saleor Documentation

Release 0.1

Mirumee Software

Jul 27, 2017

Contents

1 Getting started **3**
- 1.1 Prerequisites . . . 3
- 1.2 Installation . . . 4
- 1.3 Example data . . . 4

2 Configuration **5**
- 2.1 Environment variables . . . 5

3 Product management **7**
- 3.1 Class diagram . . . 8
- 3.2 Product variant . . . 9
- 3.3 Product . . . 9
- 3.4 Product class . . . 9
- 3.5 Attributes . . . 9
- 3.6 Example - Coffee . . . 10
- 3.7 Example - Online game item . . . 10

4 Stock management **13**
- 4.1 Product availability . . . 13
- 4.2 Allocating stock for new orders . . . 13
- 4.3 Decreasing stock after shipment . . . 14

5 Site settings **15**
- 5.1 Context processor . . . 15

6 Payments **17**
- 6.1 Supported gateways . . . 17
- 6.2 3-D Secure . . . 18
- 6.3 Fraud protection . . . 18
- 6.4 Authorisation and capture . . . 18
- 6.5 Refunds . . . 18

7 Integrations **19**
- 7.1 Static file and media storage using Amazon S3 . . . 19
- 7.2 Converting prices to local currencies using Open Exchange Rates . . . 19
- 7.3 Reporting with Google Analytics . . . 20
- 7.4 Send product data to Google Merchant Center . . . 20

7.5 Full text search with Elasticsearch . . . 20

8 Development **21**
8.1 Working with templates . . . 21
8.2 Working with front-end assets . . . 21
8.3 Working with backend code . . . 22
8.4 Running tests . . . 22
8.5 Continuous integration . . . 22
8.6 Docker . . . 22

9 Deployment **25**
9.1 Docker . . . 25
9.2 Heroku . . . 25

10 Indices and tables **27**

An open source storefront written in Python.

Contents:

CHAPTER 1

Getting started

Note: If you prefer using containers or have problems with configuring PostgreSQL, Redis and Elasticsearch, try *Docker* instructions.

Prerequisites

Before you are ready to run Saleor you will need certain software installed on your computer.

1. Python version 3.5.x or 2.7.x
2. pip if you're using an older release of Python 2.7
3. `wheel` Python package if you're using pip older than 8.1.2
4. Node.js version 6 or above

   Note: Debian and Ubuntu users who install Node.js using system packages will also need to install the `nodejs-legacy` package.

5. webpack module bundler installed globally with:

```
$ npm i webpack -g
```

6. Yarn installed globally with:

```
$ npm i yarn -g
```

7. PostgreSQL version 9.4 or above

We also strongly recommend creating a virtual environment before proceeding with installation.

Installation

1. Clone the repository (or use your own fork):

```
$ git clone https://github.com/mirumee/saleor.git
```

2. Enter the directory:

```
$ cd saleor/
```

3. Install all dependencies:

```
$ pip install -r requirements.txt
```

4. Set `SECRET_KEY` environment variable.

> **Note:** Secret key should be a unique string only your team knows. It's serious as this key is used to ensure security of your installation. Consult Django's documentation for details.
>
> We try to provide usable default values for all of the settings. We've decided not to provide a default for `SECRET_KEY` as we fear someone would inevitably ship a project with the default value left in code.

```
$ export SECRET_KEY='<mysecretkey>'
```

5. Prepare the database:

```
$ python manage.py migrate
```

6. Install front-end dependencies:

```
$ yarn
```

> **Note:** If this step fails go back and make sure you're using new enough version of Node.js.

7. Prepare front-end assets:

```
$ yarn run build-assets
```

8. Run like a normal django project:

```
$ python manage.py runserver
```

Example data

If you'd like some data to test your new storefront you can populate the database with example products and orders:

```
$ python manage.py populatedb
```

CHAPTER 2

Configuration

We are fans of the 12factor approach and portable code so you can configure most of Saleor using just environment variables.

Environment variables

`ALLOWED_HOSTS` Controls Django's allowed hosts setting. Defaults to `localhost`.

`CACHE_URL` or `REDIS_URL` The URL of a cache database. Defaults to local process memory.

Redis is recommended. Heroku's Redis will export this setting automatically.

Example: `redis://redis.example.com:6379/0`

> **Warning:** If you plan to use more than one WSGI process (or run more than one server/container) you need to use a shared cache server. Otherwise each process will have its own version of each user's session which will result in people being logged out and losing their shopping carts.

`DATABASE_URL` Defaults to a local PostgreSQL instance. See docker-compose for how to get a local database running inside a Docker container.

