ecoseller

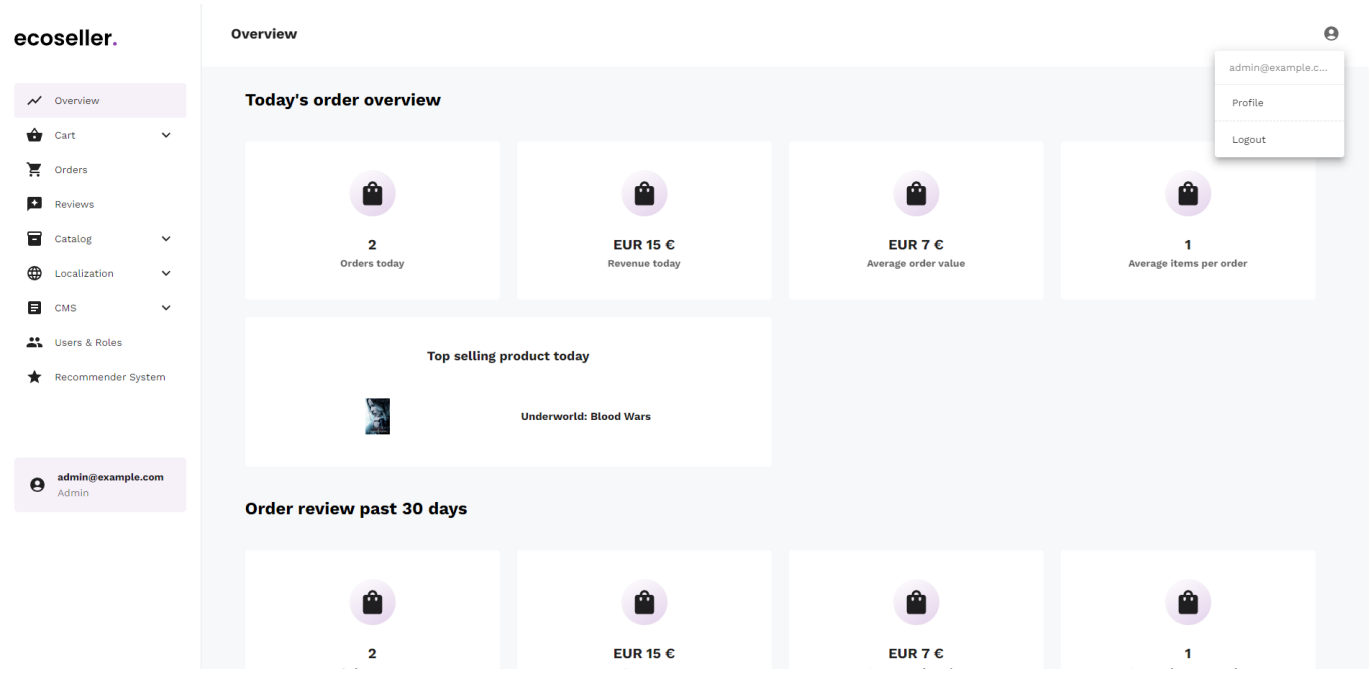
# Introduction

Unlock the future of e-commerce success with ecoseller – headless e-commerce solution that empowers your brand. ecoseller redefines the game, separating the captivating frontend from the dynamic backend, setting the stage for unparalleled flexibility and growth.

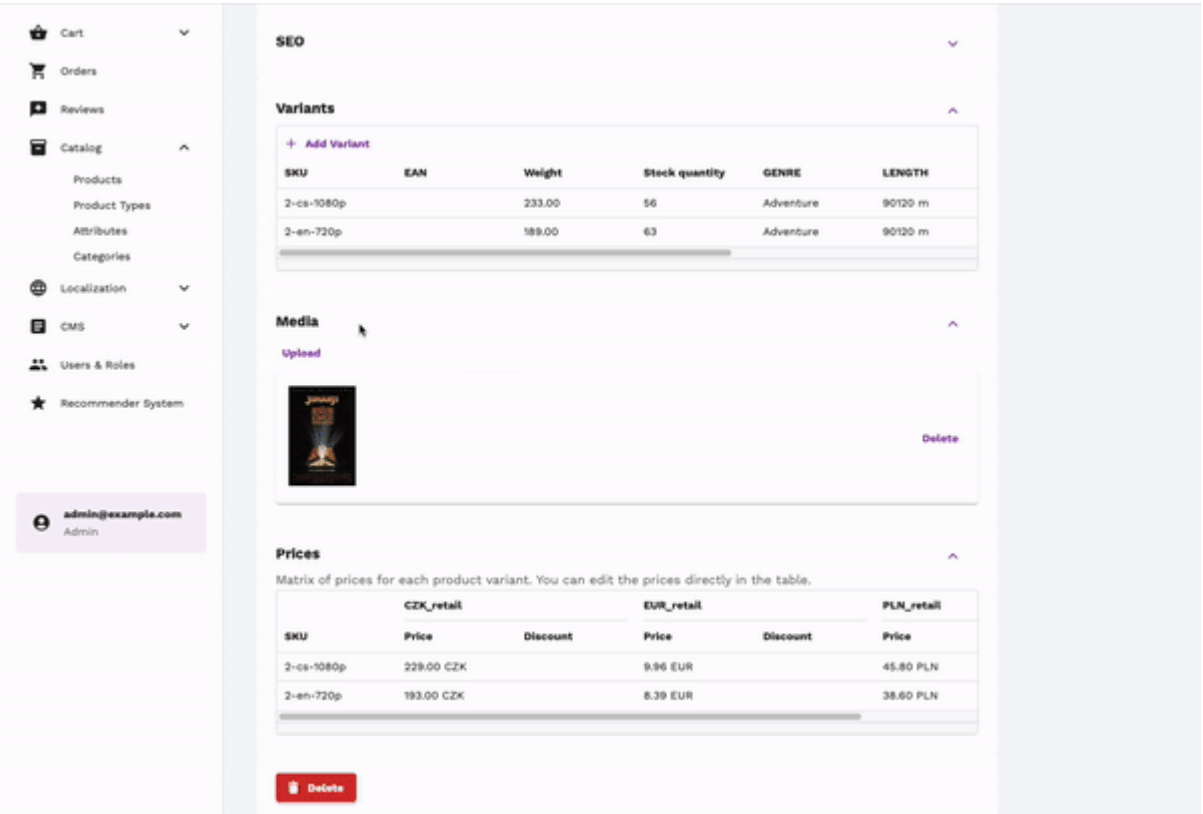
## ecoseller.io

Designed to cater to businesses of every scale, from the most daring sole proprietor to the middle size, ecoseller is your partner in success. Our platform's agility adapts to your ambitions, ensuring your digital presence resonates with impact.

ecoseller solution includes two user-facing interfaces: storefront and dashboard. Storefront serves as the online storefront for customers, while dashboard provides a comprehensive administration panel for managing the e-commerce platform.



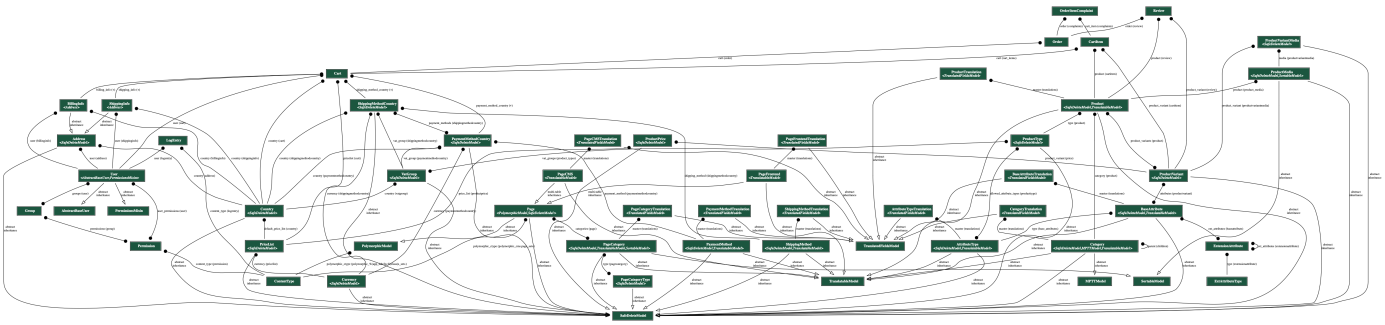

Powered by the technological prowess of Django and Next.JS, our dashboard is a symphony of innovation, delivering real-time insights and intuitive control. Feast your eyes on the future with our dashboard, a glimpse into the seamless control that fuels your journey to the top.



But that's just the beginning – we go beyond, embracing the art of AI recommendation to engage users and boosting sales like. ecoseller's AI recommendation system doesn't just sell, it captivates, creating personalized experiences that fuel engagement.

ecoseller aims to extensible platofrm, powered by our notification API. Seamlessly connect events, unlock the power to call external APIs – it's your gateway to endless possibilities, making ecoseller not just a platform but a universe of opportunities.

The data model, the dynamic canvas that lets you sell an infinite array of products with diverse attributes. Your product catalog is limitless, your customer's journey refined.



Elevate your sales, engage your audience, and embrace a future where innovation thrives – all with ecoseller by your side. Your e-commerce empire starts here, ready to take on the world?

So let's get started!

- Get started quickly and with ease.
- Get a deeper understanding of the platform and ways to extend it.
- Learn how to use ecoseller using our video tutorials

- if you prefer textual documentation, see our dashboard and storefront user docs

ecoseller

Administration documentation

# Installation

Table of contents:



## Prerequisites

Before proceeding with the installation of ecoseller, it is important to ensure that your machine meets the requirements. While ecoseller itself is not demanding and can run on less powerful devices, it is recommended to have a slightly more capable setup for the default installation, especially when including the Elasticsearch and AI recommendation system.

To run ecoseller with the AI recommendation system and Elasticsearch, we recommend using a machine with the following specifications:

- CPU: 8 cores or more
- RAM: 8GB or more
- Free Space: 10GB

It's worth noting that Elasticsearch itself requires a significant amount of memory to run efficiently, ideally around 4GB. Therefore, the suggested 8GB RAM allocation ensures smooth operation of both ecoseller and Elasticsearch.

However, if you are running ecoseller without the Elasticsearch and AI recommendation system or with a smaller dataset, you can use less powerful devices as well.

Based on our testing, we have observed that for a typical scenario with 2100 product variants and several thousand events, the training process for Level 3 recommendations requires a maximum of 1GB of RAM and completes within approximately 2 seconds. For Level 2 recommendations, the memory consumption is around 130MB, and the training process takes approximately 92 seconds.

By considering these prerequisites and performance benchmarks, you can ensure optimal performance and resource allocation for ecoseller and its AI recommendation system.

Since ecoseller is fully containerized, make sure your system is Docker-ready before proceeding with the installation. If you are new to Docker, you

can refer to the official Docker documentation for detailed instructions on installing Docker on your machine.

Also you **should have some experience** with Docker Compose, Django, Python and Next.js (React). If not - you can learn it from the official documentations. If you don't have time for that - you can hire us to do it for you (the way it was meant to without compromises). 😉

# Running ecoseller

ecoseller can be deployed in different environments depending on your needs, whether it's for development, production, or a demo environment. This section will guide you through the steps to run ecoseller in production environment.

1. You need to clone the ecoseller repository from the source code repository and navigate to the project directory in your terminal.

```
git clone https://github.com/ecoseller/ecoseller.git
```

1. Make sure `src/backend/docker-compose.env` from example exists.

2. Edit `src/backend/docker-compose.env` . Please make sure `DEBUG` flag is set to `0` and `DJANGO_ALLOWED_HOSTS` is set to your domain name in this file.

3. Make sure `src/recommender_system/docker-compose.env` from example exists.

4. If you are going to change the default ports or will use server name based routing as described in Configuration section, please make sure that you

change public URLs of backend for storefront and dashboard in `src/docker-compose.prod.yaml` file ( `NEXT_PUBLIC_API_URL` variable).

5. Navigate to `src` folder and run the following command to start ecoseller in production mode:

   ```
   docker compose -f docker-compose.prod.yaml up
   ```

   This command will start all the containers and services required for ecoseller to run. Please note that the first time you run this command, it will take some time to download the required images and build the containers. Are you having trouble with the installation? Please see the Troubleshooting section.

6. After the containers are up and running, in the default settings you can access the storefront at `http://localhost:3032` and the dashboard at `http://localhost:3033` . For advanced settings of Nginx reverse proxy please see Configuration section.

7. For initial dashboard login use the following credentials:
   - email: `admin@example.com`
   - password: `admin` Make sure you delete the default admin user and create a new one with a strong password or change the password of the default admin user.

---

# Configuration of production environment

Nginx is used as a reverse proxy to efficiently handle incoming requests and distribute them to the appropriate services. It acts as a middle layer between

the client and the backend services, enhancing performance and enabling load balancing. Two configurations are supplied for the Nginx reverse proxy:

- Simple Port Mapping: In this configuration, Nginx maps incoming requests directly to the appropriate backend service based on port numbers.
- - Please see `/src/reverse_proxy/nginx.conf`
- Server Name Based Routing: This configuration allows Nginx to utilize different server names to route requests to the corresponding backend services.
- - - Please see `/src/reverse_proxy/nginx.example.conf` If you wish to use the Server Name Based Routing configuration, you can rename the `nginx.example.conf` file to `nginx.conf` and modify it as needed or create your own configuration file (according to Nginx standards) and mount it to the Nginx container in the `docker-compose.prod.yaml` file as shown below:

```
  reverse-proxy:
    container_name: reverse_proxy
    image: nginx:latest
    ports:
      - 80:80
      - 443:443 # if you want to use https
      - 8080:8080 # remove this line if you don't want to expose services
directly
      - 3032:3032 # remove this line if you don't want to expose services
directly
      - 3033:3033 # remove this line if you don't want to expose services
directly
    volumes:
      - ./reverse_proxy/your_nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - backend
```

```
      - frontend_storefront
      - frontend_dashboard
```

Please note that if you are using the Server Name Base Routing configuration, it's good practice to remove port mapping from the Nginx container in the `docker-compose.prod.yaml` file.

# Development environment

1. To run ecoseller in a development environment, make sure `src/backend/docker-compose.env` `DEBUG` flag is set to `1` and `DJANGO_ALLOWED_HOSTS` is set to `"*"` in this file.
    - Run the following command to start ecoseller in development mode: `docker compose up .`

This command will start all the containers and services required for ecoseller to run. Please note that the first time you run this command, it will take some time to download the required images and build the containers.

**Important** Also note that both storefront and dashboard are quite slow in the developement mode since they are running in the debug mode and Next.js rebuilds every single page on every single request.

---

## Demo environment

The demo environment in Ecoseller is designed to showcase the platform's features and functionality using preloaded demo products (1400+), variants (2100+), and additional data. Setting up the demo environment is similar to the production environment, with the main difference being the utilization

of the `docker-compose.demo.yaml` file. It's very similar to `docker-compose.prod.yaml` but it has some additional services that are used to preload the demo data into the database. You can choose between using reverse proxy or accessing the services directly. However, it's recommended to use the reverse proxy configuration for the demo environment as well (please see the Reverse Proxy section for more information as well as Production environment). Our demo data live at public repository ecoseller/demo-data

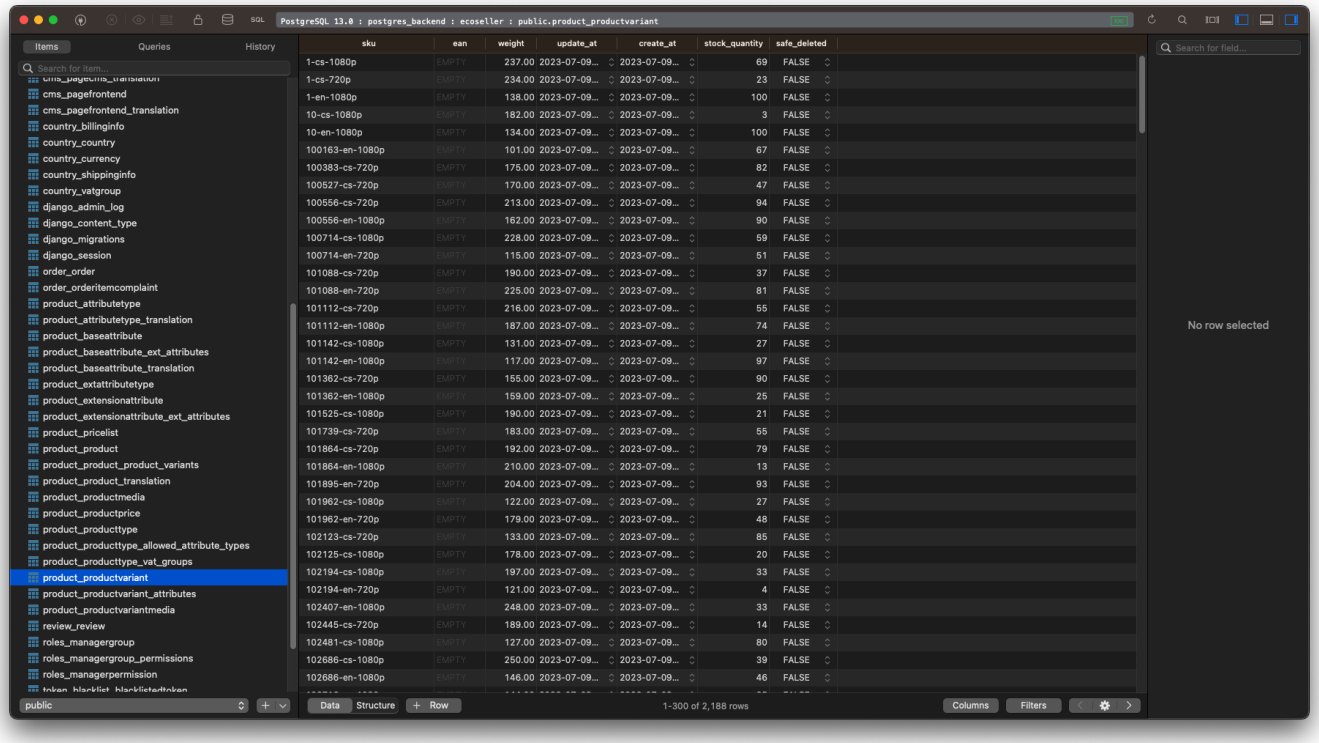
The demo environment can be started by running the following command in the `/src` directory:

```
docker compose -f docker-compose.demo.yaml up -d
```

If you compare `docker-compose.demo.yaml` and `docker-compose.prod.yaml`, you can see that main difference is in the build target of `backend` service where `demo` is utilized. It means that there're different scripts ran on startup, namely `/src/backend/demo_data_loader.sh` which clones the repository and moves the data in propriate locations.

```
git clone https://github.com/ecoseller/demo-data.git
mv /usr/src/demo-data/media /usr/src/mediafiles
PGPASSWORD=$POSTGRES_PASSWORD psql -h $POSTGRES_HOST -U $POSTGRES_USER -d
$POSTGRES_DB -p $POSTGRES_PORT -a -f /usr/src/demo-data/sql/mock_data.sql
```

**Also, please note, that the demo environment is not intended for production use. It's not setup for persistent storage so after you stop the containers all the data will be lost.**




# Troubleshooting

If the installation failed because of timeout, please firstly check that your PC meets the previously mentioned requirements

Especially, if you're using Docker on Windows, please check that your WSL resource limits meet the requirements.
If you need to update your WSL resource limits, you can follow this guide.

If you're meeting the requirements, please try to to run the startup command again. If it fails again, please try to run it with `--build` flag.

# Environment variables

ecoseller utilizes environment variables to configure various aspects of the backend and recommendation system. These environment variables are

stored in separate files, namely `docker-compose.env`. For the backend it's `src/backend/docker-compose.env` and `src/recommender_system/docker-compose.env` for recommendation system. Additionally, the storefront, dashboard, and other services have their environment variables directly specified in the YAML file for specific docker compose.

# Backend

This is an example of `src/backend/docker-compose.env` file. You can use it as a template for your own configuration. Please note that in this file you can configure Django backend and all the connections to other services that are used by the backend. You can find more information about Django environment variables in the official documentation.

---

## Example

```
DEBUG=1 # 1 for development, 0 for production
DJANGO_ALLOWED_HOSTS="*" # for development only

DATABASE=postgres
DB_ENGINE=django.db.backends.postgresql_psycopg2
POSTGRES_DB=ecoseller
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
POSTGRES_HOST=postgres_backend
POSTGRES_PORT=5432

USING_REDIS_QUEUE=1
REDIS_QUEUE_LOCATION=redis

PYTHONUNBUFFERED=1

RS_URL="http://recommender_system:8086"
```

```
STOREFRONT_URL="https://www.example.com"

NOTIFICATIONS_CONFIG_PATH="./config/notifications.json"

EMAIL_USE_SSL=1
EMAIL_PORT=465
EMAIL_HOST=smtp.example.com
EMAIL_HOST_USER=ecoseller@example.com
EMAIL_HOST_PASSWORD="yourpassword
EMAIL_FROM=Storefront<ecoseller@example.com>

USE_ELASTIC=1
ELASTIC_HOST="elasticsearch:9200"
ELASTIC_AUTO_REBUILD_INDEX=0
```

# Recommendation system

This is an example of `src/recommender_system/docker-compose.env` file. You can use it as a template for your own configuration, but please note that you need to change the `RS_URL` variable to match the URL in your ecoseller backend.

---

## Example

```
RS_SERVER_HOST=0.0.0.0
RS_SERVER_PORT=8086
RS_SERVER_DEBUG=TRUE

POSTGRES_PASSWORD=zZvyAvzG2O5gfr5

RS_PRODUCT_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/products
RS_FEEDBACK_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/feedback
RS_SIMILARITY_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/similar
RS_MODEL_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/model
```

# Reserved ports

When running ecoseller, it is important to be aware of the reserved ports used by the various services within the platform. Reserved ports ensure that different components of ecoseller can communicate with each other effectively through Docker internal network. Here are the reserved ports used in ecoseller:

## Production environment

By default, ecoseller exposes the following ports in production environment:

- **Backend Service**:
- Port 8080, however, if you're using serven name based routing, it's should not be exposed.
- **Storefront**:
- Port 3032, however, if you're using serven name based routing, it's should not be exposed.
- **Dashboard**:
- Port 3033, however, if you're using serven name based routing, it's should not be exposed.

All other services are not exposed to the outside world and are only accessible through the internal Docker network.

## Development environment

Has all services exposed to the outside world for easier development and debugging.

- **Backend Service**:
- The Ecoseller backend service runs on port 8000 by default. This is the primary entry point for accessing the backend APIs.

- **PostgreSQL Backend**:
- The Ecoseller backend PostgreSQL service is accessible on port 5433 by default. This is the primary entry point for accessing the backend PostgreSQL database.

- **Backend Redis**:
- The Ecoseller backend Redis service is accessible on port 6379 by default. This is the primary entry point for accessing the backend Redis database.

- **Elasticsearch**:
- The Elasticsearch service is accessible on port 9200 by default. This is the primary entry point for accessing the Elasticsearch database.

- **Dashboard**:
- The dashboard service is accessible on port 3030 by default. This is the primary entry point for accessing the dashboard.

- **Storefront**:
- The storefront service is accessible on port 3000 by default. This is the primary entry point for accessing the storefront.

- **Recommendation system**:
- The recommendation system service is accessible on port 8086 by default. This is the primary entry point for accessing the recommendation system.

- **PostgreSQL Recommender system**:
- The recommender PostgreSQL service is accessible on port 5432 by default. This is the primary entry point for accessing the recommender PostgreSQL database.

ecoseller

Administration documentation

# Administration

Table of contents:



# Working with ecoseller REST API

The ecosellerplatform provides a comprehensive and powerful REST API that allows developers to interact with and extend the functionality of the e-commerce platform. This section of the documentation focuses on working with the ecosellerREST API and provides detailed guidance on utilizing its endpoints and authentication mechanisms. Please note that the ecosellerREST API was designed to be used primarily for dashboard purposes and is not intended to be used as a public API for the ecosellerplatform. However, feel free to use it as you see fit. On the other hand, please consider using `NotificationAPI` for public API purposes and calling external services directly from ecosellerbackend.

# Authentication

Ecoseller's REST API authentication relies on JSON Web Tokens (JWT) to secure and authorize API requests. JWT is a compact and self-contained token format that securely transmits information between parties using digitally signed tokens. In the context of ecoseller, JWTs are utilized to authenticate and authorize API access intended for dashboard.

---

## Obtaining a JWT

To obtain a JWT, you need to send a POST request to the `/user/login/` endpoint with the following payload:

```
{
    "email": "your_email",
    "password": "your_password",
    "dashboard_login": true
}
```

If the provided credentials are valid, the API will return a response containing the JWT token:

```
{
    "access": "your_access_token",
    "refresh": "your_refresh_token"
}
```

The `access` token is used to authenticate API requests, while the `refresh` token is used to obtain a new access token once the current one expires. The access token is valid for 5 minutes, while the refresh token is valid for 24 hours. To obtain a new access token, you need to send a POST request to the `/user/refresh-token/` endpoint with the following payload:

```
{
    "refresh": "your_refresh_token"
}
```

If the provided refresh token is valid, the API will return a response containing the new access token:

```
{
    "access": "your_new_access_token"
}
```

---

# Using a JWT

Once you obtain a JWT, you need to include it in the `Authorization` header of your API requests. The header should have the following format:

```
Authorization: JWT your_access_token
```

## API documentation

The ecoseller backend provides a comprehensive API documentation that can be accessed by navigating to the `/api/docs/` endpoint. This documentation is generated automatically using the drf-yasg package and provides detailed information about the available endpoints, their parameters, and the expected responses. Please make sure to use primairly `/dashboard` endpoints since they're designed to to modify data and require authentication. Storefront endpoints don't.

# User management

The user management functionality in **ecoseller's** dashboard allows administrators to create and manage user accounts with various roles and permissions. This section of the administration documentation focuses on the process of creating a initial user in ecoseller using the Django Command-Line Interface (CLI).

A admin is a special type of user account which has access to all administrative functions and controls within the ecoseller platform. Creating an initial admin user is an essential step in setting up your ecoseller administration panel, as it provides you with all privileges to manage and configure your e-commerce platform. All other admin users can be allways setup in the dashboard, but you need at least one user to be able to login to the dashboard. For more information about roles, permissions and users in general, please refer to Authorization section in admin. documentation.

## Creating an initial admin user without dashboard

Creating an admin user without dashboard can be done through the Django CLI. To do so, run the following command:

```
python3 manage.py createsuperuser
```

# Managing database

ecoseller utilizes a PostgreSQL database to store and manage data. This section of the documentation focuses on managing a PostgreSQL database within a Docker container and connecting it to a Django and Recommender system application.

## Django <-> PostgreSQL connection

The Django application is configured to connect to a PostgreSQL database using the following environment variables:

```
POSTGRES_DB=ecoseller
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
POSTGRES_HOST=postgres_backend
POSTGRES_PORT=5432
```

## Recommender system <-> PostgreSQL connection

The Recommender system's application is configured to connect to several PostgreSQL databases using the following environment variables:

```
RS_PRODUCT_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/products
RS_FEEDBACK_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/feedback
RS_SIMILARITY_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/similar
RS_MODEL_DB_URL=postgresql://postgres:zZvyAvzG2O5gfr5@postgres_rs:5432/model
```

where each environment variable defines a connection string to one database.

# Binding database to a local folder

To persist the data in the PostgreSQL container, you can bind a local folder on your host machine to the container's data directory using Docker Compose. In your docker-compose.yml file, add the following volume configuration under the services section for the `postgres_backend` and `postgres_rs` container:

```
volumes:
    - ./backend/postgres/data:/var/lib/postgresql/data/
```

This configuration ensures that the PostgreSQL data is stored in the `./src/backend/postgres` folder on your local machine.

# Running migrations

It shouldn't be necessary to run migrations manually, but if you need to do so, you can run the following commands in the `backend` container:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

or the following commands in the `recommender_system` container:

```
python3 -m recommender_system.scripts.makemigrations {storage_name}
python3 -m recommender_system.scripts.migrate
```

where `{storage_name}` is one of `feedback_storage`, `model_storage`, `product_storage` or `similarity_storage`. This also creates migrations if there is a change in one of the storages. For more information about the Recommender system's storages, see storages section of the Recommender system page in the programming documentation.

# Backing up database

It is crucial to regularly back up your PostgreSQL database to prevent data loss and ensure data integrity. Use the pg_dump utility to create a backup of the database. Run the following command to back up the database to a file:

```
pg_dump -U your_username -d your_database_name -f /path/to/backup.sql
```

Replace `your_username` and `your_database_name` with the appropriate values. Specify the path where you want to save the backup file.

# Restoring database

To restore a PostgreSQL database from a backup file, use the pg_restore utility. Run the following command to restore the database from a backup file:

```
pg_restore -U your_username -d your_database_name /path/to/backup.sql
```

Replace `your_username`, `your_database_name`, and `/path/to/backup.sql` with the appropriate values.

# Indexing products to Elasticsearch

The process of indexing products is described in programming docs
You can also watch our video tutorial.

ecoseller

Administration documentation

# Localization

Table of contents:



Ecoseller is designed to be a versatile and comprehensive e-commerce platform that caters to a global audience. With its aim to support multi-country operations, Ecoseller provides extensive localization capabilities across all aspects of user communication. This section of the administration documentation focuses on the localization features and configuration options available within Ecoseller.

However, to understand the localization capabilities of Ecoseller, it is important to first understand the concept of locales and how they are used within the platform.

We've chosen "country first" approach. Which means, that our main localize unit is a country. For example, if you want to have a store in the US and in the UK - you will have to create two countries, but mostlikely with the same language (English). But they will differ in currency, VAT groups, shipping

methods and most likely even in the price list for products (since you might have different prices for stocking, packaging and marketing in different countries).

# Languages

Since languages are loaded on the startup of the backend and storefront, when editing them, it's neccessary to dive into the code a little bit.

---

## Backend

Languages are loaded in the `src/backend/core/core/settings.py` file under the `PARLER_LANGUAGES` variable. If you want to add a new language, you will have to add a new entry to this variable, under the `None` key. If you want to set different language as a default, you will have to change the `PARLER_DEFAULT_LANGUAGE_CODE` variable in the same file as well as `LANGUAGE_CODE`.

```
LANGUAGE_CODE = "en"
PARLER_DEFAULT_LANGUAGE_CODE = "en"
PARLER_LANGUAGES = {
    None: (
        {
            "code": "en",
        },
        {
            "code": "cs",
        },
    ),
    "default": {
        "fallbacks": ["en"],  # defaults to PARLER_DEFAULT_LANGUAGE_CODE
        "hide_untranslated": False,  # the default; let .active_translations()
return fallbacks too.
```

```
    },
}
```

Backend languages are used for database translation - so for all data stored in the database, like product names, descriptions, categories, etc. and for the e-mails sent from the backend.

If you want to edit translation of the e-mail templates, you will have to go to the container `backend` and run the following command:

```
python3 manage.py makemessages -l en -l cs -l other_language ...
```

After that, Django will create or append to the `locale` folder in the `backend` container. There you will find a folder for each language you've specified in the command above. In each folder, there will be a file called `django.po`. This file contains all the strings that are used in the backend and are marked for translation. You can edit the strings in this file and then run the following command to compile the changes:

```
python3 manage.py compilemessages
```

Editing data in the database is a little bit more complicated and is fully handled by ecoseller dashboard. Everything is described in the user documentation.

---

# Storefront

Since storefront is a Next.js application, for the localization, we use next-i18next package. It's a wrapper around i18next package, which is a very

popular localization package for JavaScript.

Languages are loaded in the `src/storefront/next-i18next.config.js` file under the `i18n.locales` variable (and `i18n.defaultLocale`).

Translations are loaded in the `src/storefront/public/locales` folder. Each language has its own folder and in each folder, there are multiple JSON files representing all i18n namespaces that can be found in the project. This file contains all the strings that are used in the storefront and are marked for translation. You can edit the strings in this file. After you're done, you need to rebuild your container so that changes are applied.

If you add new translations, then run

```
npm run translate
```

to extend the `src/storefront/public/locales` folder with new namespaces and strings.

# Country

The country is the main localization unit in the ecoseller. It's used to define the following:

- Currency
- VAT groups
- locale
- shipping methods

Countries are stored in the database and can be edited in the ecoseller dashboard. Everything is described in the localization section of the user documentation.

# Currency

Currency should allow you to make user more comfortable and feel like they're shopping at their local store. Currency is binded to the country, so you can have different currencies for different countries.

Currencies are also stored in the database and you can edit them in the ecoseller dashboard. Everything is described in the localization section of the user documentation.

# VAT groups

VAT groups are used to define different VAT rates for different products and countries. With this feature, you can have different VAT rates for different countries as well as different VAT rates for different products (usually there's a standard rate and reduced rate).

VAT groups are also stored in the database and you can edit them in the ecoseller dashboard. Everything is described in the localization section of the user documentation.

ecoseller

Administration documentation

# Authorization

Table of contents:



To achieve better security, ecoseller introduces so-called *Roles*. Each role has a set of permissions. Permissions are used to restrict access to certain parts of the application. For future use, ecoseller creates a new permission (more precisely four for each action - add/change/view/delete) for each model. At this moment, ecoseller is actively using the following permissions:

```
- add role                | group_add_permission
- change role             | group_change_permission
- can change cart         | cart_change_permission
- add category            | category_add_permission
- change category         | category_change_permission
- add page (cms)          | page_add_permission
- change page (cms)       | page_change_permission
- change product price    | productprice_change_permission
- add product price       | productprice_add_permission
- add product media       | productmedia_add_permission
- change product media    | productmedia_change_permission
- add product type        | producttype_add_permission
- change product type     | producttype_change_permission
- add product             | product_add_permission
- change product          | product_change_permission
```

```
- add user                  | user_add_permission
- change user               | user_change_permission
- add price list            | pricelist_add_permission
- change price list         | pricelist_change_permission
- add attribute type        | attributetype_add_permission
- change attribute type     | attributetype_change_permission
- add base attribute        | baseattribute_add_permission
- change base attribute     | baseattribute_change_permission
```

Ecoseller also comes with three predefined roles with the following permissions:

```
Editor permissions:
    -can change cart
    -can change product price
    -can add product price
    -can change product media
    -can add product media
    -can change product
    -can add product
    -can change category
    -can add category
    -can change page (cms)
    -can add page (cms)
    -can add product type
    -can change product type
    -can add price list
    -can change price list
    -can add attribute type
    -can change attribute type
    -can add base attribute
    -can change base attribute

UserManager permissions:
    -can add user
    -can change user
    -can add group
    -can change group

Copywriter permissions:
    -can change page (cms)
```

```
    -can change product
    -can change product media
    -can change category
```

Ecoseller also comes with a predefined admin role that has all permissions. To allow users to specify new roles on deployment, we created a special config defined in `backend/core/roles/config/roles.json` (with predefined roles mentioned above) file with the following structure:

```
[
    {
        "Role1" : {
            "permissions" : [
                {
                    "name" : "Permission1",
                    "description" : "Permission1 description"
                    "type" : "ADD",
                    "model" : "Model1"
                },
                {
                    "name" : "Permission2",
                    "description" : "Permission2 description",
                    "type" : "CHANGE",
                    "model" : "Model1"
                },
                {
                    "name" : "Permission3",
                    "description" : "Permission3 description",
                    "type" : "DELETE",
                    "model" : "Model2"
                }
            ],
            "decription" : "Description of Role1",
            "name" : "Role1Name"
        }
    },
    {
        "Role2" :{
                "permissions" : [
                    {
                        "name" : "Permission4",
```

```
                    "description" : "Permission4 description",
                    "type" : "VIEW",
                    "model" : "Model1"
                },
                {
                    "name" : "Permission5",
                    "description" : "Permission5 description"
                    "type" : "DELETE",
                    "model" : "Model3"
                },
                {
                    "name" : "Permission6",
                    "description" : "Permission6 description",
                    "type" : "CHANGE",
                    "model" : "Model3"
                },
                {
                    "name" : "Permission1",
                    "description" : "Permission1 description"
                    "type" : "ADD",
                    "model" : "Model1"
                }
            ]
            "description" : "Description of Role2",
            "name" : "Role2Name"
        }
    }
]
```

Each role has a unique name, description and a list of permissions. Each permission has a unique name, description, type and model. The type can be one of the following: `ADD`, `CHANGE`, `VIEW`, `DELETE`. The model is the name of the model to which the permission is assigned. The name of the model is the same as the name of the model in the database. For example, the name of the model for the `Product` model is `product`. The name of the model for the `ProductPrice` model is `productprice`. Each role and its corresponding permissions are created once the system is deployed during a migration process.

A description of the whole workflow with permissions (e.g. creating a new role, assigning it to specific users) can be found in the Users & Roles section.

To ensure maximum security, the way roles affect workflow differs between the frontend and the backend. The following sections describe how.

---

# Backend

The backend uses decorators to restrict access to certain endpoints. Each endpoint has a corresponding decorator that checks if the user has the required permission. If the user does not have the required permission, the decorator returns a `403` response. More information about decorators can be found in the Programming documentation.

---

# Frontend

The frontend uses context providers to hide/show or enable/disable certain components based on the user's permissions.Some components have a corresponding permission that is checked before the component is rendered. If the user does not have the required permission, the component is not rendered or is disabled. More information about context providers can be found in the Programming documentation.

## Hiding components

In the dashboard, we have a sidebar with links to various parts of the system. Some links have a corresponding permission that is checked before the link

is rendered. If the user does not have the required permission, the link is not rendered. Currently restricted links and their corresponding permissions are:

```
Products
    - product_change_permission
    - product_add_permission
Categories
    - category_change_permission
    - category_add_permission
CMS
    - page_change_permission
    - page_add_permission
Users & Roles
    - user_change_permission
    - user_add_permission
    - group_change_permission
    - group_add_permission
```

## Disabling components

We restricted disabling to specific actions - e.g. button click or dragging component. Some components have corresponding permissions that are checked before the component itself is enabled. If the user does not have the required permission, the components are disabled, preventing the user to trigger further action. Some of the currently restricted actions and their corresponding permissions are:

```
Adding new category
    - category_add_permission
Adding/Dragging media components
    - productmedia_change_permission
Creating role
    - group_add_permission
```

```
    Editing user general information
        - user_change_permission
```

ecoseller

Programming documentation

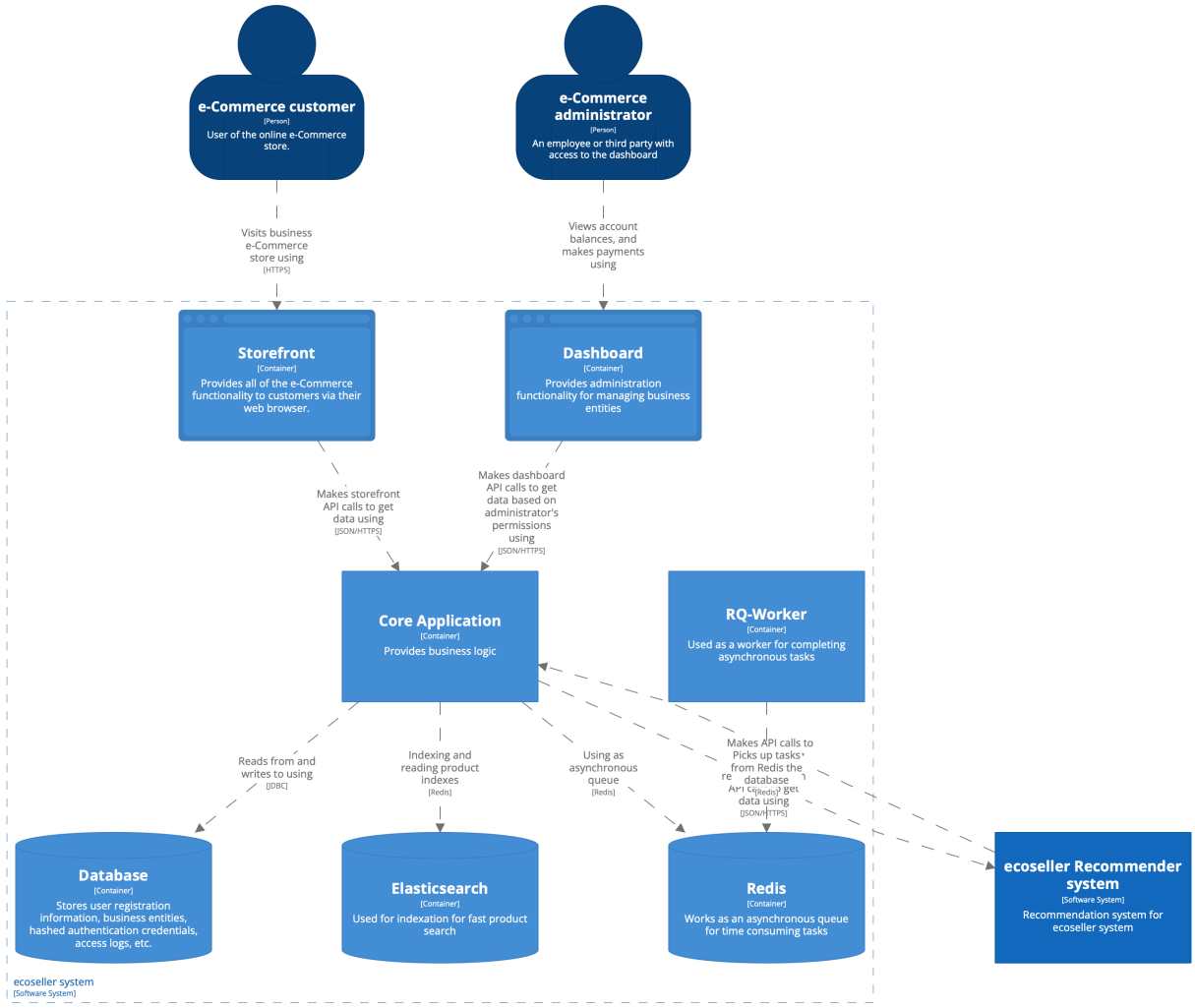
# Technical design

Table of contents:



The technical design of the **ecoseller** system combines a robust backend written in Django, a PostgreSQL database for efficient data storage, and user-facing interfaces built with Next.js for the storefront and dashboard. Elasticsearch enhances product search capabilities, while Redis enables asynchronous task processing via RQ-Worker. Additionally, the Flask-based recommender system provides personalized product recommendations. This integrated architecture ensures a performant, and programmer-friendly e-commerce platform written in Python and TypeScript.

# Architecture & Design




[Container] ecoseller system
neděle 16. července 2023 14:39 Středoevropský letní čas

The Ecoseller system is built upon an architecture incorporating various services and technologies to deliver a powerful e-commerce platform. This section provides an overview of the technical design of the **ecoseller** system, highlighting the key components and their interactions.

# Backend written in Django:

The core of the **ecoseller** system is the backend, developed using the Django Rest Framework. Django provides a solid foundation for building web applications and offers a range of features such as user management, data modeling, and API development. The backend handles crucial functionalities like product management, order processing, user authentication, and more.

We can divide the backend part into several subsections based on the area it handles.

---

## Core Application

Core part of the application is implemented in Python using the following technologies:

- `Django Rest Framework` – open source Python web framework
- `Redis` – open source data store that will help us with back-end task queuing
- `Elasticsearch` – search engine for storing and searching product data More technical details about the Core application can be found in the backend documentation.

# PostgreSQL Database:

**ecoseller** utilizes a PostgreSQL database to store and manage data efficiently. PostgreSQL is a reliable and feature-rich open-source database that ensures data integrity, scalability, and performance for the platform. It handles critical data related to products, orders, user information, and various other entities within the system. The only system accessing this database is the backend described above.

# Storefront and Dashboard written in Next.js:

The **ecoseller** platform includes two user-facing interfaces: the storefront and the dashboard. Both are developed using Next.js, a powerful React framework. Next.js enables the creation of dynamic, high-performance web applications with server-side rendering and optimized client-side navigation. The storefront serves as the online storefront for customers, while the dashboard provides a comprehensive administration panel for managing the e-commerce platform. Both applications are written in `TypeScript` and technical details can be found in the storefront & dashboard documentation.

# Elasticsearch for Fast Product Search:

**ecoseller** integrates Elasticsearch, a powerful search and analytics engine, to enhance the speed and accuracy of product searches. Elasticsearch enables efficient indexing, querying, and filtering of product data, ensuring a seamless and responsive search experience for users. The integration with Django allows for easy synchronization of product data between the backend and Elasticsearch. We use custom settings of Elasticsearch to improve the search experience. More on that in the section dedicated to supportive services.

# Redis for Asynchronous Tasks via RQ-Worker:

**ecoseller** utilizes Redis, an in-memory data structure store, to support asynchronous task processing. The RQ (Redis Queue) library leverages Redis to manage and distribute tasks across workers. The RQ-Worker, an instance of the Ecoseller backend, processes tasks from the Redis queue, enabling efficient handling of background processes and time-consuming operations. More on that in the section dedicated to supportive services.

# Recommender System written in Flask:

**ecoseller** incorporates a recommender system to provide personalized product recommendations to users. The recommender system is developed using Flask, a lightweight Python web framework. It leverages user behavior and preferences to generate relevant recommendations, enhancing the user experience and driving engagement. The recommender system is implemented in Python, the following technologies are used as well:

- `Flask` – open source Python web framework
- `NumPy` – open source Python library used to work with vectors and matrices
- `PyTorch` – open source Python library used for machine learning

The technical design of the **ecoseller** system seamlessly integrates these components, ensuring efficient data management, reliable operations, and a delightful user experience. By combining the power of Django, PostgreSQL, Next.js, Elasticsearch, Redis, and Flask, Ecoseller delivers a feature-rich and scalable e-commerce platform for businesses of all sizes with modern technologies.

# Version control

As a version control system, we use `git` . More specifically `GitHub`.

# Coding style

We use several tools for enforcing our code style. In both the dashboard and the storefront, we use:

- `Prettier` – an opinionated code formatter with support for many languages, including `JavaScript` and `TypeScript`
- `ESLint` – a static analysis tool identifying problematic patterns found in `JavaScript` and `TypeScript` code

Similarly, in `Core` and `Recommender system` components (which are written in `Python`) we use `black` code formatter and `flake8` linter. This way, we ensure consistent formatting of our code and avoid common bugs, which can be found by static analysis tools. We further use these tools in our Continuous integration setup, as described in Contribution - Continuous integration section.

`Recommender system` contains unit tests to ensure correct functionality when merging task branches to `master`.

ecoseller

Programming documentation

# Backend

Table of contents:

    - PRODUCTVARIANT_UPDATE
    - PRODUCTVARIANT_DELETE
- ProductPrice
    - PRICE_SAVE
    - PRICE_UPDATE
    - PRICE_DELETE
- ProductType
    - PRODUCTTYPE_SAVE
    - PRODUCTTYPE_UPDATE
    - PRODUCTTYPE_DELETE
- AttributeType
    - ATTRIBUTETYPE_SAVE
    - ATTRIBUTETYPE_UPDATE
    - ATTRIBUTETYPE_DELETE
- BaseAttribute
    - ATTRIBUTE_SAVE
    - ATTRIBUTE_UPDATE
    - ATTRIBUTE_DELETE
- Category
    - CATEGORY_SAVE
    - CATEGORY_UPDATE
    - CATEGORY_DELETE
- Order
    - ORDER_SAVE
    - ORDER_UPDATE
    - ORDER_DELETE
- OrderItemComplaint
    - ORDER_ITEM_COMPLAINT_CREATED
    - ORDER_ITEM_COMPLAINT_UPDATED

As mentioned in Architecture section, ecoseller's backend is mainly a django application. It consists of main `core` project and following apps:

- `api` - provides logic of NotificationAPI and PaymentsAPI for integration of 3rd party payment methods and notifying external services
- `cart` - provides functionality for user's cart
- `category` - provides functionality for product categories
- `cms` - provides functionality for content pages
- `core` - main app of the project
- `country` - provides functionality for countries, addresses, shipping and billing information, currencies and VAT groups
- `emails` - provides functionality for order confirmation emails
- `order` - provides functionality for orders
- `product` - provides functionality for products and product variants
- `review` - provides functionality for product reviews
- `roles` - provides functionality for user roles and permissions

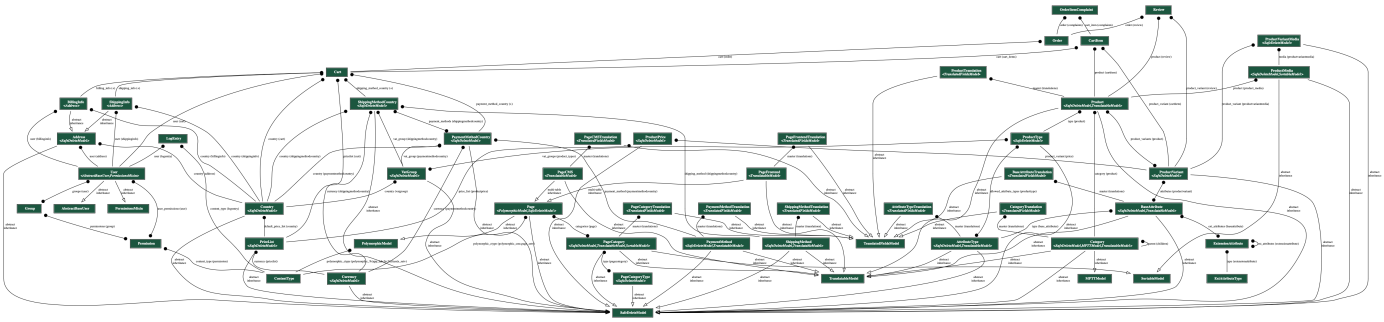
- `search` - provides functionality for searching products using ElasticSearch
- `user` - provides functionality for user related operations

To see the full core API specification, navigate to localhost:8000/swagger/ (if you are running the application locally). This will open the Swagger UI, where you can see all the endpoints and their documentation. Beware that many endpoints are protected by authorization, so you will not be able to access them without a proper JWT token.

