'custom_order_type',
  'state' => 'draft',
  'mail' => 'user@example.com',
  'uid' => 1,
  'ip_address' => '127.0.0.1',
  'order_number' => '6',
  'billing_profile' => $profile,
  'store_id' => $store->id(),
  'order_items' => [$order_item],
  'placed' => time(),
]);
$order->save();

```


Loading an order

```
// Loading is based off of the primary key [Integer]
// 1 would be the first one saved, 2 the next, etc.
$order = \Drupal\commerce_order\Entity\Order::load(1);
```