Most Heroku databases will export this setting automatically.

Example: `postgres://user:password@psql.example.com/database`

`DEBUG` Controls Django's debug mode. Defaults to `True`.

`DEFAULT_FROM_EMAIL` Default email address to use for outgoing mail.

`EMAIL_URL` The URL of the email gateway. Defaults to printing everything to the console.

Example: `smtp://user:password@smtp.example.com:465/?ssl=True`

`INTERNAL_IPS` Controls Django's internal IPs setting. Defaults to `127.0.0.1`.

Separate multiple values with whitespace.

SECRET_KEY Controls Django's secret key setting.

MAX_CART_LINE_QUANTITY Controls maximum number of items in one cart line. Defaults to `50`

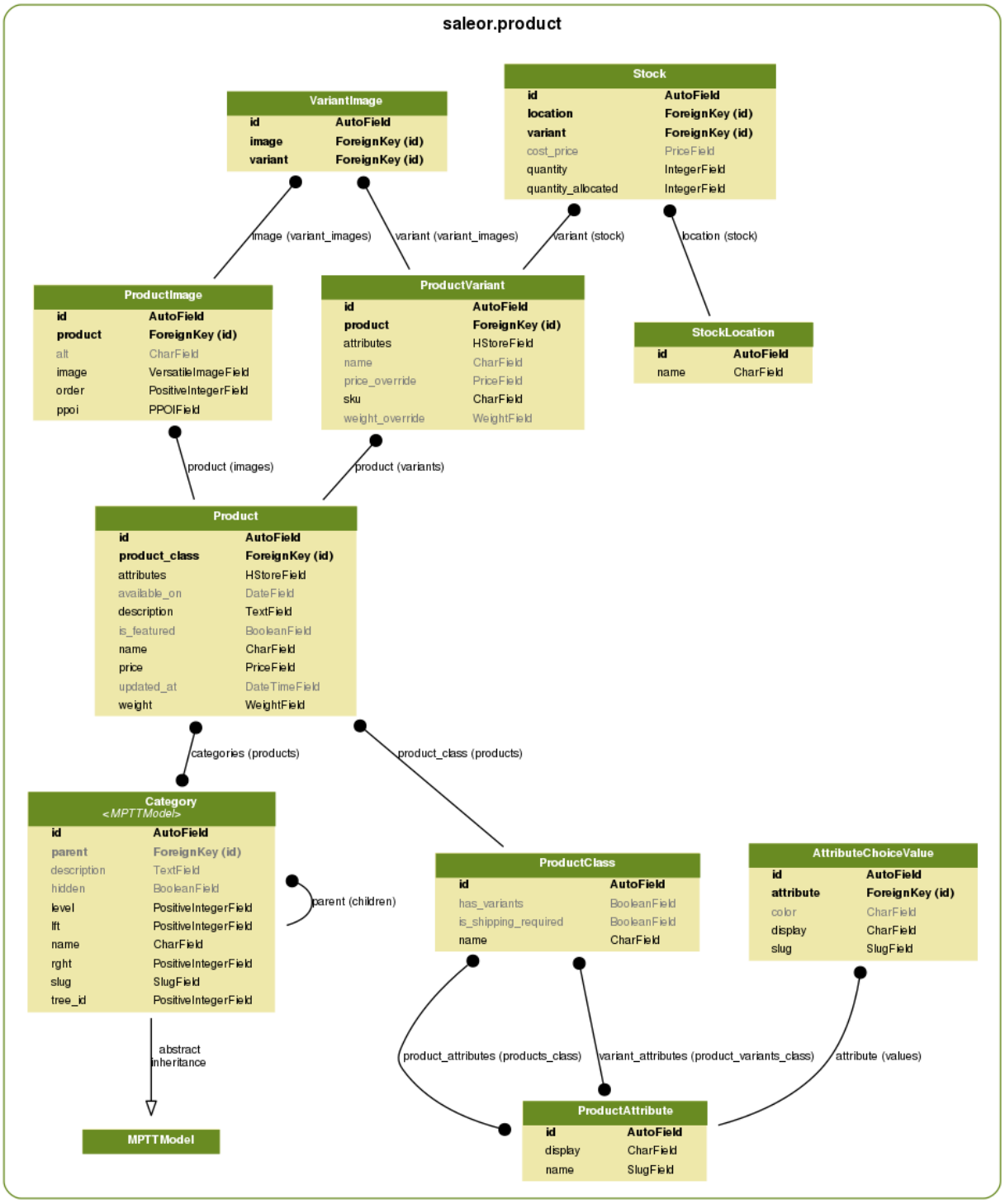
CHAPTER 3

Product management

Before filling your shop with products we need to introduce 3 product concepts - *product classes*, *products*, *product variants*.