# Data models

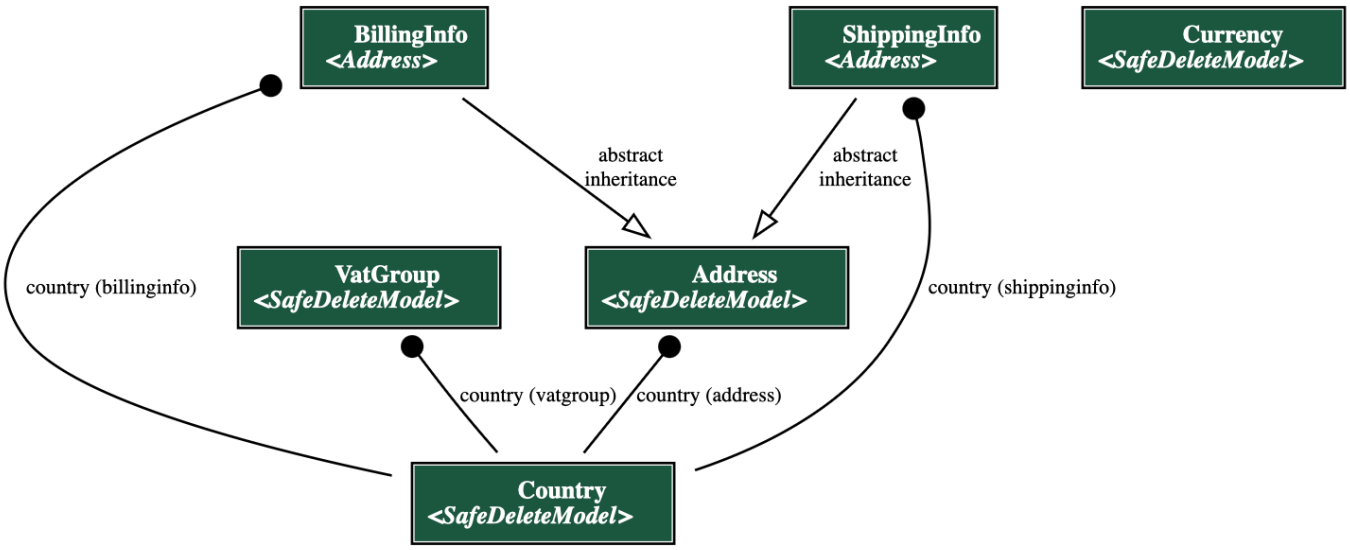
In this section we will describe data models of the backend part of the application. To do so, we will go over various parts of system and describe them in more detail using diagrams. To create diagrams, we used `django-extensions` app and its Graph models part, which generates a Graphviz `.dot` file from our django models. From that `.dot` file we used GraphvizOnline site to generate images of the diagrams.

The whole diagram of the models is shown below:



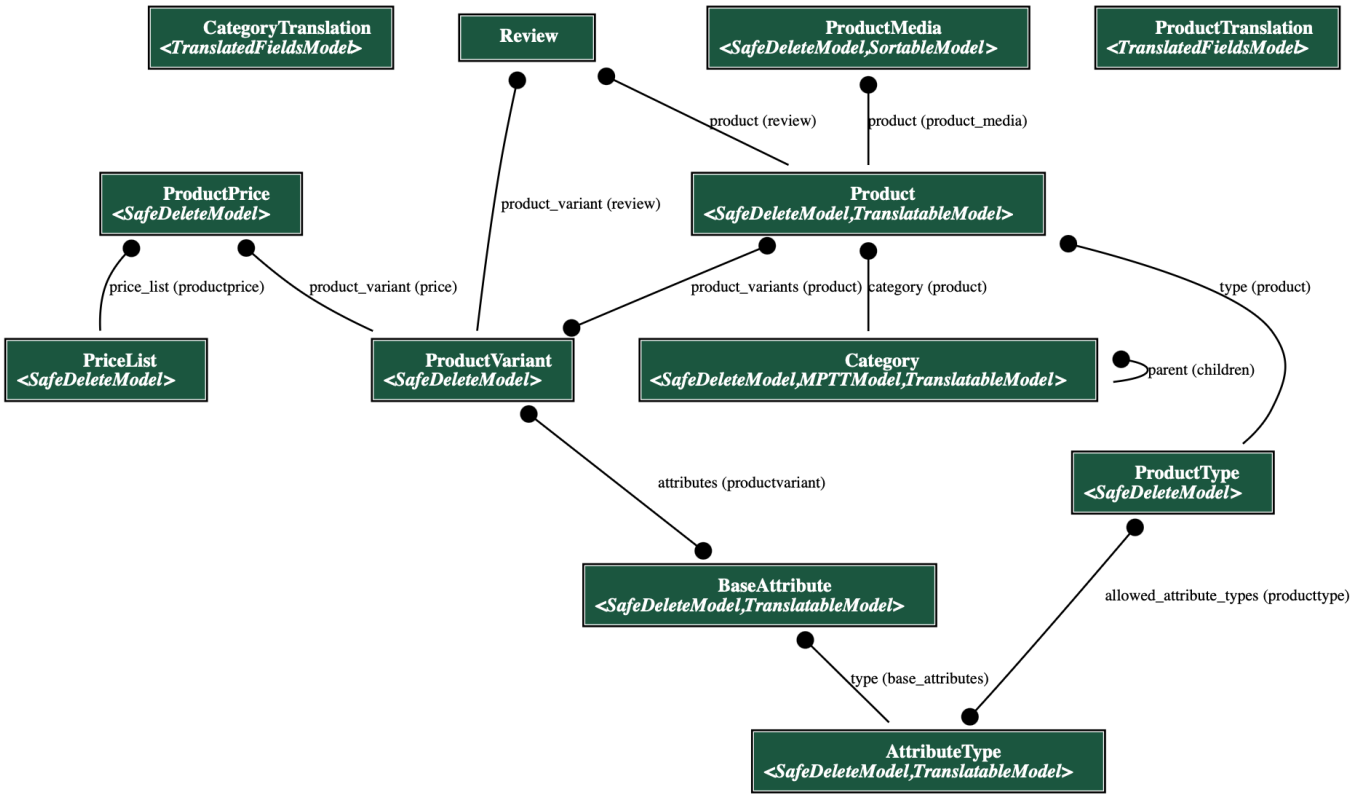
Due to the size of the diagram and complexity of the system, we will go over the models in smaller groups.

# Country




Above is the diagram of models with country specific data. The model is defined in `backend/core/country/models.py` file. The main "building" block is a `Country` model which holds all the data related to countries - like name, code, language, pricelist and vat groups. `VatGroup` itself defines binding between country and VAT percentage. `Currency` looks like a separate model with no relations, but it's mainly related to the `PriceList` model which will be described in a later sections. `Address` model is used to store addresses of users and is used during checkout process or are directly bindined to `Cart` model as well as `User` model. `ShippingInfo` and `BillingInfo` models are used to store user's shipping and billing information during checkout process. They inherit from `Address` model and add some additional fields.

# Product



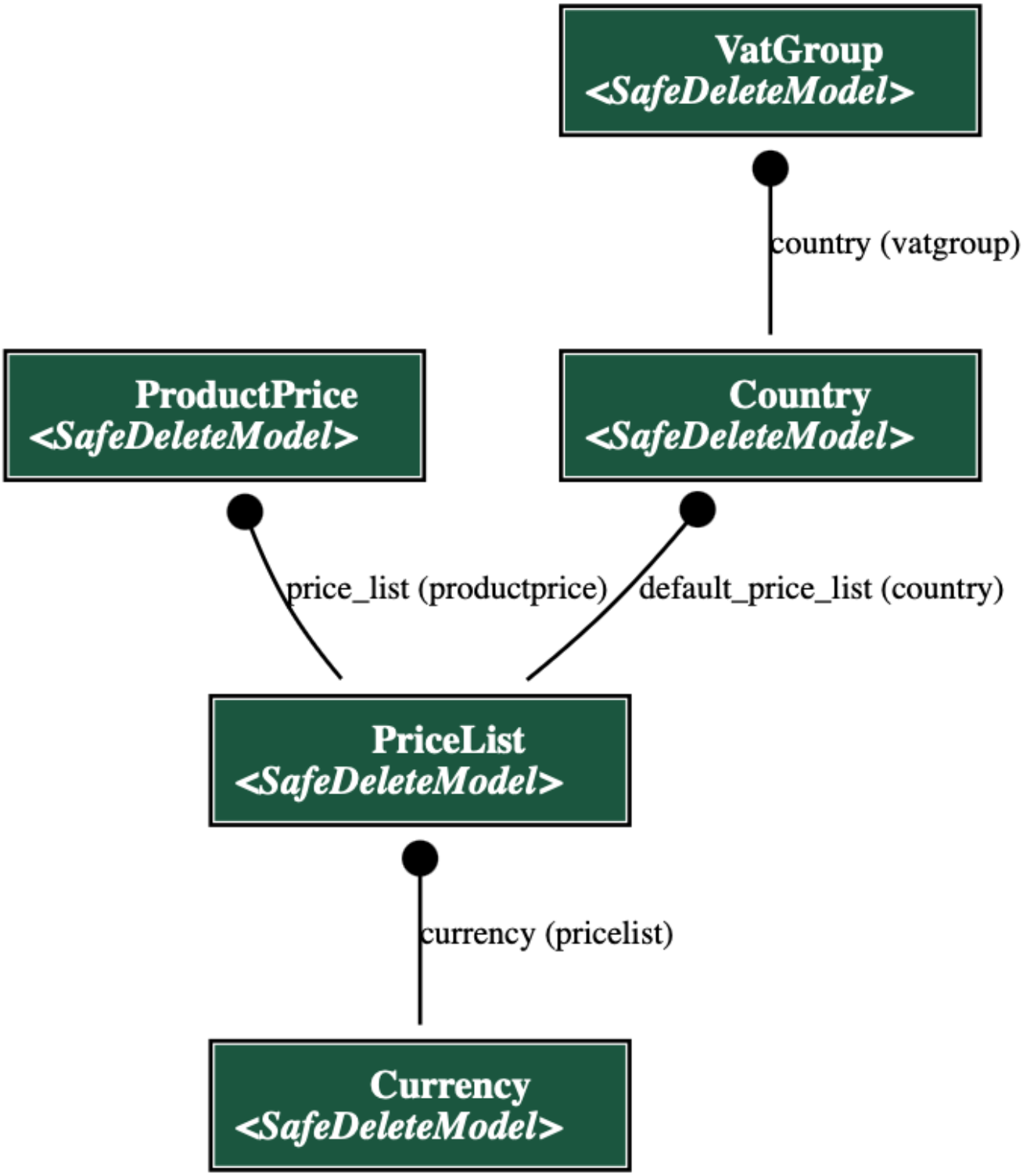

Above is the diagram of models related to products and categories. The models are defined in `backend/core/product/models.py` file and are divided into 2 groups:

- **Product models** - models that are directly related to products. They are:
  - `Product` - main product model.
  - `ProductMedia` - model for product media. It has a FK to `Product` model.
  - `ProductType` - model for product types. It defines the type of product (e.g. t-shirts, coffee, etc.). It defines allowed `AttributeType`s for product variants of this type and vat group for each country of this product.
  - `Category` - model for product categories. It's a tree structure, so it has a `parent` field which is a FK to itself.
- **Product Variant models** - models that are related to product variants. They are:
  - `ProductVariant` - main product variant model. It has a FK to `Product` model.

- `AttributeType` - model for product variant attributes. It defines the type of attribute (e.g. color, size, etc.).
- `BaseAttribute` - model for product variant attribute values. It defines the value of attribute (e.g. red, blue, etc.). It has a FK to `AttributeType` model.

Logic behind product variants is that each product variant has a set of attributes, which are defined by `AttributeType` model. Each attribute has a value, which is defined by `BaseAttribute` model. For example, if we have a product variant of type `t-shirt`, it will have 2 attributes: `color` and `size`. Each attribute will have a value, e.g. `color` will have values `red`, `blue`, `green`, etc. and `size` will have values `S`, `M`, `L`, etc.

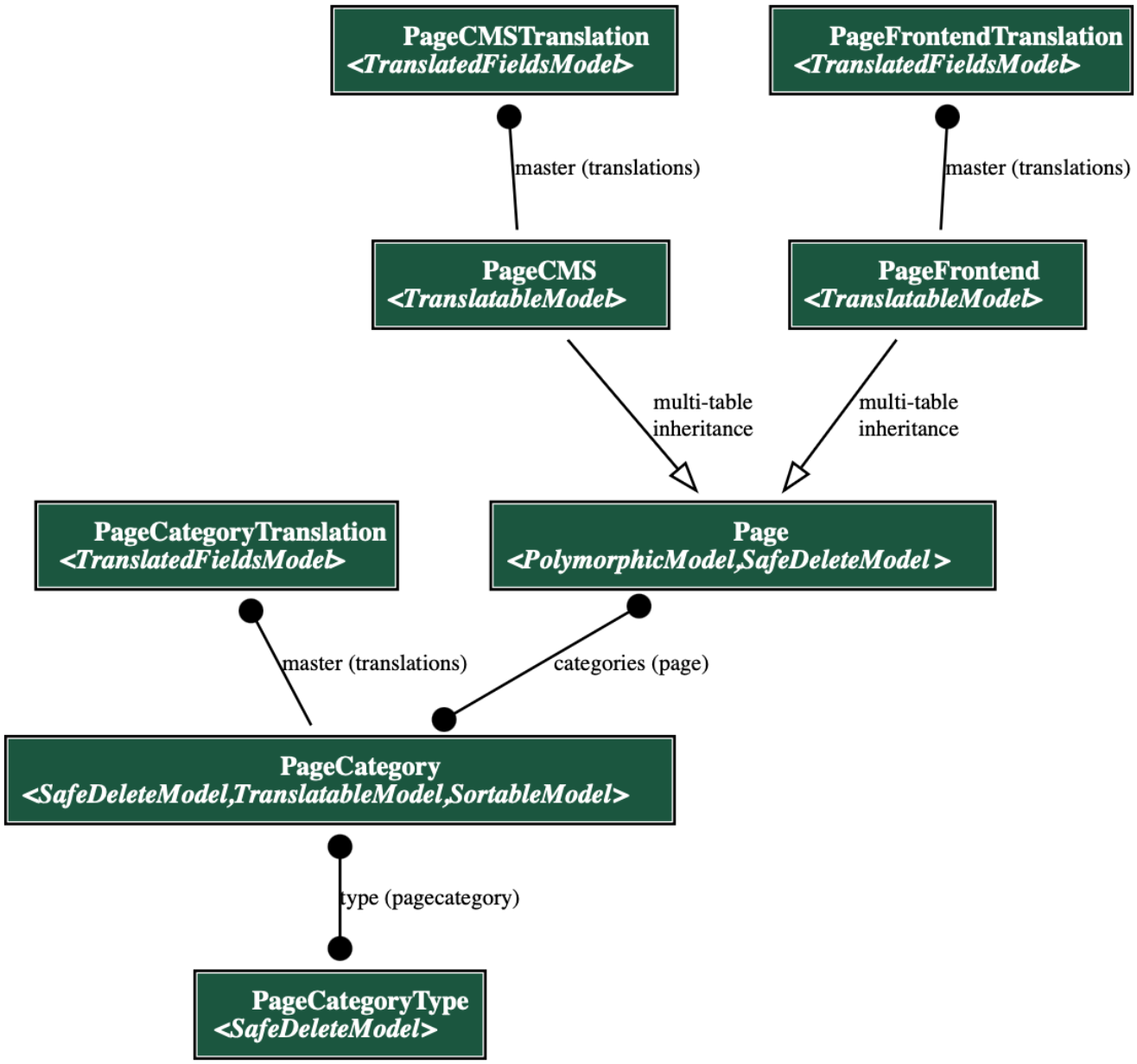
# Pricelist/Currency




Above is the diagram of models related to price lists and currencies. The models are defined in `backend/core/product/models.py` and `backend/core/country/models.py` files. Every price ( `ProductPrice` ) represents a price of `ProductVariant` in a `PriceList` . Where `PriceList` usually represents a specific

group of prices - it might be a group of prices for a specific country or a group of prices for a specific customers (like B2B or B2C). `PriceList` is also related to `Currency` model, which defines the currency of the prices in the price list. The interesting part of *ecoseller* pricing logic comes as `VatGroup` model which allows you to define different VAT groups for different countries. This allows you to have different VAT value (incl. different group of VAT - reduced, standard, ...) for different countries. With this logic, you can define a price list for a specific country and define different VAT groups for different countries. This allows you to have different prices for different countries, which is a common practice in e-commerce (for example due to different expenses for marketing, stocking, etc.).

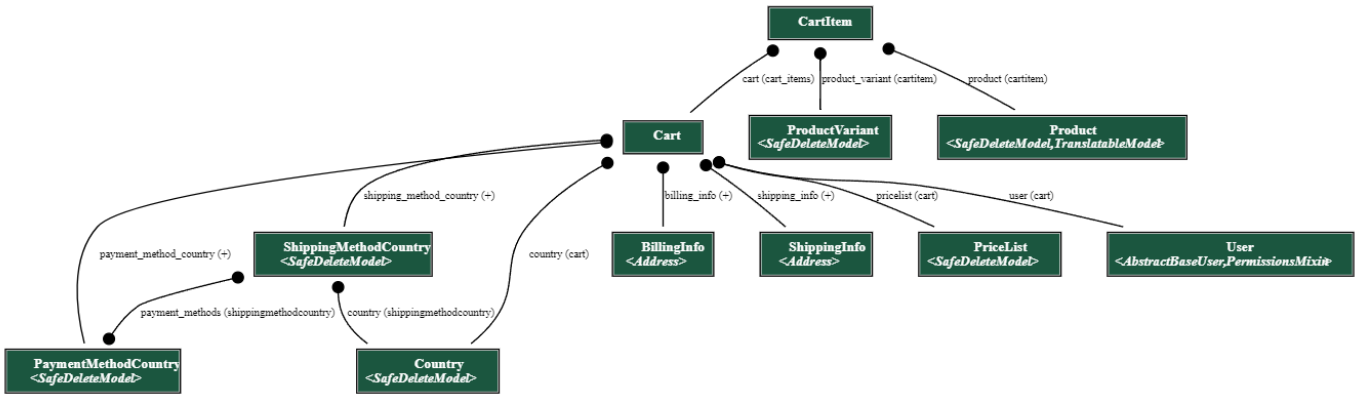
# CMS




Above is the diagram of the CMS models with its main relations. Models are defined in `backend/core/cms/models.py` file. It allows to create content pages with different types of content. The main model is `PageCMS` model, which represents a page with content in a specific language. It contains a content field in `editorjs` language. `PageFrontend` is an unussual idea in ecommerce platform. Since can have some specific pages that might not be stored in the database but would be represented as a HTML/JSX page, `PageFrontend` is simply a link to that page - or, to be clear, path of that page in the frontend. Why do we need that? Imagine a situation where you simply want some extra

CSS styles or some specific layout of the (landing) page. It's made directly in the frontend app and you simply store link in the database. This is perfectly usefull if you consider other model `PageCategory` which basically puts a page in a category. This allows you to create a group of different `PageCMS` and `PageFrontned`. For example, you can create a category `Info pages` and put all your info pages in it. We can go a bit further and create `PageCategoryType`, which can group these categories. For example, you can create a `PageCategoryType` `Footer` and put all your categoreis that should display in footer. You can then fetch those footer specific categories and display them in the footer of your website. This is a very flexible way of creating content pages and displaying them in the frontend.

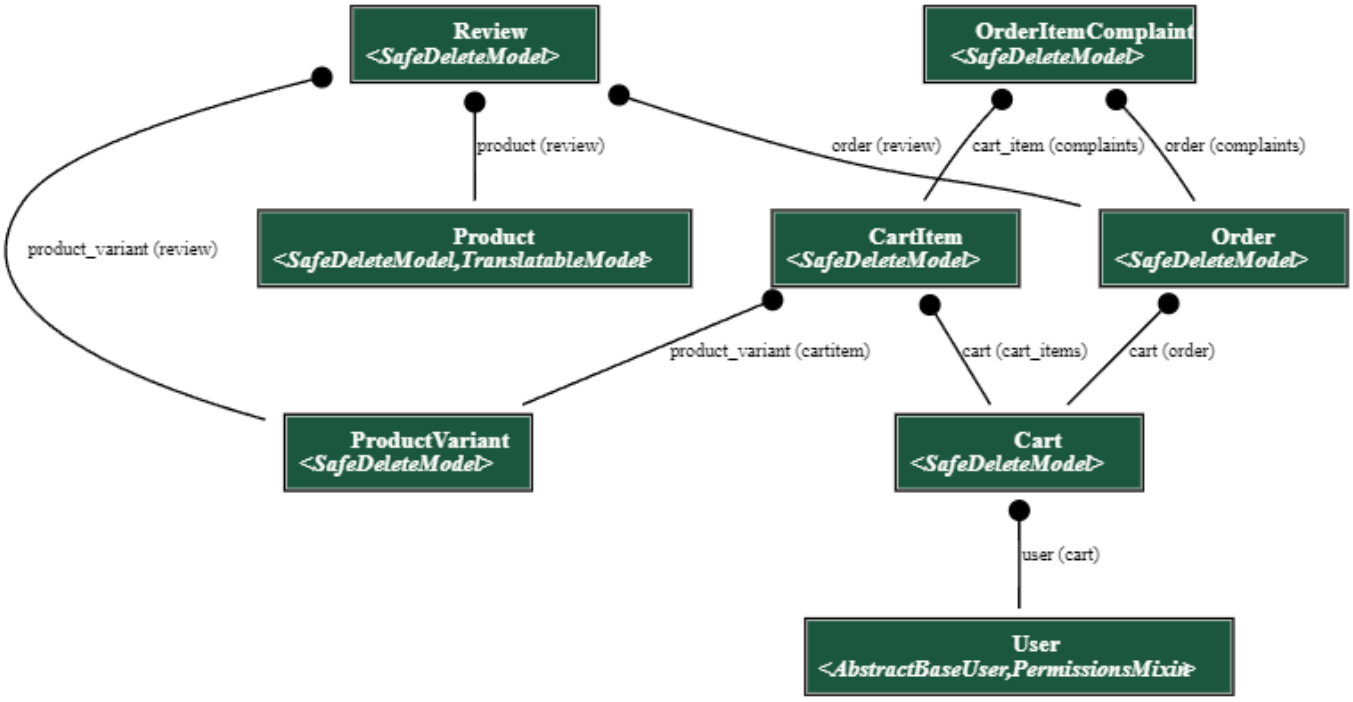
# Cart




Above is the diagram of the `Cart` model with its main relations to other models. The model is defined in `backend/core/cart/models.py` file. The `Cart` model is used to store user's cart. It has a FK to `User` model, which binds the cart to the user. It also has a FK to `ShippingMethodCountry` and `PaymentMethodCountry` models, which are used to store user's selected shipping and payment methods. We also have a `CartItem` model, which represents concrete item in the cart and has a FK relation to `Cart`. Each `CartItem` also has a FK to `ProductVariant` and `Product` models, to bind the item with the concrete product. The `Cart` model also has relations to country specific models such

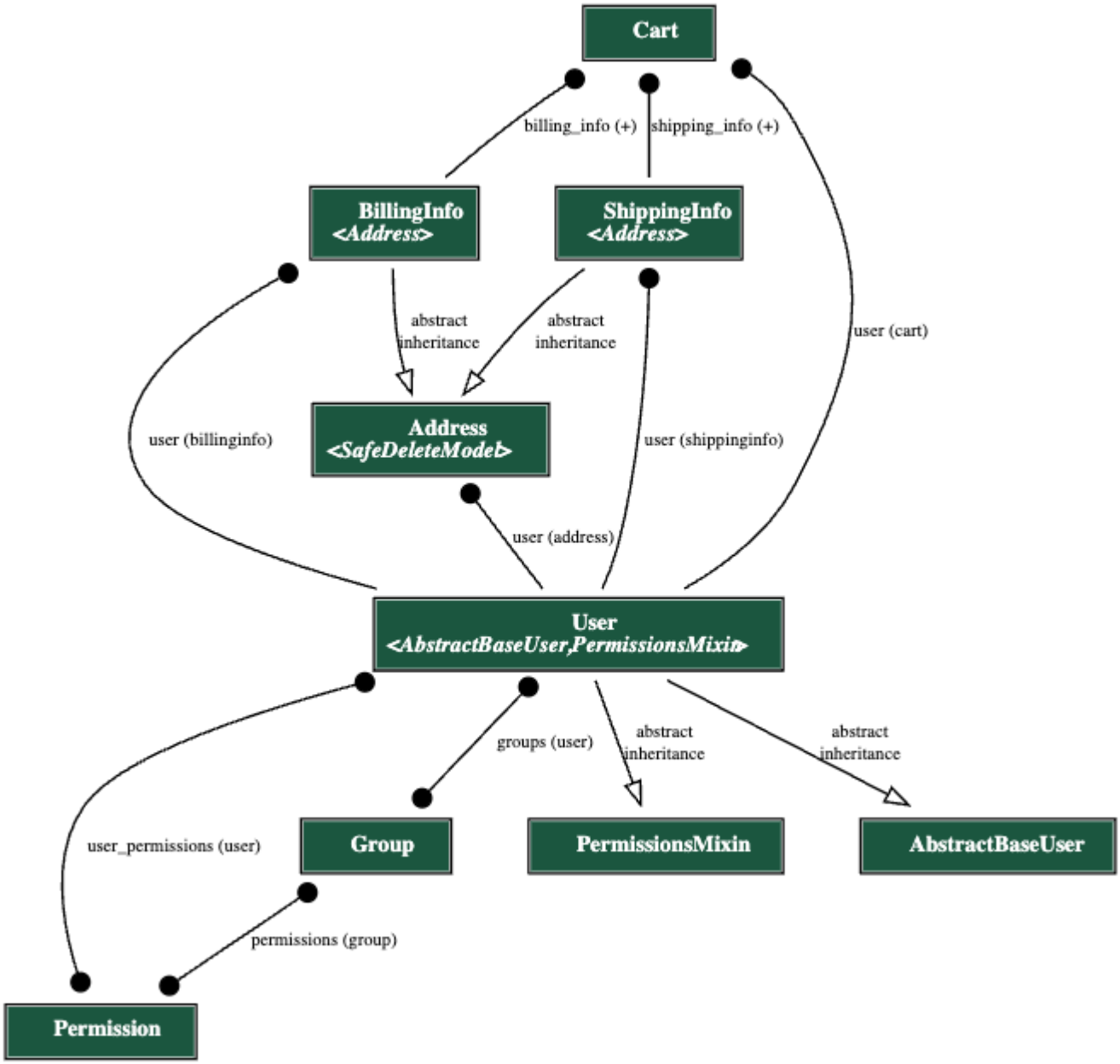
as `Country` , `PriceList` , `PaymentMethodCountry` and `ShippingMethodCountry` to ensure that the cart is bind to the concrete country specific data.

# Order




Once the user creates an order, new relation is created - a FK from `Order` model to `Cart` . We can see also another model in the diagram - `Review` , which is used to store user's reviews of products. It has a FK to `ProductVariant` , `Product` and `Order` models. Furthermore, there's also `OrderItemComplaint` model representing complaints (either warranty claim or request for return of an item), which has FK to `CartItem` and `Order` tables.

# User




Above is the diagram of the `User` model with its main relations to other models. The model is defined in `backend/core/user/models.py` file. In ecoseller, we replaced default django `User` model with our own `User` model in order to have more control over it. You can see that it has 2 abstract models as its parents: `AbstractBaseUser` and `PermissionsMixin` :

- `AbstractBaseUser` is a django abstract model that provides basic user functionality

- `PermissionsMixin` is a django abstract model that provides permissions functionality.

Another authorozation related models are `Group` and `Permission` models. They are django models that are used for authorization purposes. `Group` model is used to group users into units, while `Permission` model is used to define permissions for users. More on how we handle user authorization can be found in Authorization section. Next important relation is to `Address` model. It is used to store user's address. As we can see, there is also a connection to `ShippingInfo` and `BillingInfo`, which are used during checkout process, to store user's shipping and billing information. The last relation is to `Cart` model, which binds user to his cart.

# SafeDeleteModel

Note that all ecoseller models inherit from `SafeDeleteModel` class, which looks like this:

```
class SafeDeleteModel(models.Model):
    objects = SafeDeleteManager()
    safe_deleted = models.BooleanField(default=False)

    ...

    def delete(self, *args, **kwargs):
        self.safe_deleted = True
        self.save()

    ...
```

We can see, that this class basically adds one field ( `safe_deleted` ) and overrides `delete` method.
This way we implement safe deletion, so every time we call `delete` method

on a model, it's not physically removed from the DB, rather marked as deleted.

Also note, that we also override `objects` class variable in order to return just non-deleted products when querying.

So, if you call e.g.

```
Product.objects.all()
```

only the non-deleted products (i.e. those with `safe_deleted = False`) are returned.

# Authorization

As mentioned in Authorization section, ecoseller uses roles and permissions to restrict access to certain parts of the application.

To have better control over permissions representation and their grouping, we created 2 new models:

- `ManagerPermission` - for permission representation. It consists of:
  - `name` - name of permission with predefined format: *<model_name>*__*<permission_type>*__permission.
  - `model` - name of model to which this permission corresponds
  - `description` - text description of permission
  - `type` - type of permission. Enum of 4 possible values:
    - `view`
    - `add`
    - `change`

        - `delete`

- `ManagerGroup` - for group representation. It consists of:
    - `name` - name of group
    - `description` - text description of group
    - `permissions` - M2M field to permissions of which this group consists.

Each group/permission should be convertable to DRF group/permission.

# RolesManager

`RolesManager` is our internal python class for handling permissions and (almost) everything related to them. It consists purely of static methods, so we can call them anywhere across the code.

Its main usage is:

- Loading initial predefined roles from config and creating `ManagerGroup` and `ManagerPermission` objects from it
- Conversion between `DRF Group` and `ManagerGroup`, and also between `DRF Permission` and `ManagerPermission`

# Initial roles definitions and their loading

As mentioned earlier, we have `roles.json` config file which has initial roles definition and `RolesManager` class which is responsible for loading it. We achieved this behaviour by following adjustments:

1. We created `initial_data.py` file along with `populate_groups` method in it. In this method, we :
    1. load `roles.json` config with `RolesManager` class and create instances of `ManagerGroup` and `ManagerPermission`

    2. Create DRF Groups from loaded `ManagerGroup` objects
    3. Create general DRF permissions from `app_config`
    4. Convert all DRF permissions to `ManagerPermission` objects
    5. Assign `ManagerPermission` objects to corresponding `ManagerGroup` objects
2. We put `populate_groups` method in our `user` migration file `0002_auto_20230316_1534.py` to the `operations` part - this will ensure that when this migration runs, it will also trigger `populate_groups` method

# Protecting views with permissions

In order to apply our permission restrictions, we defined two custom decorators are defined: `@check_user_access_decorator` and `@check_user_is_staff_decorator` (their definition can be found in `backend/core/roles/decorator.py`).

## @check_user_access_decorator

The decorator is used mainly for `POST`, `PUT` and `DELETE` views. It checks if the user has the permission to perform the action. If the user has the permission, the view is executed. Otherwise, the view returns `403` status code.

---

## Parameters

- `permissions`: Set of permissions that the user needs to have to access view

---

## Usage example

To check whether the user has `product_change_permission` permission for accessing `put` method, put decorator above the method:

```
@check_user_access_decorator({"product_change_permission"})
def put(self, request, id):
    return super().put(request, id)
```

### @check_user_is_staff_decorator

The decorator is used mainly for `GET` views. It checks if the user is staff ( `is_staff` field in `User` model). If the user is staff, the view is executed. Otherwise, the view returns `403` status code.

---

## Parameters

- None: The decorator does not take any parameters

---

## Usage example

To check whether the user is staff for accessing `get` method, put decorator above the method:

```
@check_user_is_staff_decorator()
def get(self, request, id):
    return super().get(request, id)
```

# Email sending

Important part of an e-commerce application is sending emails to users. In ecoseller, we use build in `django.core.mail` module for sending emails. However there's a sophisticated logic behind it, which is described below. It allows us to have DRY code and to have better control over email sending process which are passed through our Djagno RQ.

## SMTP settings

In order to send emails, it's necessary to have SMTP server and provide configuration for it. It's done in `backend/core/settings.py` file via `EMAIL_*` variables that can be passed through environment variables.

```
EMAIL_USE_SSL=1
EMAIL_PORT=465
EMAIL_HOST=smtp.example.com
EMAIL_HOST_USER=yourusername
EMAIL_HOST_PASSWORD=yourpassword
EMAIL_FROM=Example<example@example.com>
```

## Email templates & objects

We've created `Email` object in `backend/core/emails/email/base.py` file which is used as a base class for all email objects.

---

### Email templates

We make usage of Django templating system for rendering email templates. All email templates are located in `backend/core/templates/email` directory. Each email template has its own HTML file with variables that are rendered with context generated by `generate_context` method of `Email` object.

---

## Email translation

You can, of course translate email templates and strings contained in it. It's done using Django `{% load i18n %}` tag in the template. With that included we can use `{% translate 'str_id' %}` tag provided by Django to translate strings. For more information about Django translation, please refer to Django documentation and for generating translation files, please refer to **ecoseller** administrative documentation.

---

## Email objects

Ecoseller is provided with a main class called `Email` that serves as a base class for all "emails" having following methods and attributes:

- `generate_context` - method for generating context for email template. It's used for rendering email template with context.
- `generate_subject` - method for generating email subject. It's used for generating email subject.
- `send` - method for sending email. It's used for sending email with rendered template and generated subject to the user. If object property

`use_rq` is set to `True`, it will send email via RQ. Otherwise, it will send email synchronously.

- `send_at` - method for sending email at specific time. It's used for sending email with rendered template and generated subject to the user at specific time. If object property `use_rq` is set to `True`, it will send email via RQ. Otherwise, it won't be sent.
- `use_rq` - property for determining whether email should be sent via RQ or not. It's set to `False` by default.
- `recepient_list` - property for determining recepients of the email. It's set to `[]` by default.
- `language` - property for determining language of the email. It's set to `cs` by default and can use only `settings.PARLER_LANGUAGES`

---

# Pre-defined email classes

**ecoseller** has pre-defined email objects that are used for sending emails to users. They are located in `backend/core/emails/email` directory.

## OrderItemComplaintConfirmationEmail

This email is sent to the user when they create a complaint for an order item. It's used for confirming that the complaint was created successfully.

## OrderItemComplaintStatusUpdateEmail

This email is sent to the user when the status of their complaint is updated. It's used for informing the user about the status of their complaint.

## EmailOrderConfirmation

This email is sent to the user when they create an order. It's used for confirming that the order was created successfully.

### EmailOrderReview

This email is sent to the user 14 days after the order was created. It's used for asking the user to review the order.

Let's dive into the code of `EmailOrderReview` object to see how it works:

```
class EmailOrderReview(Email):
    template_path = "email/generic_email.html"

    def __init__(self, order, recipient_list=[], use_rq=False):
        self.order = order
        self.language = order.cart.country.locale
        self.recipient_list = recipient_list
        self.use_rq = use_rq
        self.meta = {
            "order": self.order.pk,
            "type": "order_review",
            "language": self.language,
            "recipient_list": self.recipient_list,
        }

    def generate_subject(self):
        translation.activate(self.language)
        self.subject = _("Review your order")

    def generate_context(self):
        translation.activate(self.language)
        storefront_url = settings.STOREFRONT_URL
        self.context = {
            "main_title": _("Please review your order"),
            "subtitle": _("Hello,"),
            "body": _("We would like to ask you to review your order. "),
            "button_title": _("Review your order"),
```

```
            "button_link": f"{storefront_url}/review/{self.order.token}",
        }
```

As you can see, the `EmailOrderReview` object has `template_path` property set to `email/generic_email.html`. It means that the email will be rendered with `email/generic_email.html` template. Based on the `generate_context` method we can see that email will render with following context:

```
{
    "main_title": _("Please review your order"),
    "subtitle": _("Hello,"),
    "body": _("We would like to ask you to review your order. "),
    "button_title": _("Review your order"),
    "button_link": f"{storefront_url}/review/{self.order.token}",
}
```

We can see that the context contains `button_link` variable which is used for generating button in the email. The button will have `Review your order` title and will redirect the user to the `storefront_url` with `/review/{self.order.token}` path.

`generic_email.html` can be used for multiple usecases - for informational email or for email with CTA button. It's up to you how you use it.

---

# Sending emails

There're situations when we need to send emails to users. For example, when the user creates an order, we need to send them an email with order confirmation. For that, we use `EmailOrderConfirmation` object. But how and where do we call it? For this purpose, we've `NotificationsAPI` which is used to

react to events that happen in the system. For example, when the user creates an order, we call `ORDER_SAVE` event and send `order` object as a payload. Then, via proper configuration, we can call `backend/core/api/notifications/connectors/email.py` which will send `EmailOrderConfirmation` object to the user based on `send_order_confirmation` passed from `backend/core/config/notifications.json` as `method` to the `type EMAIL`.

```
"ORDER_SAVE": [
    {
      "type": "EMAIL",
      "method": "send_order_confirmation"
    },
    ...
  ],
```

# Product filtering & ordering

In this chapter, we'll describe product filtering and ordering, which is used for storefront, in more detail.

First of all, it's important to mention several important things

- On the storefront category page, products are displayed, however attributes (and therefore filters) are assigned to product variants. So, the filtering is done in a following way:
  **A product matches the filters if there's at least 1 of its variants matching them.**
- In Core, there are 2 endpoints for getting the products in a given category:
  - GET `category/storefront/<int:pk>/products/` – used for initial getting of the products (with no filters selected)

    - POST `category/storefront/<int:pk>/products/` – used for getting products with filtering & order selected. It's neccessary to pass the object containing selected filters and ordering in the request body. Note that this object can be possible quite complex, so we decided to implement it using POST method and pass the data in request body, instead of another GET.
- Both filtering and ordering is done using DB queries, in order to optimize the performance.

When filtering products (using the POST request mentioned before), you should pass JSON with the following structure in a body:

```
{
  "filters":{
    "numeric": [...],
    "textual": [...],
  },
  "sort_by": "...",
  "order': "..."
}
```

- `filters` – contains serialized filters
- `sort_by` – denotes a value which should be used for ordering products
- `order` – denotes ordering of products (ascending/descending), possible values are: `asc` (default) and `desc`

We're going to describe `sort_by` field in more detail, because the logic is a little bit more complex here.
That's because we often want to sort products by a value, that's not directly present in `Product` or `ProductVariant` objects (e.g. *title*, which is stored in multiple translations or *price*, which is stored in a separate object `ProductPrice`, as you can see here).

We decided to implement ordering in a following way: `CategoryDetailProductsStorefrontView` contains static property `SORT_FIELDS_CONFIG`, which contains configuration of possible orderings and looks as follows:

```
SORT_FIELDS_CONFIG = {
  "title": {
      "sort_function": _order_by_title,
      "additional_params": [locale],
  },
  "price": {
      "sort_function": _order_by_price,
      "additional_params": [pricelist],
  },
  "recommended": {
      "sort_function": _order_by_recommendation,
      "additional_params": [recommendations],
  }
]
```

We see that there are 3 possible orderings:

- by title
- by price
- by recommendation

Each of the possible orderings needs to have configured

- `sort_function` – reference to function used for sorting by this value
- `additional_params` – array of additional parameters that are passed to the sort function, except the list of products and ordering type (asc/desc) which are passed always. You can omit this field if there are no additional params, the default value – empty list will be used

Then, the sort function looks like this:

```
def _order_by_price(products, is_reverse_order, pricelist: PriceList):
    """
    Extend product query by extra field with lowest price of the product's
variant prices
    """

    products = products.annotate(
        price=Subquery(
            ProductPrice.objects.filter(
                product_variant__in=OuterRef("product_variants"),
                price_list__code=pricelist.code,
            )
            .order_by("price")
            .values("price")[:1]
        )
    ).order_by("price" if not is_reverse_order else "-price")
    return products
```

We see that the function above sorts the products by their price; note that it contains an extra parameter ( `pricelist` ), which was specified in `additional_params` .
Again, it's important to mention that **the first 2 arguments (**`products` **and** `is_reverse_order`**) are passed to any sort function.**

If your goal is to add another product ordering, you need to firstly implement a sort function in a similar way as above and then add it to `SORT_FIELDS_CONFIG` as well.

Also note, that on storefront, filters for each category are stored in session storage, so when an user e.g. filters the products, goes to product detail page and then back, the same filters as before are applied.

# Static files and media

ecosellercurrently supports storing static and media files using local storage. While it does not natively integrate with object storage services like Amazon S3, it is possible to implement such functionality using the Python package s3boto3.

However, in most cases, storing static and media files locally is sufficient for the needs of an e-commerce platform. Hence, why we decided to use simplest solution possible using WhiteNoise package. It was neccassary to use this package because of the way Django works. Django does not serve static files in production, so serving the app via Gunicon or uWSGI would not work propely. WhiteNoise is a middleware that allows Django to serve static files in production.

If you want to disable WhiteNoise, you can change `MIDDLEWARE` in `backend/core/settings.py` to:

```
MIDDLEWARE = [
    # ...
    "django.middleware.security.SecurityMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # Remove this line
    # ...
]
```

# Implementing payment methods (`PaymentAPI`)

This guide will walk you through the process of extending the Core application with new payment methods without encountering conflicts with

the existing codebase. By following the provided guidelines and leveraging the system's flexible architecture, you'll be able to seamlessly integrate various online payment gateways into your ecosellerecommerce platform.

Integrating online payment gateways into your ecommerce system offers numerous advantages. It allows your customers to securely make payments using their preferred payment methods, which can boost conversion rates and provide a seamless checkout experience. The ecosellersystem's architecture has been designed to make the implementation of new payment methods straightforward, enabling you to expand your payment gateway options as your business grows.

# Payment Gateway Integration Process

To implement a new payment method within the ecoseller ecommerce system, you will need to follow these steps:

1. Choose the appropriate base class: Your new payment method should inherit from either the `PayBySquareMethod` class or the `OnlinePaymentMethod` class. Both of these classes are derived from the `BasePaymentMethod` class and can be imported from `core.api.payments.modules.BasePaymentMethod`. Select the base class that aligns with the requirements of the payment gateway you are integrating. `PayBySquareMethod` is used in situations where it's neccessary to provide user payment QR code. On the other hand `OnlinePaymentMethod` is used for generating link for third party payment gateway where is user usually redirected. **Note that classes inherited from `BasePaymentMethod` obtain instance of `Order` model (see `core.order.models`) on initialization. So you can freely access data of `Order` and `Cart`.**

2. Create a new payment method class: In your codebase, create a new class that extends the chosen base class. Ideally put your code into separate file

stored in `core/api/payments/modules`. Provide a meaningful name for the class that reflects the payment gateway you are integrating. For example, if integrating the "XYZ Gateway," you could name your class XYZGatewayMethod. Implement the necessary methods:

---

## PayBySquareMethod

`PayBySquareMethod` is used to return Base64 encoded image and provide payment data such as IBAN, BIC, etc. In order to implement a payment method inherited from `PayBySquareMethod` you need to define two methods:

- `pay` - it's expected that this method return a dictonary containing two required keys.
- - `qr_code` - Base64 encoded image of the payment QR
- - `payment_data` - dictionary containing payment information stored in text with keys such as `amount`, `IBAN`, `payment_identification`, etc.
- `status` - method returning `PaymentStatus` (`core.api.payments.conf.PaymentStatus`). So for example - calling API of your bank and checking icomming payments.

---

## OnlinePaymentMethod

`OnlinePaymentMethod` is used to return payment link so that user can be redirected. In order to implement a payment method inherited from `OnlinePaymentMethod` you need to define two methods:

- `pay` - it's expected that this method return a dictonary containing two required keys.
  - `payment_url` - link to the payment gateway (usually provided by the payment gateway API with `payment_id` in it).
  - `payment_id` - ID of the payment in the payment gateway
- `status` - method returning `PaymentStatus` (`core.api.payments.conf.PaymentStatus`). Usually implemented as a wrapper around payment gateway's status getter.

Examples:

```
from .BasePaymentMethod import OnlinePaymentMethod, PayBySquareMethod
from ..conf import PaymentStatus


class TestGateway(OnlinePaymentMethod):
    def pay(self):
        return {"payment_url": "https://payment.url", "payment_id": "1234567890"}

    def status(self) -> PaymentStatus:
        """
        Moc status and return paid with some probability
        """
        import random

        if random.random() < 0.5:
            return PaymentStatus.PAID
        return PaymentStatus.PENDING

class BankTransfer(PayBySquareMethod):
    def pay(self):
        self.bic = self.kwargs.get("bic")
        self.iban = self.kwargs.get("iban")
        self.currency = self.kwargs.get("currency")
        self.variable_symbol = 123456789
        self.amount = 100
```

```
            return {
                "qr_code": "base64 encoded image",
                "payment_data": {
                    "amount": self.amount,
                    "currency": self.currency,
                    "variable_symbol": self.variable_symbol,
                    "iban": self.iban,
                    "bic": self.bic,
                },
            }

        def status(self) -> PaymentStatus:
            """
            Mock status and return paid with some probability
            """
            import random

            if random.random() < 0.5:
                return PaymentStatus.PAID
            return PaymentStatus.PENDING
```

1. Registering payment method in the `Core` In order to let the `Core` know about your payment methods, you need to define JSON configuration file. This file can be stored anywhere within accessible space for the `core` . However, to keep ecosellerpractices, we recommend to store this file in `core/config/payments.json` (default path). Your custom path must be stored in the `PAYMENT_CONFIG_PATH` environment variable.

It’s a dictionary containing unique identifiers of payment methods. Those identifiers are up to you, the only requirement is that you keep the unique constraint and that the name makes somehow sense. You will use this name also in the `dashboard` to link the payment method with your backend implementation.

Every payment method is required to have `implementation` key which is in the format `{module}.{class}` , so for example

`api.payments.modules.BankTransfer.BankTransfer` or `api.payments.modules.TestGateway.TestGateway`. You can use optional key `kwargs` (keyword arguments) into which you can store everything constant that you want to access in the `BasePaymentMethod` implementation (it's stored into `self.kwargs` variable 😉). Usually this is used to pass your IBAN, ... into `PayBySquareMethod` or public key or path to public certificate into `OnlinePaymentMethod`.

Example:

```
{
    "BANKTRANSFER_EUR": {
        "implementation": "api.payments.modules.BankTransfer.BankTransfer",
        "kwargs": {
            "currency": "EUR",
            "bankName": "Deutsche Bank",
            "accountNumber": "DE12500105170648489890",
            "swiftCode": "DEUTDEDBBER"
        }
    },
    "BANKTRANSFER_CZK": {
        "implementation": "api.payments.modules.BankTransfer.BankTransfer",
        "kwargs": {
            "currency": "CZK",
            "bankName": "CŠOB",
            "accountNumber": "CZ5855000000001265098001",
            "swiftCode": "CEKOCZPP"
        }
    },
    "TEST_API": {
        "implementation": "api.payments.modules.TestGateway.TestGateway",
        "kwargs": {
            "merchant": "123456",
            "secret": "1234567890abcdef1234567890abcdef",
            "url": "https://payments.comgate.cz/v1.0/"
        }
    }
}
```

1. Binding payment method to the implementation So you have your payment method implementation ready and want to bind to your payment method object. On the `PaymentMethodCountry` model is a field ready for this situation. There're two ways to do it:

- **Using dashboard:** Navigate to the detail of payment method (Cart/Payment Methods) in the dashboard, scroll to *Country variants* and set `API Request` for required country variant.

- **Using direct database access:** Find `cart_paymentmethodcountry` table in your database and set `api_request` field for the specific row the the value which you used as unique identifier of your payment method in the `JSON` config. So for example `BANKTRANSFER_CZK` . However direct database access is not recommended.

1. Now you only need to process the data correctly on the storefront and you're ready to go. So either redirect you user automatically, show the payment square or do something else. We tried to make it generic so that it's not anyhow limiting for your specific use-case.

# Recommendations

Because online payments are crucial part for customer's safety and comfort, we recommend to use online payment gateways that are known to the users in specific country. For example, don't use czech payment gateway for german customers and vice-versa. Use something that your customers know and are familiar with. Due to that we decided that we will allow to bind payment method implementation to every country variant.

# Connecting external services (`NotificationAPI`)

This comprehensive guide will provide you with all the necessary information to seamlessly extend ecoseller's functionality by leveraging external APIs. With the Notification API, you can effortlessly integrate your own systems and services to respond to specific events within the ecosellerplatform, such as product save, order save, and more.

The ecosellerNotification API empowers you to enhance your ecosellerexperience by enabling real-time communication and synchronization with external applications. By leveraging this API, you can ensure that your external systems stay up to date with the latest changes and events happening within ecoseller, allowing for a seamless and efficient workflow.

This documentation will walk you through the entire process of working with the Notification API in your application. You'll learn how to configure endpoints and interpret the data sent by ecoseller.