Example: Book store - one of *products* would be “Introduction to Saleor”. The book is available in hard and soft cover, so there would be 2 *product variants*. Type of cover is only attribute which creates separate variants in our store, so we use *product class* named “Book” with activated variants and ‘Cover type’ variant attribute.

Class diagram



Product variant

It's most important object in shop. All cart and stock operations use variants. Even if your *product* don't have multiple variants, we create one under the hood.

Product

Describes common details of few *product variants*. When shop displays category view, items on the list are distinct *products*. If variant has no overrode property (example: price), default value is taken from *product*.

- **available_on** Until this date product is not listed in storefront and is unavailable for users.
- **is_featured** Featured products are displayed on front page.

Product class

Think about it as template for your *products*. Multiple *products* can use same *product class*. Note: In interface product class is called *Product Type*

- **product_attributes** Attributes shared with all *product variants*. Example: Publisher - all book variants are published by same company
- **variant_attributes** It's what distinguishes different *variants*. Example: Cover type - your book can be in hard or soft cover.
- **is_shipping_required** Mark as false for *products* which does not need shipping. Could be used for digital products.
- **has_variants** If your *product* has no different variants or if you want to create separate *product* for every one of them - turn this option off. Note: this option simplifies dashboard. There is always *variant* created under the hood.

Warning: Changing a *product class* affects all *products* of this class.

Warning: You can't remove *product class* if it has any *products*.

Attributes

Attributes can help you better describe your products. Also, the can be used to filter items in category views. There are 2 types of *attributes* - choice type and text type. If you don't provide choice values, then attribute is text type.

Examples

- *Choice type*: Every shirt you sell can be made in 3 colors. Then you create *attribute* Color with 3 choice values (for example 'Red', 'Green', 'Blue')
- *Text type*: Number of pages

Example - Coffee

Your shop sells Coffee from around the world. Customer can order 1kg, 500g and 250g packages. Orders are shipped by couriers.

Attributes

Name	*Values*
Country of origin	Brazil, Vietnam, Colombia, Indonesia
Package size	1kg, 500g, 250g

Product class

Name	Coffee
Product Attributes	Country of origin
Variant Attributes	Package size
Has variants	Yes
Is shipping required	Yes

Product

Product class	Coffee
Name	Best Java Coffee
Country of Origin attribute	Indonesia
Description	Best coffee found on Java island!

Variants

Package size attribute	*Price*
1kg	$20
500g	$12
250g	$7

Example - Online game item

You have great selection of online games items. Each item is unique, important details are included in description. Bought items are shipped directly to buyer account.

Attributes

Name	*Values*
Game	Kings Online, War MMO, Target Shooter
Max attack	—

Product class

Name	Game item
Product Attributes	Game, Max attack
Variant Attributes	None
Has variants	No
Is shipping required	No

Product

Product class	*Name*	*Price*	*Game attribute*	*Max attack attribute*	*Description*
Game item	Magic Fire Sword	$199	Kings Online	8000 damage	Unique sword for any fighter. Set your enemies in fire!
Game item	Rapid Pistol	$2500	Target Shooter	250 damage	Fastest pistol in whole game.

CHAPTER 4

Stock management

Each product variant has a stock keeping unit (SKU) and can have any number of stock records.

A stock record represents that variant's availability in a single location. Multiple stock records are often used to represent different warehouses, different fulfilment partners or separate shipments of the same product that were obtained at different prices.

Each stock record holds information about *quantity* at hand, quantity *allocated* for already placed orders and quantity *available*.

Example: There are five boxes of shoes in warehouse A. Three of them have already been sold to customers but were not yet dispatched for shipment. The stock records **quantity** is **5**, **quantity allocated** is **3** and **quantity available** is **2**.

Each stock records also has a *cost price* (the price that your store had to pay to obtain it).

Product availability

A variant is *in stock* if at least one of its stock records has unallocated quantity.

The highest quantity that can be ordered is the maximum available quantity in any of its stock records. It's *not* the sum of all quantities to allow each order line to be fulfiled by a single stock record.

Allocating stock for new orders

Once an order is placed, a stock record is selected to fulfil each order line. Default logic will select the stock record with the *lowest cost price* that holds enough stock. Quantity needed to fulfil the order line is immediately marked as *allocated*.

Example: A customer places an order for another box of shoes and warehouse A is selected to fulfil the order. The stock records **quantity** is **5**, **quantity allocated** is now **4** and **quantity available** becomes **1**.

Decreasing stock after shipment

Once a delivery group is marked as shipped, each stock record used to fulfil its lines will have both its quantity at hand and quantity allocated decreased by the number of items shipped.

Example: Two boxes of shoes from warehouse A are shipped to a customer. The stock records **quantity** is now **3**, **quantity allocated** becomes **2** and **quantity available** stays at **1**.

CHAPTER 5

Site settings

Site settings module allows your users to change common shop settings from dashboard like its name or domain. Settings object is chosen by pk from `SITE_SETTINGS_ID` variable.

Context processor

Thanks to `saleor.site.context_processors.settings` you can access *Site settings* in template with `settings` variable.

CHAPTER 6

Payments

Supported gateways

Saleor uses django-payments library to process payments.

Default configuration uses the *dummy* backend. It's meant to allow developers to easily simulate different payment results.

Here is a list of supported payment providers:

- Authorize.Net
- Braintree
- Coinbase
- Cybersource
- Dotpay
- Google Wallet
- PayPal
- Sage Pay
- Sofort.com
- Stripe

Please note that this list is only provided here for reference. Please consult django-payments documentation for an up to date list and instructions.

Note: All payment backends default to using sandbox mode. This is very useful for development but make sure you use production mode when deploying to a production server.

3-D Secure

3-D Secure is a card protection protocol that allows merchants to partially mitigate fraud responsibility. In practice it greatly lowers the probability of a chargeback.

Saleor supports 3-D Secure but whether it's used depends on the payment processor and the card being used.

Fraud protection

Some of the payment backends provide automatic fraud protection heuristics. If such information is available Saleor will show it in the order management panel.

Authorisation and capture

Some of the payment backends support pre-authorising payments. Please see django-payments documentation for details.

Authorisation and capture is a two step process.

First the funds are locked on the payer's account but are not transferred to your bank.

Then depending on the provider and card type you have between a few days and a month to charge the card for an amount not exceeding the authorised amount.

This is very useful when an exact price cannot be determined until after the order is prepared. It is also useful if your business prefers to manually screen orders for fraud attempts.

When viewing orders with pre-authorised payments Saleor will offer options to either capture or release the funds.

Refunds

You can issue partial or full refunds for all captured payments. When you edit an order and remove items Saleor will also offer to automatically issue a partial refund.

CHAPTER 7

Integrations

Static file and media storage using Amazon S3

If you're using containers for deployment (including Docker and Heroku) you'll want to avoid storing files in the container's volatile filesystem. This integration allows you to delegate storing such files to Amazon's S3 service.

Set the following environment variables to use S3 to store and serve media files:

`AWS_ACCESS_KEY_ID` Your AWS access key.

`AWS_SECRET_ACCESS_KEY` Your AWS secret access key.

`AWS_MEDIA_BUCKET_NAME` The S3 bucket name to use for media files.

By default static files (such as CSS and JS files required to display your pages) will be served by the application server.

If you intend to use S3 for your static files as well, set an additional environment variable:

`AWS_STORAGE_BUCKET_NAME` The S3 bucket name to use for static files.

Converting prices to local currencies using Open Exchange Rates

This integration will allow your customers to see product prices in their local currencies. Local prices are only provided as an estimate, customers are still charged in your store's default currency.

Before you begin you will need an Open Exchange Rates account. Unless you need to update the exchange rates multiple times a day the free Subscription Plan should be enough but do consider paying for the wonderful service that Open Exchange Rates provides. Start by signing up and creating an "App ID".

Export the following environment variable:

`OPENEXCHANGERATES_API_KEY` Your store's Open Exchange Rates "App ID."

To update the exchange rates run the following command at least once per day:

```
$ python manage.py update_exchange_rates --all
```

Note: Heroku users can use the Scheduler add-on to automatically call the command daily at a predefined time.

Reporting with Google Analytics

Because of EU law regulations, Saleor will not use any tracking cookies by default.

We do however support server-side Google Analytics out of the box using Google Analytics Measurement Protocol.

This is implemented using google-measurement-protocol and does not use cookies at the cost of not reporting things impossible to track server-side like geolocation and screen resolution.

To get it working you need to export the following environment variable:

GOOGLE_ANALYTICS_TRACKING_ID Your page's Google "Tracking ID."

Send product data to Google Merchant Center

Saleor has tools for generating product feed which can be used with Google Merchant Center. Final file is compressed CSV and saved in location specified by `saleor.data_feeds.google_merchant.FILE_PATH` variable.

To generate feed use command:

```
$ python manage.py update_feeds
```

It's recommended to run this command periodically.