---

## Key Features of the Notification API:

**Event-based Triggers:** The notifications API allows you to define specific events within ecoseller, such as `PRODUCT_SAVE` and `ORDER_SAVE`. These events serve as triggers for the notifications.

**Multiple Notification Types:** The API supports various notification types, including `RECOMMENDERAPI`, `HTTP`, and `EMAIL`. You can choose the appropriate type based on your integration requirements.

**Flexible Methods:** Each notification type can have different methods associated with it. For example, for the `RECOMMENDERAPI` type, the method `store_object` is used, while for the `HTTP` type, methods like `POST` are utilized.

**HTTP Integration:** The API allows you to send HTTP requests to external endpoints by specifying the URL. This enables seamless integration with other systems or services that can receive and process the notifications.

**Email Notifications:** With the "EMAIL" type, you can send email notifications related to specific events. In the given configuration file, the "send_order_confirmation" method is used to trigger the sending of an order confirmation email. Customization: The JSON configuration file provides flexibility for customization. You can easily add or modify notification types, methods, and URLs based on your specific integration requirements.

**Expandable Event List:** The JSON configuration can be extended to include additional events and corresponding notifications. This allows you to adapt the API to match a wide range of events and actions within the ecosellerplatform.

By leveraging these key features of the notifications API, you can extend the functionality of ecosellerby seamlessly integrating with external systems, such as recommender engines, HTTP-based APIs, and email services. This enables you to create powerful workflows and automate processes based on specific events occurring within ecoseller.

# Usage

Here is some example (default) NotificationAPI configuration.

---

## Configuring Notification API configuration

To configure your notifications, you need to edit provided JSON configurations in the Core component.

The provided configuration might look like this:

```
{
    "PRODUCT_SAVE": [
        {
            "type": "RECOMMENDERAPI",
            "method": "store_object"
        },
        {
            "type": "HTTP",
            "method": "POST",
            "url": "http://example.com/api/product"
        }
    ],
    "ORDER_SAVE": [
        {
            "type": "HTTP",
            "method": "POST",
            "url": "http://example.com/api/order"
        },
        {
            "type": "EMAIL",
            "method": "send_order_confirmation"
        }
    ]
}
```

As you can see, for every trigger you can setup list of events that will be performed.

# List of connectors

There are multiple actions you can perform using predefined connectors:

- `HTTP` type: this action requires to have `method` and `url` provided. As the title says, `method` is mean as an HTTP Method. You can use all methods utilized by Python `requests` module.

- `EMAIL` type: you can control sending e-mails using internal `email app` . Feel free to remove e-mail events that you don't want to be sent using the Django interface.

- `RECOMMENDER` type: if you don't want to use provided recommendation system feature, feel free to remove events providing data to the recommender.

We recommend to edit configuration JSON directly ( `core/config/notifications.json` ). However, you can define your custom one and installing it by setting `NOTIFICATIONS_CONFIG_PATH` as your environment variable.

# List of triggers

The triggers that you can respond to are derived from the `ecoseller` models. It's usually an action based on `save` , `update` or `delete` .

# Model based triggers

Make sure you are fammiliar with ecoseller data models. Events are then pretty self-explanatory. Here is the list of all events that you can respond to:

## Product

Whenever product is saved, updated or deleted, you can respond to it using following events:

`PRODUCT_SAVE`

JSON Payload sent to the connector is the product itself simmilar to the example below:


```
{
   "_model_class":"Product",
   "id":2,
   "published":true,
   "type":3,
   "category_id":3,
   "product_translations":[
      {
         "id":3,
         "language_code":"en",
         "title":"Jumanji",
         "meta_title":"Jumanji",
         "meta_description":"When two kids find and play a magical board game,
they release a man trapped in it for decades - and a host of dangers that can
only be stopped by fin",
         "short_description":"None",
         "slug":"jumanji"
      },
      {
         "id":4,
         "language_code":"cs",
         "title":"Jumanji",
         "meta_title":"Jumanji",
         "meta_description":"Když dvě děti najdou a hrají kouzelnou deskovou hru,
osvobodí muže, který v ní byl po desetiletí uvězněn - a spoustu nebezpečí, která
```

```
lze zastavit pou",
         "short_description":"None",
         "slug":"jumanji"
      }
   ],
   "product_variants":[
      "2-cs-1080p",
      "2-en-720p"
   ],
   "update_at":"2023-07-09T17:35:11.713935+00:00",
   "create_at":"2023-07-09T17:35:11.713935+00:00",
   "deleted":false
}
```

PRODUCT_UPDATE

## JSON Payload sent to the connector is the product itself simmilar to the example below:

```
{
   "_model_class":"Product",
   "id":2,
   "published":true,
   "type":3,
   "category_id":3,
   "product_translations":[
      {
         "id":3,
         "language_code":"en",
         "title":"Jumanji",
         "meta_title":"Jumanji",
         "meta_description":"When two kids find and play a magical board game,
they release a man trapped in it for decades - and a host of dangers that can
only be stopped by fin",
         "short_description":"None",
         "slug":"jumanji"
      },
      {
         "id":4,
         "language_code":"cs",
         "title":"Jumanji",
```

```
            "meta_title":"Jumanji",
            "meta_description":"Když dvě děti najdou a hrají kouzelnou deskovou hru, osvobodí muže, který v ní byl po desetiletí uvězněn - a spoustu nebezpečí, která lze zastavit pou",
            "short_description":"None",
            "slug":"jumanji"
         }
      ],
      "product_variants":[
         "2-cs-1080p",
         "2-en-720p"
      ],
      "update_at":"2023-07-09T17:35:11.713935+00:00",
      "create_at":"2023-07-09T17:35:11.713935+00:00",
      "deleted":false
   }
```


PRODUCT_DELETE

## JSON Payload sent to the connector is the product itself simmilar to the example below:


```
{
   "_model_class":"Product",
   "id":2,
   "published":true,
   "type":3,
   "category_id":3,
   "product_translations":[
      {
         "id":3,
         "language_code":"en",
         "title":"Jumanji",
         "meta_title":"Jumanji",
         "meta_description":"When two kids find and play a magical board game, they release a man trapped in it for decades - and a host of dangers that can only be stopped by fin",
         "short_description":"None",
         "slug":"jumanji"
      },
      {
```

```
            "id":4,
            "language_code":"cs",
            "title":"Jumanji",
            "meta_title":"Jumanji",
            "meta_description":"Když dvě děti najdou a hrají kouzelnou deskovou hru,
    osvobodí muže, který v ní byl po desetiletí uvězněn - a spoustu nebezpečí, která
    lze zastavit pou",
            "short_description":"None",
            "slug":"jumanji"
         }
      ],
      "product_variants":[
         "2-cs-1080p",
         "2-en-720p"
      ],
      "update_at":"2023-07-09T17:35:11.713935+00:00",
      "create_at":"2023-07-09T17:35:11.713935+00:00",
      "deleted":false
   }
```


## ProductVariant

Whenever product variant is saved, updated or deleted, you can respond to it using following events:

`PRODUCTVARIANT_SAVE`

JSON Payload sent to the connector is the product variant itself simmilar to the example below:


```
{
   "_model_class":"ProductVariant",
   "sku":"2-en-720p",
   "ean":"",
   "weight":189.0,
   "stock_quantity":63,
   "recommendation_weight":1.0,
   "update_at":"2023-07-18T10:27:19.559905+00:00",
   "create_at":"2023-07-09T17:38:10.811513+00:00",
   "attributes":[
      3,
```

```
        5,
        8,
        9,
        10
    ],
    "deleted":false
}
```

PRODUCTVARIANT_UPDATE

JSON Payload sent to the connector is the product variant itself simmilar to the example below:

```
{
    "_model_class":"ProductVariant",
    "sku":"2-en-720p",
    "ean":"",
    "weight":189.0,
    "stock_quantity":63,
    "recommendation_weight":1.0,
    "update_at":"2023-07-18T10:27:19.559905+00:00",
    "create_at":"2023-07-09T17:38:10.811513+00:00",
    "attributes":[
        3,
        5,
        8,
        9,
        10
    ],
    "deleted":false
}
```

PRODUCTVARIANT_DELETE

JSON Payload sent to the connector is the product variant itself simmilar to the example below:

```
{
    "_model_class":"ProductVariant",
    "sku":"2-en-720p",
```

```
    "ean":"",
    "weight":189.0,
    "stock_quantity":63,
    "recommendation_weight":1.0,
    "update_at":"2023-07-18T10:27:19.559905+00:00",
    "create_at":"2023-07-09T17:38:10.811513+00:00",
    "attributes":[
       3,
       5,
       8,
       9,
       10
    ],
    "deleted":false
 }
```


## ProductPrice

Whenever product price is saved, updated or deleted, you can respond to it using following events:

`PRICE_SAVE`

JSON Payload sent to the connector is the product price itself simmilar to the example below:


```
 {
    "_model_class":"ProductPrice",
    "id":13,
    "price_list_code":"CZK_retail",
    "product_variant_sku":"2-cs-1080p",
    "price":229.0,
    "update_at":"2023-07-18T10:31:56.386815+00:00",
    "create_at":"2023-07-09T17:37:28.340365+00:00",
    "deleted":false
 }
```


`PRICE_UPDATE`

JSON Payload sent to the connector is the product price itself simmilar to the example below:


```
{
   "_model_class":"ProductPrice",
   "id":13,
   "price_list_code":"CZK_retail",
   "product_variant_sku":"2-cs-1080p",
   "price":229.0,
   "update_at":"2023-07-18T10:31:56.386815+00:00",
   "create_at":"2023-07-09T17:37:28.340365+00:00",
   "deleted":false
}
```


`PRICE_DELETE`

JSON Payload sent to the connector is the product price itself simmilar to the example below:


```
{
   "_model_class":"ProductPrice",
   "id":13,
   "price_list_code":"CZK_retail",
   "product_variant_sku":"2-cs-1080p",
   "price":229.0,
   "update_at":"2023-07-18T10:31:56.386815+00:00",
   "create_at":"2023-07-09T17:37:28.340365+00:00",
   "deleted":false
}
```


## ProductType

Whenever product type is saved, updated or deleted, you can respond to it using following events:

`PRODUCTTYPE_SAVE`

JSON Payload sent to the connector is the product type itself simmilar to the example below:


```
{
   "_model_class":"ProductType",
   "id":3,
   "name":"Movie",
   "attribute_types":[
      1,
      2,
      3,
      4
   ],
   "products":[
      6,
      10,
      16,
      1
   ],
   "update_at":"2023-07-18T10:34:06.786793+00:00",
   "create_at":"2023-07-08T15:38:32.982680+00:00",
   "deleted":false
}
```


PRODUCTTYPE_UPDATE

JSON Payload sent to the connector is the product type itself simmilar to the example below:


```
{
   "_model_class":"ProductType",
   "id":3,
   "name":"Movie",
   "attribute_types":[
      1,
      2,
      3,
      4
   ],
```

```
    "products":[
       6,
       10,
       16,
       1
    ],
    "update_at":"2023-07-18T10:34:06.786793+00:00",
    "create_at":"2023-07-08T15:38:32.982680+00:00",
    "deleted":false
 }
```

`PRODUCTTYPE_DELETE`

JSON Payload sent to the connector is the product type itself simmilar to the example below:

```
 {
    "_model_class":"ProductType",
    "id":3,
    "name":"Movie",
    "attribute_types":[
       1,
       2,
       3,
       4
    ],
    "products":[
       6,
       10,
       16,
       1
    ],
    "update_at":"2023-07-18T10:34:06.786793+00:00",
    "create_at":"2023-07-08T15:38:32.982680+00:00",
    "deleted":false
 }
```

## AttributeType

Whenever product attribute type is saved, updated or deleted, you can respond to it using following events:

ATTRIBUTETYPE_SAVE

JSON Payload sent to the connector is the attribute type itself simmilar to the example below:

```
{
   "_model_class":"AttributeType",
   "id":1,
   "type":"CATEGORICAL",
   "type_name":"GENRE",
   "unit":"None"
}
```

ATTRIBUTETYPE_UPDATE

JSON Payload sent to the connector is the attribute type itself simmilar to the example below:

```
{
   "_model_class":"AttributeType",
   "id":1,
   "type":"CATEGORICAL",
   "type_name":"GENRE",
   "unit":"None"
}
```

ATTRIBUTETYPE_DELETE

JSON Payload sent to the connector is the attribute type itself simmilar to the example below:

```
{
   "_model_class":"AttributeType",
   "id":1,
   "type":"CATEGORICAL",
   "type_name":"GENRE",
   "unit":"None"
}
```

## BaseAttribute

Whenever product base attribute is saved, updated or deleted, you can respond to it using following events:

`ATTRIBUTE_SAVE`

JSON Payload sent to the connector is the base attribute itself simmilar to the example below:

```
{
    "_model_class":"Attribute",
    "id":9,
    "type":1,
    "raw_value":"Adventure",
    "order":"None",
    "ext_attributes":[

    ],
    "deleted":false
}
```

`ATTRIBUTE_UPDATE`

JSON Payload sent to the connector is the base attribute itself simmilar to the example below:

```
{
    "_model_class":"Attribute",
    "id":9,
    "type":1,
    "raw_value":"Adventure",
    "order":"None",
    "ext_attributes":[

    ],
    "deleted":false
}
```

ATTRIBUTE_DELETE

JSON Payload sent to the connector is the base attribute itself simmilar to the example below:

```
{
    "_model_class":"Attribute",
    "id":9,
    "type":1,
    "raw_value":"Adventure",
    "order":"None",
    "ext_attributes":[

    ],
    "deleted":false
}
```

## Category

Whenever category is saved, updated or deleted, you can respond to it using following events:

CATEGORY_SAVE

JSON Payload sent to the connector is the category itself simmilar to the example below:

```
{
    "_model_class":"Category",
    "id":3,
    "parent_id":2,
    "deleted":false
}
```

CATEGORY_UPDATE

JSON Payload sent to the connector is the category itself simmilar to the example below:

```
{
   "_model_class":"Category",
   "id":3,
   "parent_id":2,
   "deleted":false
}
```

CATEGORY_DELETE

JSON Payload sent to the connector is the category itself simmilar to the example below:

```
{
   "_model_class":"Category",
   "id":3,
   "parent_id":2,
   "deleted":false
}
```

## Order

Whenever order is saved, updated or deleted, you can respond to it using following events:

ORDER_SAVE

JSON Payload sent to the connector is the order itself simmilar to the example below:

```
{
   "token":"545107d5-59ad-41d6-9f70-782828afdce2",
   "customer_email":"jdoe@example.com",
   "order":{
      "token":"545107d5-59ad-41d6-9f70-782828afdce2",
      "cart":{
         "token":"c637d1b1-d5fd-4a15-a14f-1e57b6b94a4d",
         "cart_items":[
            {
```

```
                "product_id":159858,
                "product_variant_sku":"159858-en-720p",
                "unit_price_without_vat":"170.00",
                "unit_price_incl_vat":"205.70",
                "quantity":1
             }
          ],
          "shipping_method_country":1,
          "payment_method_country":1,
          "create_at":"2023-07-18T10:44:40.222194Z",
          "status":"PENDING",
          "marketing_flag":true,
          "agreed_to_terms":true,
          "payment_id":"None"
       },
       "_model_class":"Order",
       "session_id":"cc024f1d-160a-427c-a821-1e84126eb45f"
    }
 }
```


ORDER_UPDATE

## JSON Payload sent to the connector is the order itself simmilar to the example below:


```
 {
    "token":"545107d5-59ad-41d6-9f70-782828afdce2",
    "customer_email":"jdoe@example.com",
    "order":{
       "token":"545107d5-59ad-41d6-9f70-782828afdce2",
       "cart":{
          "token":"c637d1b1-d5fd-4a15-a14f-1e57b6b94a4d",
          "cart_items":[
             {
                "product_id":159858,
                "product_variant_sku":"159858-en-720p",
                "unit_price_without_vat":"170.00",
                "unit_price_incl_vat":"205.70",
                "quantity":1
             }
          ],
          "shipping_method_country":1,
```

```
            "payment_method_country":1,
            "create_at":"2023-07-18T10:44:40.222194Z",
            "status":"PENDING",
            "marketing_flag":true,
            "agreed_to_terms":true,
            "payment_id":"None"
         },
         "_model_class":"Order",
         "session_id":"cc024f1d-160a-427c-a821-1e84126eb45f"
      }
   }
```


`ORDER_DELETE`

JSON Payload sent to the connector is the order itself simmilar to the example below:


```
{
   "token":"545107d5-59ad-41d6-9f70-782828afdce2",
   "customer_email":"jdoe@example.com",
   "order":{
      "token":"545107d5-59ad-41d6-9f70-782828afdce2",
      "cart":{
         "token":"c637d1b1-d5fd-4a15-a14f-1e57b6b94a4d",
         "cart_items":[
            {
               "product_id":159858,
               "product_variant_sku":"159858-en-720p",
               "unit_price_without_vat":"170.00",
               "unit_price_incl_vat":"205.70",
               "quantity":1
            }
         ],
         "shipping_method_country":1,
         "payment_method_country":1,
         "create_at":"2023-07-18T10:44:40.222194Z",
         "status":"PENDING",
         "marketing_flag":true,
         "agreed_to_terms":true,
         "payment_id":"None"
      },
      "_model_class":"Order",
```

```
        "session_id":"cc024f1d-160a-427c-a821-1e84126eb45f"
    }
}
```

### OrderItemComplaint

`ORDER_ITEM_COMPLAINT_CREATED`

JSON Payload sent to the connector is the order item complaint is created.

```
{
    "complaint_id": 1,
}
```

`ORDER_ITEM_COMPLAINT_UPDATED`

JSON Payload sent to the connector is the order item complaint is created.

```
{
    "complaint_id": 1,
}
```

---

## Action based triggers

Action based triggers are based on user actions. They are not related to any model, they're usually sent from storefront. Here is the list of all events that you can respond to:

### Product page

`PRODUCT_DETAIL_ENTER`

This trigger reacts to the situation when user enters product detail page. Data passed to this action trigger are simply those that are sent from the storefront so you can easily extend it. In a basic configuration it's just:

```
{
    "product_id": 1,
    "product_variant_sku": "1-en-720p",
    "session_id": "cc024f1d-160a-427c-a821-1e84126eb45f"
}
```

Make sure you don’t change these data, because they are required by recommender system.

PRODUCT_DETAIL_LEAVE

This trigger reacts to the situation when user leaves product detail page. Data passed to this action trigger are simply those that are sent from the storefront so you can easily extend it. In a basic configuration it’s just:

```
{
    "product_id": 1,
    "product_variant_sku": "1-en-720p",
    "session_id": "cc024f1d-160a-427c-a821-1e84126eb45f"
}
```

Make sure you don’t change these data, because they are required by recommender system.

PRODUCT_ADD_TO_CART

This trigger reacts to the situation when user adds product to the cart.

```
{
    "product_id": 1,
    "product_variant_sku": "1-en-720p",
    "session_id": "cc024f1d-160a-427c-a821-1e84126eb45f"
}
```

# Use cases

NotificationAPI is here to help you. It’s up to you how you will use it, since it very convinient and flexible. Here are some examples of how we imagine you can use it. If you have any other ideas, feel free to share them with us.

---

## Connecting cutom e-mail service

If you want to connect your custom e-mail service, you can do it by removing `EMAIL` events from the configuration and adding your own `HTTP` events that will send data to your custom e-mail service.

---

## Connecting company internal API

If you want to connect your company internal API, you can do it by adding `HTTP` events that will send data to your custom API. For example data about orders ( `ORDER_SAVE` , ...) and complaints.

---

## Connecting custom analytics

Make usage of action based triggers to connect your custom analytics. For example, you can track how many users are entering product detail page ( `PRODUCT_DETAIL_ENTER` ), how many of them are leaving it ( `PRODUCT_DETAIL_LEAVE` ) and how many of them are adding product to cart ( `PRODUCT_ADD_TO_CART` ).

# Disconnecting recommendation system

If you don’t want to use provided recommendation system, you can remove `RECOMMENDER` events from the configuration. However, you can still use action based triggers to connect your custom recommendation or analytical system (if you keep storefront sending those data).

ecoseller

Programming documentation

# Dashboard and Storefront

Table of contents:

# Dashboard & Storefront

The dashboard is a part of the application that is used by the administrators to manage the application. The storefront on the other hand, is a part of the application that is used by the customers to browse and buy products. Both of these parts are standalone applications built using Next.js framework and are served by the same server. Their directory structure consists of various parts, among which the most important are:

- `api` - contains various API route functions that are used to send requests to the backend
- `components` - contains a collection of React components that are used in the dashboard
- `pages` - contains React components that are used as pages in the dashboard, also ensures routing (more information about routing can be found in API routes section)
- `public` - contains static files that are used in the dashboard
- `styles` - contains styles definitions that are used in the dashboard
- `types` - contains TypeScript type definitions that are used in the dashboard
- `utils` - contains various utility functions that are used in the dashboard along with context providers and interceptors (more information about context providers and interceptors can be found in Context providers section and Interceptors section)

# Context providers

To be able to access various data in different parts of the application, we use React Context. More information about React Context can be found on the following links:

- Passing data deeply with context
- useContext

In further parts of this section, we assume that the reader is familiar with React Context and its usage from the links above.

In ecoseller, we use various context providers, and now we will describe them in more detail.

# UserProvider

`UserProvider` is a context provider that provides information about the currently logged in user to its children. It is used in both `Dashboard` and `Storefront` component, although they differ a bit in data they provide.

---

## Parameters

- `children` : React component that is wrapped by the provider

---

## Return value

- `user` : fetched data from `/user/detail/` endpoint. Consists of:

    - `email` - email of the user
    - `first_name` - first name of the user
    - `last_name` - last name of the user
    - `birth_date` - birth date of the user
    - `is_active` - whether the user is active
    - `is_admin` - whether the user is admin
    - `is_staff` - whether the user is staff
- `roles` : fetched data from `/roles/user-groups/${email}` . Consists of:
    - `name` - name of the role
    - `description` - description of the role
    - `permissions` - list of permissions of the role. Each permission consists of:
        - `name` - name of the permission
        - `description` - description of the permission
        - `type` - type of the permission
        - `model` - model to which the permission corresponds

`UserProvider` in `Storefront` only provides `user` data, while `Dashboard` provides both `user` and `roles` data.

---

# Usage example

`UserProvider` already wraps whole application in both `Dashboard` and `Storefront` components, so we can access user data in any child component. To access user data, we use `useUser` hook:

- In dashboard:

```
const ChildComponent = () => {
  ...
  const { user, roles } = useUser();
  ...
  return (
      ...
  );
};
```

- In storefront:

```
const ChildComponent = () => {
  ...
  const { user } = useUser();
  ...
  return (
      ...
  );
};
```

# PermissionProvider

As mentioned in Authorization section, ecoseller uses roles and permissions to restrict access to certain parts of the application. `PermissionProvider` is a context provider that provides information about user's permissions to its children. It is used in `Dashboard` component.

To ensure proper usage, we defined `ContextPermissions` type with permissions that may be passed to the provider. The type is defined in `dashboard/utils/context/permission.tsx` file.

---

# Parameters

- `allowedPermissions` : Array of `ContextPermissions` - permissions the user needs to have to gain access to the component
- `children` : React component that is wrapped by the provider

---

## Return value

- `hasPermission` : boolean - true if the user has all permissions from `allowedPermissions` array, false otherwise

---

## Usage example

To check whether the user has `user_add_permission` permission for adding new user, wrap the component with `PermissionProvider` :

```
<PermissionProvider allowedPermissions={["user_add_permission"]}>
    <EditableContentWrapper>
        <CreateUser />
    </EditableContentWrapper>
</PermissionProvider>
```

Now, we can check in respective component whether the user has the permission:

```
const CreateUser () => {
    ...
    const { hasPermission } = usePermission();
    ...
    return (
```

```
            ...
            <TextField
                disabled={!hasPermission}
            >
            Email
            </TextField>
            ...
        );
    };

const EditableContentWrapper = () => {
    ...
    const { hasPermission } = usePermission();
    ...
    return (
        ...
        <Button
            disabled={!hasPermission}
        >
        Save
        </Button>
        ...
    );
};
```

This will disable the `TextField` and `Button` components if the user does not have `user_add_permission` permission.

# CartProvider

`CartProvider` is a context provider that provides information about the user's cart as well as some usefull functions to its children. It is used only in `Storefront` component.

# Parameters

- `children` : React component that is wrapped by the provider

---

# Return value

- `cart` : fetched data from `/cart/storefront/<str:token>` endpoint. Consists of:
  - `token` - token of the cart
  - `cart_items` - items of the cart
  - `update_at` - date of the last update of the cart
  - `total_items_price_incl_vat_formatted` - total price of the cart including VAT
  - `total_items_price_without_vat_formatted` - total price of the cart without VAT
  - `total_price_incl_vat_formatted` - total price of the cart including VAT and shipping
  - `total_price_without_vat_formatted` - total price of the cart without VAT and shipping
  - `shipping_method_country` - id to `ShippingMethodCountry` object
  - `payment_method_country` - id to `PaymentMethodCountry` object
- `cartSize` : number of items in the cart
- `addToCart` - function for adding item to the cart
- `removeFromCart` - function for removing item from the cart
- `updateQueantity` - function for updating quantity of the item in the cart
- `clearCart` - function for clearing the cart
- `cartProductQuantity` - function for getting quantity of the product in the cart

# Functions provided by `CartProvider`

### `addToCart`

Adds item to the cart. If the item is already in the cart, it updates its quantity. Takes following parameters:

- `sku` : SKU of the product
- `qty` : quantity of the product
- `product` : product ID
- `pricelist` : pricelist ID
- `country` : country ID

### `removeFromCart`

Deletes item from the cart. Takes following parameters:

- `sku` : SKU of the product

### `updateQuantity`

Updates quantity of the item in the cart. Takes following parameters:

- `sku` : SKU of the product
- `quantity` : new quantity of the product

### `clearCart`

Clears the cart. Takes no parameters.

### `cartProductQuantity`

Returns quantity of the product in the cart. Takes following parameters:

- `sku` : SKU of the product

---

## Usage example

`CartProvider` already wraps whole application, so we can access data or functions in any child component. To do so, we use `useCart` hook:

```
const ChildComponent = () => {
    ...
    const { cart, cartSize, addToCart, removeFromCart, updateQuantity, clearCart,
cartProductQuantity } = useCart();
    ...
    return (
        ...
    );
};
```

# CookieProvider

`CookieProvider` is a context provider that provides information about the user's cookies as well as some usefull functions to its children. It is used only in `Storefront` component.

---

## Parameters

- `children` : React component that is wrapped by the provider

# Return value

- `cookieState` - set consisting of set of boolean flags:
  - `neccessaryCookies` - whether the user has accepted neccessary cookies
  - `preferenceCookies` - whether the user has accepted preference cookies
  - `statisticalCookies` - whether the user has accepted statistical cookies
  - `adsCookies` - whether the user has accepted ads cookies
  - `openDisclaimer` - whether to show cookie disclaimer
- `setCookieState` - function for setting cookie state
- `setCookieSettingToCookies` - function for setting cookie setting to cookies
- `toggleDisclaimer` - function for toggling cookie disclaimer

---

# Functions provided by `CookieProvider`

### `setCookieState`

Sets cookie state. Takes following parameters:

- `key` : type of cookie
- `value` : boolean value to set to the cookie

### `setCookieSettingToCookies`

Sets cookie setting to cookies. Takes following parameters:

- `allTrue` : whether all cookies are accepted

### `toggleDisclaimer`

Toggles cookie disclaimer. Takes following parameters:

- `value` : whether to show cookie disclaimer

---

## Usage example

`CookieProvider` already wraps whole application, so we can access data or functions in any child component. To do so, we use `useCookie` hook:

```
const ChildComponent = () => {
    ...
    const { cookieState, setCookieState, setCookieSettingToCookies,
toggleDisclaimer } = useCookie();
    ...
    return (
        ...
    );
};
```

# CountryProvider

`CountryProvider` is a context provider that provides information about the country that is currently set by the user, as well as some usefull functions to its children. It is used only in `Storefront` component.

---

## Parameters

- `children` : React component that is wrapped by the provider

## Return value

- `country` : object representing country. Consists of:
  - `code` - id of the country
  - `name` - name of the country
  - `locale` - locale of the country
  - `default_price_list` - id of the default price list of the country
- `countryList` : list of `country` objects - all available countries
- `setCountryCookieAndLocale` - function for setting country cookie and locale

---

## Functions provided by `CountryProvider`

### `setCountryCookieAndLocale`

Sets country cookie and locale. Takes following parameters:

- `countryCode` : code of the country

---

## Usage example

`CountryProvider` already wraps whole application, so we can access data or functions in any child component. To do so, we use `useCountry` hook:

```
const ChildComponent = () => {
    ...
```

```
    const { country, countryList, setCountryCookieAndLocale } = useCountry();
    ...
    return (
        ...
    );
};
```

# RecommenderProvider

`RecommenderProvider` is a context provider that provides information about the user's recommender session as well as some usefull functions to send either recommender event or retrive recommendations. It is used only in `Storefront` component. This context provider creates a new recommender session for each user (if it does not exist yet) and stores it in the cookie storage under `rsSession` key.

---

## Parameters

- `children` : React component that is wrapped by the provider

---

## Recommender events

Recommender events are used to track user's behaviour on the website. They are sent to the recommender server and are used to generate recommendations. They are defined in `RS_EVENT` variablle. Each event has its own payload. More information about recommender events can be found on the following links: #TODO: add links to recommender documentation

# Recommender situations

Recommender situations are used to define the context in which the user is currently in. They are defined in `RS_RECOMMENDATIONS_SITUATIONS` variable. Each situation has its own payload. More information about recommender situations can be found on the following links: #TODO: add links to recommender documentation

---

# Return value

- `session` : string `uuid` session id
- `sendEvent` - function for sending recommender event
- `getRecommendations` - function for getting recommendations for given situation

---

# Functions provided by `RecommenderProvider`

User does not have to send session id directly, it's injected to each request automatically.

### sendEvent

Sends recommender event. For example it can be information about adding product to the cart, or leaving product page. Takes following parameters:

- `event: RS_EVENT` : event to send

- `payload: any` : payload of the event (depends on the event) It returns noting (void).

### `getRecommendations`

Gets recommendations for given situation. It returns recommmended products for the given session. Takes following parameters:

- `situation: RS_RECOMMENDATIONS_SITUATIONS` : situation for which to get recommendations
- `payload: any` : payload of the situation (depends on the situation - might be product_id, etc.)

---

## Usage example

`RecommenderProvider` already wraps whole application, so we can access data or functions in any child component. To do so, we use `useRecommender` hook:

```
const ChildComponent = () => {
    ...
    const { session, sendEvent, getRecommendations } = useRecommender();
    ...
    return (
        ...
    );
};
```

# Interceptors

Interceptors are used to intercept requests and responses before they are handled by the application. In ecoseller, we use them to add authorization token and other data to requests and to handle errors. We use `axios` library for handling requests and responses. More information about interceptors can be found on the following links:

- Axios - Getting started
- Axios interceptors

In further parts of this section, we assume that the reader is familiar with `axios` library and its usage from the links above. Interceptors in the `Dashboard` and `Storefront` differ a bit, so we will describe them separately.

# Request interceptor - Dashboard

In the `Dashboard` , we use request interceptor to add authorization token to requests. The interceptor is defined in `dashboard/utils/interceptors/api.ts` file. Fisrtly, we define `api` axios instance with base url and headers:

```
export const api = axios.create({
  baseURL,
  headers: {
    "Content-Type": "application/json",
  },
  withCredentials: true,
});
```

Then, we add request interceptor to the `api` instance:

- for the request

```
  api.interceptors.request.use((config) => {
  let access = "";
  let refresh = "";
```

```
  if (isServer()) {
    access = getCookie("accessToken", { req, res }) as string;
    refresh = getCookie("refreshToken", { req, res }) as string;
  } else {
    access = Cookies.get("accessToken") || "";
    refresh = Cookies.get("refreshToken") || "";
  }

  if (access) {
    config.headers.Authorization = `JWT ${access}`;
  }
  return config;
  });
```

- for the response

```
  api.interceptors.response.use(
  (response) => {
    return response;
  },
  (error: AxiosError) => {
    // check conditions to refresh token
    if (
      (error.response?.status === 401 || error.response?.status === 403) &&
      !error.response?.config?.url?.includes("user/refresh-token") &&
      !error.response?.config?.url?.includes("user/login")
    ) {
      return refreshToken(error);
    }
    return Promise.reject(error);
  }
  );
```

Where the `refreshToken` is a function responsible for fetchiing a new access token and retrying the request. It is defined in `dashboard/utils/interceptors/api.ts` file.

# Request interceptor - Storefront

In the `Storefront`, we use request interceptor to add authorization token and country locale to requests. The interceptor is defined in `storefront/utils/interceptors/api.ts` file. Axios instance in the `Storefront` is defined similarly as in the `Dashboard` (see above). The difference is in the request interceptor, where we also add a country locale to the `Accept-Language` header:

```
api.interceptors.request.use((config) => {

  ... // similar to the Dashboard

  // set locale (if present)
  const locale = getLocale();
  if (locale) {
    config.headers["Accept-Language"] = locale;
  }

  return config;
});
```

# API Routes

Next.js API Routes provide a convenient way to create server-side endpoints within your Next.js application. These API routes allow you to handle server-side logic and expose custom API endpoints. This section of the documentation explores the usage of Next.js API Routes in the context of ecoseller.

In the Ecoseller architecture, it is recommended to avoid exposing the backend service directly to the client. By using Next.js API Routes, you can encapsulate server-side logic within your Next.js application, ensuring a secure and controlled environment for handling data and performing server-side operations. This approach provides several benefits:

- **Security**: By not exposing the backend service to the client, you reduce the risk of potential security vulnerabilities. It prevents unauthorized access or tampering of sensitive data and operations that should be restricted to server-side execution only.
- **Imporved control**: Keeping the backend service separate from the client-side code gives you better control over the server-side operations and access to the underlying data. It allows you to enforce business logic, perform validations, and apply necessary security measures within the API routes.
- **Simplified Architecture**: The separation of concerns between the client-side code and the server-side logic simplifies the overall architecture. It enables cleaner code organization and promotes modularity, making it easier to maintain and scale the application in the long run.

This approach also helped us with cookie handling and token refreshing. Setting cookie client side caused some problems with refreshing tokens, so we decided to set cookies in API routes instead.

# API Routes in the Dashboard

All API routes can be found in the `dashboard/pages/api` directory. All those routes use simmilar logic and send requests to the backend via `api` interceptor.

```
export const productDetailAPI = async (
  method: HTTPMETHOD,
  id: number,
  req?: NextApiRequest,
  res?: NextApiResponse
) => {
  if (req && res) {
    setRequestResponse(req, res);
  }
```

```
  const url = `/product/dashboard/detail/${id}/`;

  switch (method) {
    case "GET":
      return await api
        .get(url)
        .then((response) => response.data)
        .then((data: IProduct) => {
          return data;
        })
        .catch((error: any) => {
          throw error;
        });
    case "DELETE":
      return await api
        .delete(url)
        .then((response) => response.data)
        .then((data: IProduct) => {
          return data;
        })
        .catch((error: any) => {
          throw error;
        });
    case "PUT":
      const body = req?.body;
      if (!body) throw new Error("Body is empty");
      return await api
        .put(url, body)
        .then((response) => response.data)
        .then((data: IProduct) => {
          return data;
        })
        .catch((error: any) => {
          throw error;
        });

    default:
      throw new Error("Method not supported");
  }
};

const handler = async (req: NextApiRequest, res: NextApiResponse) => {
```

```
  /**
   * This is a wrapper for the product detail api in the backend
   */

  const { id } = req.query;
  const { method } = req;

  if (!id) return res.status(400).json({ message: "id is required" });
  if (Array.isArray(id) || isNaN(Number(id)))
    return res.status(400).json({ message: "id must be a number" });

  if (method === "GET" || method === "PUT" || method === "DELETE") {
    return productDetailAPI(method, Number(id), req, res)
      .then((data) => res.status(200).json(data))
      .catch((error) => res.status(400).json(null));
  } else {
    return res.status(400).json({ message: "Method not supported" });
  }
};

export default handler;
```

# API Routes in the Storefront

All API routes can be found in the `storefront/pages/api` directory. All those routes use simmilar logic and send requests to the backend via `api` interceptor.

```
export const cartAPI = async (
  method: HTTPMETHOD,
  req: NextApiRequest,
  res: NextApiResponse
) => {
  if (req && res) {
    setRequestResponse(req, res);
  }

  const url = `/cart/storefront/`;

  switch (method) {
    case "POST":
```

```
        return await api
          .post(url, req.body)
          .then((response) => response.data)
          .then((data) => {
            return data;
          })
          .catch((error: any) => {
            throw error;
          });
      default:
        throw new Error("Method not supported");
    }
  };

  const handler = async (req: NextApiRequest, res: NextApiResponse) => {
    const { method } = req;

    if (method == "POST") {
      return cartAPI("POST", req, res)
        .then((data) => res.status(201).json(data))
        .catch((error) => res.status(400).json(null));
    }
    return res.status(404).json({ message: "Method not supported" });
  };

  export default handler;
```

ecoseller

Programming documentation

# Recommender system

Table of contents:

Recommender system is a web application that provides recommended products and their variants to the backend of the **ecoseller**.

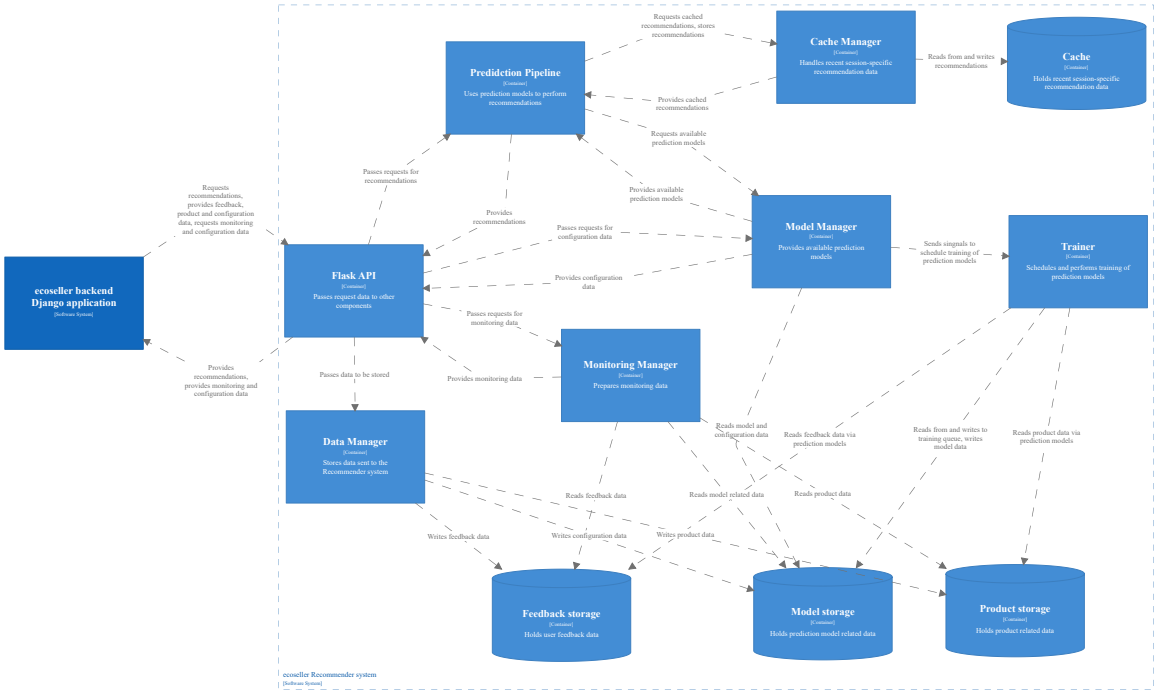
This application is split into two docker containers - web server providing recommendations ( `recommender_system` ) and trainer ( `recommender_system_trainer` ). The web server uses models prepared by the trainer to provide relevant products and their variants. Only the web server has exposed API that is used by the backend. Trainer runs does not communicate with other components

of **ecoseller**. It reads Recommender system's database and checks if any model needs training. The trainer trains models if needed and prepares them for the web server to be used.

# Architecture

The Recommender system's web server is implemented as a Flask application. The trainer is a simple Python process as no API endpoints are exposed.

Dependency injector is used throughout the application to handle proper initialization and usage of all the components of the Recommender system. The web server includes a container that defines all the components that are used via dependency injection. These components include storages and managers (*Cache manager*, *Data manager*, *Model manager*, *Monitoring manager* and *Prediction pipeline*).




# API

There are several API endpoints exposed by the Recommender system.

---

# Healthcheck

There is a `GET` healthcheck endpoint at `/` that is used by Docker to check if Recommender system's web server is up and running. Trainer starts running after this endpoint of the web server responds so that database migrations are run only once and only by the web server.

---

## Storing objects

There are two endpoints that handle storing objects to the Recommender system.

One handles only one object, the second saves list of objects, both are `POST` endpoints, their paths are `/store_object` and `/store_objects`, respectively.

The data sent to this endpoint contain the class name of the object being saved along with the object's representation in JSON. For more information about the data sent to this endpoint, see the Data manager section.

---

## Providing recommendations

The Recommender system provides the recommendations via two `POST` endpoints.

Endpoint at `/predict` returns list of product variant SKUs ordered by their relevance to the request.

Endpoint at `/predict/product_positions` returns a dictionary mapping product ID to the position of its most relevant product variant.

Only the category list ordering calls the second endpoint in order to create Django query that orders the products to be displayed in the category list.

These endpoints expect the following data:

```
{
    "recommendation_type": str,  # "HOMEPAGE" | "PRODUCT_DETAIL" |
"CATEGORY_LIST" | "CART"
    "session_id": str,
    "user_id": Optional[int],
    "category_id": int,
    "variants": List[str],
    "variants_in_cart": List[str],
    "limit": Optional[int],
}
```

Where `recommendation_type`, `session_id`, `user_id` and `limit` are expected for all recommendations. `category_id` is filled only for category list ordering, `variants` for product detail recommendations (it contains SKUs of all variants of the current product) and `variants_in_cart` for cart recommendations (it contains product variant SKUs in the current cart).

---

## Providing monitoring data

Monitoring data are obtained from the Recommender system via `GET` endpoint at `/dashboard`. It expects two arguments - `date_from` and `date_to`, that specify for which time period the monitoring data should be displayed.

This endpoint returns all the data that are needed to be displayed on the dashboard. More details about the monitoring data are described in monitoring manager and dashboard's recommender system sections.

# Data manager

Data manager handles all data coming to the Recommender system. It stores objects to the corresponding storage inside the Recommender system.

Data manager obtains data from the request, parses API model(s) from them, converts them to Stored models and then saves those Stored models to the appropriate storage.

---

## API models

API models are objects, that correspond to the format which is sent via API. Each API model is a subclass of Pydantic's `BaseModel` .

Parsing these models by the Data manager is handled by mapping a field `_model_class` to the corresponding API model's type and then parsing the data using `parse_obj` method.

Each API model has `save` method, that performs the conversion to Stored model and saves that model to the storage.

API models include the following objects:

- `Attribute`
- `AttributeType`
- `Category`
- `Config`
- `Order`
- `Product`
- `ProductAddToCart`
- `ProductDetailEnter`

- `ProductDetailLeave`
- `ProductPrice`
- `ProductTranslation`
- `ProductType`
- `ProductVariant`
- `RecommendationView`
- `Review`

---

# Stored models

Stored models represent objects that are stored into a storage, but they are storage-implementation independent. This means that changing storage from SQL to filesystem does not affect these objects in any way. All operations with their storage are done by calling the appropriate methods on that storage. Each model has its storage as a `_storage` attribute that is set during the model's initialization.

These models are also subclasses of the Pydantic's `BaseModel`, this makes it easy to work with SQLAlchemy ORM.

## Immutable

Some objects are immutable, editing these objects raises `TypeError`. These objects can only be created once.

## Many-to-many relations

Several objects represent *many-to-many* relations, for example `ProductModel` and `ProductVariantModel` have *many-to-many* relation among them.

Related models can be obtained by calling the `get_target_model_class` of the `ManyToManyRelationMixin`, which provides you with the `target_model_class`, and primary key field names of both classes of this relation. This allows the storage to implement just one general method to obtain *many-to-many* related objects.

# Storages

The Recommender system uses several storages to store the objects it needs to provide recommendations.

Each storage is initialized as a Singleton in the application's container and is accessed via dependency injection.

---

## ORM

Working with SQL (via SQLAlchemy) takes advantage of its ORM functionality. Tables are defined as subclasses of SQLAlchemy's `DeclarativeBase`. Mapping from Stored models to those table definitions is performed by `SQLModelMapper`, each table object has `origin_model` attribute of its `Meta` class to tell the mapper which Stored model to map to which table.

---

## Cache

Cache storage is implemented as a filesystem storage. It saves category list recommendations performed by the Recommender system.

It has limited size since old data will not be used once a session ends.

More information about this storage's usage is described in the Cache manager section.

---

# Feedback

Feedback storage is implemented as a PostgreSQL database. All models regarding feedback data are stored here. They include the following:

- `ProductDetailEnterModel`
- `ProductDetailLeaveModel`
- `ProductAddToCartModel`
- `ReviewModel`
- `RecommendationViewModel`

There are two other models stored in this storage: `PredictionResultModel` and `SessionModel`. Prediction results are stored here so that monitoring manager can evaluate user feedback data based on the provided recommendations by applying SQL joins inside a single database.

---

# Model

Model storage includes models that are related to the prediction models in general. The following models are stored here:

- `ConfigModel` that contains configuration of the whole Recommender system
- `LatestIdentifierModel` that contains the identifier of the latest trained model for each of the prediction models
- `TrainerQueueItemModel` that represents item in the queue of the prediction models to be trained
- `TrainingStatisticsModel` that contains statistics describing each training performed by the Recommender system

Model storage is implemented as a PostgreSQL database.

---

# Product

Product storage is a PostgreSQL database containing all product-related models. They are stored in the same format as in Django backend application, some models or fields are missing as they are not needed by the Recommender system.

Converting those objects to a more Recommender system-friendly structure is done during training in order to keep the complexity of storing models as low as possible.

---

# Prediction model-specific

Some prediction models have their own storages implemented, namely *EASE* (filesystem), *GRU4Rec* (filesystem) and *Similarity* (PostgreSQL). These storages are used to store the models parameters that are loaded when given model is being used. *Similarity* model also saves distances of all product variants into the database and performs ordered queries on those data during prediction.

More information about these storages is provided in the sections describing the prediction models themselves.

---

## Migrations

All SQL storages use Alembic to manage migrations. Each storage has its own `alembic.ini` file and `alembic` folder containing its versions.

Each `alembic.ini` file contains path to the migration script of the corresponding file that runs the migrations ( `env.py` ), this file needs to have `target_metadata` variable assigned to the proper base class in order to generate the correct migrations.

---

## Changing storage type

In order to change storage type (for example changing *EASE* storage from filesystem to SQL) the following steps are necessary:

1. Create a new subclass of the `SQLBase` class located in the file `recommender_system/storage/sql/models/base.py`.
2. Define the tables by subclassing the created base class in a newly created file `recommender_system/storage/sql/models/ease.py`.
3. Create a new subclass of the appropriate *Abstract storage class* (in this case `AbstractEASEStorage`) and implement all the defined methods. Make this class a subclass of `SQLStorage` as well to be able to use SQL-specific functionality.
4. Change the Dependency injector's Singletion initialization in the application's container to this new class.
5. Create new database for this storage in the `../setup_database.sql` file, add connection string to environment variables files and use this connection string when initializing the storage in the container.
6. Add this storage to the `recommender_system/scripts/migrate.py` file so that migrations are applied when the **ecoseller** starts.
7. Copy `alembic.ini` file and `alembic` folder from different SQL storage and adjust paths and metadata as described in the migrations section above. Delete all version files.
8. Generate migrations by running `python3 -m recommender_system.scripts.makemigrations {storage_name}` (`{storage_name}` in this case is `ease_storage`).

*All paths above are relative to `src/recommender_system/app`.*

To add a new storage, use similar process to the one described above, it does not differ much.

# Prediction models

The Recommender system contains several prediction models that perform the recommendations. This section describes the models, their usage is described in the Prediction pipeline section.

Each model's task is to select a subset of product variant from the Product storage. Input differs based on the type of recommendation, the differences are described in the Prediction pipeline section.

Some of the models can not be used in all situations based on their properties.

---

## Dummy (level 0)

This is the simplest model, which is used only if an error occurs when more complex models perform their prediction.

Dummy model returns randomly selected subset of product variants.

This model does not need any training.

---

## Selection (level 1)

This model is also randomized, but users can select product variants that should be recommended more often. Each product variant has its recommendation weight specified, this value is used as weight when the random sampling is performed.

This model does not need any training.

## Popularity (level 2)

Popularity-based model tends to recommend popular product variants more. The popularity is represented by the number of orders of that product variant.

The recommendations are sampled, similarly to the Selection model, popularity is used as the sampling weight.

This model does not need any training.

## Similarity (level 3)

Similarity-based model is a content-based model. It recommends the most similar product variants to the product variant passed as input.

Training computes distances of all pairs of product variants and saves them to the Similarity storage. Each record contains identifier of the model, SKUs of both product variants and their distance.

This model selects the closest product variants to the given product variant during prediction.

During training, each product variant is represented by two vectors. One is in numerical attribute space, the second in categorical attribute space.

Vectors contain values of all existing attributes so that product variants of different product types can be compared.

If a product variant has no value of a numerical attribute, the average value is used.

If a product variant has no value of a categorical attribute, the most common value is used.

The distance of two product variants is computed as addition of Euclidean distances in numerical and categorical space. Distance in each space is multiplied by a coefficient that decreases when the product variants share more attributes. This coefficient is defined as the size of union of the attributes defined for the product variants divided by the size intersection of the attributes defined for the product variants.

If product variant $p$ has $m$ attributes, product variant $q$ has $n$ attributes and they share $k$ product variants, then the coefficient is $\frac{m+n-k}{k}$. If $k = 0$, then the coefficient is $m + n$.

---

## GRU4Rec (level 4)

This is a session-based model based on this article.

It uses a recurrent neural network. The input of the network is a vector representing the current session. Each value of the input vector represents one product variant. The session is represented as a list of visited product variants, each visited product variant has value of $x^k$ where $x \in [0, 1]$ is a parameter of this model and $k$ is the number of product variants visited after that product variant.

The neural network used here consists of three layers: embedding, GRU and output. Embedding and output layers are linear.

The output of the network consists of scores for each product variant, the product variants with the highest scores are selected.

---

## EASE (level 5)

EASE is a collaborative-filtering model. This model approximates a user-item rating matrix. Value at position $(i, j)$ is user $i$'s rating of item $j$.

The EASE algorithm predicts the user's preferences by a dot product the user's representation and a matrix it computes during training. The user is represented by a vector of ratings of all the project variants - it is basically the same as a row in the user-item rating matrix.

The output contains estimated ratings of all the product variants, the top-rated ones are recommended to the user.

The training computes the matrix that is used during prediction, it is done by inversion of a slightly modified user-item rating matrix. The resulting

matrix's diagonal is set to 0.

---

# Adding new prediction model

New prediction models can be implemented and added to be used by the Recommender system by following the steps below:

1. Create class of your model by subclassing `AbstractPredictionModel` and implementing its abstract methods.
2. Add your class to `ModelManager`'s `get_all_models` method.
3. Add your model to cascade on dashboard, so it can be used by the Recommender system.

# Prediction pipeline

Prediction pipeline takes care of recommending products. It consists of three phases - retrieval, scoring and ordering.

## Retrieval

Retrieval phase of prediction pipeline selects product variants to be considered when recommending product variants to the user. It typically uses simple models so that the time complexity is kept low.

The default number of selected product variants is 1000. This step is different for category list recommendations - product variants of given category are selected in that case.

## Scoring

Scoring phase orders the retrieved product variants based on scores obtained from the used prediction model.

### Ordering

Ordering phase of the prediction pipeline re-orders the top product variants obtained by the scoring phase.

It takes the top $k$ (default is 50) product variants and maximizes the *intra list distance* among them. This phase uses the Similarity prediction model, this phase is skipped if the model is not ready.

---

## Cache manager

The recommendations provided by the prediction pipeline are cached. Only category list recommendations are cached. Only the category that was visited last by a user is cached.

The cache size can hold up to 1000 items by default. This value can be changed via `RS_CACHE_SIZE` environment variable.

---

## Model manager

The models used in the retrieval and scoring phases are selected by *Model manager*. The model is selected from the corresponding cascade. Cascade is an ordered list of models where if the first one is not available, the second is

used. If no model is available, Dummy model is used - this one is available all the time.

There are cascades for all recommendation situations (homepage, product detail, category list and cart) and both phases of the pipeline (without retrieval for category list).

## Monitoring manager

Monitoring data to be displayed on dashboard are prepared by monitoring manager. It simply selects prediction and training-related data from the database.

## Trainer

Trainer runs in a separate container to keep the Recommender system's response fast.

It checks a database containing items representing training requests and once there are new requests, it starts training the corresponding prediction model.

It also schedules models for training based on signals sent from Model manager, the Python object inside the Recommender system's server takes care of this in order to keep this functionality inside single object.

# Dockerization

The recommender system consists of one server (Recommender system), one trainer and one PostgreSQL instance.

The Recommender system uses Gunicorn in production mode, default Flask WSGI otherwise.

Only the recommender system's server performs database migrations, this is possible due to healthchecks of PostgreSQL and the Recommender system's server. PostgreSQL instance is started first, once it is up and running, the Recommender system's server starts. It performs migrations and starts the server. Once the server is started and starts responding, the trainer is started with all its dependencies ready.

# Configuration

The Recommender system is configured via `ConfigModel` object that contains options for the Recommender system as a whole as well as for individual prediction models.

This object is editable from dashboard, each version is saved to the database with the appropriate timestamp to keep track of changes. The most current version is used each time the configuration is being accessed.

# Importing data

It is possible to import data from two datasets to the Recommender system to test its offline performance.

Both datasets fill the storages with product and feedback data.

## Demo

Demo data are imported to the Recommender system using `mock_data_rs_feedback.sql` and `mock_data_rs_products.sql` scripts during container initialization. These contain product and feedback data representing a subset of MovieLens dataset.

Product variant sequences visited by the users were generated randomly.

These data are imported when the `demo` version of *ecoseller*'s `docker-compose` file is used:

```
docker compose -f docker-compose.demo.yaml up
```

This dataset contains ~ 1000 products, ~ 2000 product variants and ~ 500 users.

# Retailrocket

Retailrocket RS dataset can be imported to the Recommender system as well.

It is necessary to save the files of the dataset into the folder `src/recommender_system/data`. Running the script `recommender_system/scripts/fill_data.py` saves those data to the database.

The whole Retailrocket dataset contains over 400k products and over 1M users.

# Unit testing

The Recommender system contains several unit tests to ensure proper functionality of individual components of the Recommender system.

The tests are written using `pytest` framework.

ecoseller

Programming documentation

# Supportive services

Table of contents:



This section focuses on the seamless integration of additional supportive services such as Elasticsearch, Redis, and PostgreSQL with the Django backend, enhancing the functionality and performance of the ecoseller platform.

# Elasticsearch

Elasticsearch is a powerful search and analytics engine that enables fast and efficient full-text search capabilities in ecoseller. By integrating Elasticsearch with

the Django backend, users can benefit from advanced search features, including filtering, ranking, and suggestions. Elasticsearch enhances the user experience by providing quick and accurate search results, making it an integral part of ecoseller's search functionality.

# Integration

The integration of Elasticsearch with the Django backend is achieved using the Django Elasticsearch DSL and Django Elasticsearch DSL DRF packages. The Django Elasticsearch DSL package provides a simple API for defining Elasticsearch indexes, while the Django Elasticsearch DSL DRF package provides a set of classes and filters for integrating Elasticsearch with the Django REST framework. The integration of these packages is done in `src/backend/core/core/settings.py` where are defined the Elasticsearch connection settings and the Elasticsearch index settings based on the `env` variables. We use `USE_ELASTIC` variable to enable or disable the Elasticsearch integration. If `USE_ELASTIC` is set to `True`, the Elasticsearch integration is enabled, otherwise it is disabled. Indexes are created only if the Elasticsearch integration is enabled and are defined as `ELASTICSEARCH_INDEX_NAMES`.

# Analyzers

The following analyzers are defined in ecoseller:

- `czech_autocomplete_hunspell_analyzer` - Czech analyzer for autocomplete based on the Hunspell dictionary
- `slovak_autocomplete_hunspell_analyzer` - Slovak analyzer for autocomplete based on the Hunspell dictionary
- `general_autocomplete_hunspell_analyzer` - General analyzer for autocomplete based on the Hunspell dictionary for english

Please refer to `src/backend/core/search/analyzers.py` for more details. Hunspell dictionaries are downloaded on the build of Elasticsearch container.

# Indexes

Ecoseller is indexing products so that users can search in products. The following index is defined for products:

---

### `products`

This index is used to store the product information. It is defined in `src/backend/core/products/documents.py` and is based on the `Product` model defined in `src/backend/core/products/models.py`. Following data are stored:

- `id` - the product id
- `title` - dictionary with the product title in different languages (and different analyzers)
- `short_description` - dictionary with the product short description in different languages (and different analyzers)
- `attribute_list` - list of dictionaries with the product attributes sepperated by comma in different languages (and different analyzers)

# Search

In order to retrive indexed data, ecoseller uses `PaginatedElasticSearchAPIView` in the `src/backend/core/search/views.py` for HTTP requests. This view has to be extended by a view that defines the `serializer_class` and `document_class` attributes. Also it's neccessary to define custom `generate_q_expression` method that will be used for querying. For example, we can see `SearchProducts` defined in `src/backend/core/products/views.py`.

```
class SearchProducts(PaginatedElasticSearchAPIView):
    serializer_class = ProductStorefrontListSerializer
    document_class = ProductDocument
```

```
    serializer_as_django_model = True

    def generate_q_expression(self, query):
        return Q(
            "multi_match",
            query=query,
            fields=[
                "id^10.0",
                f"title.{self.langauge}^5.0",
                f"short_description.{self.langauge}^3.0",
                f"attribute_list.{self.langauge}^2.0",
            ],
        )
```

For more information about creating `Q` expressions, please refer to Elasticsearch DSL documentation here.

# Indexing products to Elasticsearch

To ensure efficient product searches and recommendations within Ecoseller, it is crucial to index your products in Elasticsearch. ecosellerprovides a convenient CLI command within the backend container to perform this indexing process.

To index your products using the CLI command, follow these steps:

1. Access the `backend` container: If you are running ecoseller locally using Docker, open your terminal and navigate to the ecosellerproject directory. Use the following command to access the `backend` container:

   ```
   docker exec -it <your_backend_container_id_or_name> /bin/bash
   ```

2. Run the indexing command: Once inside the `backend` container, run the following command to index the products in Elasticsearch:

   ```
   python3 manage.py search_index --rebuild
   ```

   This command triggers the indexing process, where the products will be parsed, analyzed, and stored in Elasticsearch for efficient searching and recommendation functionalities.

Note: Ensure that you are in the correct directory within the backend container (usually the project's root directory) before executing the command.

The indexing process may take some time, depending on the size of your product database. Once the process is complete, your products will be fully indexed and ready for efficient searching and recommendation generation within Ecoseller.

---

## Automation with CRON job

You can also automate the indexing process by scheduling a CRON job to run the indexing command at specified intervals. This ensures that your Elasticsearch index stays up to date with any changes in your product database. Set up a CRON job with the following command:

```
0 2 * * * docker exec <your_backend_container_id_or_name> python3 manage.py search_index --rebuild
```

# Turning off Elasticsearch

If you no longer wish to use Elasticsearch in your ecosellersetup, you can easily disable it by adjusting the environment variables and stopping the Elasticsearch container. Follow the steps below to turn off Elasticsearch:

1. Update environment variables: (please see dedicated section for environment variables in the installation guide) Set the `USE_ELASTIC` variable to `0` in the `backend` env file.
2. Stop the Elasticsearch container
3. Restart the `backend` container

With these steps completed, Elasticsearch will be disabled in your ecosellersetup. However, please note that this will also disable the fast search functionality within Ecoseller. Therefore, it is recommended to keep Elasticsearch enabled for user experience.

# Redis

Redis, an in-memory data structure store, is utilized in ecoseller. By integrating Redis with the Django backend, ecoseller leverages its key-value store capabilities to efficiently manage and process background tasks.

# Integration

The integration of Redis and mainly Redis Queue with the Django backend is achieved using the Django RQ. There's a special flag defined as `env` variable `USING_REDIS_QUEUE` that enables or disables the Redis Queue integration.

# Redis Queue

Redis Queue is used for running background tasks. For example, we can take a look at `src/backend/core/emails/email/base.py` with class `Email` that is used to inherit from when creating new email classes. This class defines `send` method that is used to send emails in the background. Since sending e-mails from main thread can take a long time, we use Redis Queue to send emails in the background.

```
def send(self):
    self.generate_subject()
    self.generate_context()

    queue = django_rq.get_queue(
        "high", autocommit=True, is_async=True, default_timeout=360
    )

    args = (
        self.subject,
        "",
        settings.EMAIL_FROM,
        self.recipient_list,
    )

    kwargs = {"html_message": self.generate_msg_html()}

    if self.use_rq:
        queue.enqueue(
            send_mail,
            args=args,
```

```
            kwargs=kwargs,
            meta=self.meta,
        )
    else:
        send_mail(*args, **kwargs)
```

The message is simply attached to the queue and the queue is processed in the background.

## Worker

In order to process the queue, we need to run a worker. We can do that by running `python manage.py rqworker` command. This command will start a worker that will process the queue. However single worker is already included in all Docker Compose files supplied with ecoseller.

# PostgreSQL

PostgreSQL, a robust and feature-rich open-source relational database, forms the core of ecoseller's data storage and management. By integrating PostgreSQL with the Django backend, ecoseller ensures secure, reliable, and scalable storage for critical data, such as product information, user details, and order history. PostgreSQL's advanced features, including ACID compliance and support for complex data structures, make it an excellent choice for managing structured data in ecoseller.

## Integration

The integration of PostgreSQL with the Django backend is achieved using the psycopg2 package. The PostgreSQL connection settings are defined in `src/backend/core/core/settings.py` based on the `env` variables.

Since Django ORM is used for database management, the integration of PostgreSQL is done automatically by Django.

ecoseller

Programming documentation

# Deployment

Table of contents:



Ecoseller offers deployment using Docker and Docker Compose, providing a consistent and reproducible environment across different deployment scenarios. This section of the programming documentation outlines parts and logic behind included Docker Compose files: `docker-compose.yml`, `docker-compose.demo.yml`, and `docker-compose.prod.yml`.

## Docker Compose files

Docker Compose is a powerful tool that allows you to define and manage multi-container Docker applications. It simplifies the process of deploying and running complex applications by defining the services, networks, and volumes required for your application in a single YAML file. With Docker Compose, you can easily orchestrate the deployment of your application stack, specify environment variables, and configure inter-container communication. If you are not familiar with Docker Compose, we recommend you to read the official documentation.

All our docker-compose files define the same services (except for `docker-compose.yaml` - it doesn't contain Nginx reverse proxy), but with different configurations. The services are:

- `backend` - Core backend service, written in Python using Django framework. It can be accessed via port 8000 and has three targets - `development`, `demo`, and `production`.
- `postgres_backend` - PostgreSQL database for backend service. It can be accessed via port 5433.
- `redis` - Redis database for backend service. It can be accessed via port 6379.
- `rq-worker` - RQ worker for backend service. It picks up tasks from Redis database and executes them. It's basicaly a copy of `backend` service, but with different command ran on startup.
- `elasticsearch` - ElasticSearch database for backend service. It can be accessed via port 9200.
- `frontend_dashboard` - Dashboard service written in Next.js. It can be accessed via port 3030. It has two targets - `development` and `production`. The `developemnt` target is used for local development (`npm run dev`), while `production` target runs the service in production mode (`npm run build && npm run start`).
- `frontned_storefront` - Dashboard service written in Next.js. It can be accessed via port 3031. It has two targets - `development` and `production`. The `developemnt` target is used for local development (`npm run dev`), while `production` target runs the service in production mode (`npm run build && npm run start`).
- `postgres_rs` - PostgreSQL database for recommender system. It can be accessed via port 5434.

- `recommender_system` - Recommender system service written in Python using Flask framework. It can be accessed via port 8001 and has two targets - `development` and `production`.
- `recommender_system_trainer` - Short term living service that trains recommender system. It's basicaly a copy of `recommender_system` service, but with different command ran on startup.

# Dockerfile

Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession. This page describes the commands you can use in a Dockerfile. For the full reference on Dockerfile, we recommend you to read the official documentation.

Here are some interesting parts of our Dockerfile that are worth mentioning.

---

## Multi-stage builds

Multi-stage builds are a new feature requiring Docker 17.05 or higher on the daemon and client. Multistage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain. They allow you to go from a Dockerfile that has multiple `FROM` instructions to a single `FROM` instruction. You can read more about multi-stage builds in the official documentation. We used this feature on both frontend services, `frontend_dashboard` and `frontend_storefront` and `backend` service. Here is an example of `frontend_dashboard` Dockerfile:

```
# dependencies
FROM node:18-alpine3.14 as dependencies
WORKDIR /usr/src/app
COPY ./package.json .
COPY ./package-lock.json .

RUN npm install -g npm@8.8.0
RUN npm install --legacy-peer-deps

# builder
FROM dependencies as builder
WORKDIR /usr/src/app
COPY . .

# development environment
FROM builder as development
WORKDIR /usr/src/app
ENV NODE_ENV development
CMD ["npm", "run", "dev"]

# production environment
FROM builder as production
WORKDIR /usr/src/app
ENV NODE_ENV production
RUN npm run build
CMD ["npm", "start"]

# demo environment
FROM builder as demo
WORKDIR /usr/src/app
ENV NODE_ENV production
RUN npm run build
CMD ["npm", "start"]
```

As you can see from the example, we have five stages in this Dockerfile:

- `dependencies` - This stage installs all dependencies for the project. It's used in all other stages.

- `builder` - This stage copies all files from the project to the image. It's used in all other stages.
- `production` / `demo` - This stage builds the project and runs it in production mode.
- `development` - This stage runs the project in development mode.

This way, we can have a single Dockerfile for all three environments, and we can easily switch between them by changing the target in `docker-compose.yml` file. For example, if we want to run `frontend_dashboard` in development mode, we would use the following command:

```
docker compose -f docker-compose.yml up frontend_dashboard
```

If we want to run it in production mode, we would use the following command:

```
docker compose -f docker-compose.prod.yml up frontend_dashboard
```

The same logic applies to `backend` service. Here is an example of `backend` Dockerfile:

```
FROM python:3.9-slim as base

RUN mkdir /usr/src/app
WORKDIR /usr/src/app

RUN apt-get update
RUN apt-get install -y gcc

COPY ./requirements.txt requirements.txt

RUN pip3 install -U setuptools
RUN pip3 install -r ./requirements.txt
```

```
COPY ./core/ .

# Development branch of a Dockerfile
FROM base as development
RUN chmod +x /usr/src/app/entrypoint.sh
ENTRYPOINT [ "sh", "/usr/src/app/entrypoint.sh" ]

# Demo branch of a Dockerfile
FROM base as demo
RUN apt-get install -y git postgresql-client
RUN chmod +x /usr/src/app/entrypoint.sh
# load demo data
ENTRYPOINT [ "sh", "-c", "/usr/src/app/entrypoint_demo.sh &&
/usr/src/app/entrypoint.sh" ]

# Production branch of a Dockerfile
FROM base as production
RUN chmod +x /usr/src/app/entrypoint.sh
ENTRYPOINT [ "sh", "/usr/src/app/entrypoint.sh" ]
```

As you can see from the example, we have four stages in this Dockerfile:

- `base` - This stage installs all dependencies for the project. It's used in all other stages.
- `development` - This stage runs the project in development mode using typical `python3 manage.py runserver` command.
- `demo` - This stage runs the project in production mode. It also loads demo data into the database. Production mode is ran using `gunicorn` server as `gunicorn core.wsgi -c ./gunicorn/conf.py`.
- `production` - This stage runs the project in production mode. Production mode is ran using `gunicorn` server as `gunicorn core.wsgi -c ./gunicorn/conf.py`.

## Gunicon

The basic settings for Gunicorn are defined in the `backend/core/gunicorn/conf.py` file. Here is an example of `gunicorn/conf.py` file:

```
import multiprocessing
# gunicorn.conf.py
# Non logging stuff
bind = "0.0.0.0:8000"
workers = multiprocessing.cpu_count() * 2 + 1
threads = 2
# Access log - records incoming HTTP requests
accesslog = "/var/log/gunicorn.access.log"
# Error log - records Gunicorn server goings-on
errorlog = "/var/log/gunicorn.error.log"
# Whether to send Django output to the error log
capture_output = True
# How verbose the Gunicorn error logs should be
loglevel = "info"
```

All logs are stored in `/var/log` directory. This directory is not mounted to the host machine, so you can't access it directly. However, you can access it using `docker exec` command. For example, if you want to see the content of `gunicorn.access.log` file, you would use the following command:

```
docker exec -it backend cat /var/log/gunicorn.access.log
```

---

# Docker cache

Docker can cache the results of each build step. This is useful when you are building an image that is based on another image. If the previous build step has not changed, Docker will reuse the cache and skip the build step. This can significantly speed up the build process. However, if you are not careful, this can lead to unexpected results. For example, if you change the order of

the build steps, Docker will not reuse the cache. This can lead to unexpected results.

ecoseller

User documentation

# Getting started

Table of contents:



The following sections will guide you through the **core features** of the application and help you get started with it using video tutorials.
Please go through the tutorials in the order listed (as they often depend on the previous ones).

In case you want to see all of the ecoseller features, or you just prefer documentation in a textual form, see also

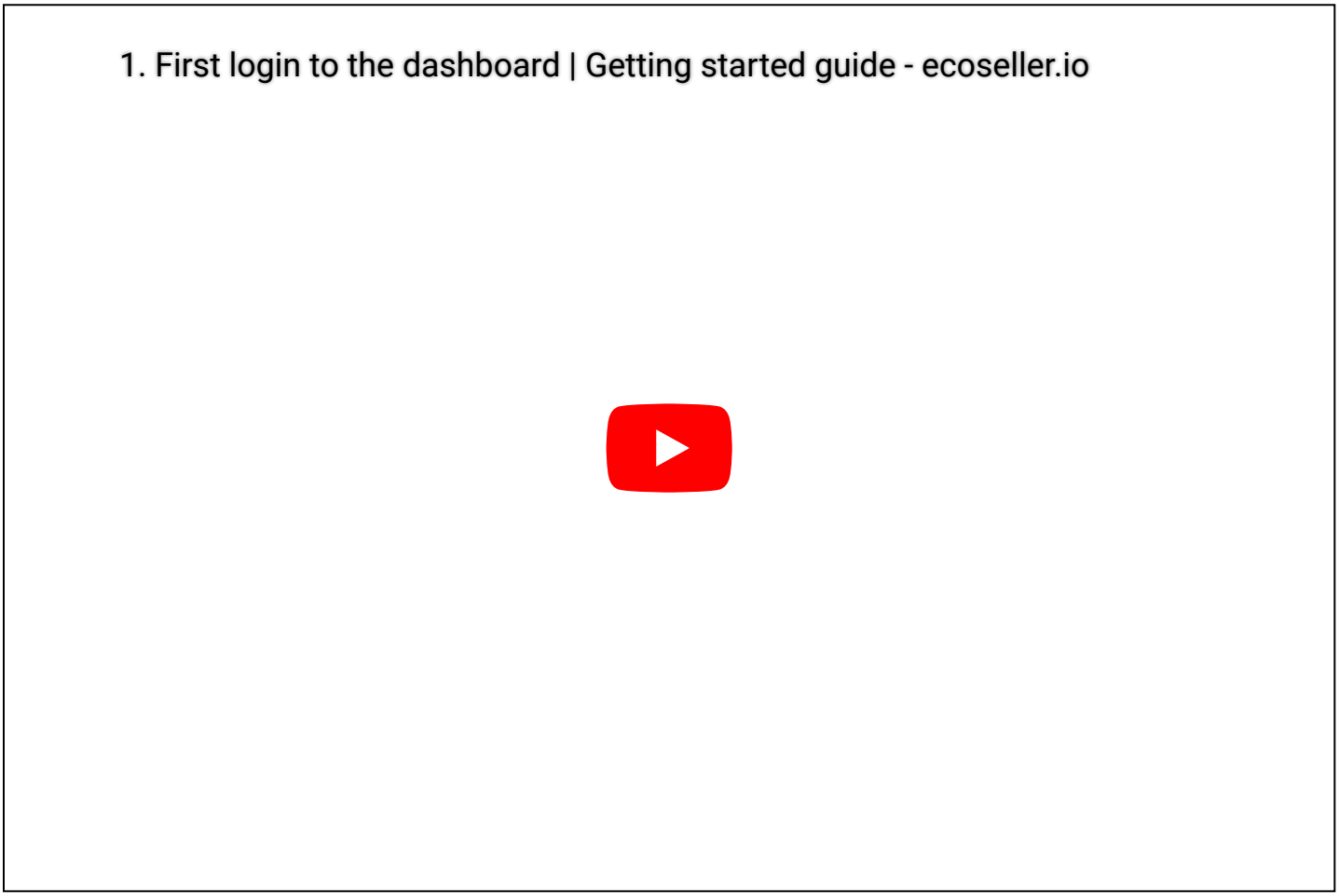
- dashboard docs
- storefront docs

# First login to the dashboard

In the following tutorial, you will learn the following:

- Initial login

- Initial login
- Changing your password
- Creating a new user with given roles
- Explaining pre-defined roles

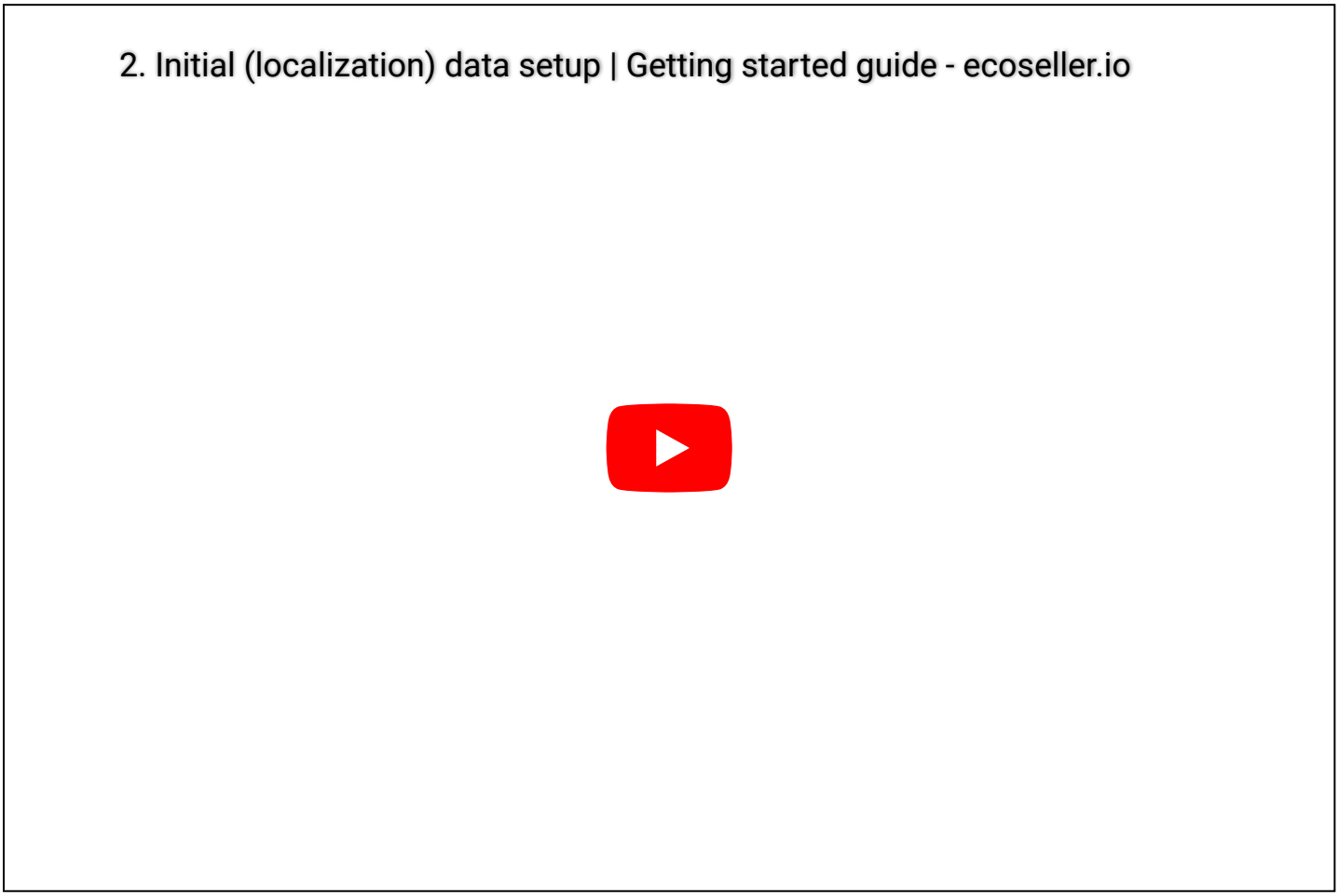



# Setting up localization-related data

This tutorial is focused on setting up localization-related data. You will learn the overall structure of the data and how they relate to each other in the following steps:

- Structure of the currency data and how to create a new currency

- Structure of the price list data, how to create a new price list and how to assign a currency to it
- Structure of the country data and how to create a new country with a given price list
- Structure of VAT group data, how to create a new VAT group and bind it to a country

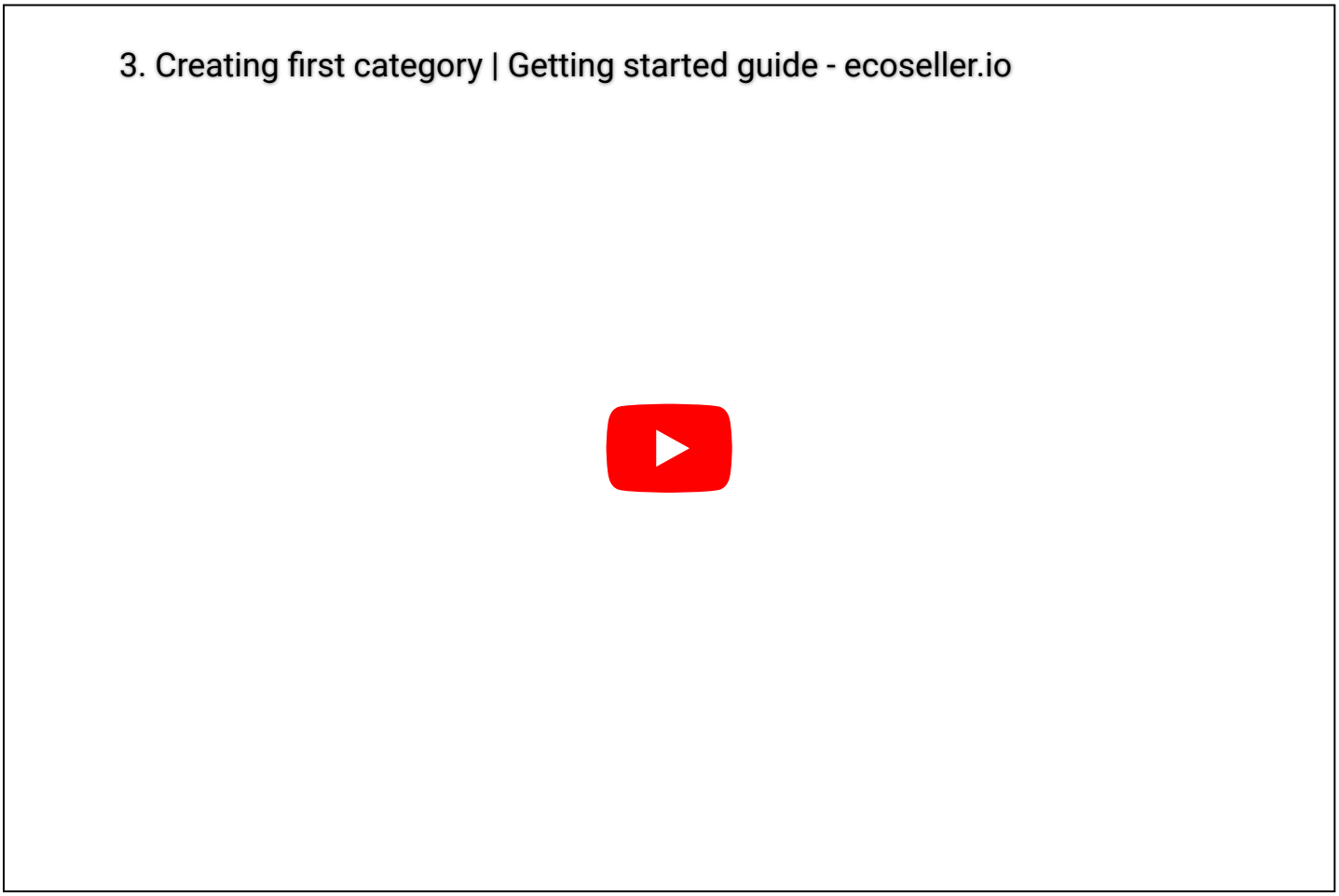



# Creating your first product category

This tutorial focuses on creating product categories, you'll learn how to

- create new category

- set parent category

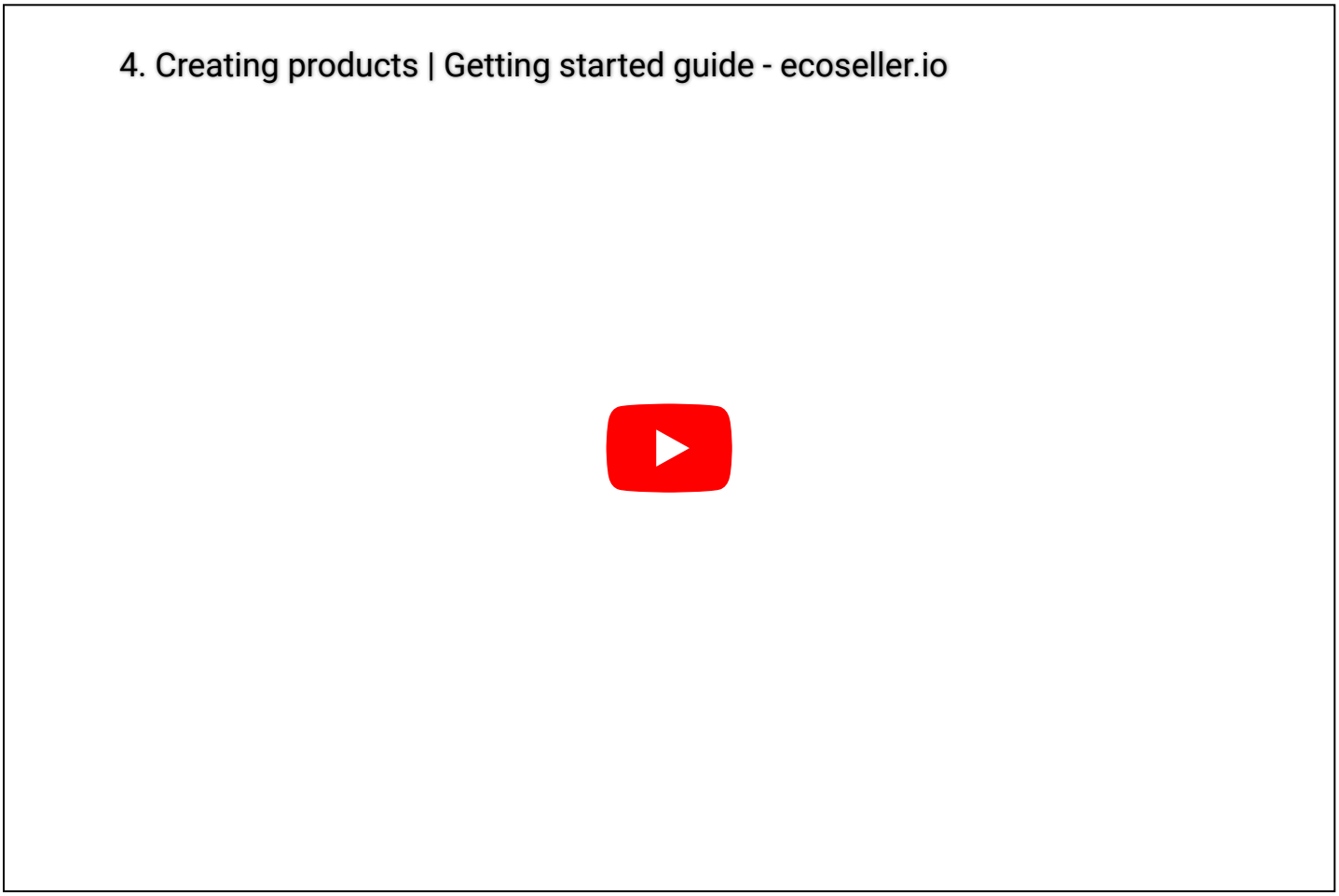



# Creating your first product

This tutorial describes step-by-step how to create your first product.

You'll learn how to:

- Create new attributes (both textual and numeric)
- Create new product types and assign attributes to them
- Create new product and set its product type
- Create new product variants
- Set product variant prices
- Upload media (e.g. photos) to a given product




# Browsing & indexing products

In this tutorial you'll learn how to

- browse and filter products on storefront
- index the products to Elasticsearch

5. Browsing & indexing the data | Getting started guide - ecoseller.io

ecoseller

User documentation

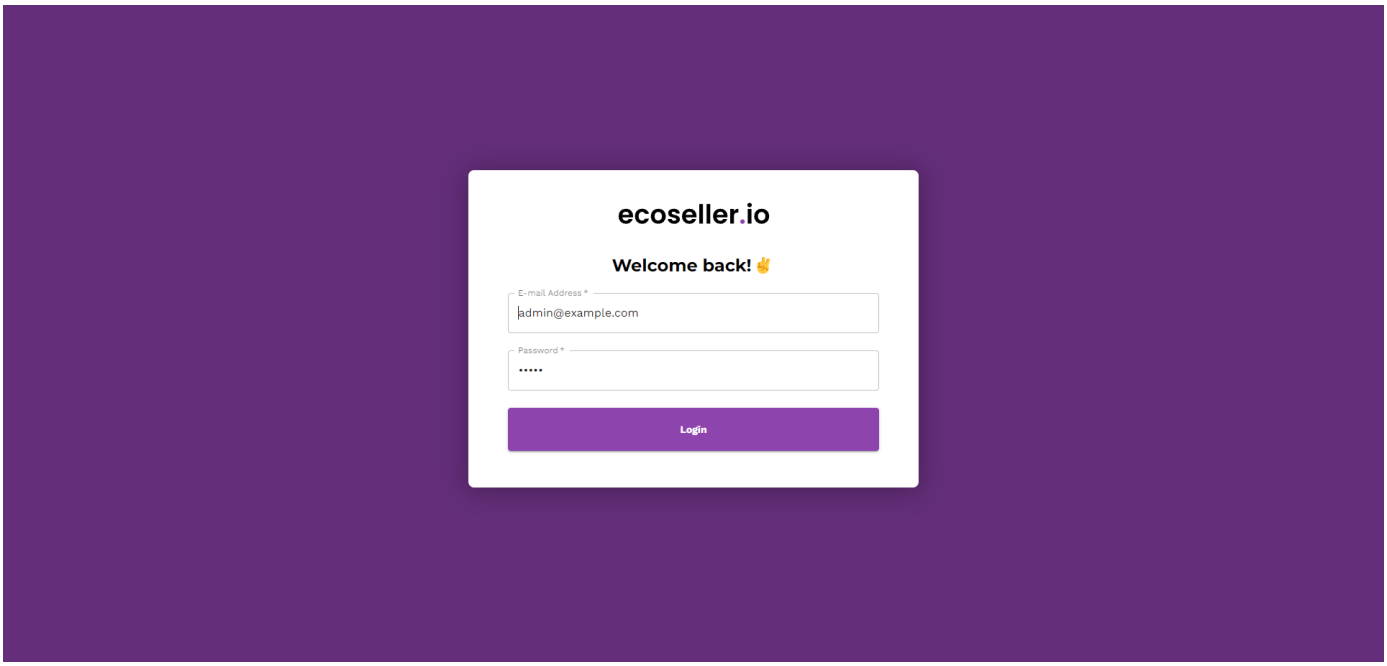
# Dashboard

Table of contents:

- Performance
- Training
- Configuration

# Login page

When you first open dashboard, the login page is displayed.

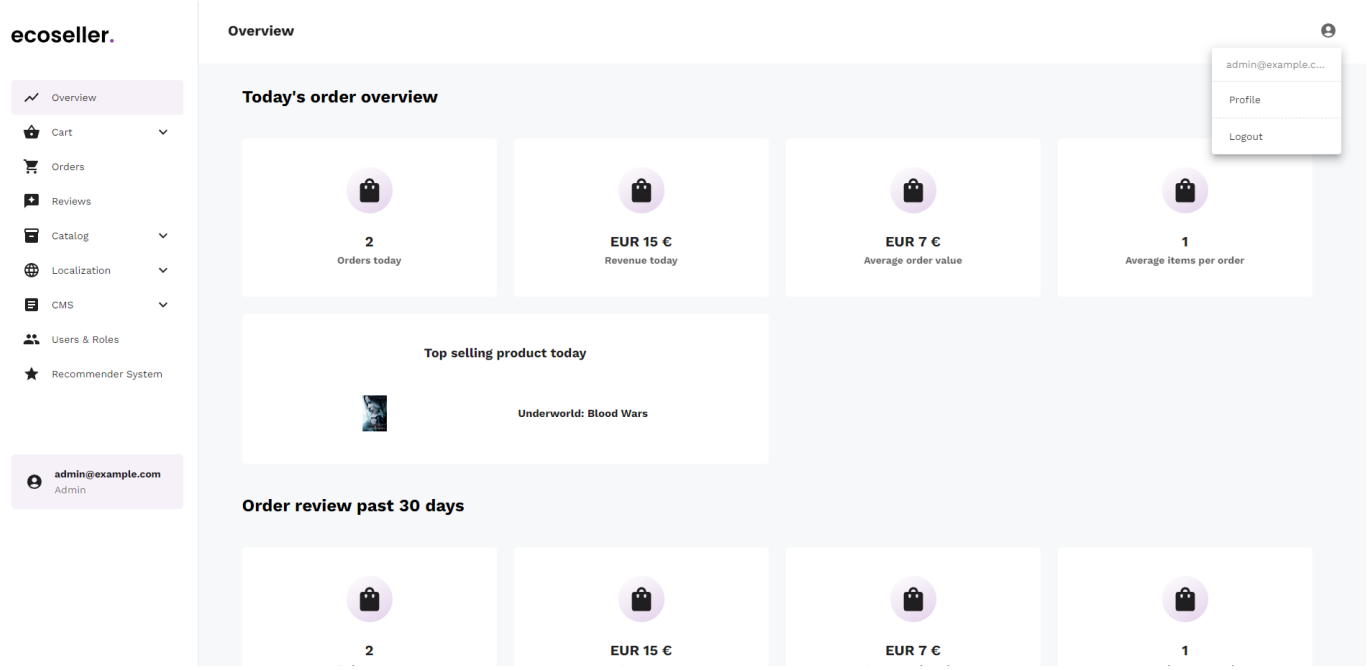



After you login, you're redirected to *Overview* page

# Overview

The Overview page provides summarized information about the store. It is the first page that is shown to the user after logging in. The page is divided into two main sections:

- Today's statistics
- Statistics for the last 30 days



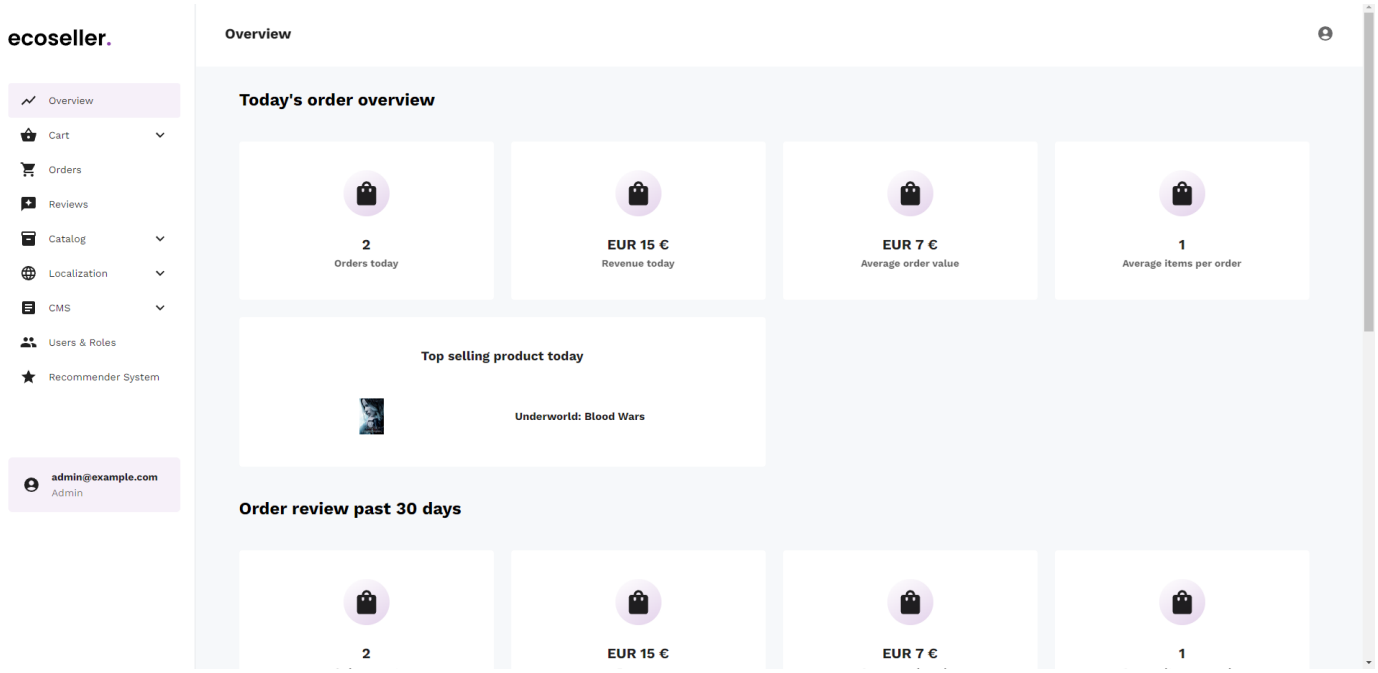

Using the icon on the top right, you can display your profile or logout.

**There's a sidebar menu on the left, which is used for navigation and it's shown on all dashboard pages.**
The chapters of documentation correspond to these menu items.

# Today's statistics

For today's statistics, the following information is shown:

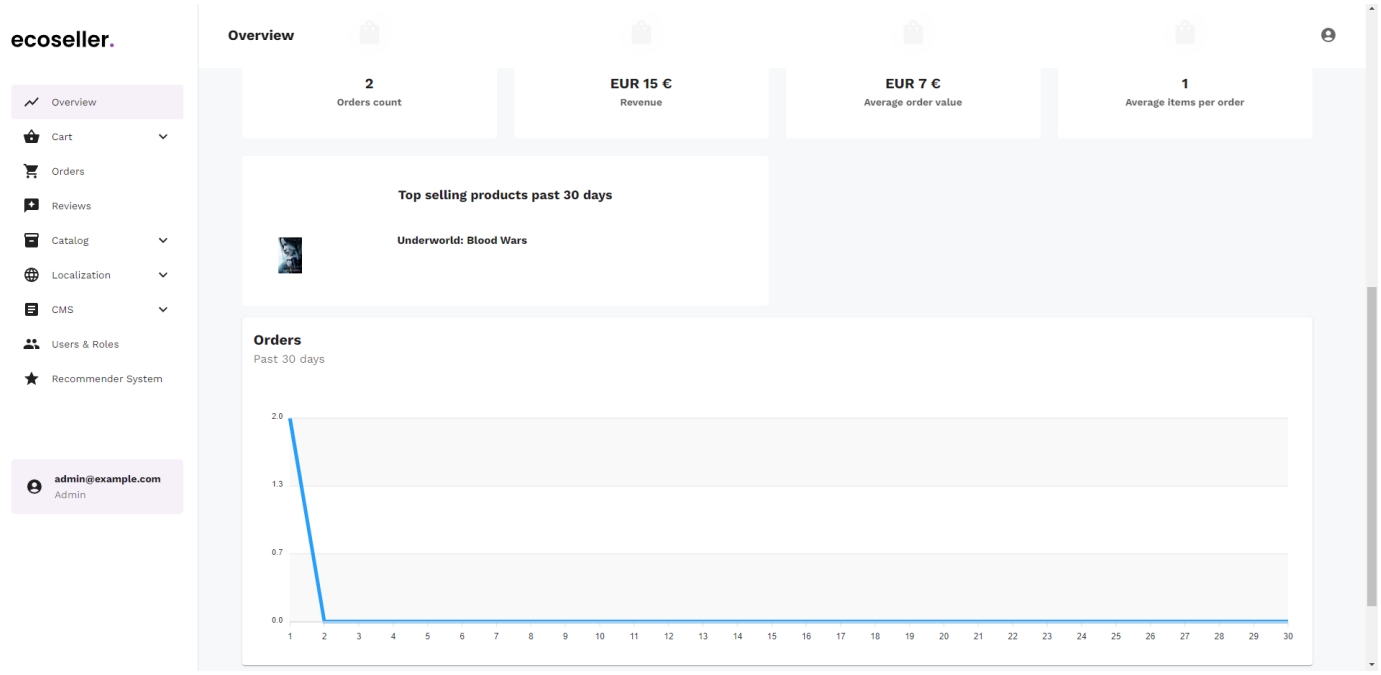
- Orders count
- Revenue
- Average order value
- Average items per order
- Top selling product




# Statistics for the last 30 days

For the last 30 days' statistics, the following information is shown:

- Orders count
- Revenue
- Average order value
- Average items per order
- Up to 5 top selling products
- Line graph showing orders count for the last 30 days



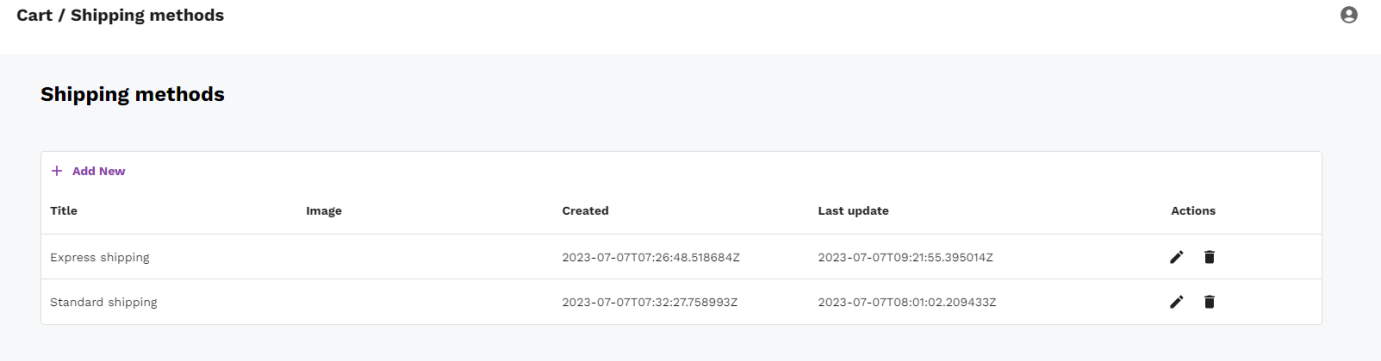

# Cart

Cart section is used to manage shipping and payment methods that can customers use.
We will describe the both parts in more detail in the following sections.