Merchant Center has few country dependent settings, so please validate your feed at Google dashboard. You can also specify there your shipping cost, which is required feed parameter in many countries. More info be found at Google Support pages.

One of required by Google fields is *brand* attribute. Feed generator checks for it in variant attribute named *brand* or *publisher* (if not, checks in product).

Feed can be downloaded from url: `http://<yourserver>/feeds/google/`

Full text search with Elasticsearch

You can use optional Elasticsearch integration. To get it working you need to export following environment variable:

ELASTICSEARCH_URL URL to Elasticsearch server, for example: *"http://localhost:9200"*. Defaults to `None`

For more details see elasticsearch

CHAPTER 8

Development

Working with templates

Default storefront templates are based on Bootstrap 4.

You can find the files under `/templates/`.

Working with front-end assets

All static assets live in subdirectories of `/saleor/static/`.

Stylesheets are written in Sass and rely on postcss and autoprefixer for cross-browser compatibility.

JavaScript code is written according to the ES2015 standard and relies on Babel for transpilation and browser-specific polyfills.

Everything is compiled together using webpack module bundler.

The resulting files are written to `/saleor/static/assets/` and should not be edited manually.

During development it's very convenient to have webpack automatically track changes made locally. This will also enable *source maps* that are extremely helpful when debugging.

To run webpack in *watch* mode run:

```
$ yarn start
```

Warning: Files produced this way are not ready for production use. To prepare static assets for deployment run:

```
$ yarn run build-assets --production
```

Working with backend code

Python dependencies

To guarantee repeatable installations all project dependencies are managed using pip-tools. Project's direct dependencies are listed in `requirements.in` and running `pip-compile` generates `requirements.txt` that has all versions pinned.

We recommend you use this workflow and keep `requirements.txt` under version control to make sure all computers and environments run exactly the same code.

Running tests

Before you make any permanent changes in the code you should make sure they do not break existing functionality.

The project currently contains very little front-end code so the test suite only covers backend code.

To run backend tests use pytest:

```
$ py.test
```

You can also test against all supported versions of Django and Python. This is usually only required if you want to contribute your changes back to Saleor. To do so you can use Tox:

```
$ tox
```

Continuous integration

The storefront ships with a working CircleCI configuration file. To use it log into your CircleCI account and enable your repository.

Docker

Using Docker to build software allows you to run and test code without having to worry about external dependencies such as cache servers and databases.

> **Warning:** The following setup is only meant for local development. See *Docker* for production use of Docker.

You will need to install Docker and docker-compose before performing the following steps.

To build assets you will need node, yarn and webpack module bundler.

Note: Our configuration exposes PostgreSQL, Redis and Elasticsearch ports. If you have problems running this docker file because of port conflicts, you can remove 'ports' section from docker-compose.yml

---

1. Install JavaScript dependencies

```
$ yarn
```

2. Prepare static assets

```
$ yarn run build-assets
```

3. Build the containers using `docker-compose`

```
$ docker-compose build
```

4. Prepare the database

```
$ docker-compose run web python manage.py migrate
$ docker-compose run web python manage.py collectstatic
$ docker-compose run web python manage.py populatedb --createsuperuser
```

   The `--createsuperuser` switch creates an admin account for `admin@example.com` with the password set to `admin`.

5. Run the containers

```
$ docker-compose up
```

By default, the application is started in debug mode, will automatically reload code and is configured to listen on port `8000`.

CHAPTER 9

Deployment

Docker

You will need to install Docker first.

Before building the image make sure you have all of the front-end assets prepared for production:

```
$ yarn run build-assets --production
$ python manage.py collectstatic
```

Then use Docker to build the image:

```
$ docker build -t mystorefront .
```

Heroku

First steps

```
$ heroku create --buildpack https://github.com/heroku/heroku-buildpack-nodejs.git
$ heroku buildpacks:add https://github.com/heroku/heroku-buildpack-python.git
$ heroku addons:create heroku-postgresql
$ heroku addons:create heroku-redis
$ heroku config:set SECRET_KEY='<your secret key here>'
$ heroku config:set ALLOWED_HOSTS='<your hosts here>'
```

Note: Heroku's storage is volatile. This means that all instances of your application will have separate disks and will lose all changes made to the local disk each time the application is restarted. The best approach is to use cloud storage such as Amazon S3. See *Static file and media storage using Amazon S3* for configuration details.

Deploy

```
$ git push heroku master
```

Prepare the database

```
$ heroku run python manage.py migrate
```

CHAPTER 10

Indices and tables

- genindex
- modindex
- search