## Shipping methods

Shipping method page displays list of all shipping methods with the following fields:

- title
- image
- created
- last update



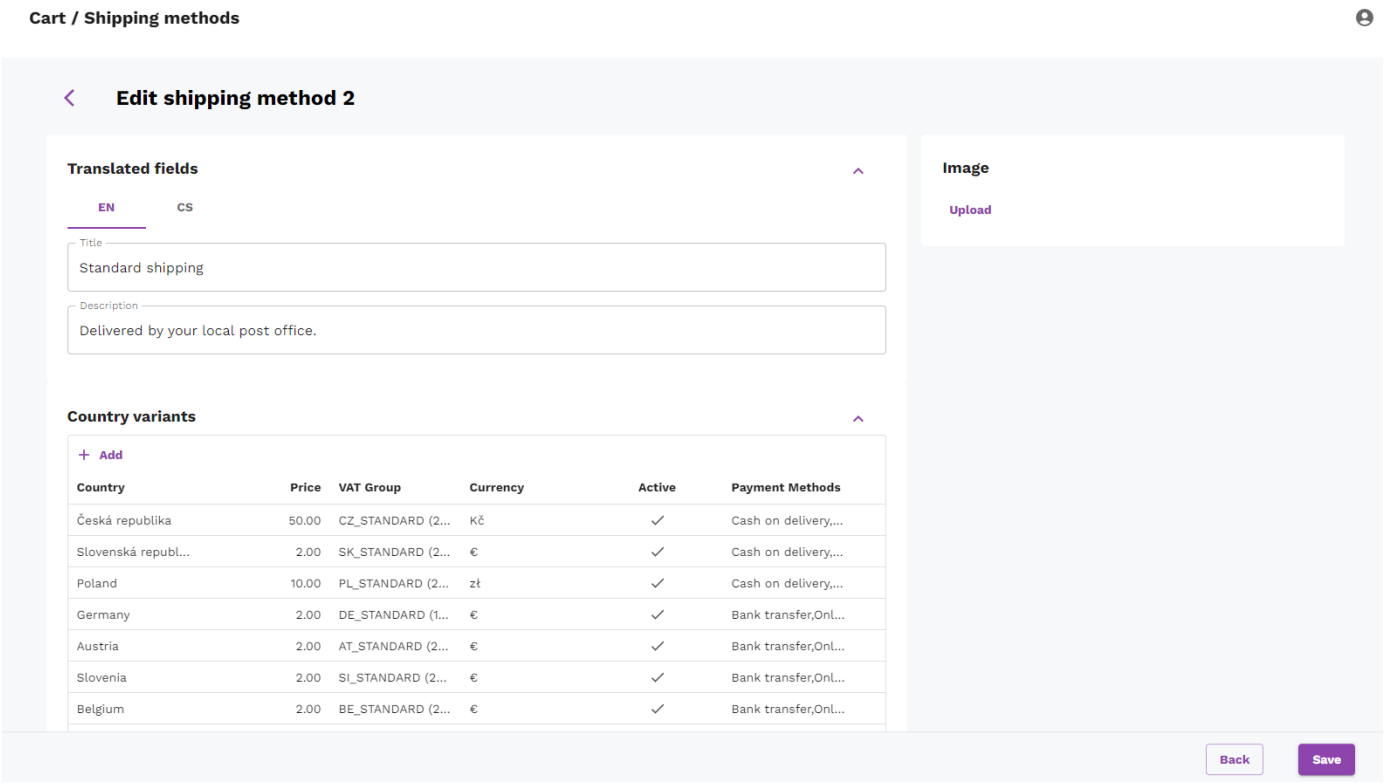

There are also 2 action buttons:

- *Edit* - after you click on this button, you'll be redirected to Shipping metod detail page (see Shipping method detail)
- *Delete* - used for deleting the given shipping method

---

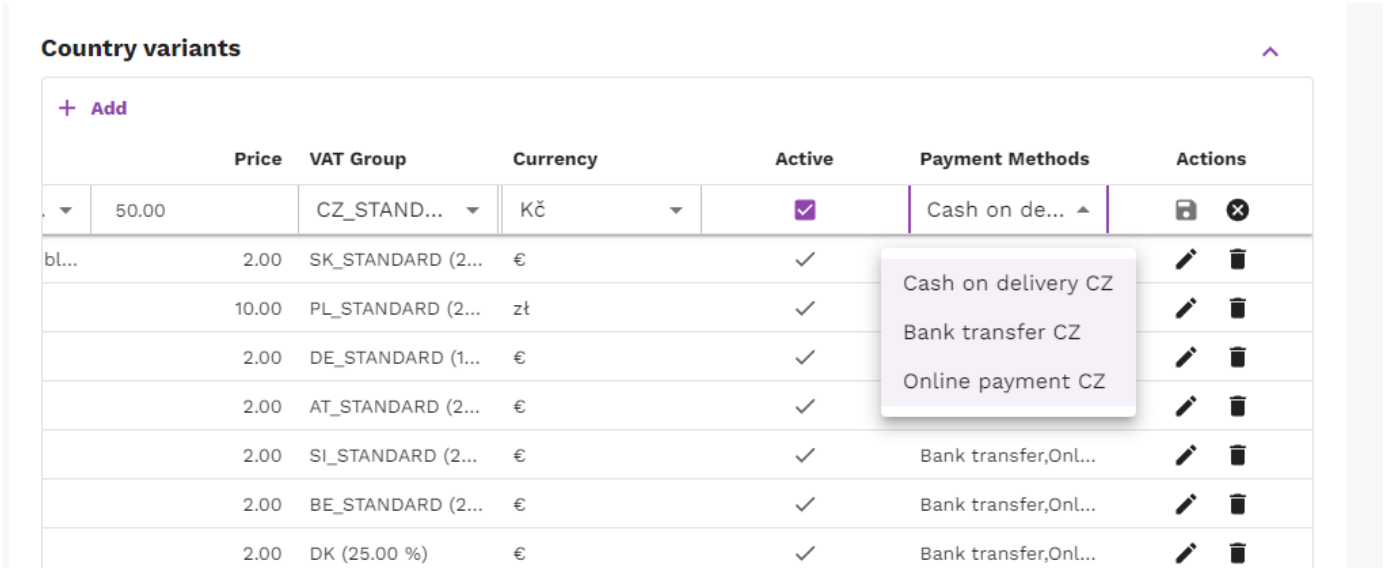
# Shipping method detail

This page contains detailed info about a shipping method




You can edit all of the fields and also add an image (which is shown to customers when they select shipping method). Note that title and description is set individually for each language.
Below, there's also a table of individual country variants.

This table is editable - you can edit & remove individual rows (using action buttons) as well as add new ones.



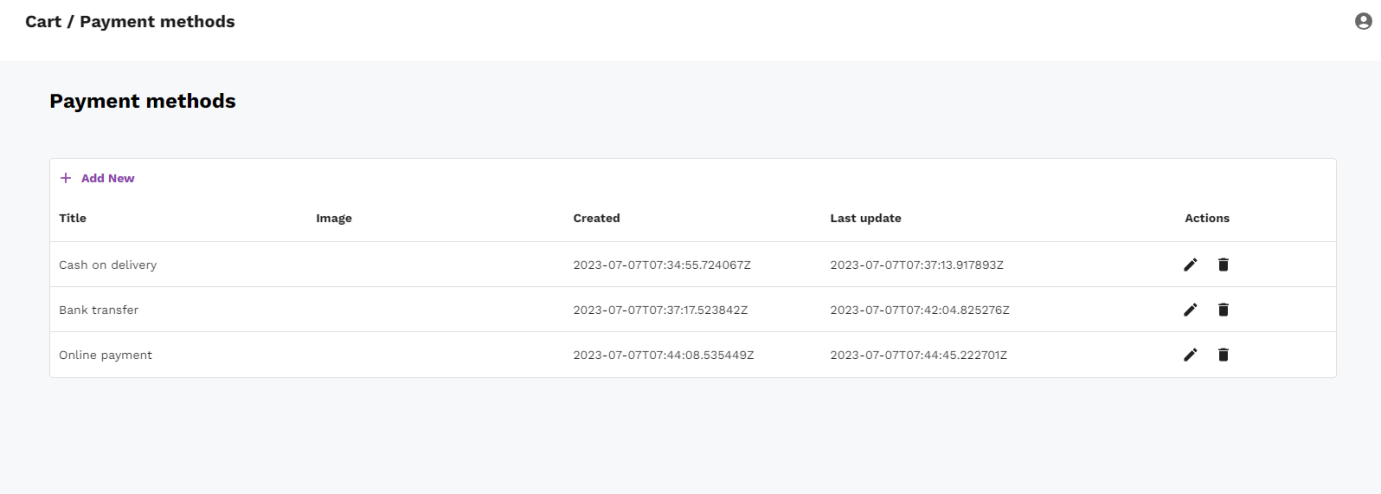
The image above shows that you're able to edit all fields and select available payment methods as well. In this case we selected all of the 3 available payment methods.

# Payment methods

Payment method page displays list of all shipping methods with the following fields:

- title
- image
- created
- last update

Note that it looks basically the same as shipping methods page.

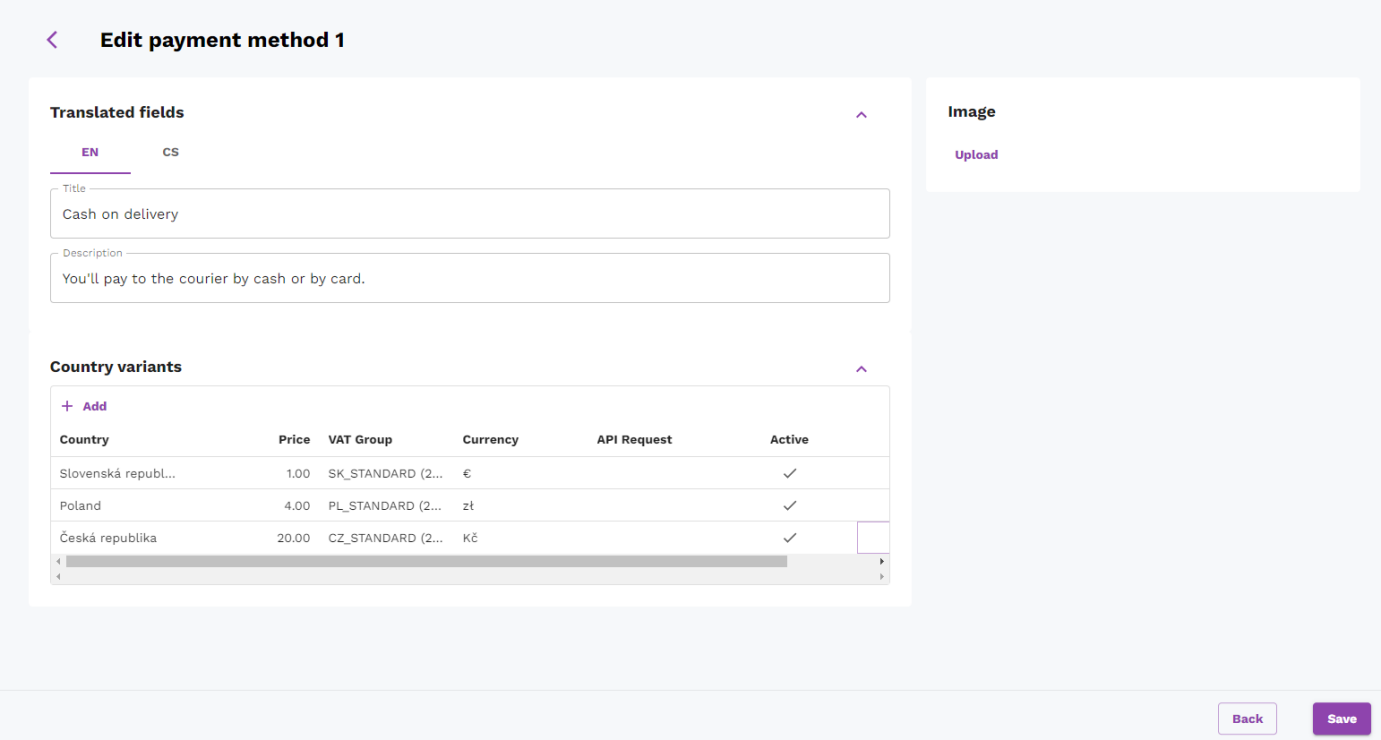



There are also 2 action buttons:

- *Edit* - after you click on this button, you'll be redirected to Payment metod detail page (see Payment method detail)
- *Delete* - used for deleting the given payment method

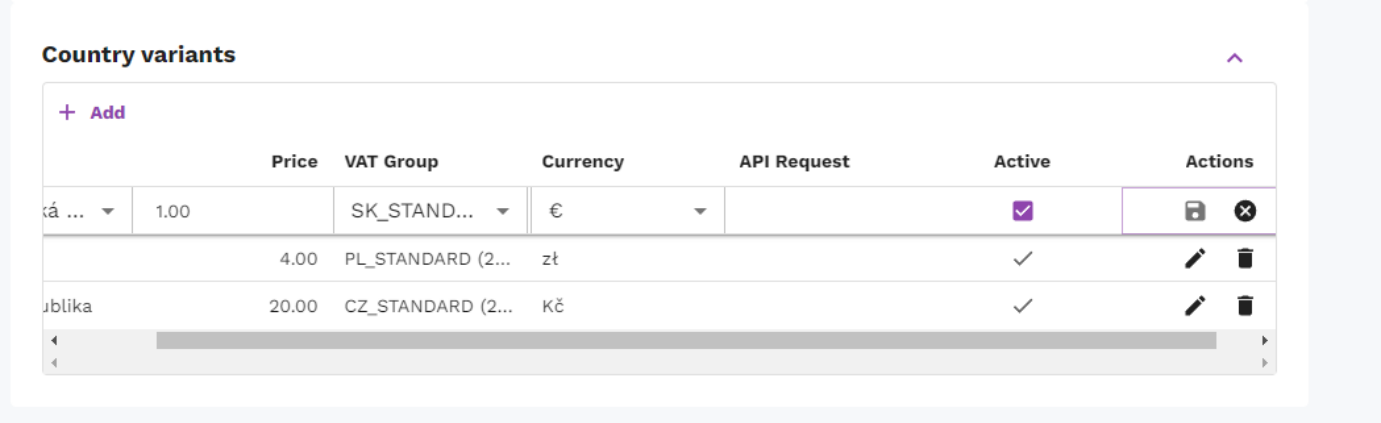
# Payment method detail

This page contains detailed info about a payment method.




You can edit all of the fields and also add an image (which is shown to customers when they select payment method). Note that title and description is set individually for each language.
Below, there's also a table of individual country variants.

This table is editable - you can edit & remove individual rows (using action buttons) as well as add new ones.

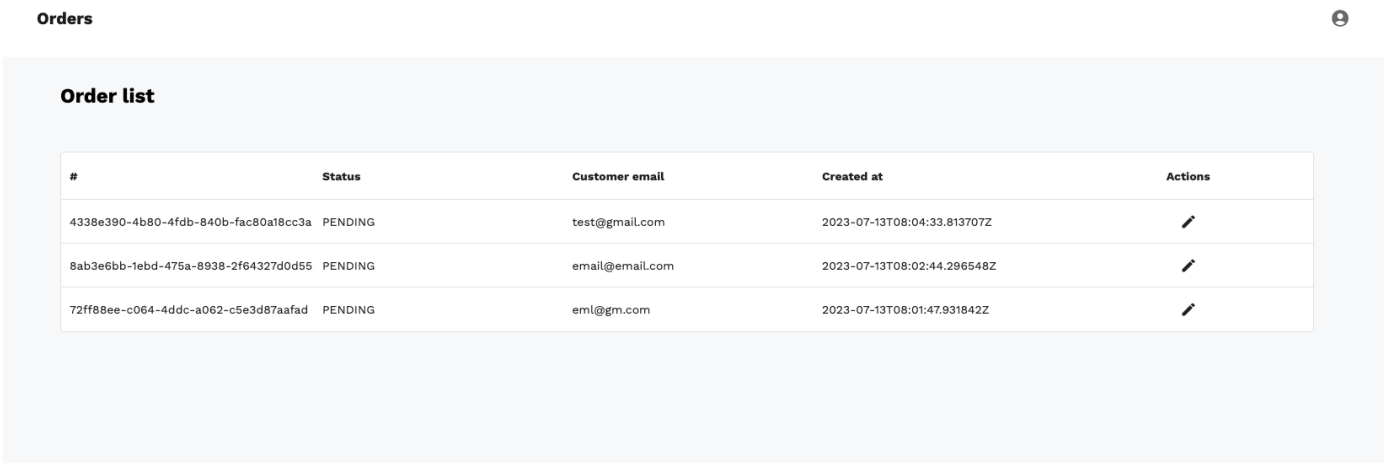



The image above shows that you're able to edit all fields including API request - this way, you're able to select which code should be executed when the customer is about to pay.
See Payment Gateway integration process section for further information.

# Orders

The orders page consists of a list showing all orders. The list has the following columns:

- Order token
- Status
- Customer email
- Created at
- Actions




The actions column contains the following buttons:

- Edit

# Edit

Click on the edit button opens the order details page.

# Order details

This page shows full information about the order. The page is divided into the following sections:

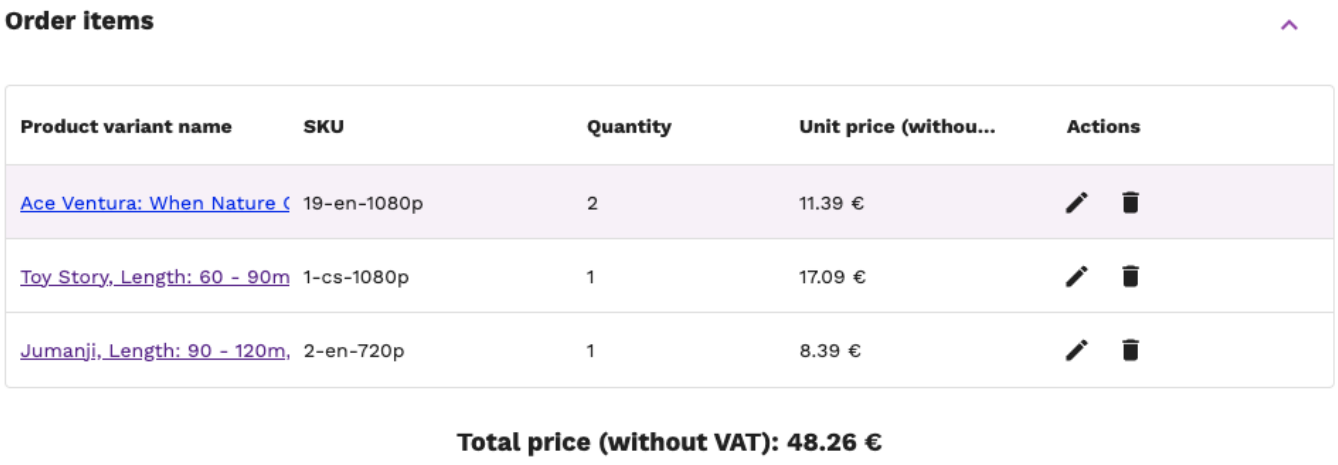
- Order items
- Status
- Shipping Info
- Billing info
- Shipping and payment methods

# Order items

This section shows a list of all items in the order. The list has the following columns:

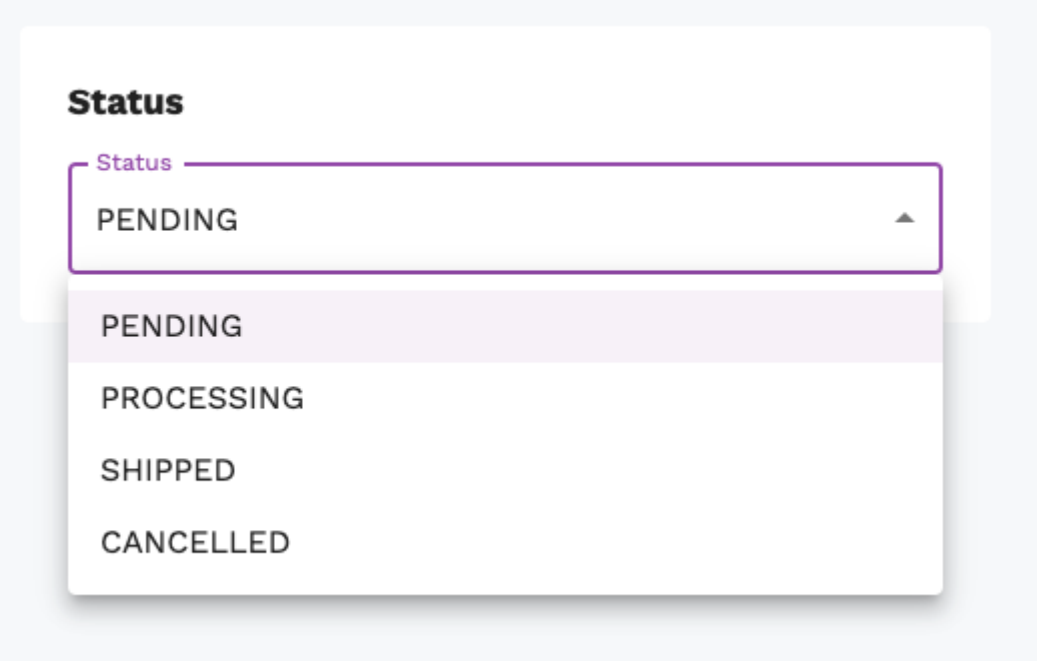
- Product variant name - click on the name opens the product variant details page (described here)
- SKU
- Quantity
- Unit price (without VAT)
- Complaints - this column is displayed only if there are any complaints for this item
- Actions
  - Edit - admin can change the quantity of the product variant
  - Delete - admin can delete the product variant from the order

This section also shows the total price (without VAT) of a given order.



## Status

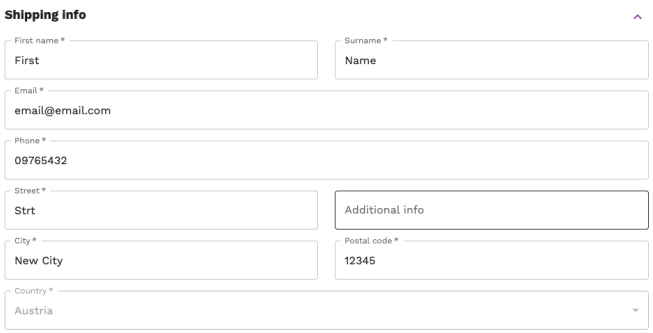
This section shows the current status of the order. The status can be changed by the admin using a drop-down menu.

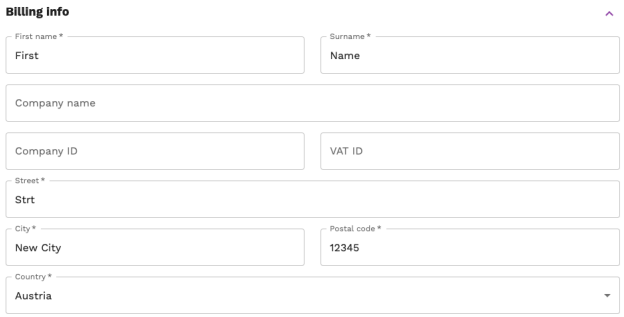



## Shipping and billing info

This section shows the shipping and billing information of the order. It contains the same information as the shipping and billing information in the checkout process. Information is shown in the form view, and the admin is again able to modify its content (if the order's status is set to *PENDING* or *PROCESSING*)

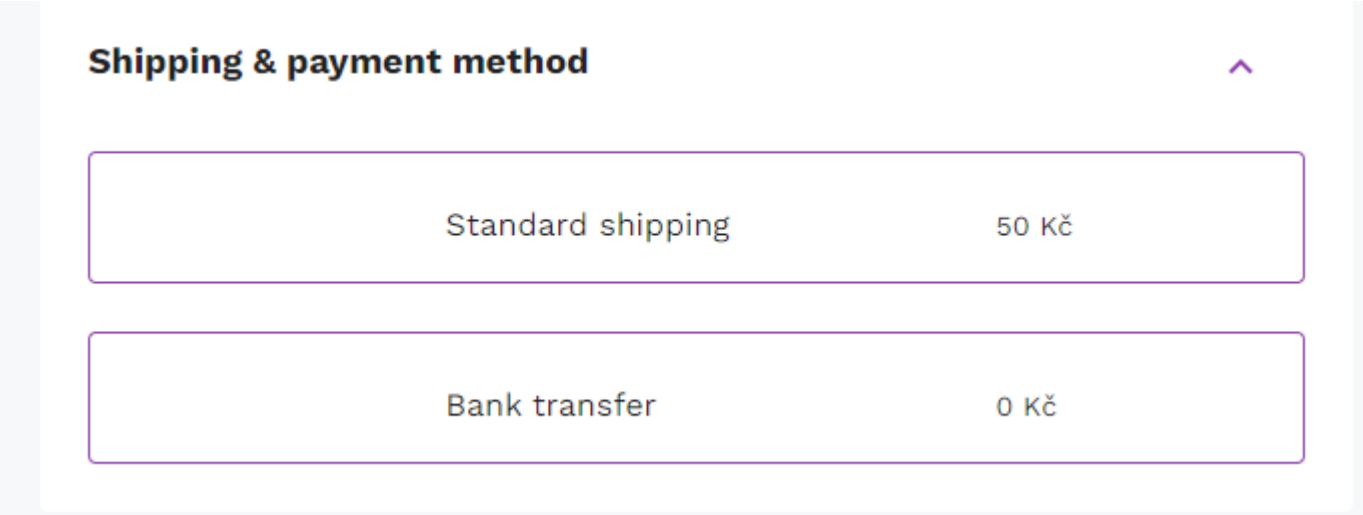
**Shipping info** | **Billing info**




# Shipping and payment method

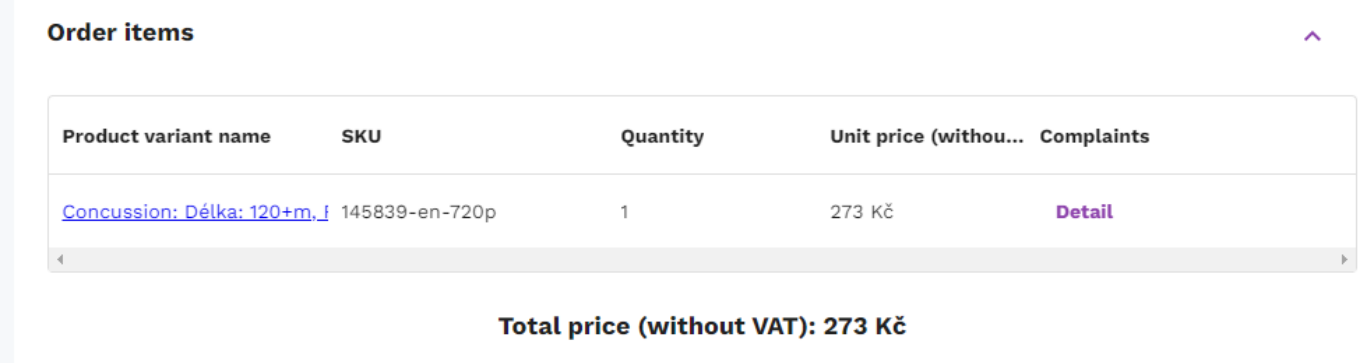
This section shows selected shipping and payment methods along with their prices.



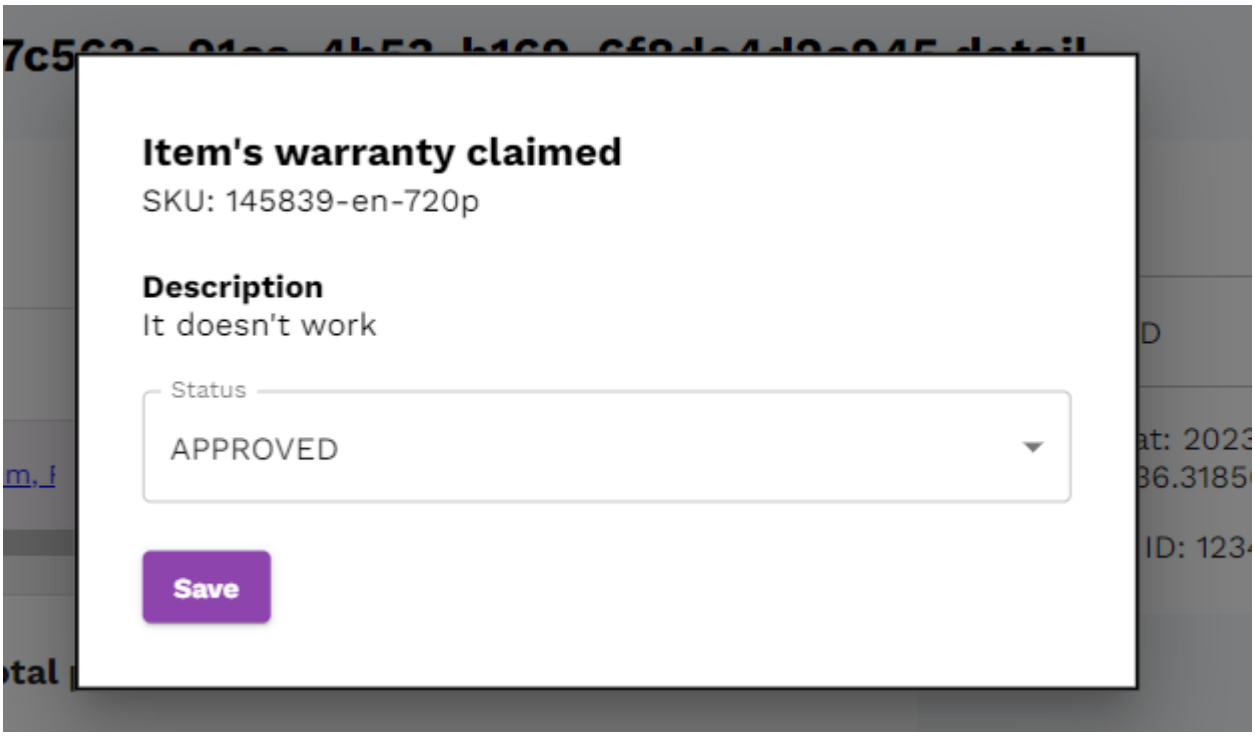

Note that these values cannot be changed.

# Order item complaints

If there are any complaints (warranty claim / return requests) for an order item, you'll see it in `Complaints` column.




After you click on the complaint detail button, pop-up with complaint details is displayed.

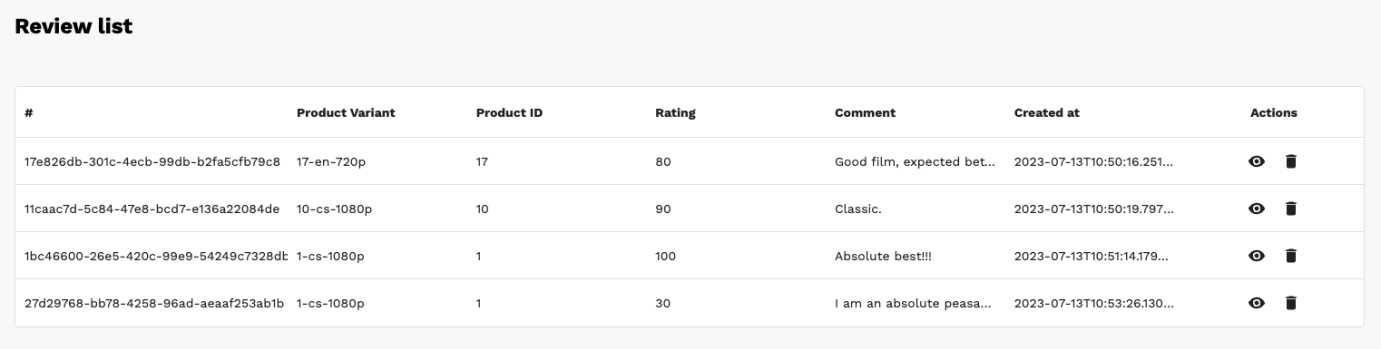



In this pop-up, all complaint details are displayed, and you're able to change its status.

# Reviews

The reviews page consists of a list showing all reviews. The list has the following columns:

- Review token
- Product variant
- Product ID
- Rating
- Comment
- Created at
- Actions
  - Detail - click on the detail button opens the review details page (see Review details)
  - Delete - click on the delete button deletes the review

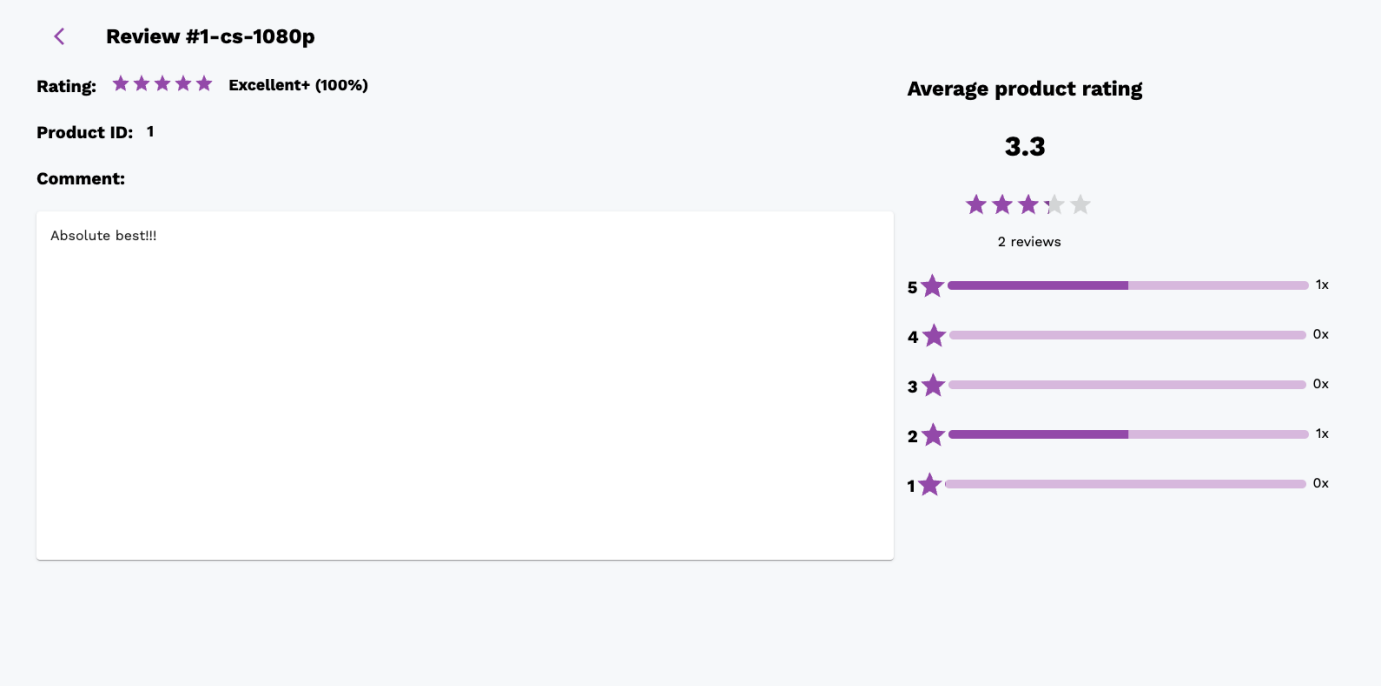



# Review details

This page shows full information about the review and overall rating of the product. The page is divided into the following sections:

- Rating - shown via stars and percentage

- Product ID
- Comment
- Average product rating




## Average product rating

This section shows the average rating of the product. The rating is shown via start and average score (value from 0 to 5). It also shows the number of reviews for the product and the distribution of ratings. Distribution values are rounded up - this means that if the user submitted a rating of 4.5, it will be shown as 5 in the distribution.

# Catalog

Catalog is the place where you can manage all products and categories. The catalog page consists of:

- Attributes - Attributes binded to the product variants
- Product types - Product types define the structure of the product (what attributes it has)
- Products - Products are the actual products that are shown in the storefront and their variants, prices, etc.
- Categories - Categories are used to group products into categories

# Attributes

Attributes are used to define the structure of the product variant. For example, if you want to sell t-shirts, you need to define attributes like size, color, etc. Attributes are then used in the product type to define the structure of the product variant. Attributes are also used in the product variant to define the actual values of the attributes.

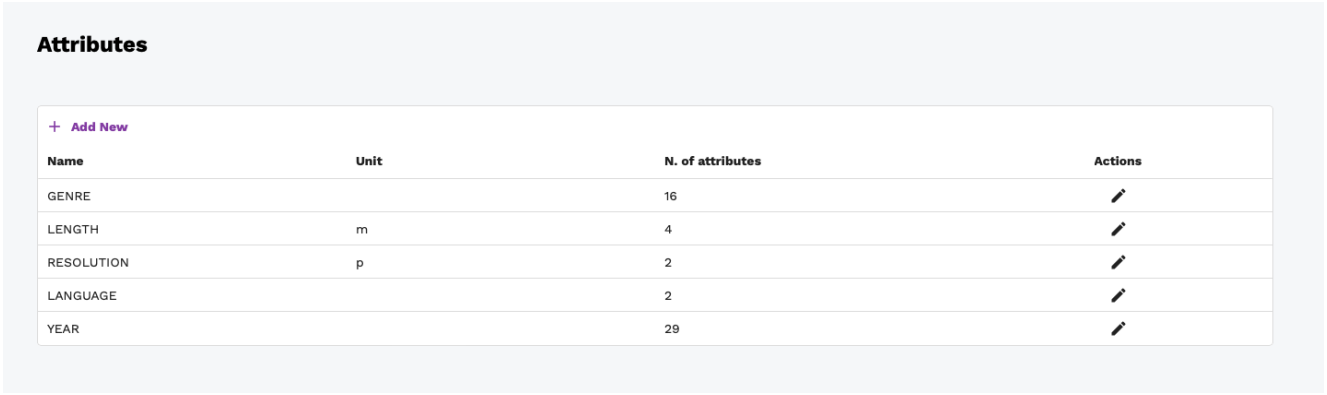
---

## Attribute list

Under the attributes section, you can see a list of all attributes. The list has the following columns:

- Name - unique name of the attribute, it’s not shown in the storefront and is used only in the administration
- Unit - unit of the attribute, for example, cm, kg, etc. It’s shown in the storefront next to the value of the attribute (for example 10 cm)

- N. of attributes - just a number of values under the given attribute
- Actions - edit button




# Creating new attribute

To create a new attribute, click on the button in the upper left corner of the attribute table - *Add New*. You will be redirected to the attribute details page (see Editing attributes).

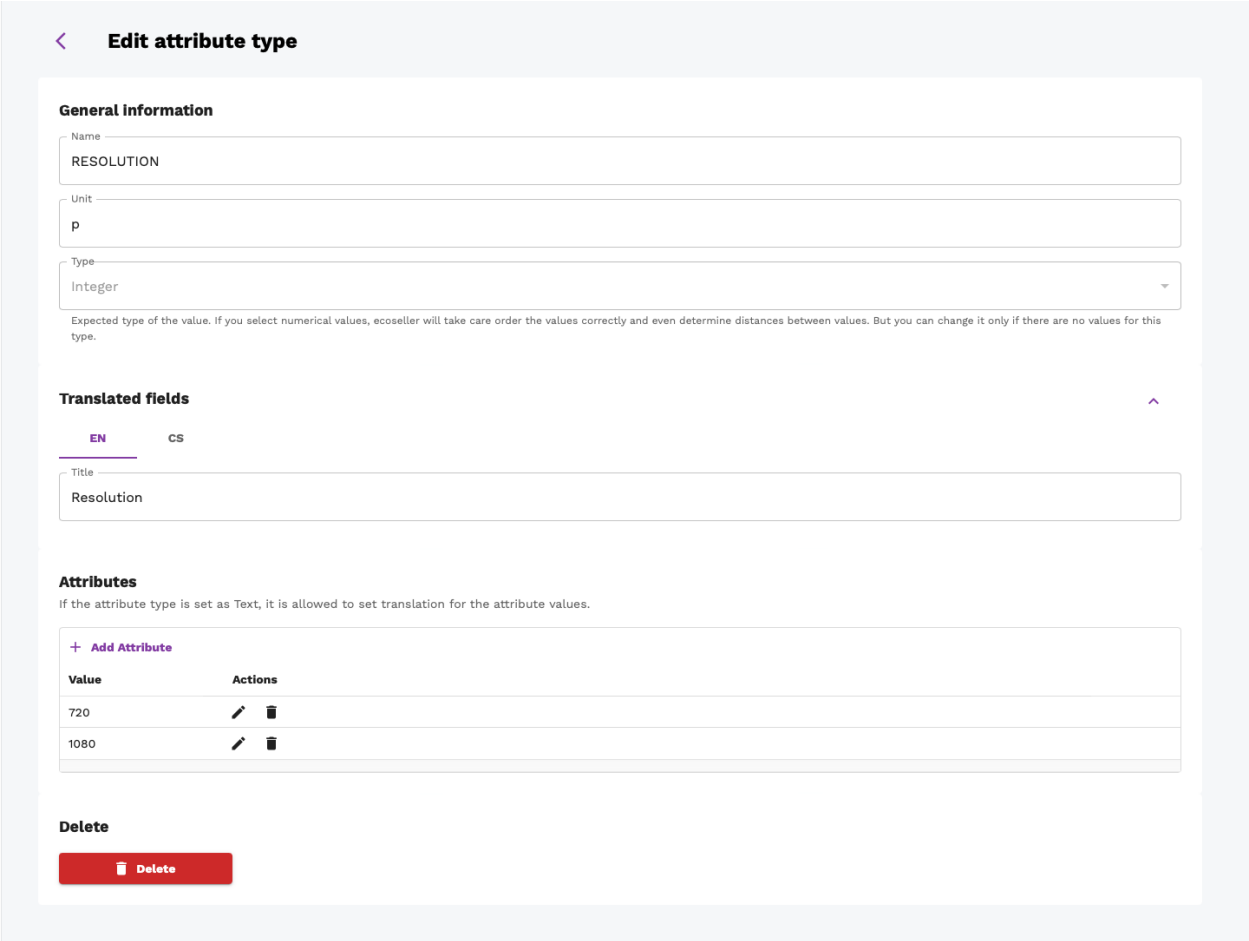
# Editing attribute

To edit an attribute, click on the edit button in the attribute list. This opens the attribute details page. The detail page consists of the following sections:

1. *General information* - this section contains the name and unit of the attribute. If you've just created the attribute or it has no values, you can change type of attribute (see Attribute types)
2. *Translated fields* - this section contains the translated name of the attribute. You can translate the name of the attribute into multiple

languages.

3. *Attributes* - this section contains the list of values of the attribute. The list has the following columns:
   - Value - the value of the attribute
   - List of languages (if it's `text` attribute type) - the value of the attribute in the given language
   - Actions - edit and delete buttons
   - Edit - click on the edit button opens the edit attribute value page (see Editing attribute values)
   - Delete - click on the delete button deletes the attribute value

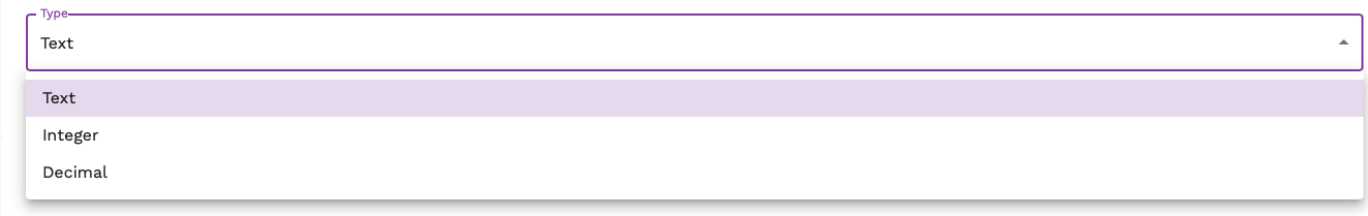



## Attribute types

There are three expected type of the value. If you select numerical values, ecoseller recommender system will take care order the values correctly and

even determine distances between values. But you can change it only if there are no values for this type. The types are:

- Text
- Integer
- Decimal




# Product Types

Product types define the structure of the product. For example, if you want to sell t-shirts, you need to define attributes like size, color, etc. Product types are then used in the product to define the structure of the product variant. Product types are also used in the product variant to define the actual values of the attributes.

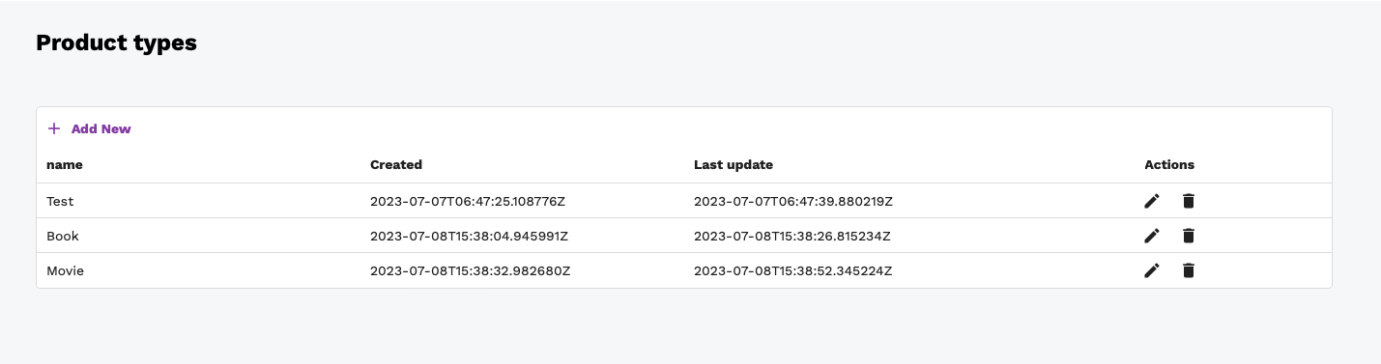
---

## Product type list

Under the product types section, you can see a list of all product types. The list has the following columns:

- Name - unique name of the product type, it’s not shown in the storefront and is used only in the administration
- Created - date when the product type was created

- Last updated - date when the product type was last updated
- Actions - edit button




---

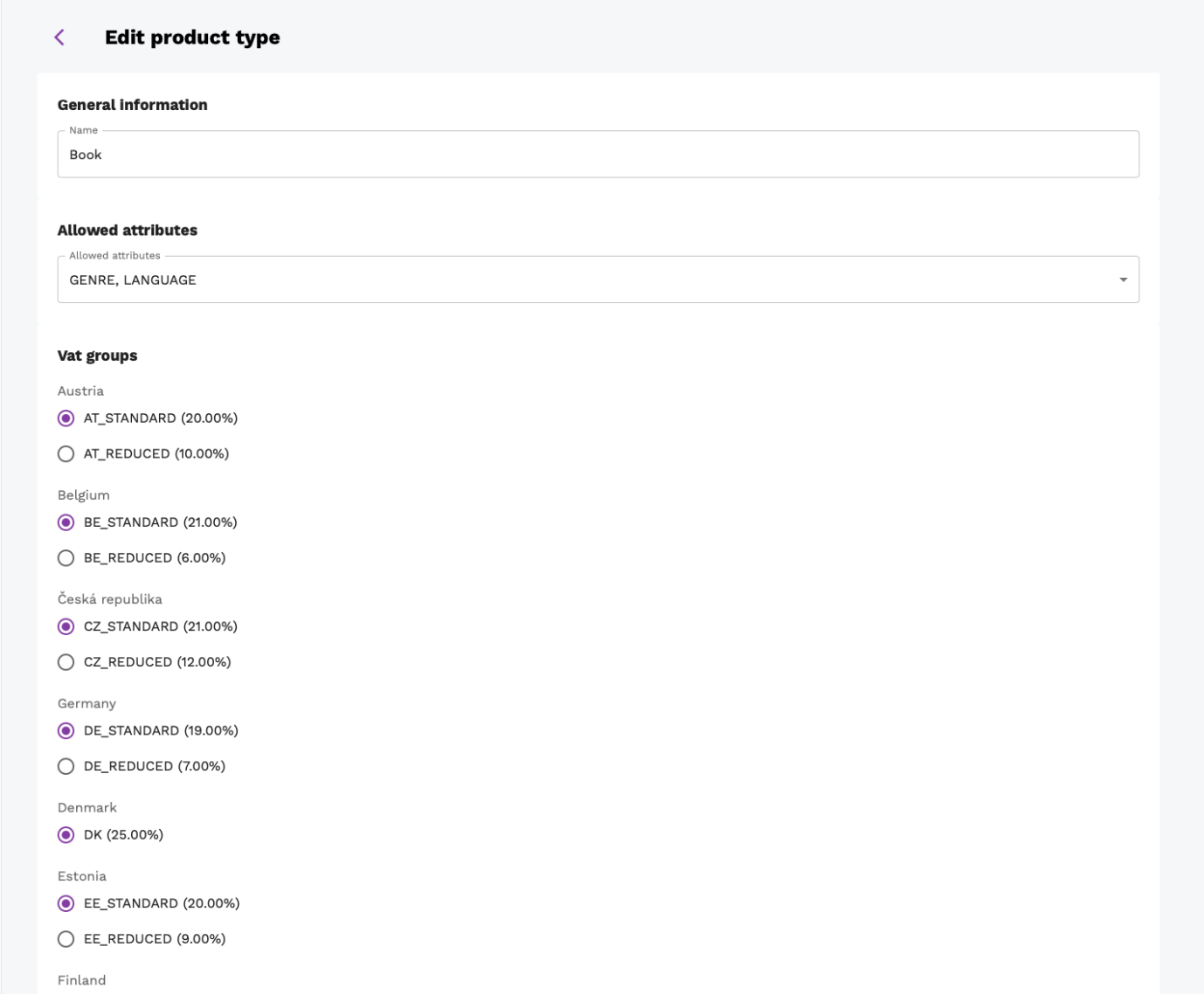
## Creating new product type

To create a new product type, click on the button in the upper left corner of the product type table - *Add New*. You will be redirected to the product type details page (see Editing product type).

---

## Editing product type

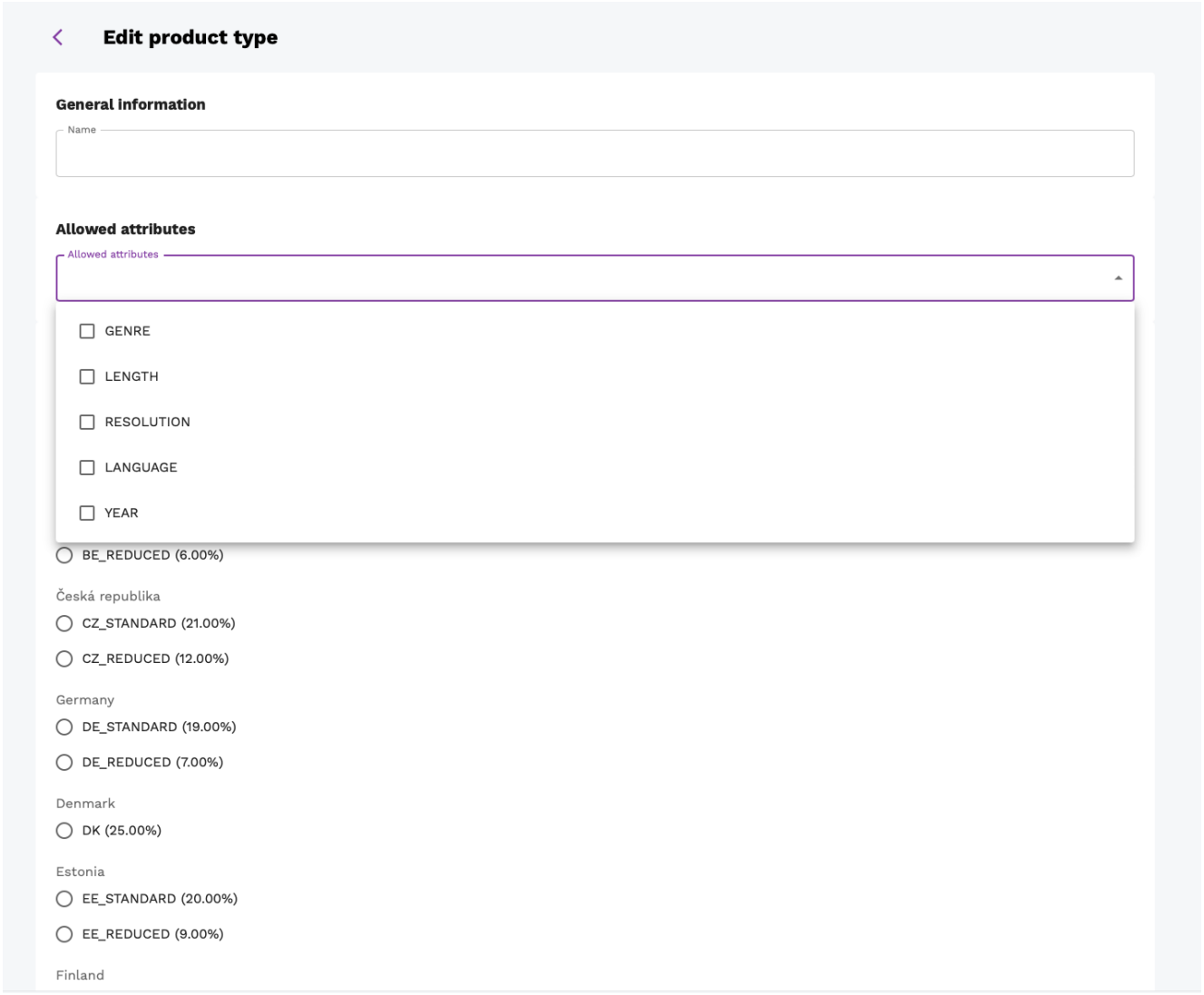
To edit a product type, click on the edit button in the product type list. This opens the product type details page.




The detail page consists of the following sections:

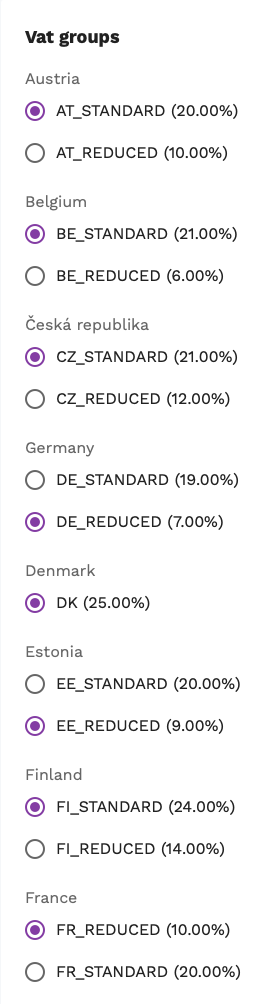
1. *General information* - this section contains the name of the product type. You'll see this name in the Product edit page (see Editing product) when you select the product type. It's not visible in the storefront.
2. *Allowed attributes* - this section contains the list of attributes that are allowed in the product type. It's a dropdown list from which you can select the attributes. You can select multiple attributes. Those are the attribtues that you'll be able to select in the product edit page (see

Editing product) when you select the product type.




3. *Vat groups* - this section contains the list of vat groups that are allowed in the product type. They're listed in a sections groupped by the country. You need to select *Vat group* for each country. You can select only one vat group per country. When calculating price incl VAT for each product x

country, selected vat group will be used.




# Products

Products are the core of the catalog. They're the actual products that are shown in the storefront and their variants, prices, etc.

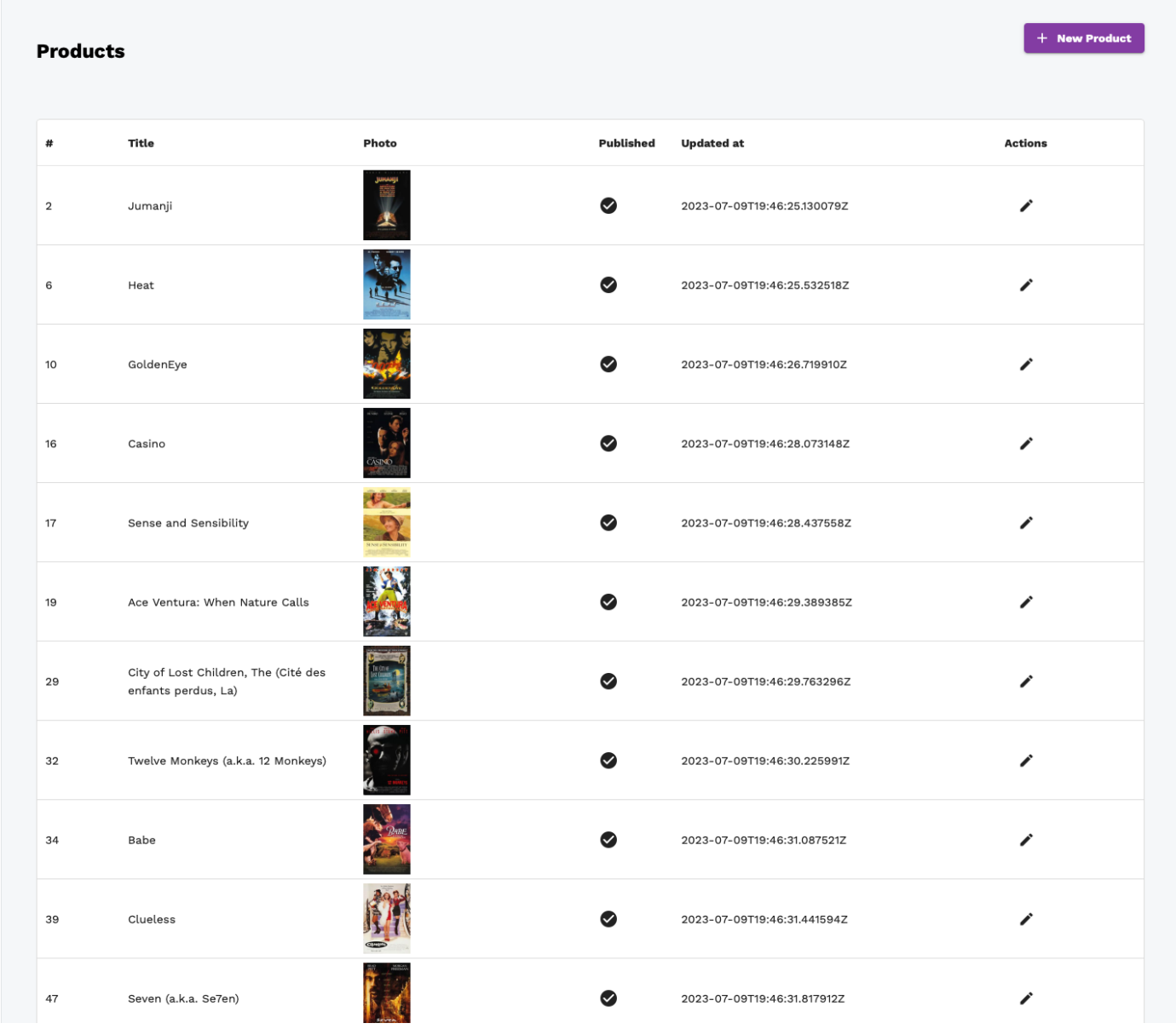
---

## Product list

Under the products section, you can see a list of all products. The list has the following columns:

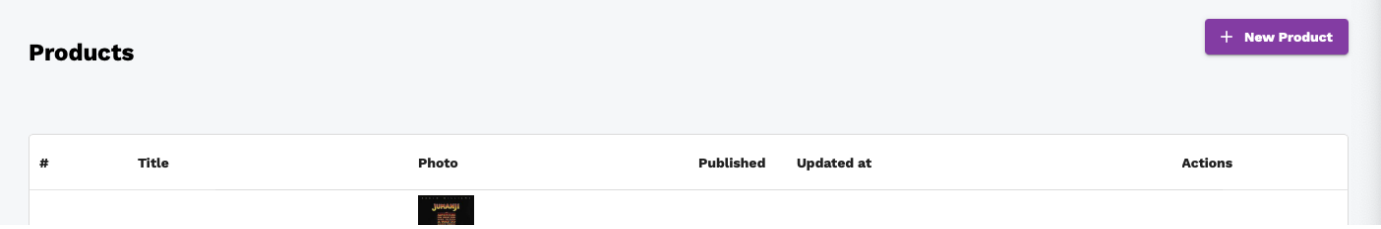
- ID – unique ID of the product

- Title - title of the product
- Photo - primary image of the product
- Published - whether the product is published or not
- Updated at
- Actions - edit button

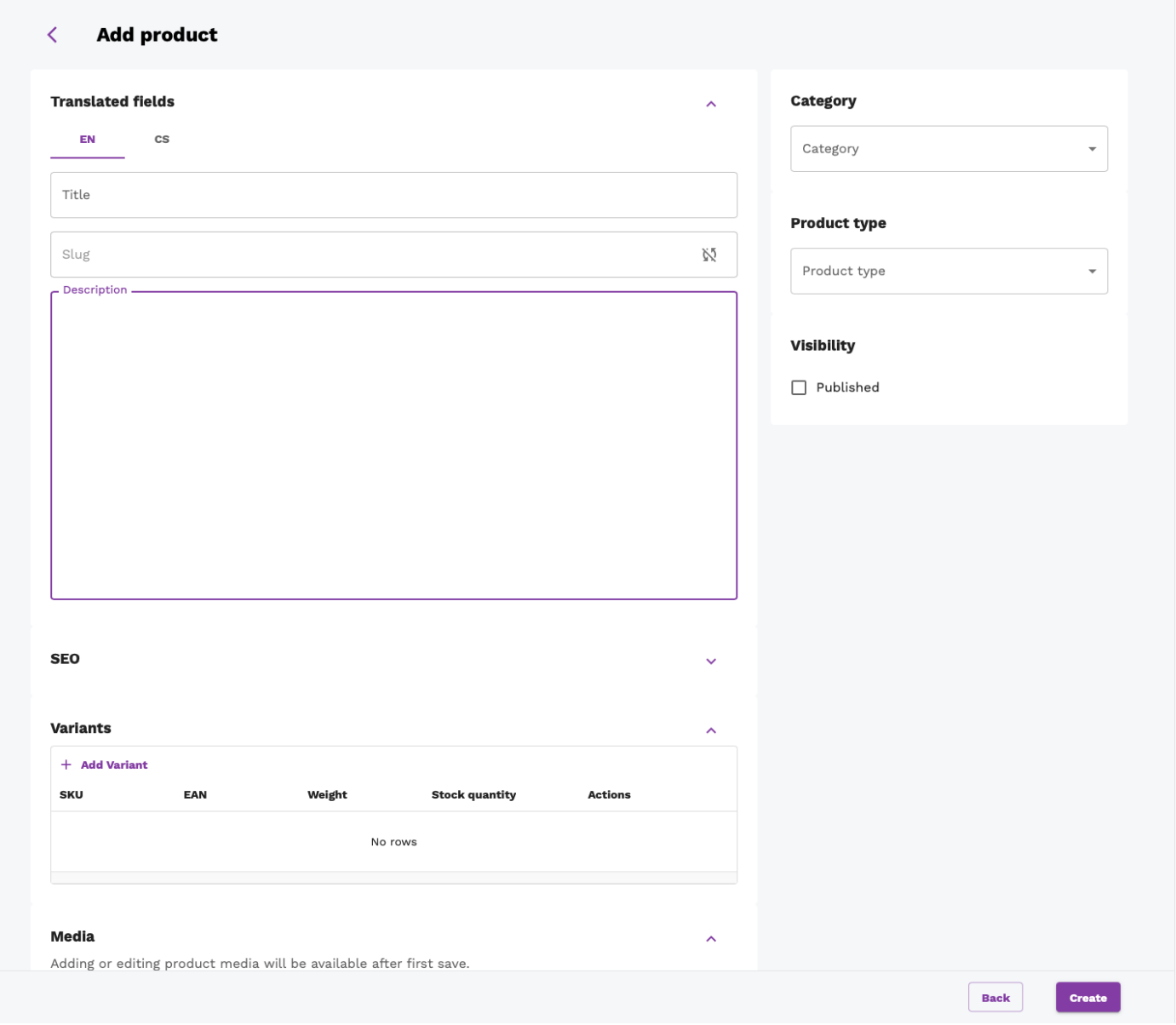


# Creating new product

If you want to create a new product, click on the button in the upper right corner above the product table - *New product*. You will be redirected to the product details page (see Editing product).



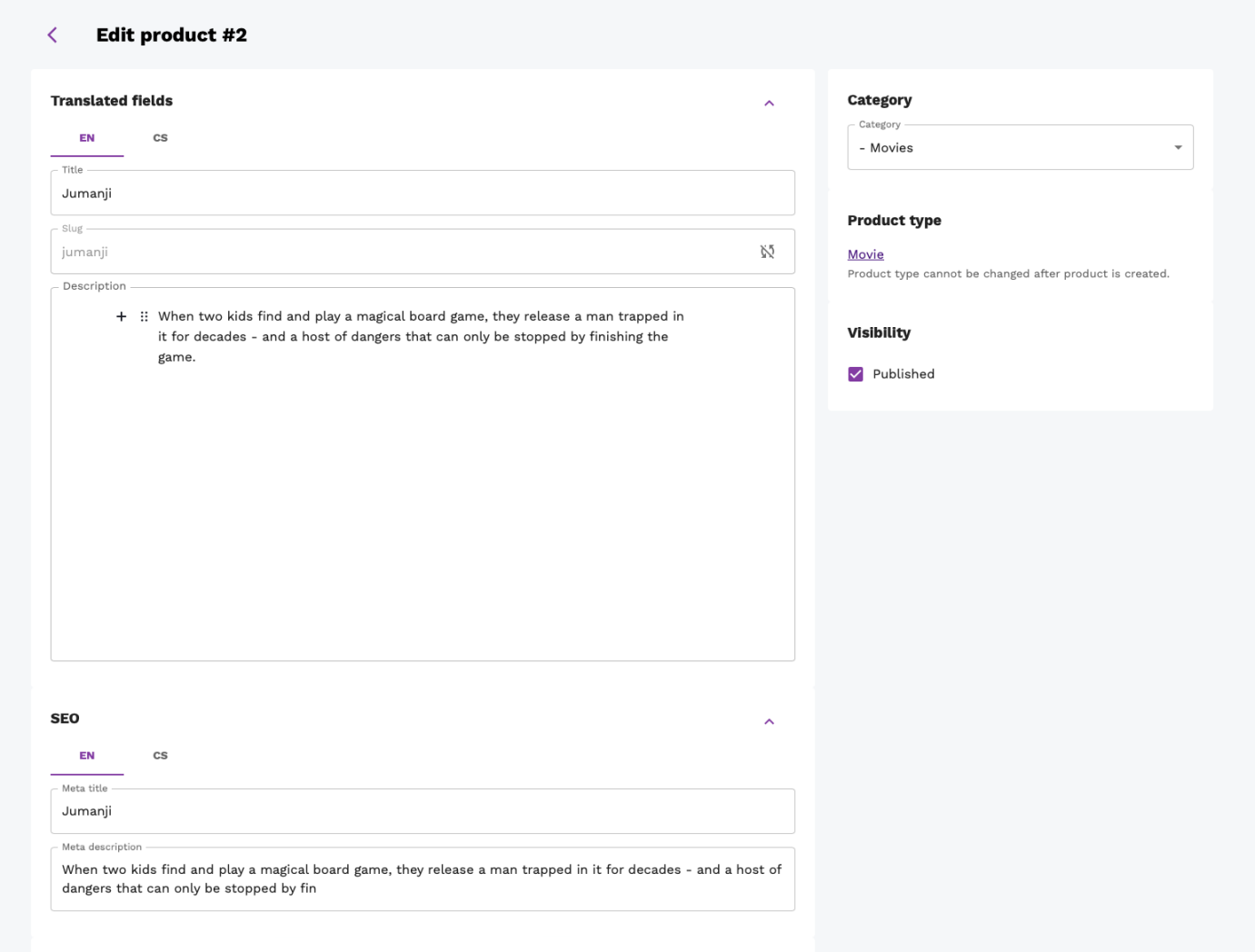

At this point, you can start setting data fields of the product. Make sure you select the product type first. The product type defines the structure of the product variant and it cannot be changed after the product is created.



After you firstly save the product, you'll be redirected to the product edit page (see Editing product).

# Editing product

Editing a product is quite large topic. It's because the product is the core of the catalog and it has many different fields. The product edit page is divided into multiple sections. Each section is described in the following subsections.



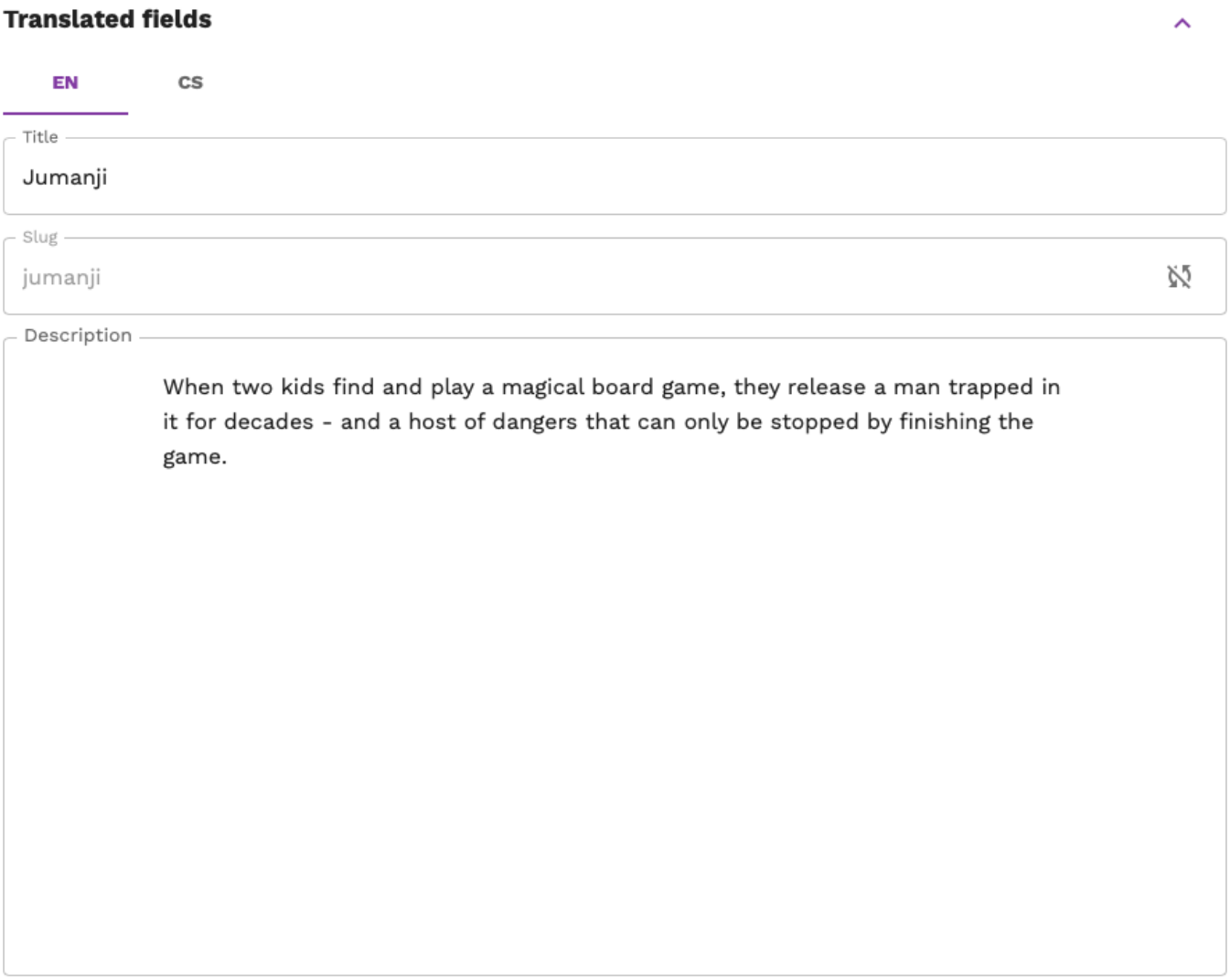

## Translated fields

This section contains the translated fields of the product. You can translate:

- Title - title of the product
- Slug - slug of the product
  - Slug is a part of the URL of the product. It's the part that comes after the domain name. For example, if the domain name is `www.example.com` and the slug is `my-product`, the URL of the product will be `www.example.com/product/{id}/my-product`.
  - It must be unique across all products
  - Slug is automatically generated from the title of the product. If you change the title, the slug will be automatically updated. If you don't want to use the automatically generated slug, you can change it manually by clicking on the sync button in the slug field.


- Description - description of the productin editorjs editor.

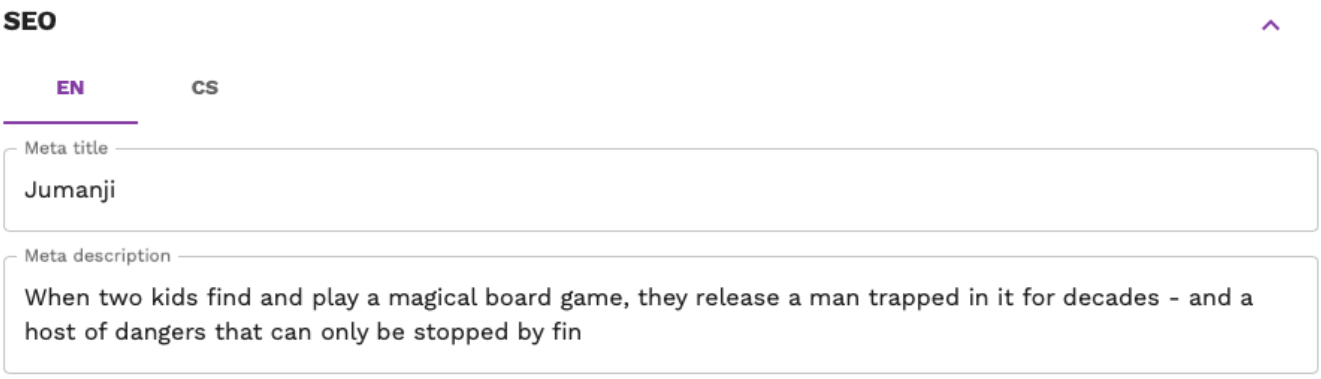
of the product into your defined languages.




## SEO

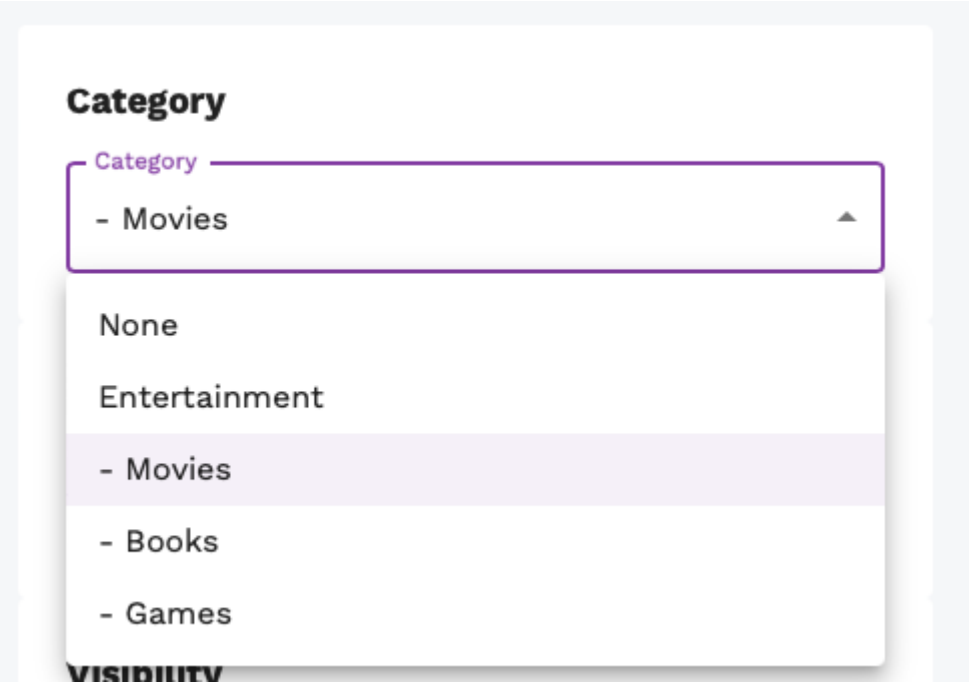
This section contains SEO fields of the product. You can set:

- Meta title - title of the product that is shown in the browser tab or in the search engine results
- Meta description - description of the product that is shown in the search engine results.




## Category

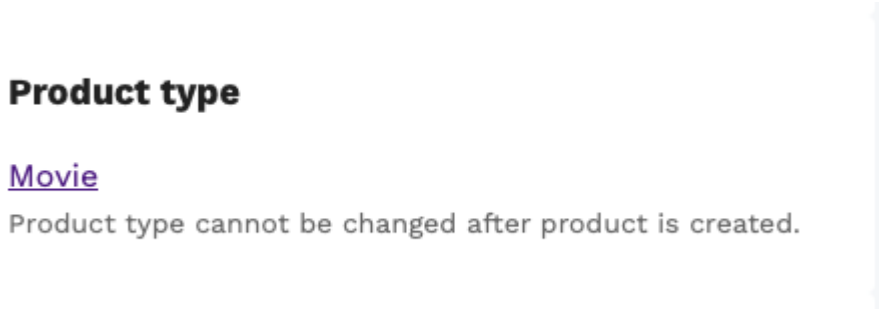
This section contains the category of the product. You can select only one category for each product.




## Product type

Product type cannot be changed after the product is created. Please see Product types and Q: Why can't I change product type after I've created the

product for more information.




## Visibility

This section contains the visibility of the product. You can set:

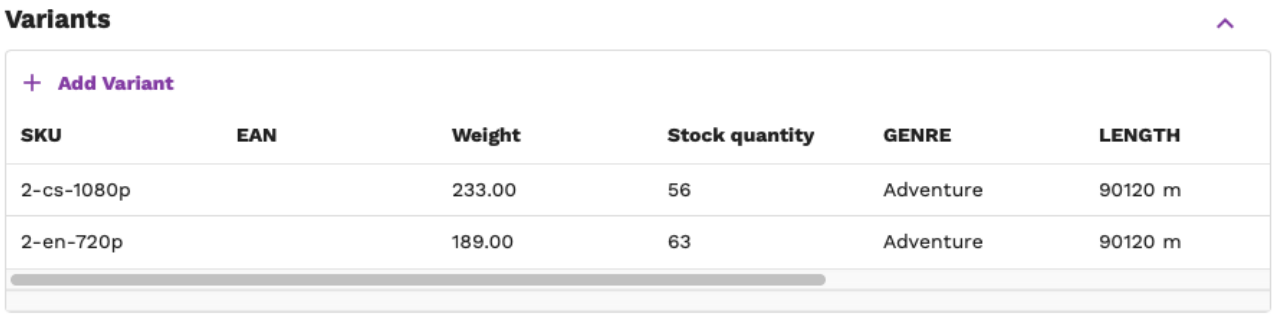
- Published - whether the product is published or not. If the product is not published, it won't be shown in the storefront.

## Product variants

This section contains the product variants of the product. You can add multiple product variants to the product. Each product variant has the following fields:

- SKU - unique identifier of the product variant. It's used to identify the product variant in the warehouse. It must be unique across all product variants.
- EAN - EAN code of the product variant. It's used to identify the product variant in the warehouse. It's not required.
- Weight - weight of the product variant in grams. It's not required.
- Stock quantity - stock of the product variant. It's not required.
- Attributes - You will see list of attributes based on your selected product type.

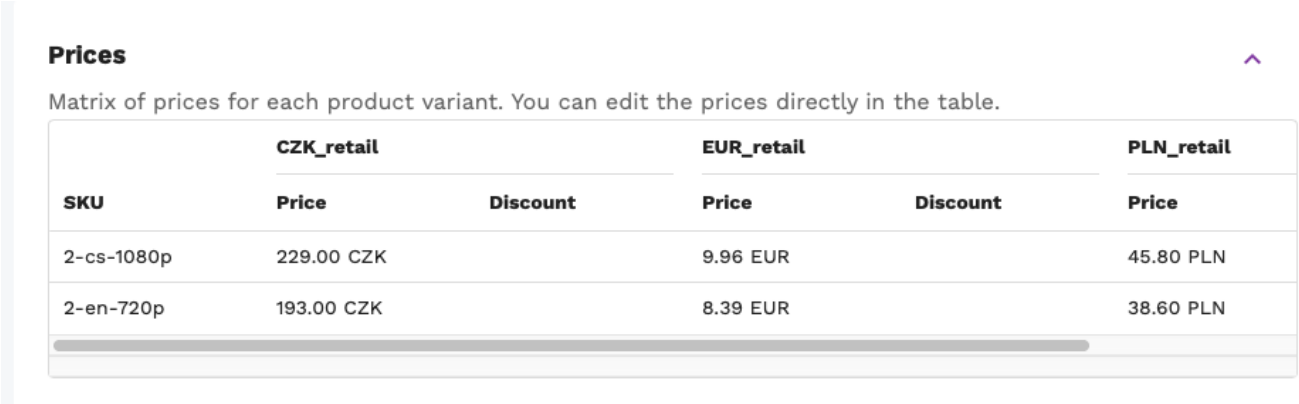
- Actions - edit and delete button




## Product prices

This section contains the prices of the product. It comes from defined price lists in the Localization section. You can set for each SKU and pricelist:

- Price excl VAT - price of the product variant without VAT. Vat is calculated based on the selected vat group in the product type for given country.
- Discount - discount of the product variant in percentage. It's not required.




## Product media

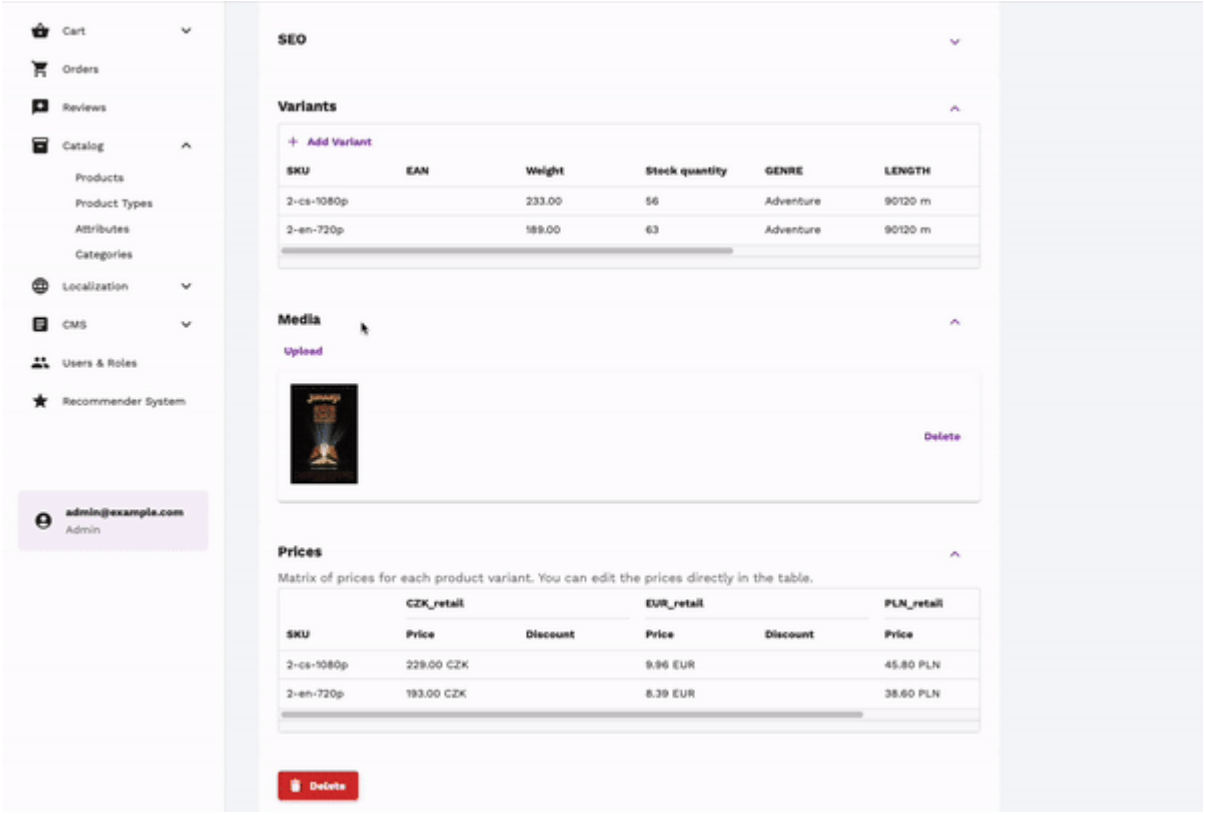
This section contains the media of the product.

### Adding images

You can add multiple media (images) to the product using the *Upload* button.

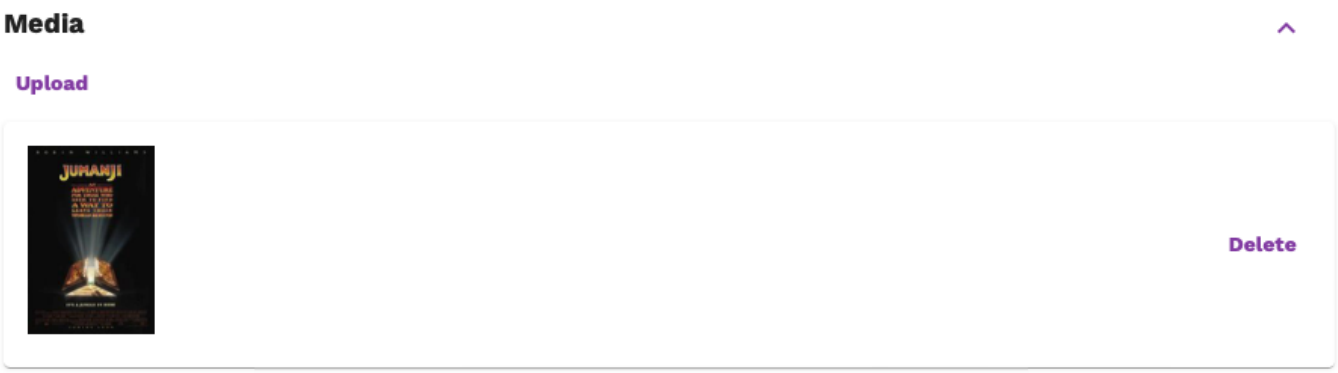
## Reordering images (setting primary image)

The primary image (shown in the category) is the first image in the list. You can reorder the images by drag and drop.




## Deleting images

You can also delete the images by clicking on the *Delete* icon.




# General FAQ about products

Here you can find some general questions about editing products and variants.

### Q: What is the difference between product and product variant?

A: Product is a wrapper around multiple product variants. Imagine that you're selling t-shirts. You have a t-shirt in two sizes (S and M) and two colors (red and blue). This means that you have 4 product variants. But you have only one product. The product is the t-shirt and the product variants are the t-shirt in different sizes and colors. Product variant is what a actually ship from your warehouse. It's the actual product that the customer buys. In the example above, the product variant is the t-shirt in size S and color red.

### Q: Why do I need to create product type?

A: With selected product type at product creation, you can define the structure of the product variant. For example, if you want to sell t-shirts, you need to define attributes like size, color, etc. Product types are then used in the product to define the structure of the product variant. Product types are also used in the product variant to define the actual values of the attributes. So imagine you have product type `CLOTHING` with attribtues `size` and `color`. If your product is type `CLOTHING`, it's variants need to define values for `size` and `color` attributes.

### Q: Why can't I change product type after I've created the product?

A: You can't change product type after you've created the product because the product type defines the structure of the product variant. If you change the product type, you would need to change the structure of the product variant. This would mean that you would need to change the values of the attributes of the product variant.

### Q: Why do I need to create product variant?

A: Product variant is what a actually ship from your warehouse. It's the actual product that the customer buys.

# Categories

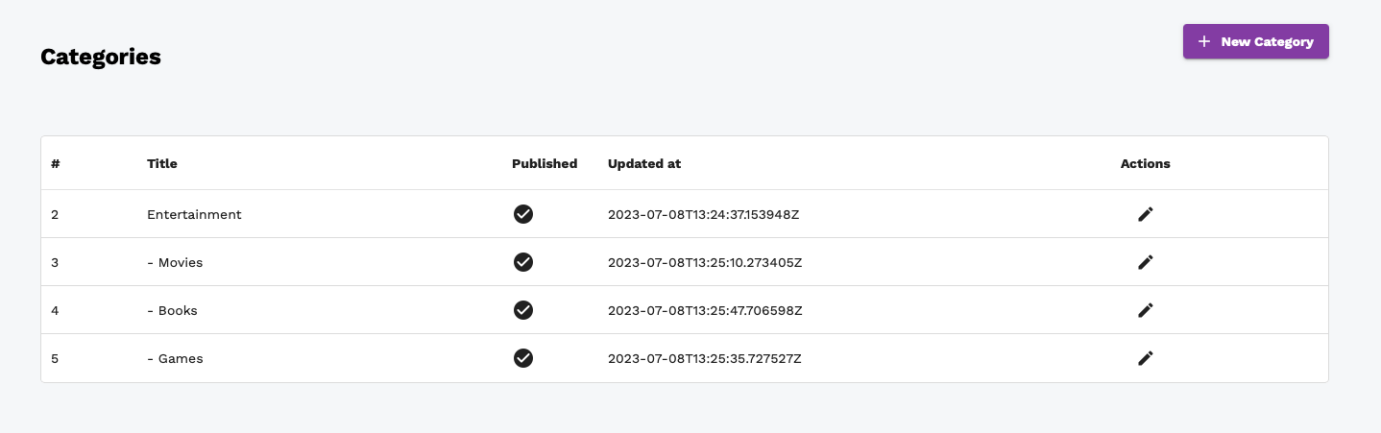
In order to group products into categories, you need to create categories. Categories can be nested - this means that you can create a category and then create a subcategory of that category. You can create as many subcategories as you want. The nesting is not limited. The categories are shown in the storefront in the navigation menu up to the third level of nesting.

---

## Category list

The category list shows all categories in the store. The list has the following columns:

- ID
- Title - the title of the category with the nesting level shown via indentation
- Published - whether the category is published or not (visible and browsable in the storefront)
- Updated at
- Actions - edit button

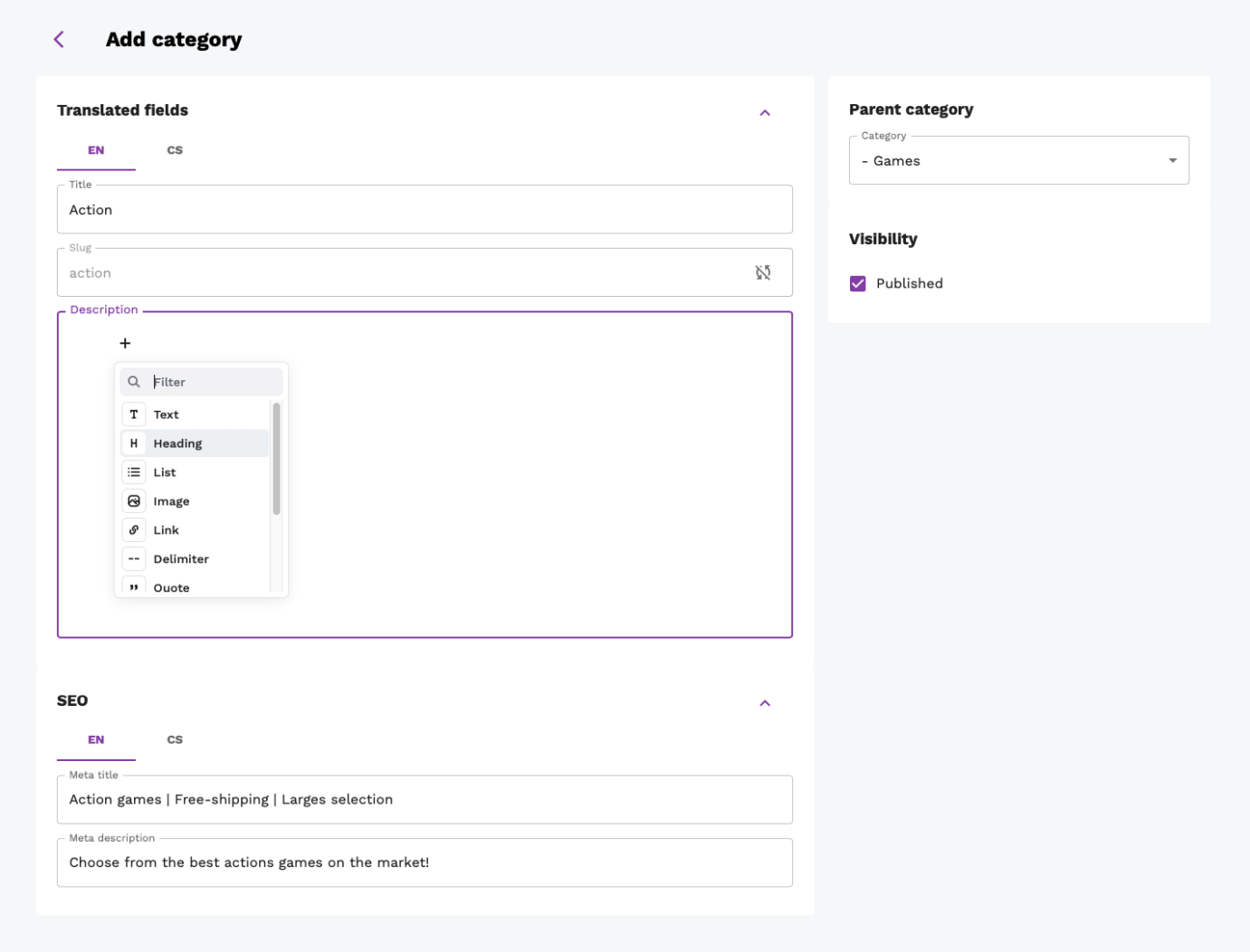



## Creating new category

To create a new category, click on the button in the upper right corner *New Category*. You will be redirected to the category details page. Now you can follow steps described in the editing category section.

## Editing category

To edit a category, click on the edit button in the category list. This opens the category details page.




1. In the *Translated fields* section, you can:
    1. Edit or add title of the category. The title is required and doesn't have to unique.
    2. Edit or add slug of the category. Slug is required and has to be unique. The slug is used in the URL of the category page in the storefront and is generated automatically from the title. You can change the slug to anything you want, but it has to be unique. If you change the slug, the URL of the category page in the storefront will change. In order to change the slug, click the button in the right corner of the slug field.
    3. You can also add a description of the category. The description is optional and is in the *editorjs format*. The description is shown in the storefront on the category page.

2. In the *SEO* section, you can:
    1. Edit or add meta title of the category. The meta title is optional and doesn't have to be unique. The meta title is shown in the browser tab and in the search results. Otherwise, title of the category is shown.
    2. Edit or add meta description of the category. The meta description is optional and doesn't have to be unique. The meta description is shown in the search results.
3. In the *Parent category* section, you can:
    1. Select parent category of the category. The parent category is optional. If you select a parent category, the category will be nested under the parent category. The nesting is not limited. The categories are shown in the storefront in the navigation menu up to the third level of nesting. If you don't select parent, the category will be a root category.
4. In the *Visibility* section, you can:
    1. Select whether the category is published or not. If the category is published, it is visible and browsable in the storefront. If the category is not published, it is not visible and not browsable in the storefront.

# Localization

The localization part of the dashboard is used to manage country-specific parts of the system. This includes:

- Countries
- Vat Groups
- Price Lists
- Currency

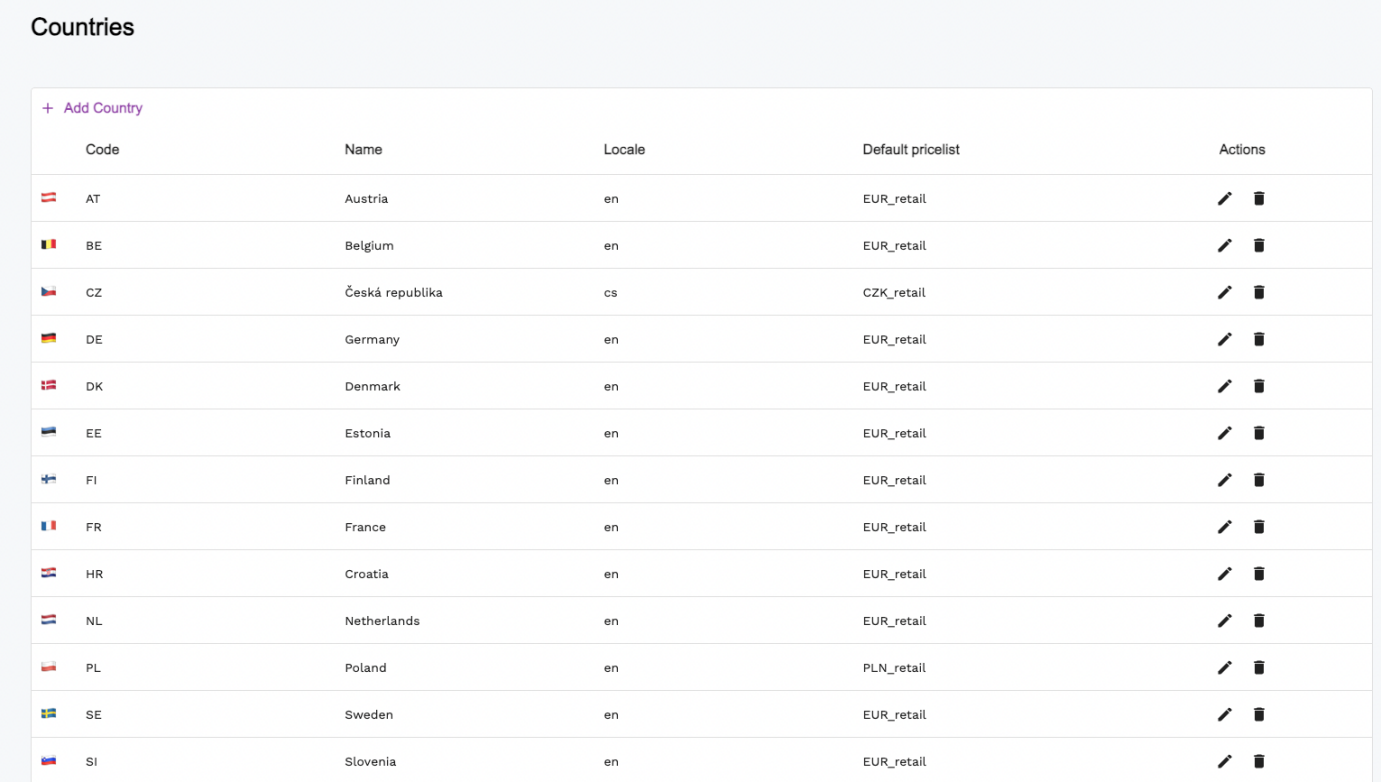
Each of these parts has its page in the dashboard. We will describe them in more detail in the following sections.

# Countries

This page consists of a list showing all countries. The list has the following columns:

- Code
- Name
- Locale
- Default pricelist
- Actions
    - Edit - click on the edit button unlocks the row for editing
    - Delete - click on the delete button deletes the country

To add a new country, click on the *Add New* button in the upper left corner of the table. This adds a new row to the table. Fill in the code, name, locale and default pricelist of the country and click on the *Save* icon.

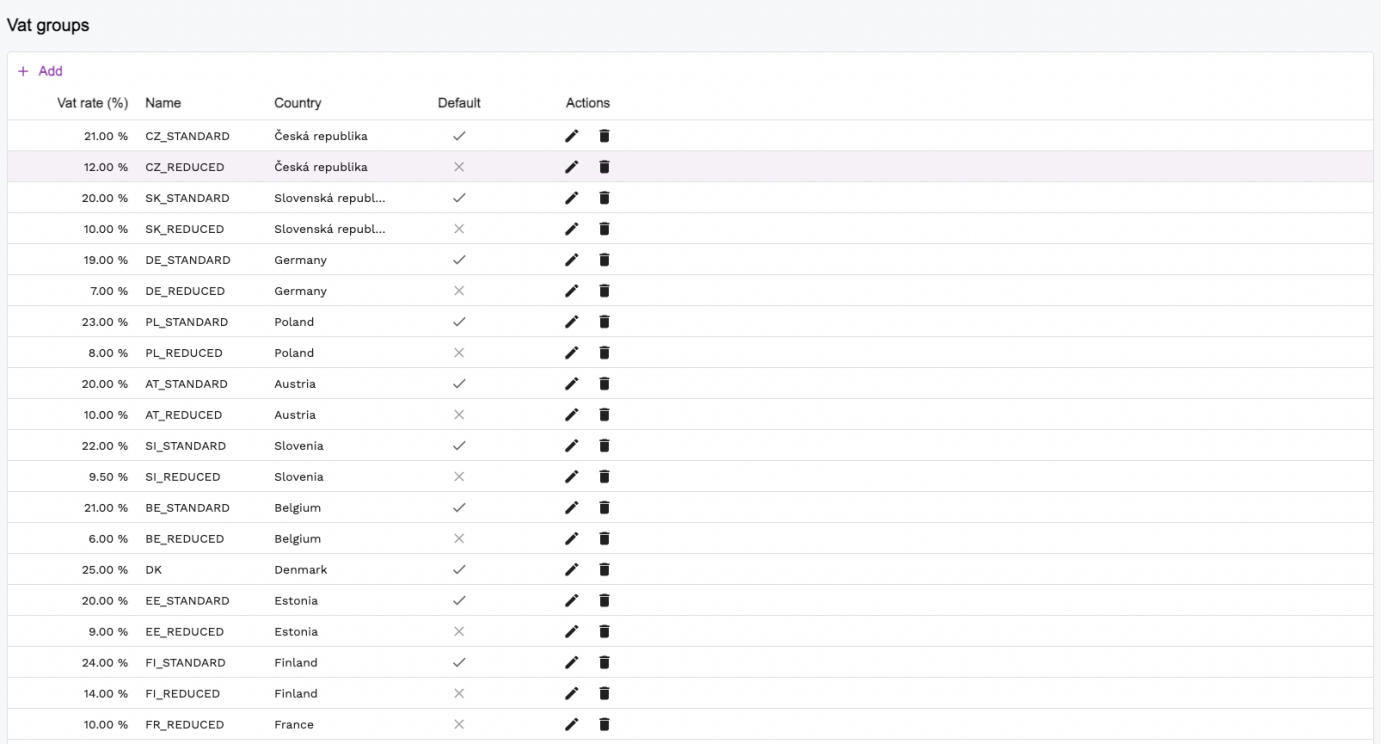



# Vat Groups

This page consists of a list showing all vat groups. The list has the following columns:

- Vat rate (in %)
- Name
- Country
- Default
- Actions
  - Edit - click on the edit button unlocks the row for editing
  - Delete - click on the delete button deletes the vat group

To add a new vat group, click on the *Add New* button in the upper left corner of the table. This adds a new row to the table. Fill in the vat rate, name, and country, set the default option and click on the *Save* icon.

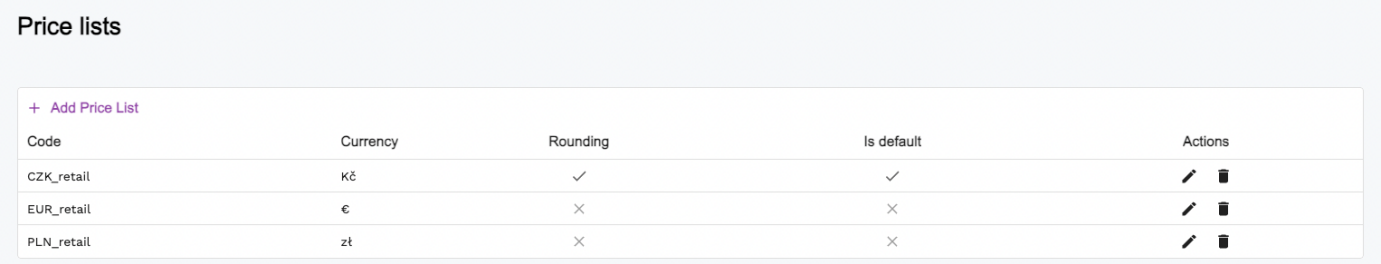



# Price Lists

This page consists of a list showing all price lists. The list has the following columns:

- Code
- Currency
- Rounding
- Is default
- Actions
  - Edit - click on the edit button unlocks the row for editing
  - Delete - click on the delete button deletes the price list

To add a new price list, click on the *Add New* button in the upper left corner of the table. This adds a new row to the table. Fill in the code and currency, set rounding and default option and click on the *Save* icon.

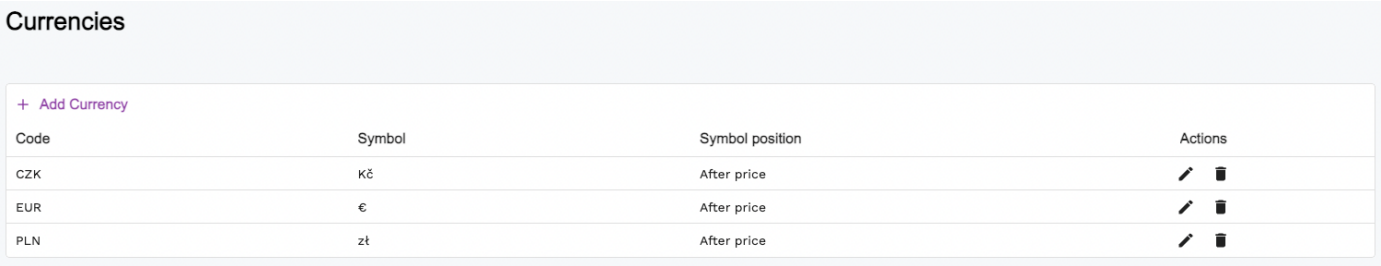



# Currency

This page consists of a list showing all currencies. The list has the following columns:

- Code
- Symbol
- Symbol position
  - Before price
  - After price
- Actions
  - Edit - click on the edit button unlocks the row for editing
  - Delete - click on the delete button deletes the currency

To add a new currency, click on the *Add New* button in the upper left corner of the table. This adds a new row to the table. Fill in the code and symbol, set the symbol position and click on the *Save* icon.



# CMS

This section describes creating and editing pages and menus (of pages). We have two types of pages:

- CMS Page
  - CMS Page is a page that is created and edited by the admin in the dashboard. It can contain any content that the admin wants to show to the user. The admin can create as many CMS pages as he wants.
- Storefront Link
  - A storefront page/link is a special type of page that lives in the storefront and is created and edited by the storefront programmer. In the dashboard, you can just create a piece of information about the page and link it to the storefront page. This is useful when you want to create a link to a page that is not a CMS page (e.g. something with more CSS and Java/TypeScript).

Both pages can be categorized into “menus” - this means that you can create a menu and add pages to it. The menu can then be fetched from the storefront.

## Pages

The pages page consists of a list showing all pages. The list has the following columns:

- ID
- Title
- Type - CMS or Storefront

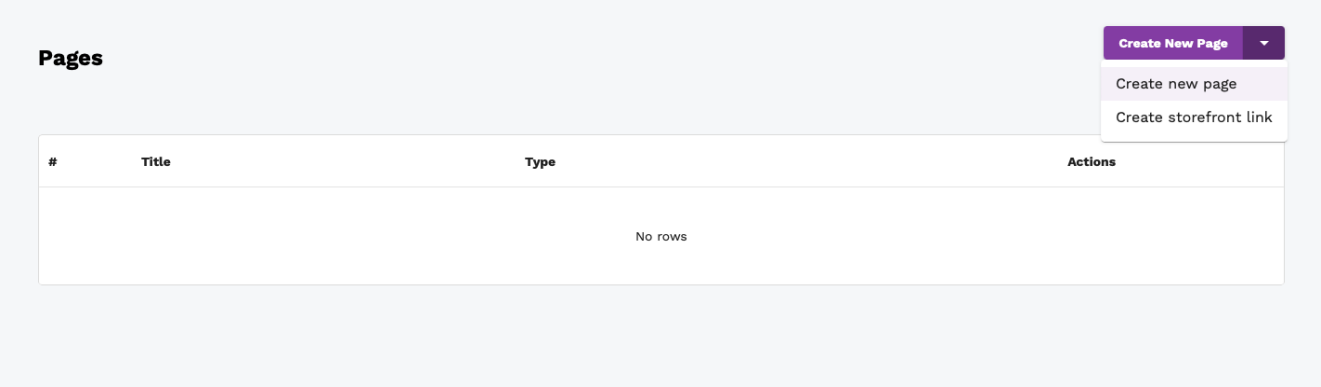
- Actions

---

# Creating a new CMS page

To create a new CMS page, click on the arrow next to *Create New Page* button in the upper right corner. This opens a drop-down menu with two options:

- Page
- Storefront link Select the *Page* option to create a new CMS page and click the button.




---

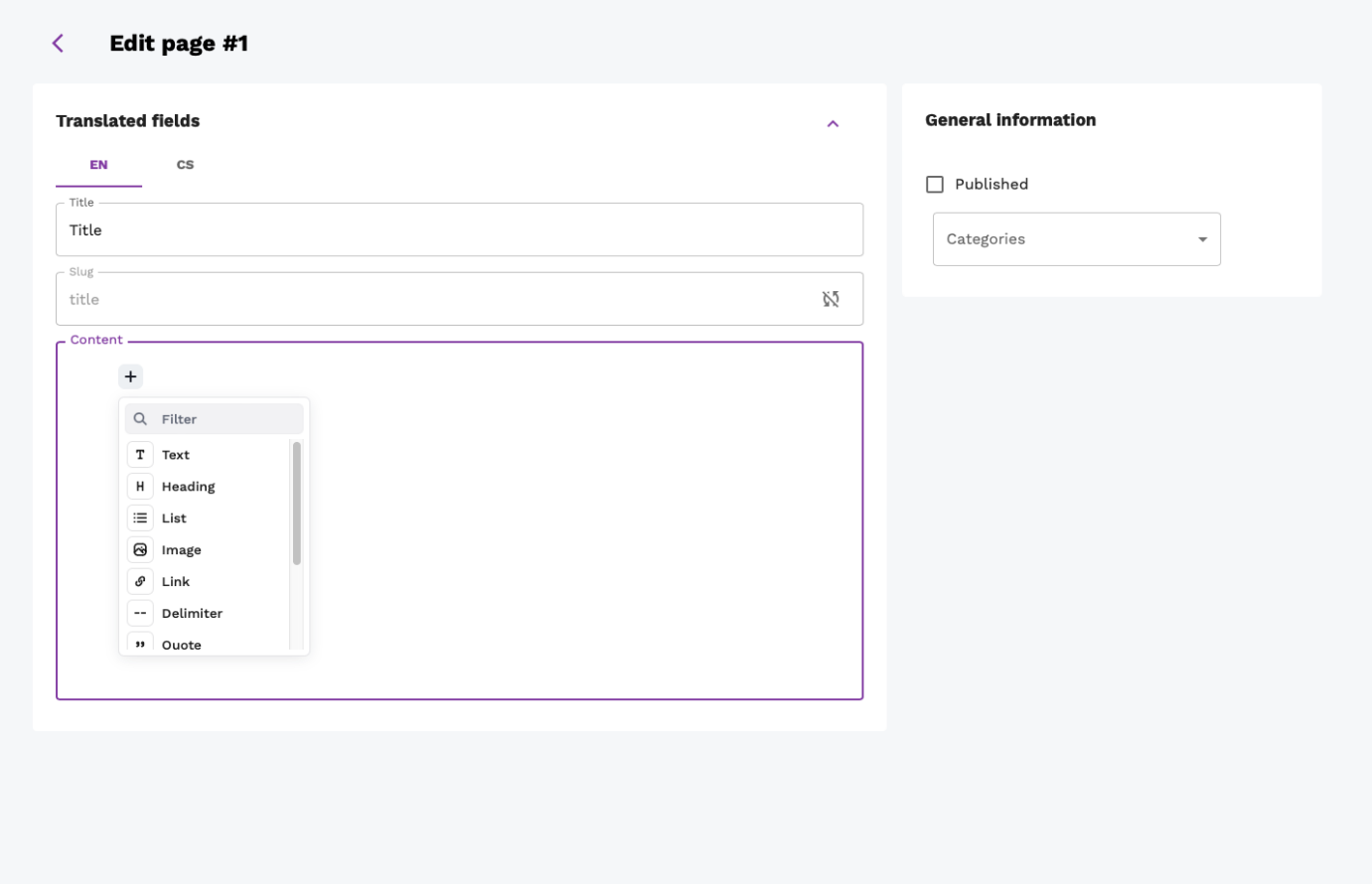
# Editing CMS page

Simply click on the edit icon of the page that you want to edit in the list. This opens a page with the following fields:

- Title
- Slug - this is the URL of the page. It is automatically generated from the title, but you can change it if you want. Just click on the *sync* button in the

slug field. Just make sure that the slug is unique.

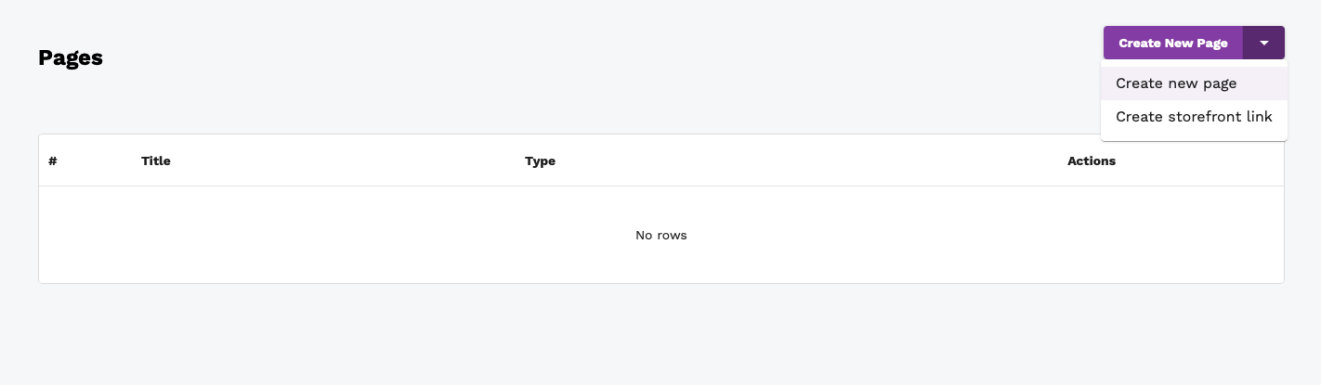
- Content - this is the content of the page. Use *editorjs* to create the content. You can find more information about *editorjs* here.
- Published - this is a checkbox that indicates whether the page is published or not. If the page is not published, it will not be shown in the storefront and will not be accessible via the URL - a 404 error will be shown instead.
- Categories - this is a list of categories that the page belongs to.




# Creating a new storefront link

To create a new storefront link, click on the arrow next to *Create New Page* button in the upper right corner. This opens a drop-down menu with two options:

- Page
- Storefront link Select the *Storefront link* option to create a new storefront link and click the button.

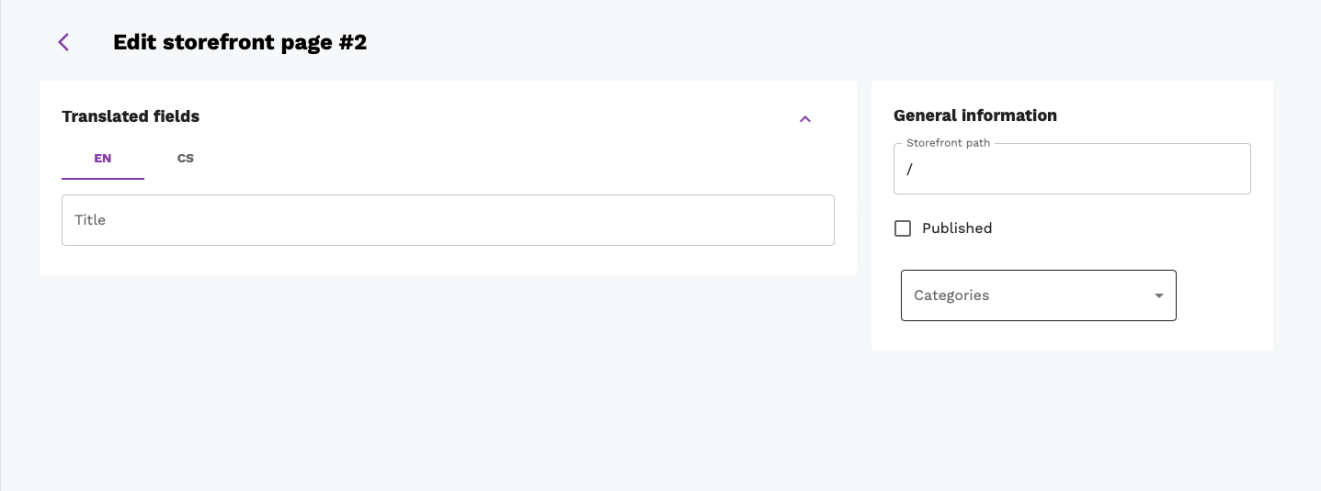



# Editing storefront link

Simply click on the edit icon of the page that you want to edit in the list. This opens a page with the following fields:

- Title - this is the title of the page. It is shown on the storefront.
- Storefront path - this is the path to the storefront page.
- Published - this is a checkbox that indicates whether the page is published or not. If the page is not published, it will not be shown in the storefront and will not be accessible via the URL - a 404 error will be shown instead.

- Categories - this is a list of categories that the page belongs to.




# Categories & Types

This page consists of two parts:

- Categories
- Page category types

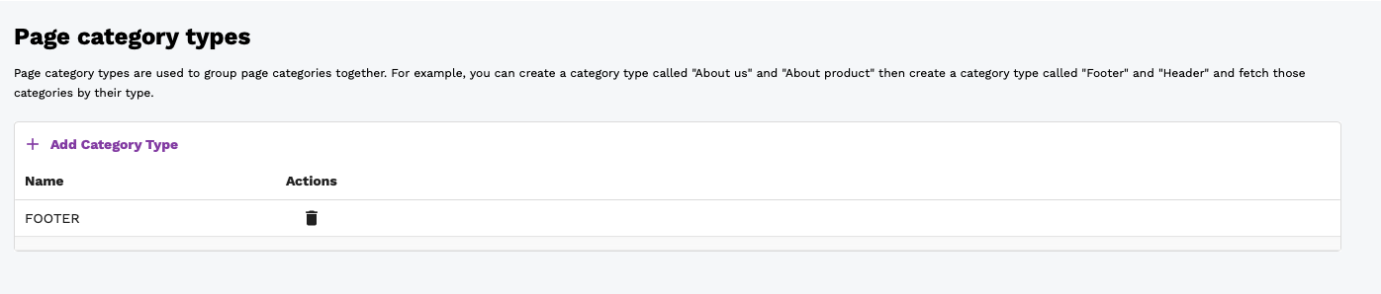
**Page category type** works as a grouping of **categories***. *It is used to group categories that are used for the same purpose. For example, you can create a page category type called *FOOTER* and add categories *About us*, *Contact us* and *Terms and conditions* to it. Then you can use this page category type in the footer of the storefront - since it's automatically fetched from the dashboard.

---

## Page category type

This section shows a list of all page category types. The list has the following columns:

- Name (unique code)
- Actions (delete)




## Creating a new category type

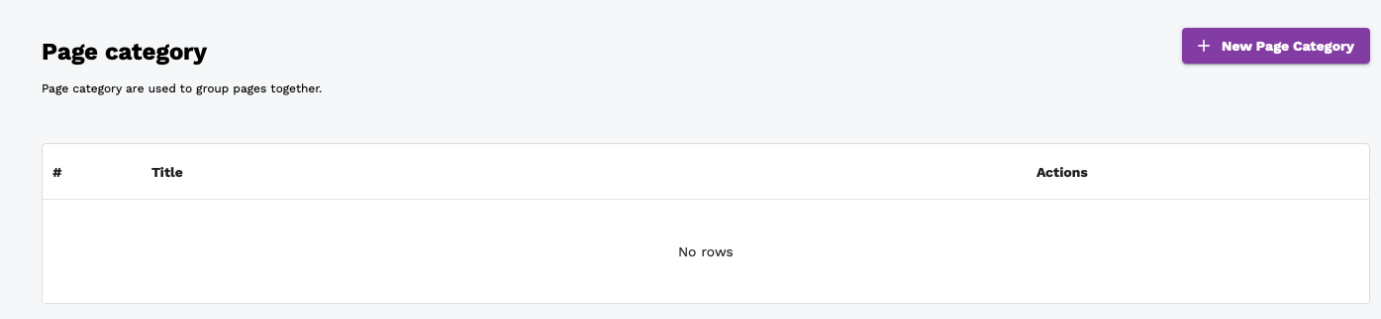
In order to create a new page category type, click on the *Add category type* button in the upper left corner of the table. This adds a new row to the table. Fill in the unique name of the page category type and click on the *Save* button. Since the **Name** field is unique and serves as an identifier of the page category type, it cannot be changed thus editing is not allowed.

---

# Categories

This section shows a list of all categories. The list has the following columns:

- ID
- Title
- Actions



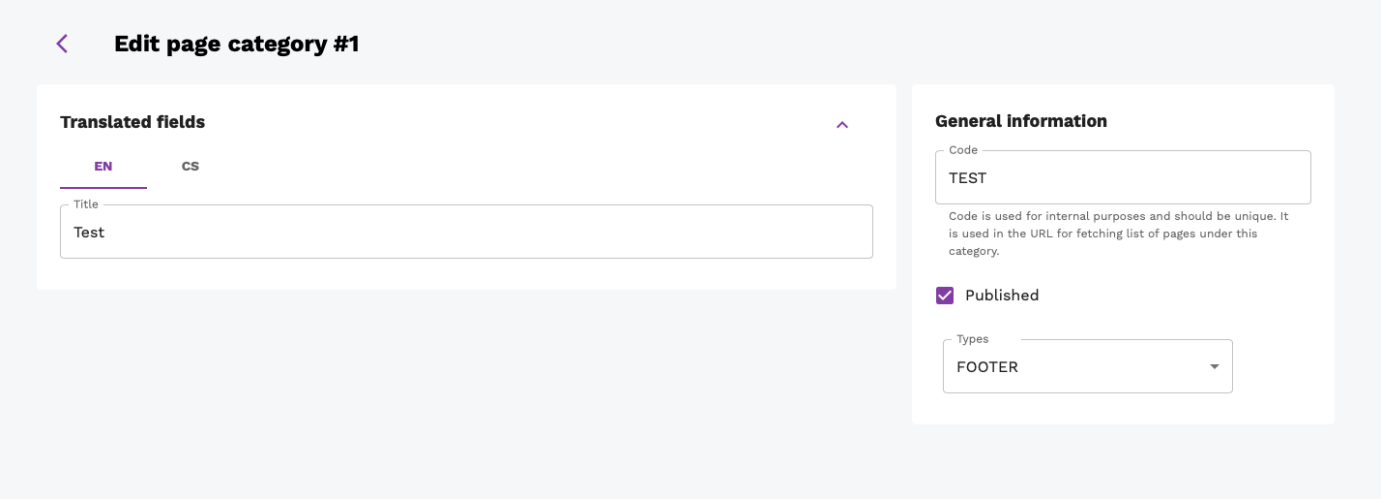

## Creating a new category

In order to create a new category, click on the *New page category* button in the upper left corner of the table. This redirects you to the detail category page. Now follow the steps the same as for editing a category (see Editing a category).

## Editing a category

In order to edit a category, click on the edit icon in the table. This redirects you to the detail category page.

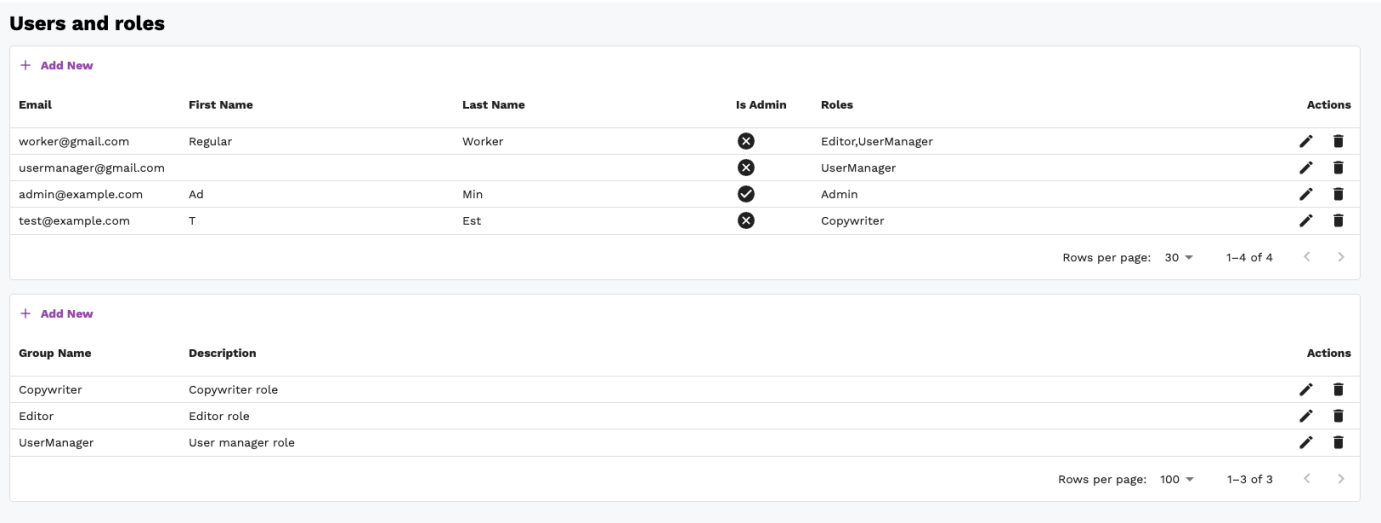
1. Fill in the title of the category
2. Select the page category type from the drop-down menu. Set the unique code of the category. This code is used to identify the category in the storefront. It is also used to fetch the category as a group. Set the published checkbox. If the category is not published, it will not be shown on the storefront. Then click on the *Save* button.




# Users & Roles

This page provides an overview of all users and roles. It is into two main parts:

- Users
- Roles




# Users

This section shows a list of all users. The list has the following columns:

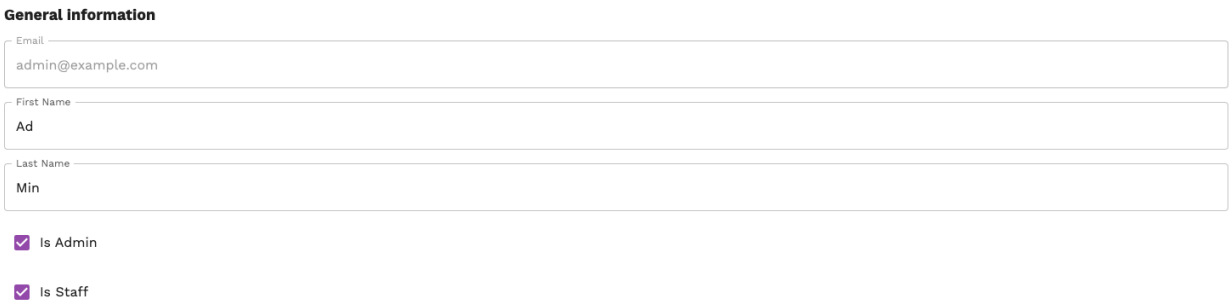
- Email
- First name
- Last name
- Is Admin
- Roles
- Actions
  - Edit - click on the edit button opens the user details page (see User details)
  - Delete - click on the delete button deletes the user

---

# User details

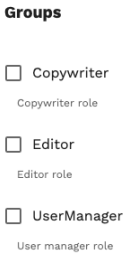
The user details page shows full information about the user. The page is divided into three main parts:

- General information:
  - Email (cannot be changed)
  - First Name
  - Last Name
  - Is Admin - only admin can change this value
  - Is Staff



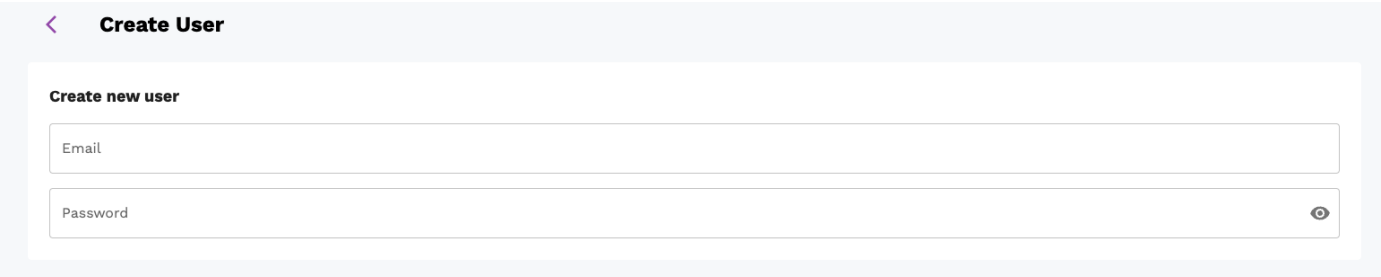
- Password
  - Old password - displayed only if a user is editing his profile
  - New password
  - New password confirmation


- Roles - a checklist of all roles available in the system. If a user has a role assigned, the checkbox is checked.




# Create user

To create a new user, click on the *Add New* button in the users list. This opens a new page where the user can set the email and password.




# Roles

Roles are used to group various permissions into one unit. This plays a crucial role in the authorization process as it restricts access to certain parts of the system. Only the admin or users with the `user_change_permission` permission can edit the user (for more detailed info about permissions see Authorization page). The system comes with three predefined roles:

- Editor
- Copywriter
- UserManager

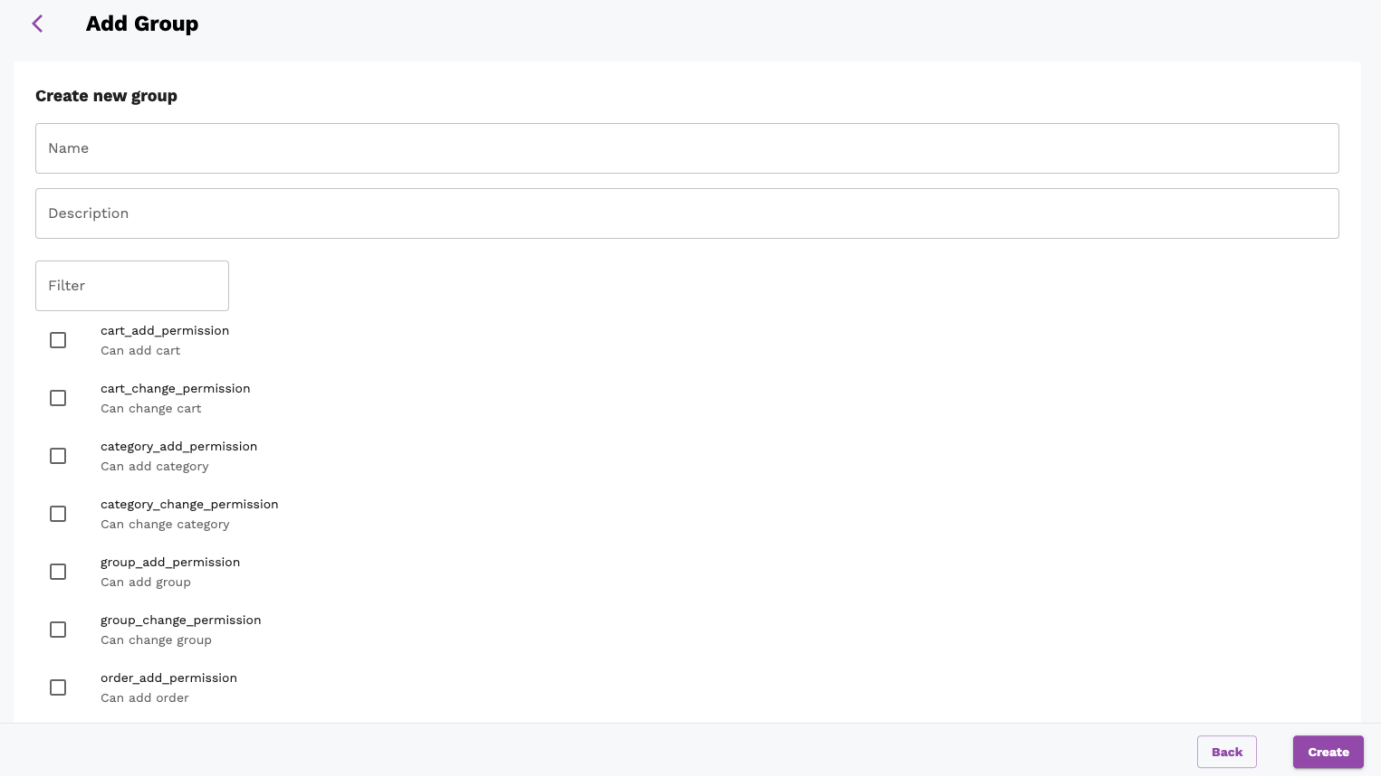
Admin counts as a special role that has all permissions. Authorized users can create new roles or edit existing ones.

---

## Create role

To create a new role, click on the *Add New* button in the roles list. This opens a new page where the user can enter the name of the role, its description and

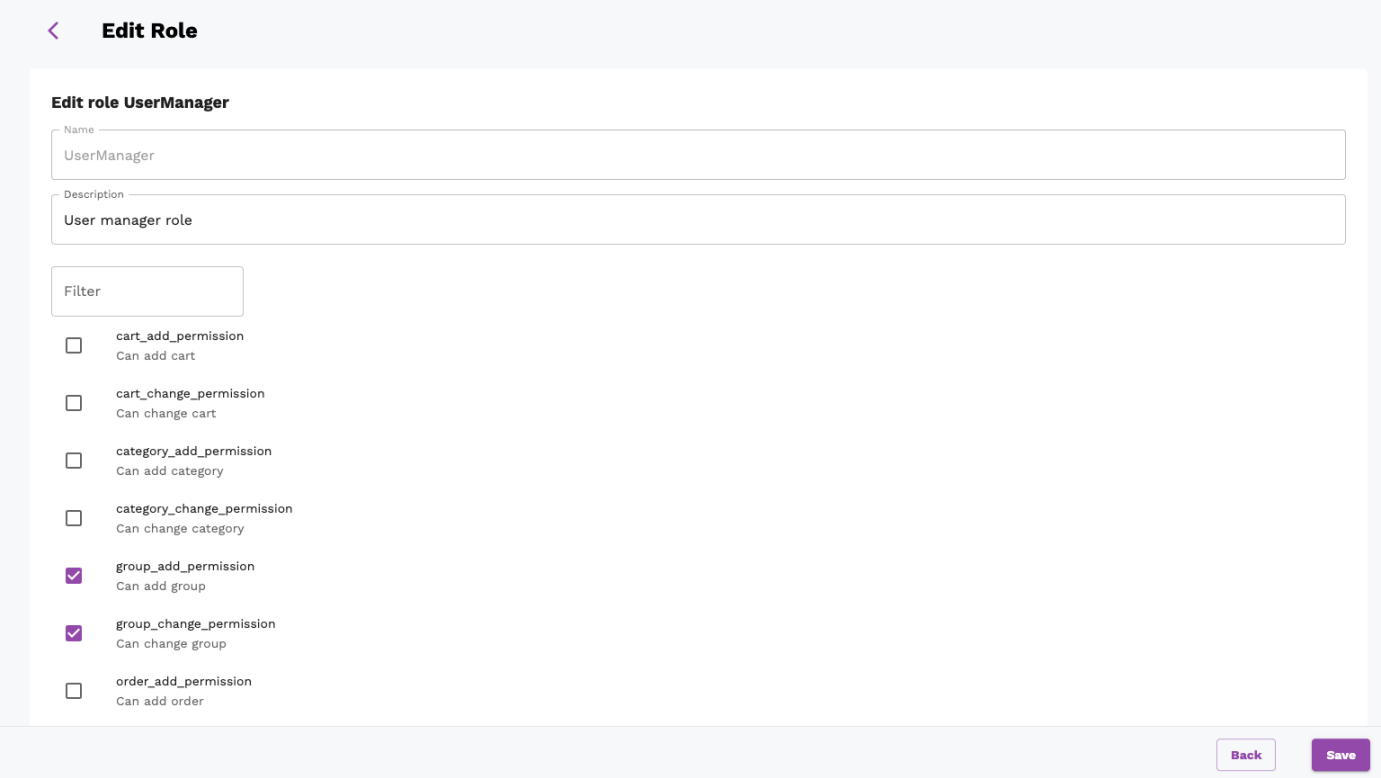
select permissions that will be assigned to the role.




# Edit role

To edit an existing role, click on the edit button in the roles list. This opens a new page where the user can edit a description of a given role and select

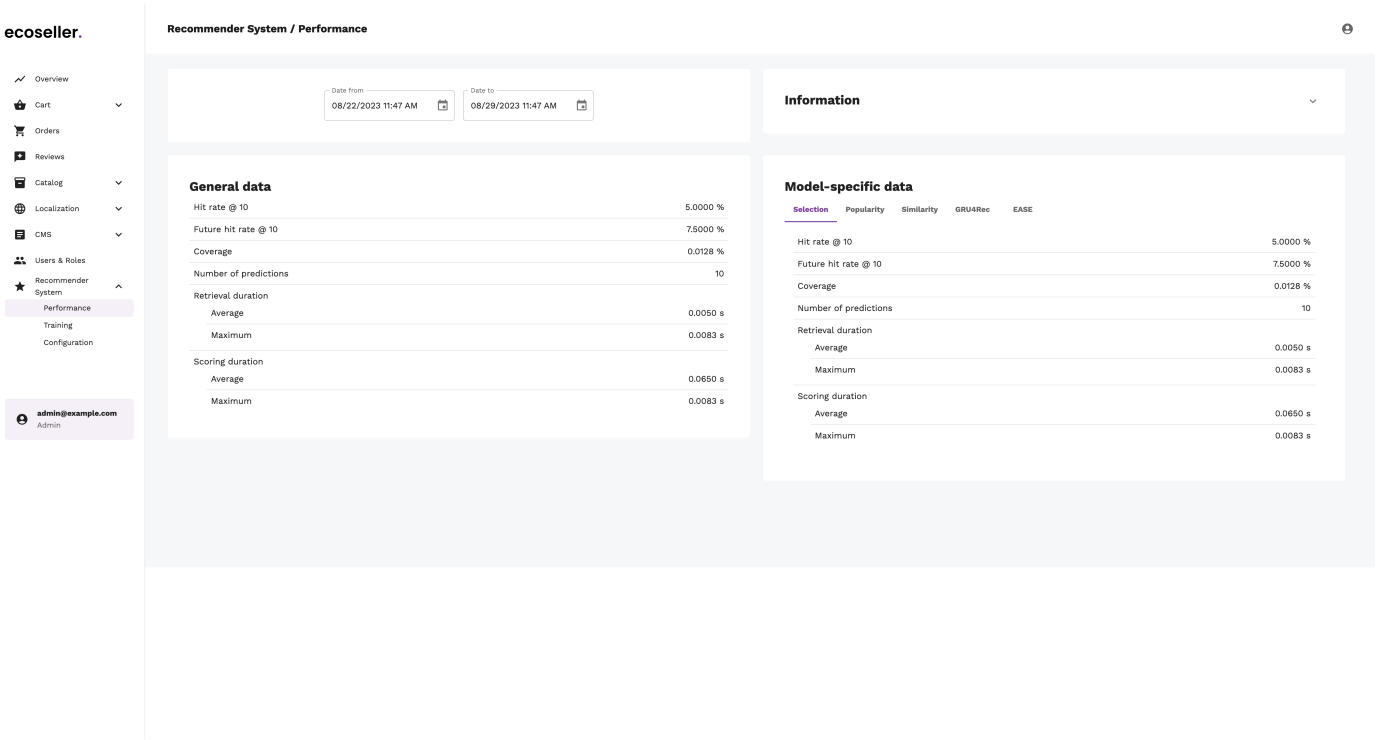
permissions that will be assigned to the role.



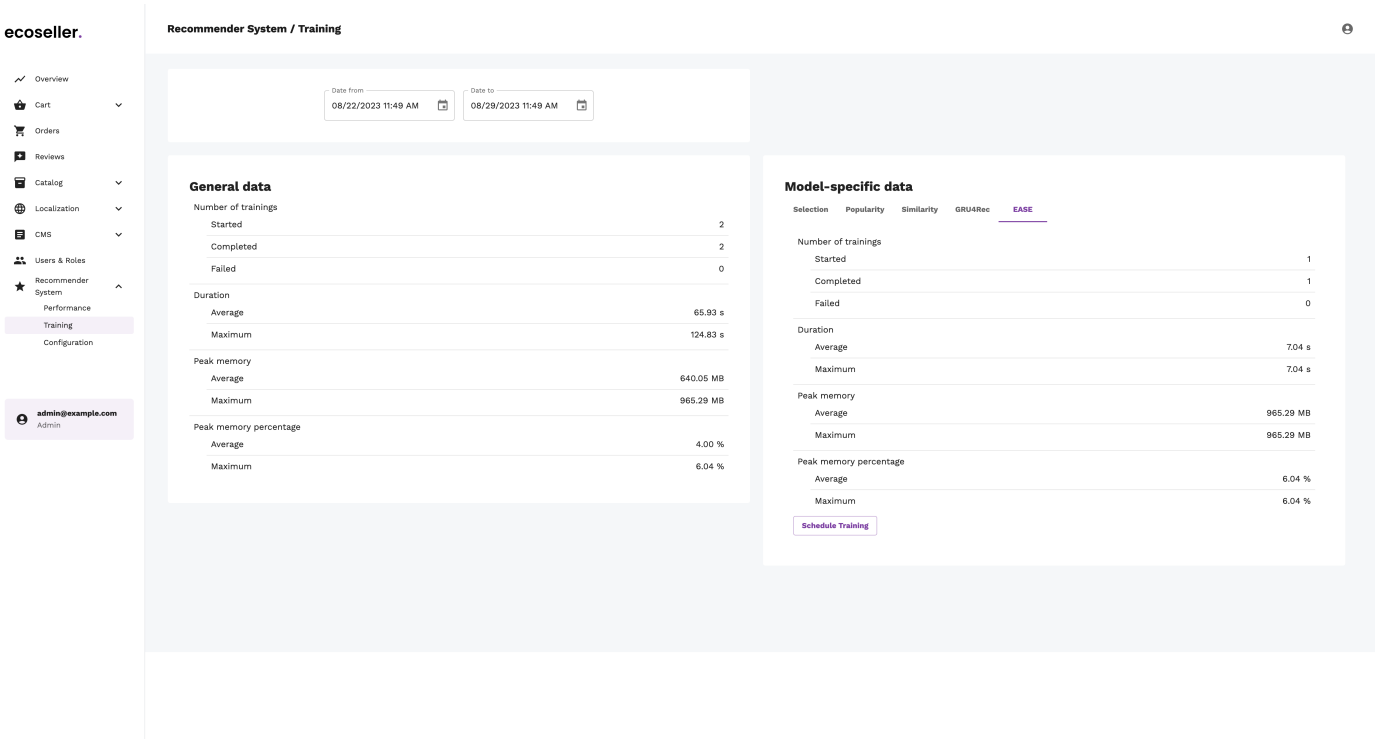

# Recommender system

This section of the dashboard contains information regarding the Recommender system. It is divided into 3 pages.

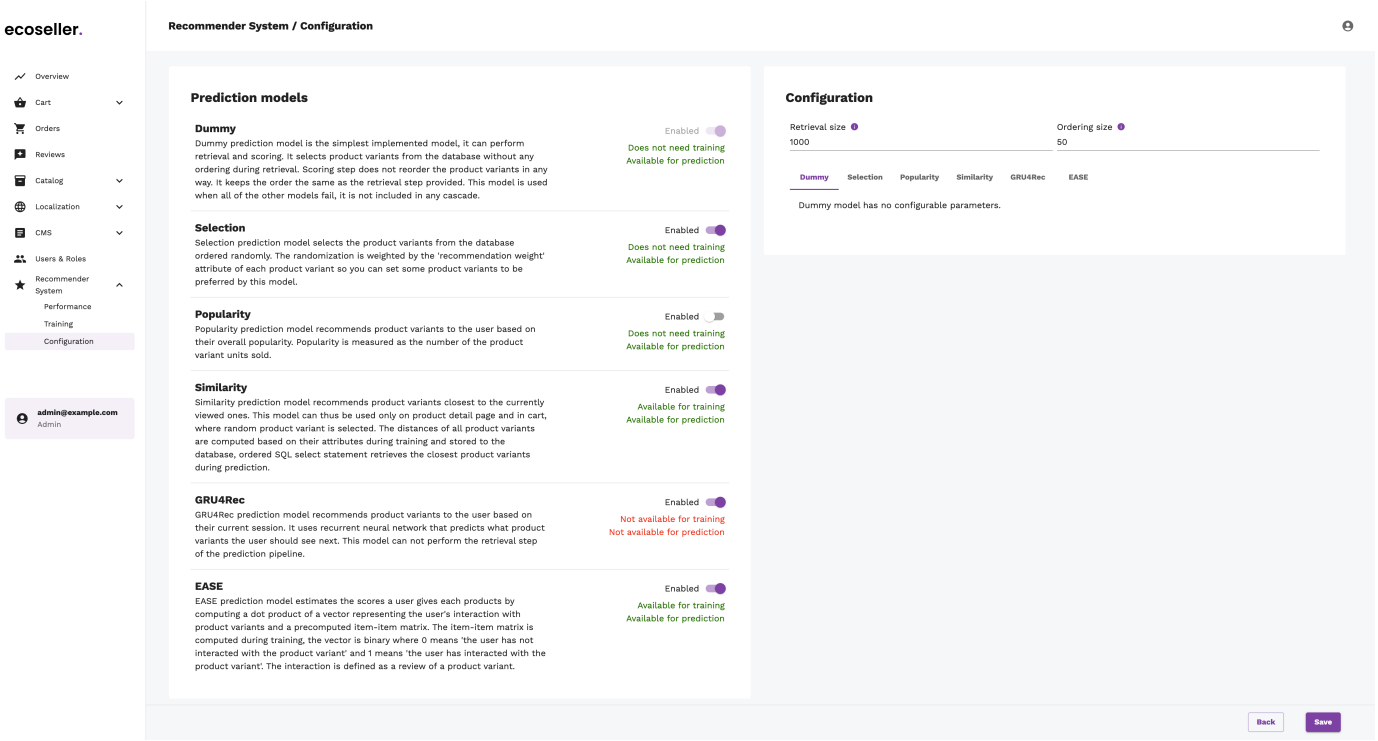
*Performance* page contains information about the Recommender system's performance in the selected time window. Users can inspect metrics and other prediction-related information here.




*Training* page contains information about trainings of all prediction models. Each model can also be scheduled to be re-trained.



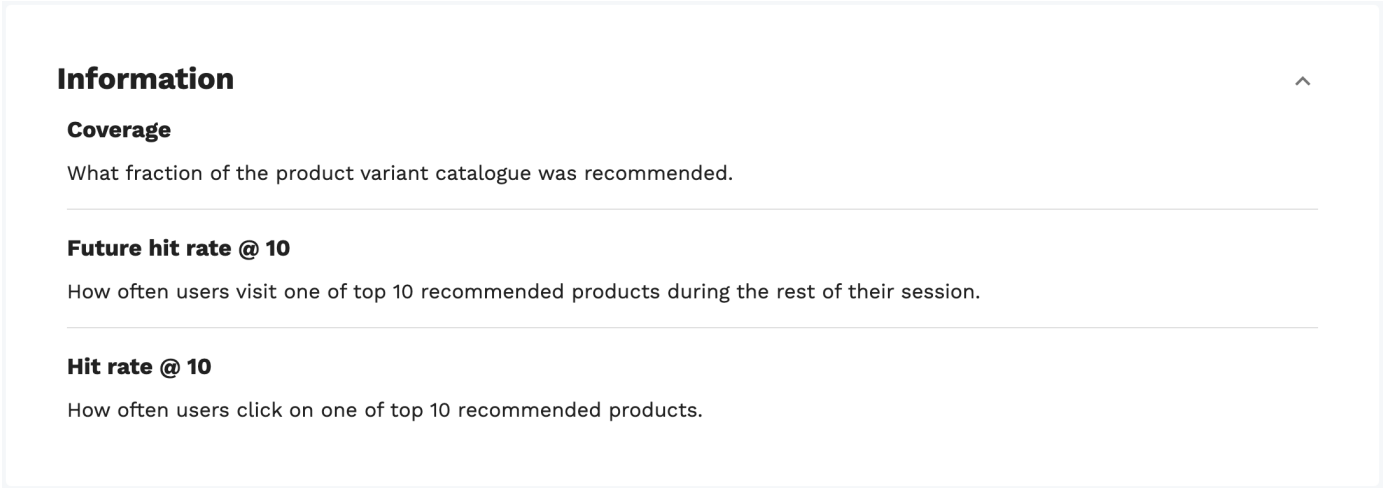

*Configuration* page enables user to configure the Recommender system and its individual models in real-time. Each model can also be disabled.




# Performance

Users can select time window for which to display performance data.

All metrics are described in the *Information* section on this page. The default metrics are coverage, hit rate @ $k$ and future hit rate @ $k$.




Coverage is the fraction of the product variant catalogue that was recommended, hit rate @ $k$ specifies how often the users click on one of top

$k$ recommended product variants. Future hit rate @ $k$ specifies how often users visit one of the top $k$ recommended product variants during the rest of their session.

Number of predictions and their duration is displayed for the user as well.

These statistics are computed for the whole Recommender system (*General data*) and for each prediction model individually (*Model-specific data*).

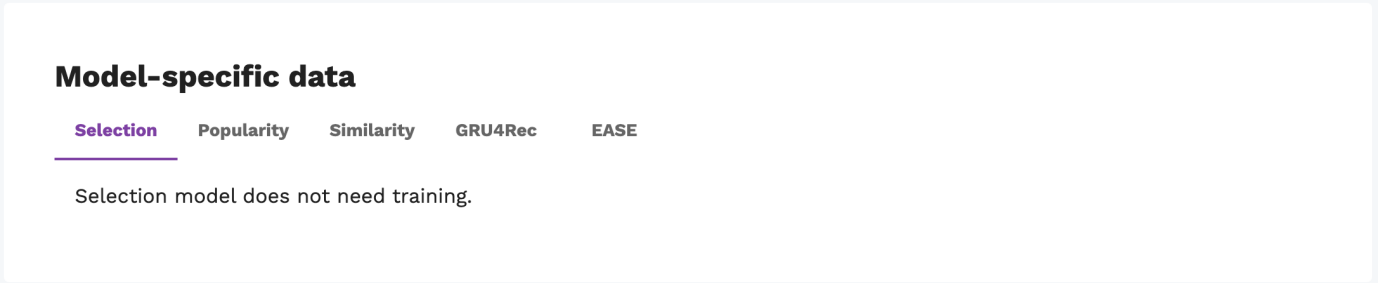
# Training

Training page has almost identical layout to the performance page. It enables user to select time window for which to display training data.

The training data are displayed for the whole Recommender system (*General data*) and for each prediction model individually (*Model-specific data*).

The data contain number of trainings based on their status (*Completed* or *Failed*), duration of the trainings and their memory consumption.

The user can also schedule training of a prediction model, the model is added to the queue of models to be trained.

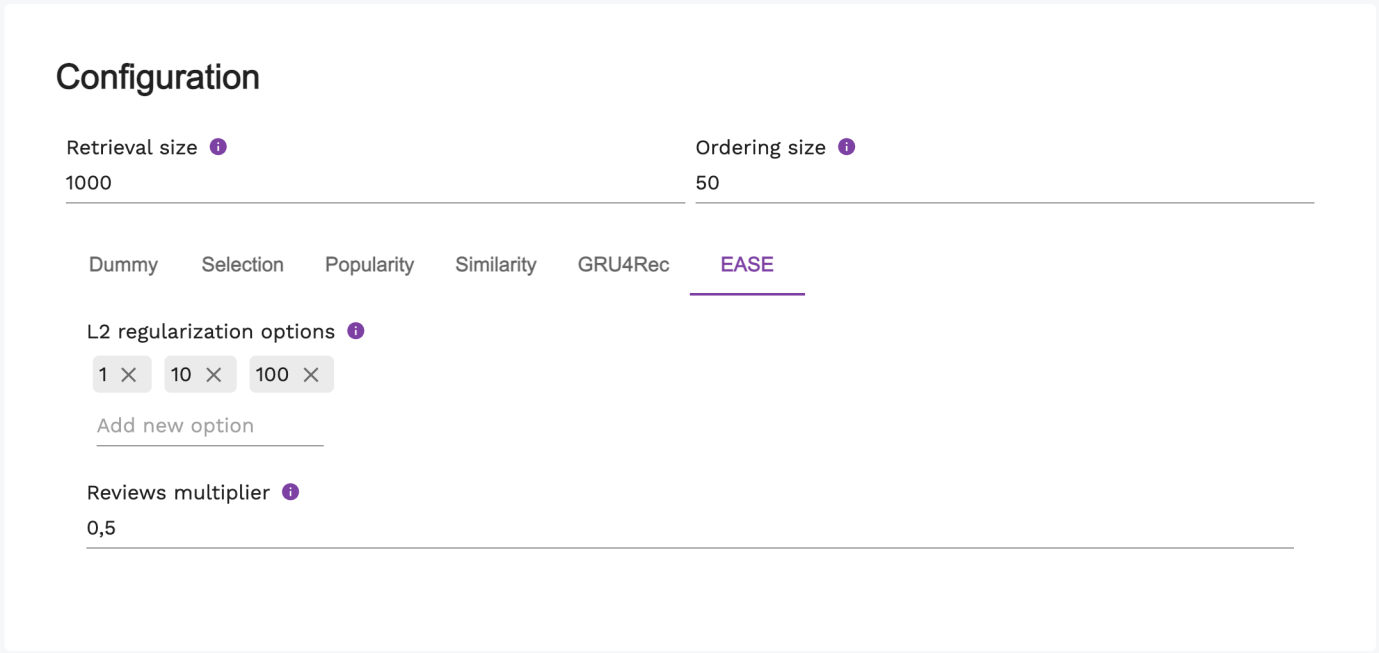



Some prediction models can not be trained, this information is displayed to the user.

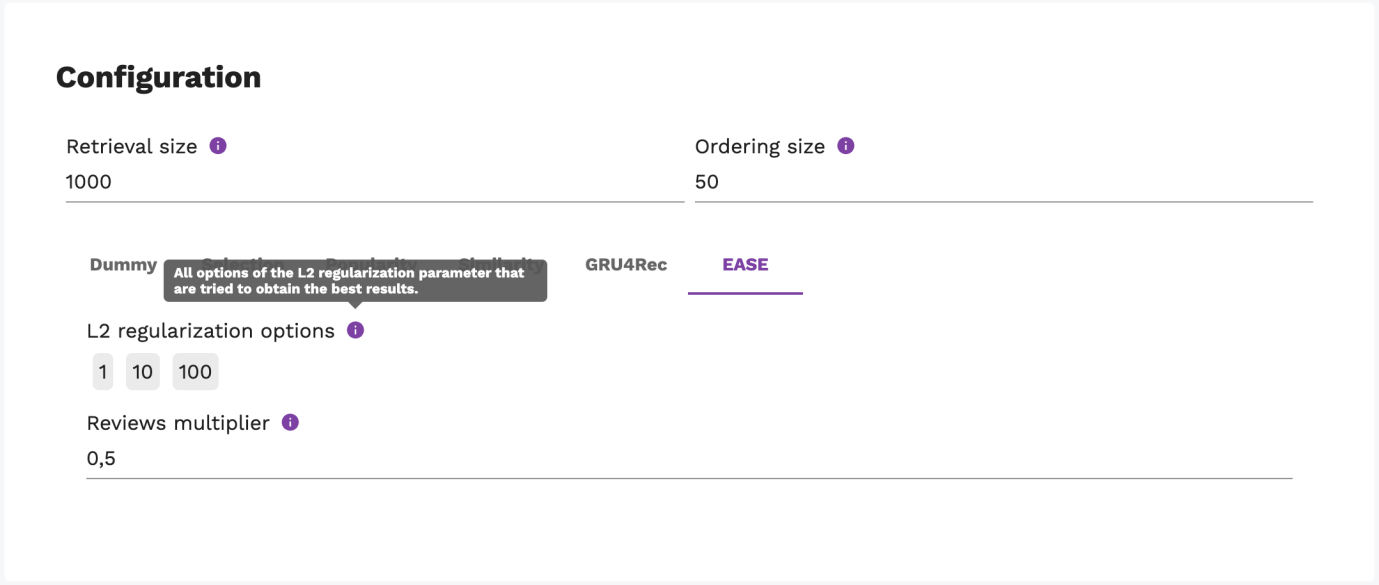
# Configuration

Configuration page contains information about the individual prediction models. Each model can also be disabled in real-time. This list of models contains also information whether the model is available for training (if enough data is present) or whether it does not need any training.

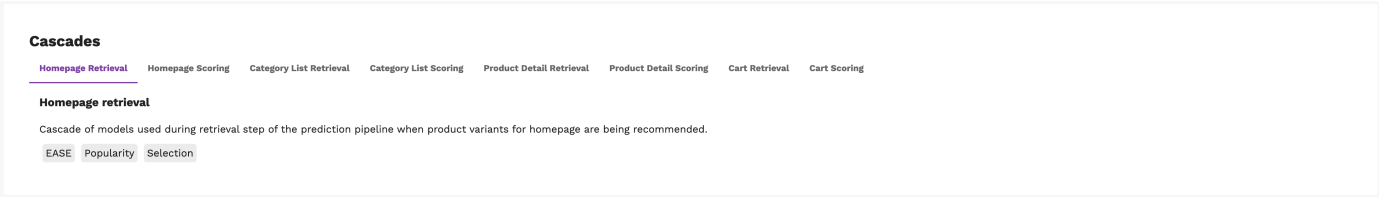
The Recommender system can be configured on this page. The configurable parameters contain *retrieval size* and *ordering size*.




Each prediction model can also be configured. Some of the parameters are options that the training algorithm chooses from - it tries all combinations of the parameters and selects the best combination based on the model's performance. Each parameter has description in a tooltip above a *info* icon.




The prediction model cascades can be edited on this page as well. Each cascade defines the order of models that is tried when performing the corresponding recommendations. If the first model can not be used, the second is selected and so on.




Each cascade can be re-ordered using a drag&drop feature.

ecoseller

User documentation

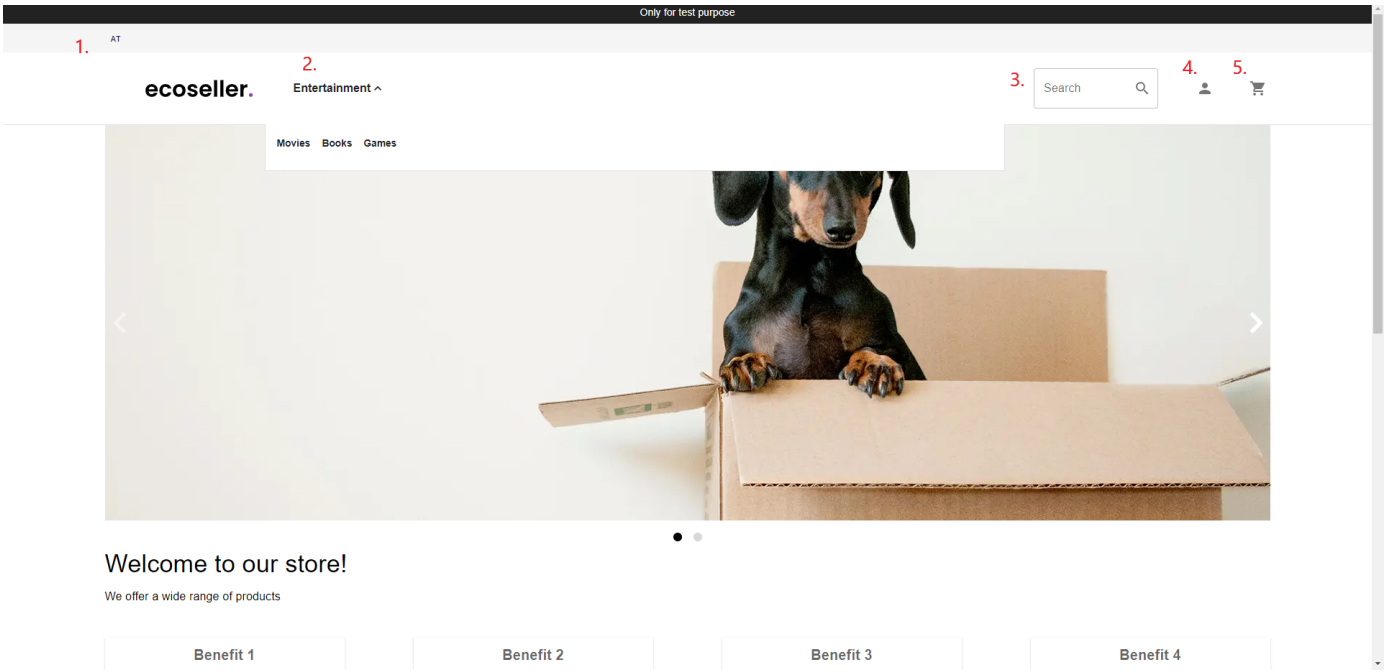
# Storefront

Table of contents:



# Homepage

When you first open storefront, the homepage is displayed.



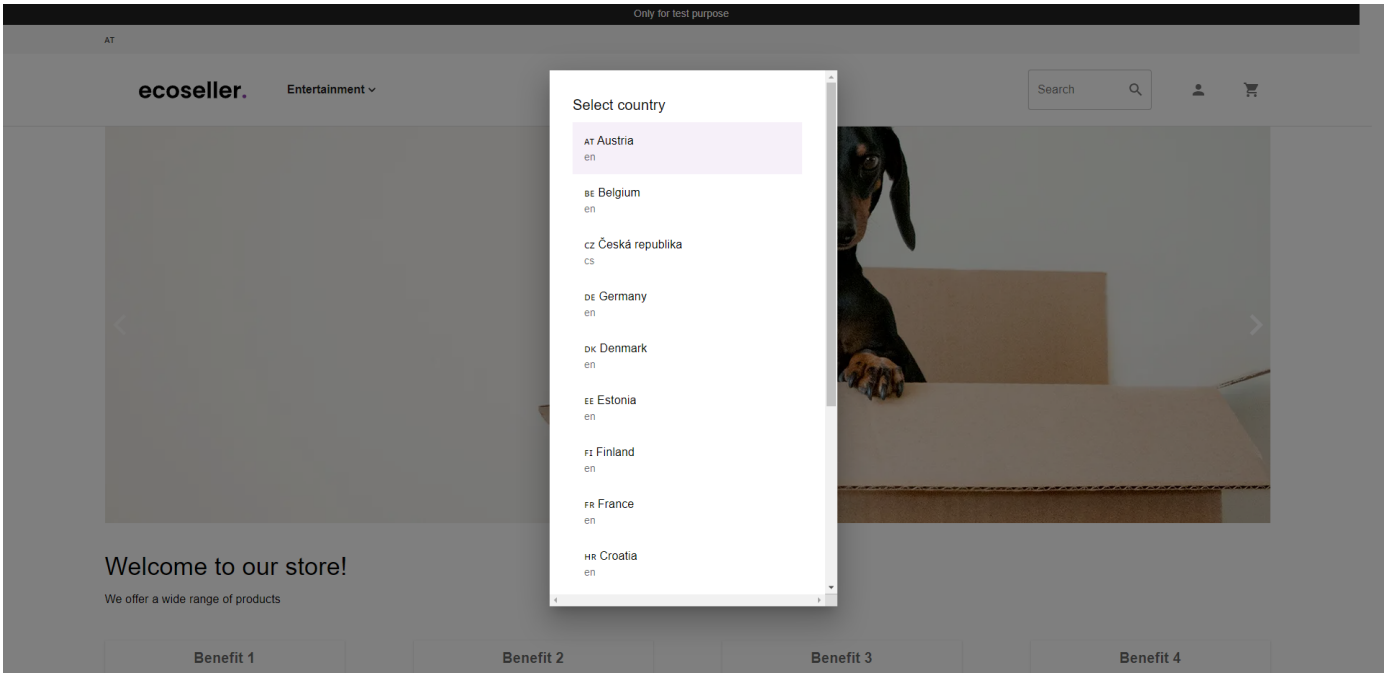

There are several components denoted at the image above

1. Country change button (see Country change)
2. Category menu - using this menu, you can navigate to individual categories
3. Search bar - used for searching products
4. Profile icon - using this icon, you can login or display your profile info and logout (if already logged-in)
5. Cart icon - after you click on this icon, you're redirected to the cart

The most common use-cases are described below.

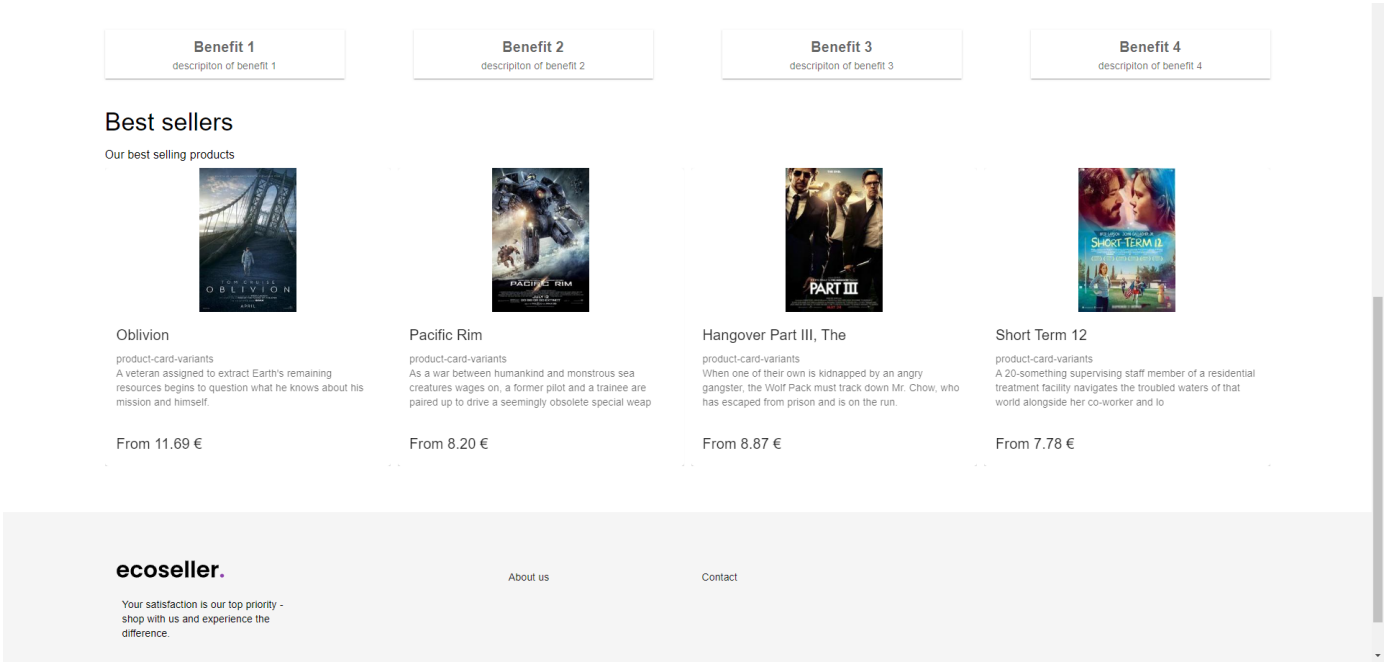
# Country change

To change country, firstly click on the country button mentioned above and then select the desired country.



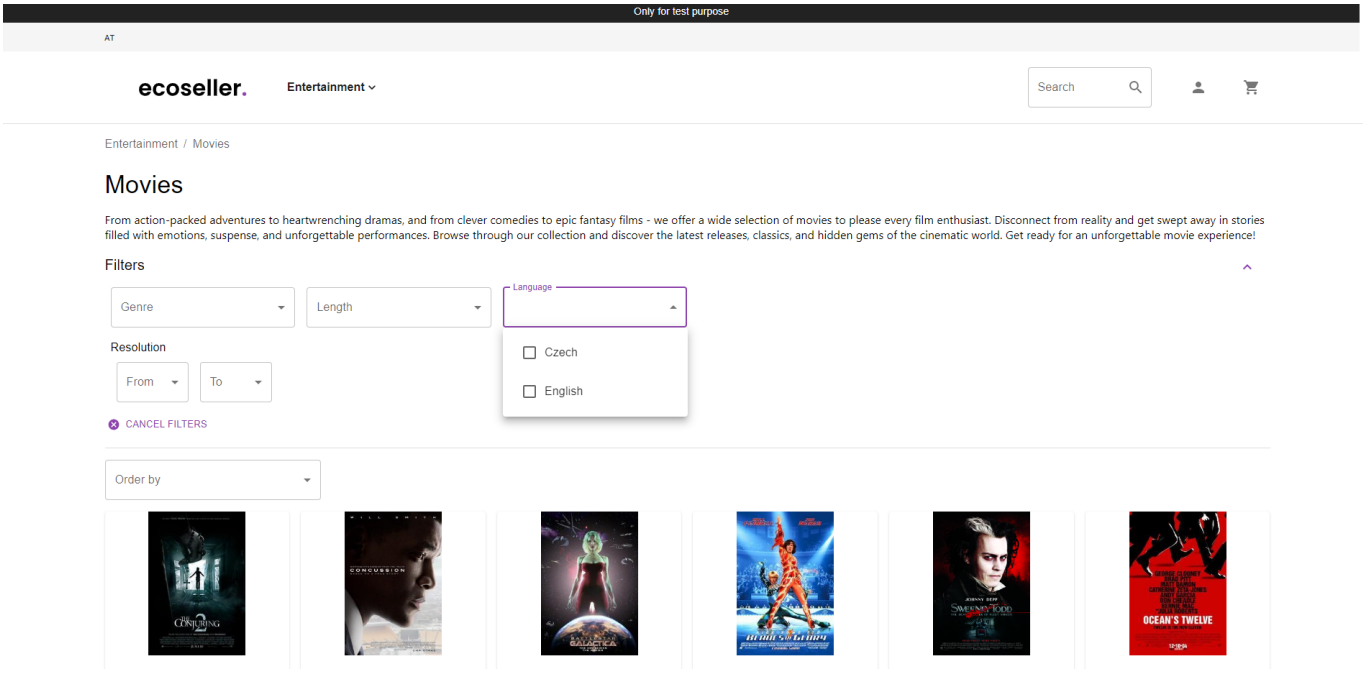

# Best selling products

When you scroll down on homepage, you can see our best selling products.



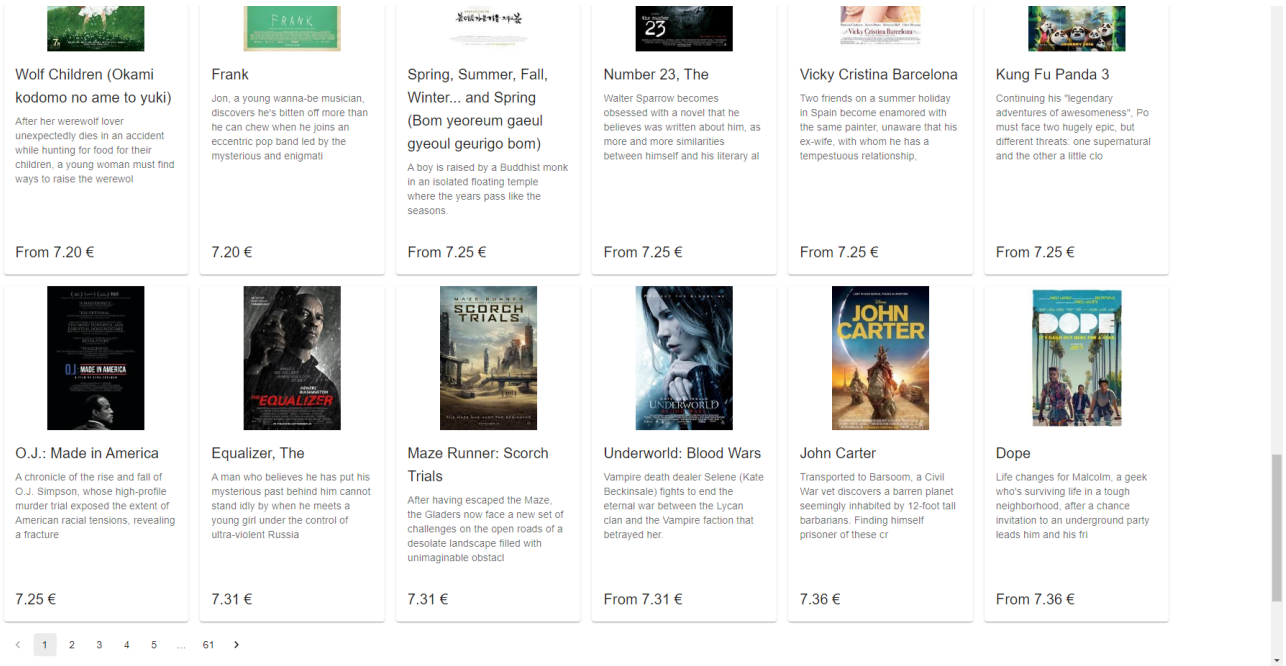

# Product browsing

After you select category in the top menu, you're redirected to category page




You can filter the products and select their ordering.

Below, there is a grid of products displayed (using pagination).




Each product card shows its

- photo
- title

- brief description
- price. Note that some products show the price like *From $ 7.31*, which means that there are multiple product variants with different prices and the cheapest of them is displayed.

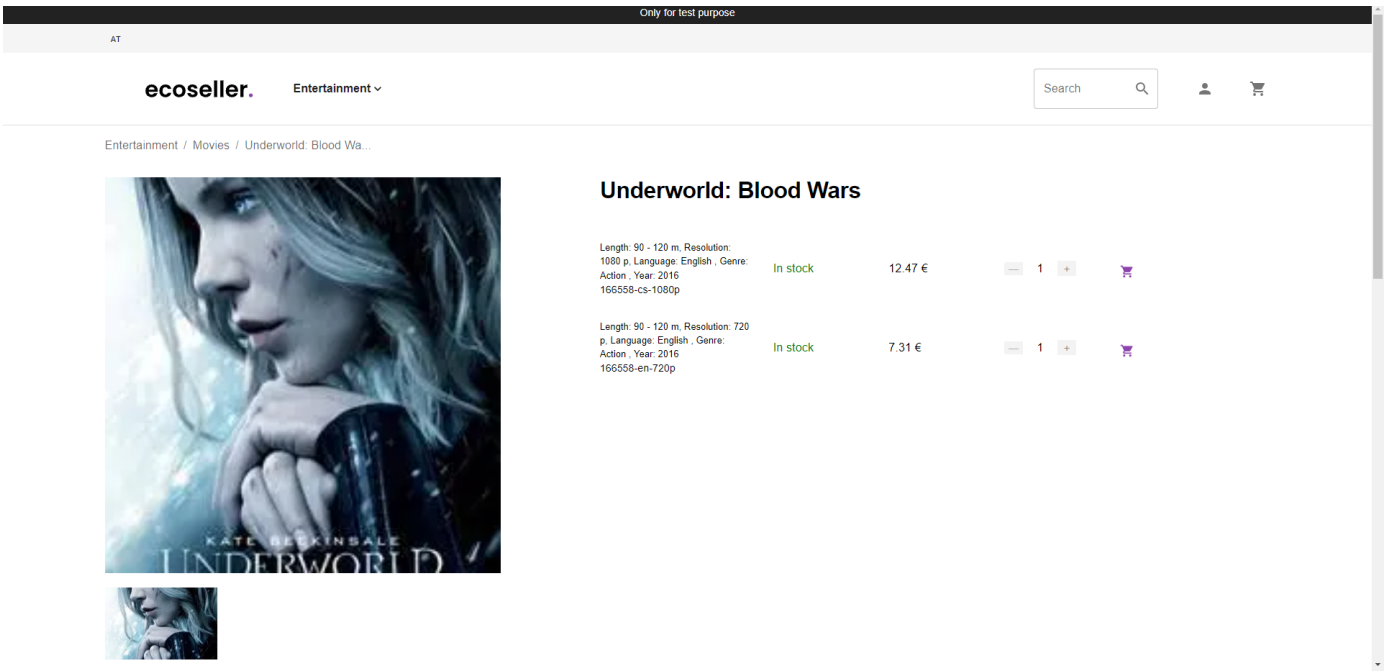
## Go to product detail

After you click on product card, product detail page is displayed

# Product detail

Product detail page shows product's

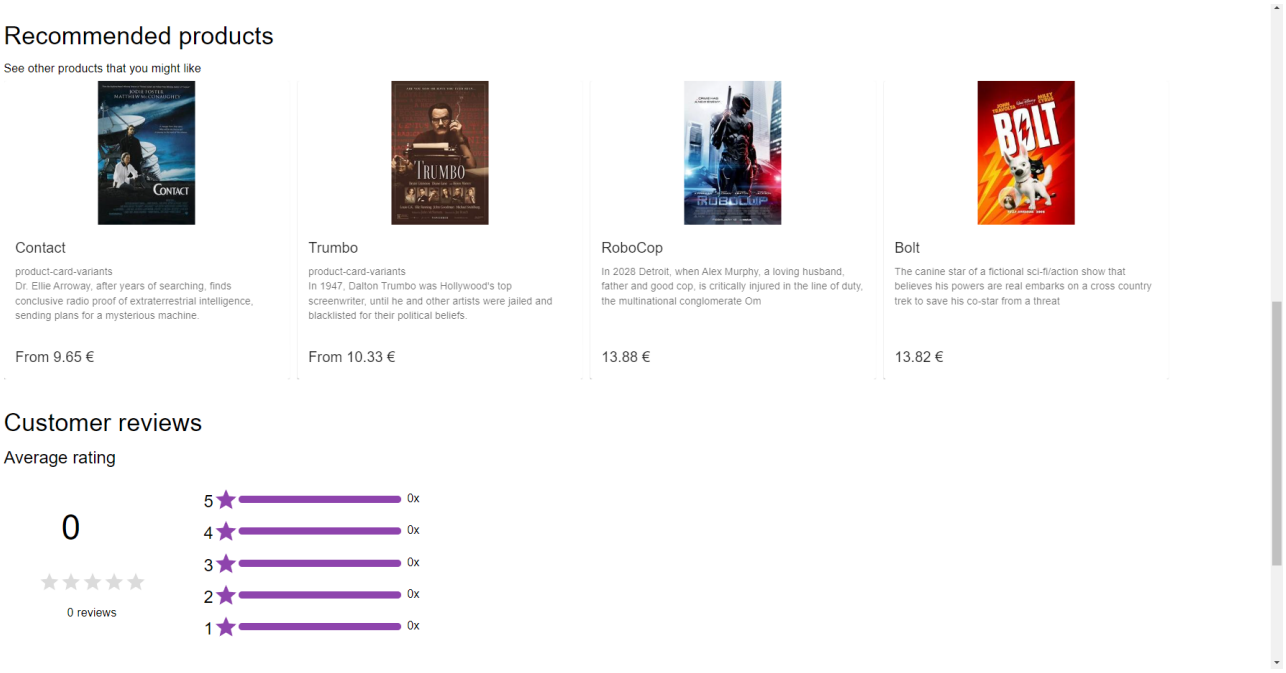
- title
- description
- gallery of its photos
- list of its variants including prices.



# Add to cart

In order to add product variant to cart, firstly select its quantity and then click on the *Cart* button next to it.

At the bottom of the page, you can also see recommended products and product reviews.

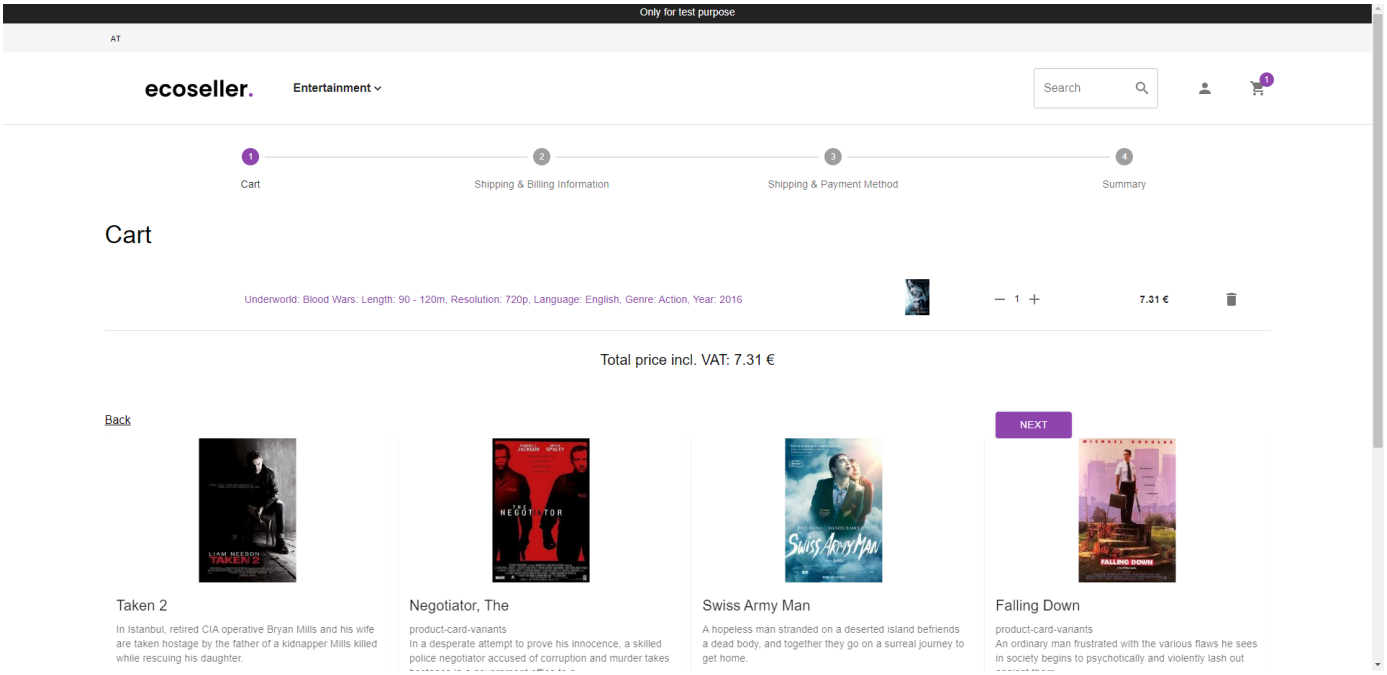



# Creating an order

After you click on the *Cart* button in the top menu, you're redirected to Cart detail page.

# Cart detail

This page shows all of your cart items including their prices



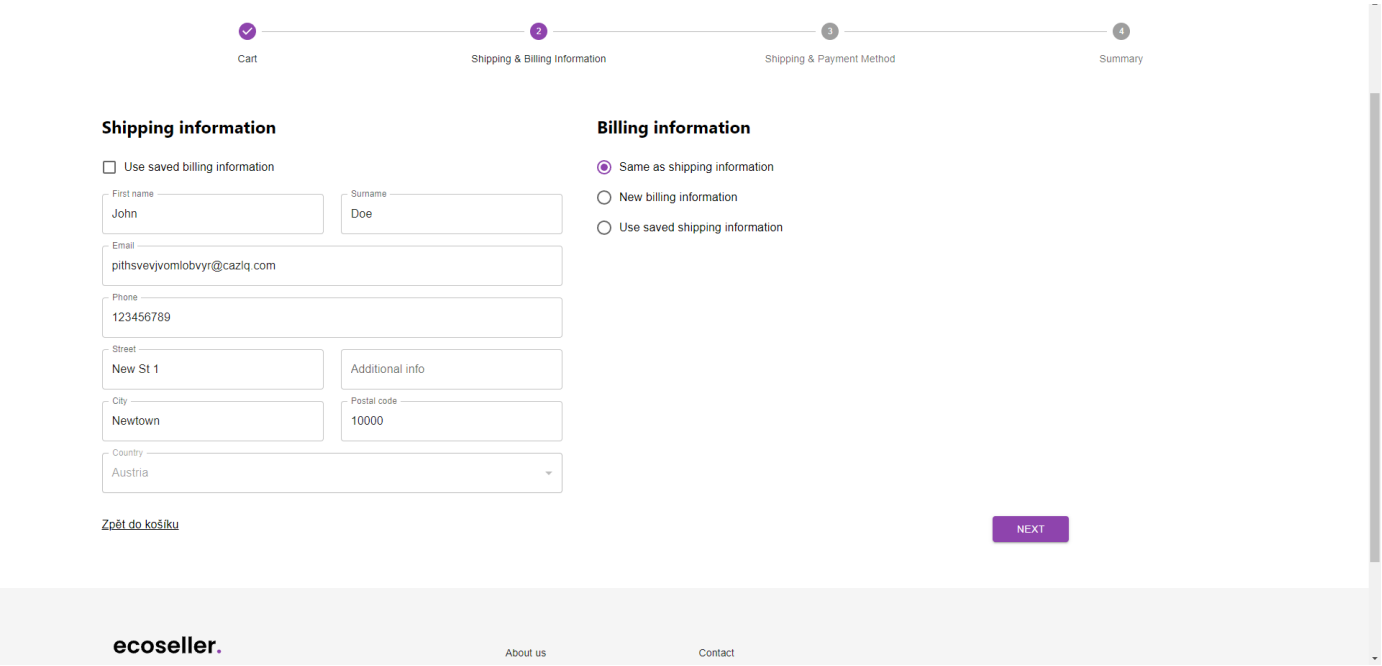

You can update their quantities or remove them.
Also note that the *Cart* icon in the top menu contains a badge showing number of items in your cart (cool, right? 😎).

At the bottom, there's a list of recommended products.

After you select *Next*, you're redirected to shipping & billing info page.

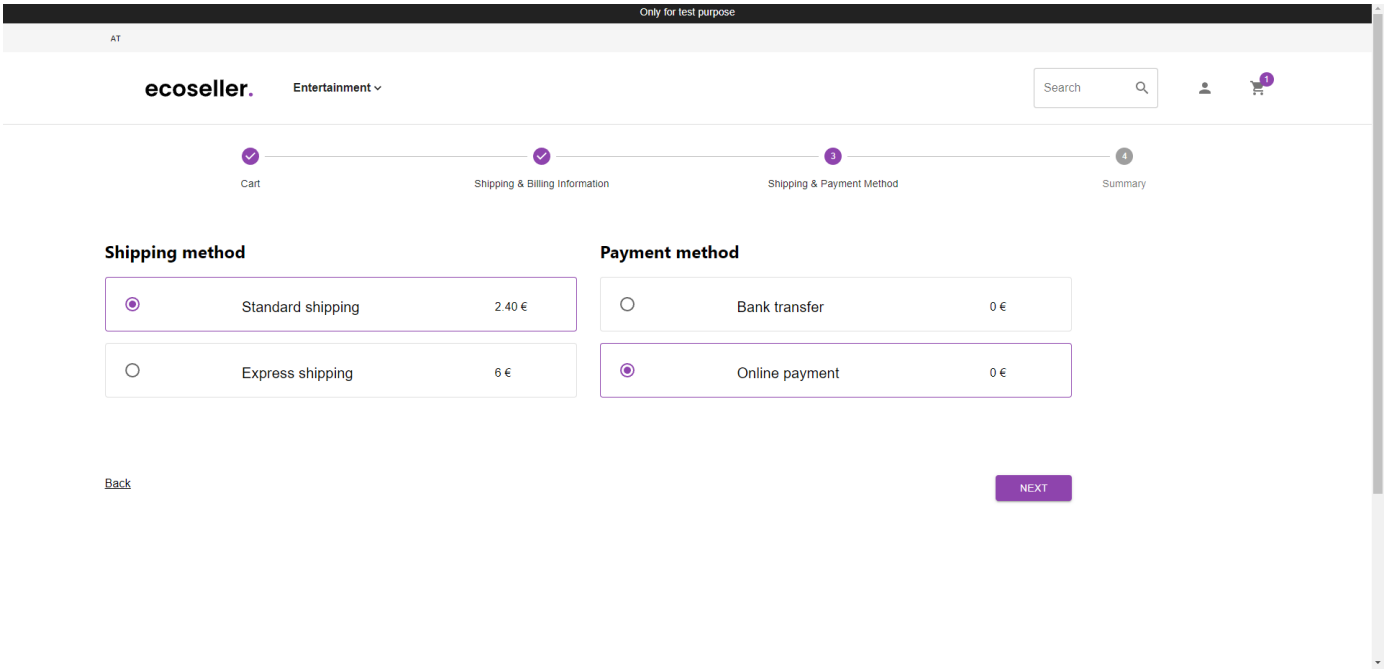
# Shipping & billing information

Here, you need to specify your shipping and billing information.
If you're logged in and you've saved your billing & shipping info, you can use these saved values. Otherwise, you must fill in the forms.



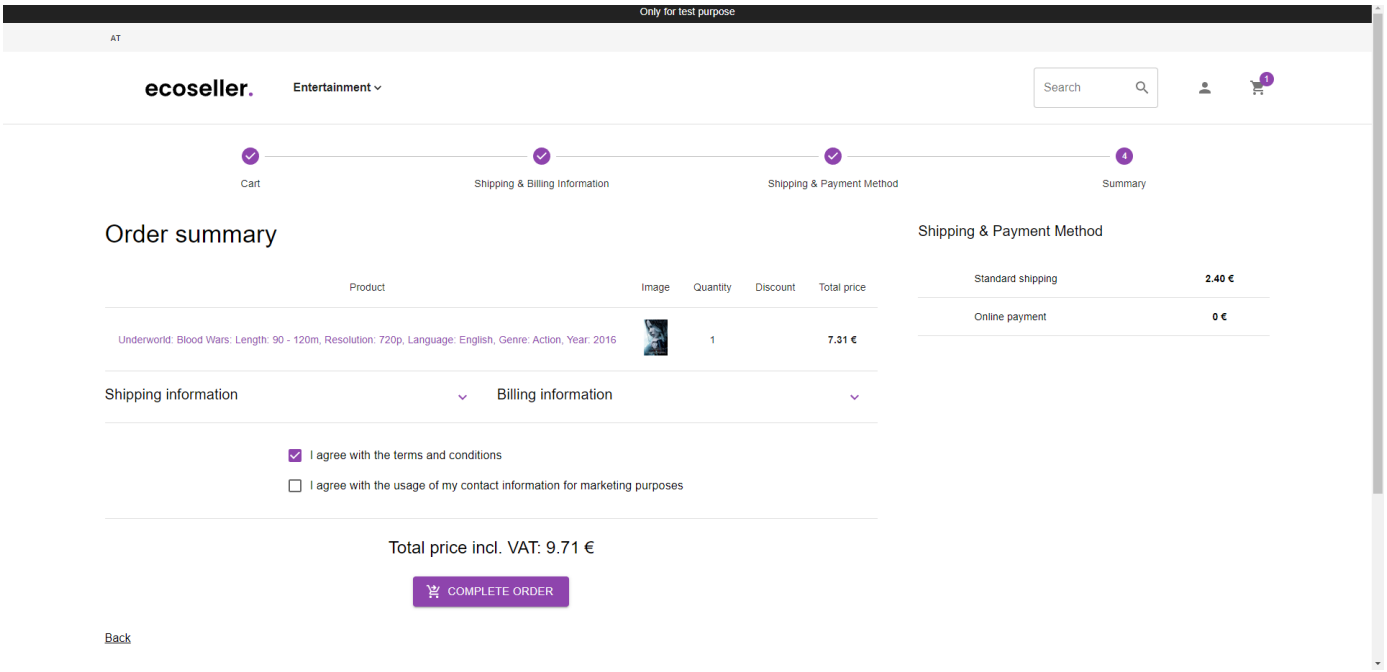

# Shipping & payment method

In the next step, you select firstly the desired shipping method and then payment method as well.




# Order summary

In the last step, summary of your order is displayed.

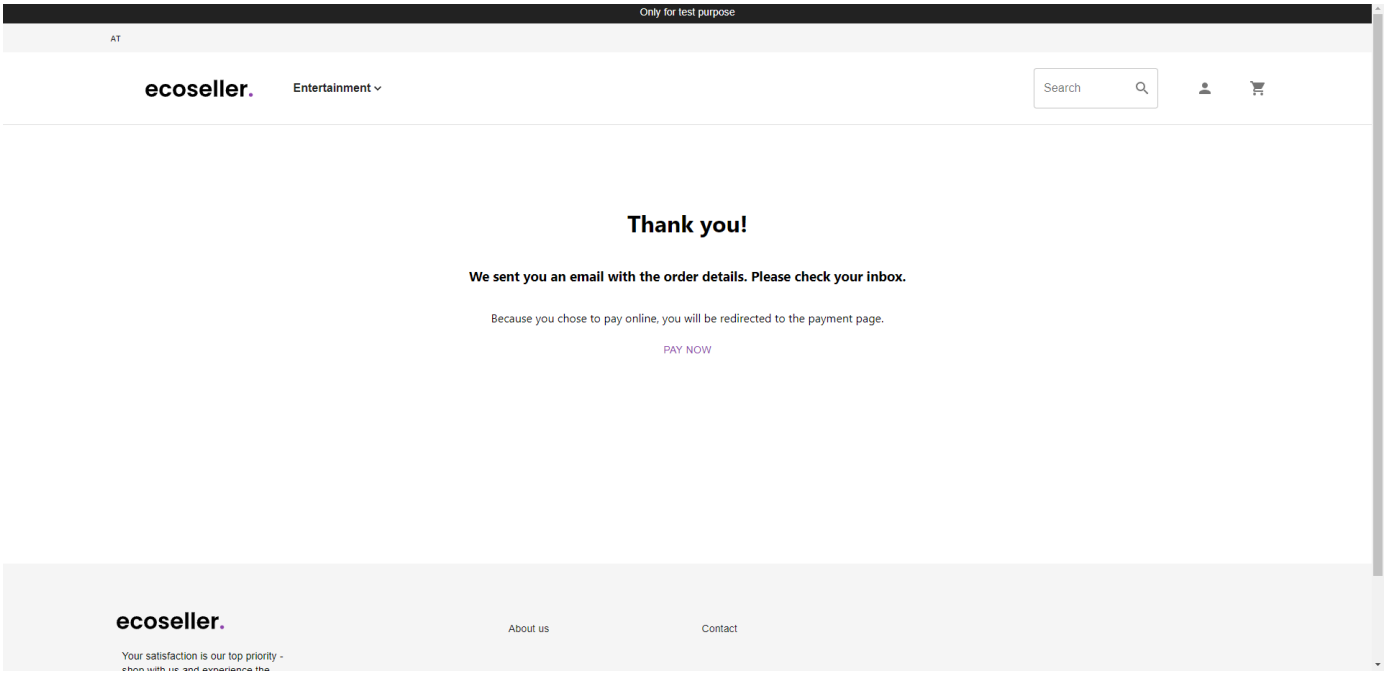



To complete the order, you must

1. Agree with our terms and conditions
2. Click on *Complete order* button

# Order completed page

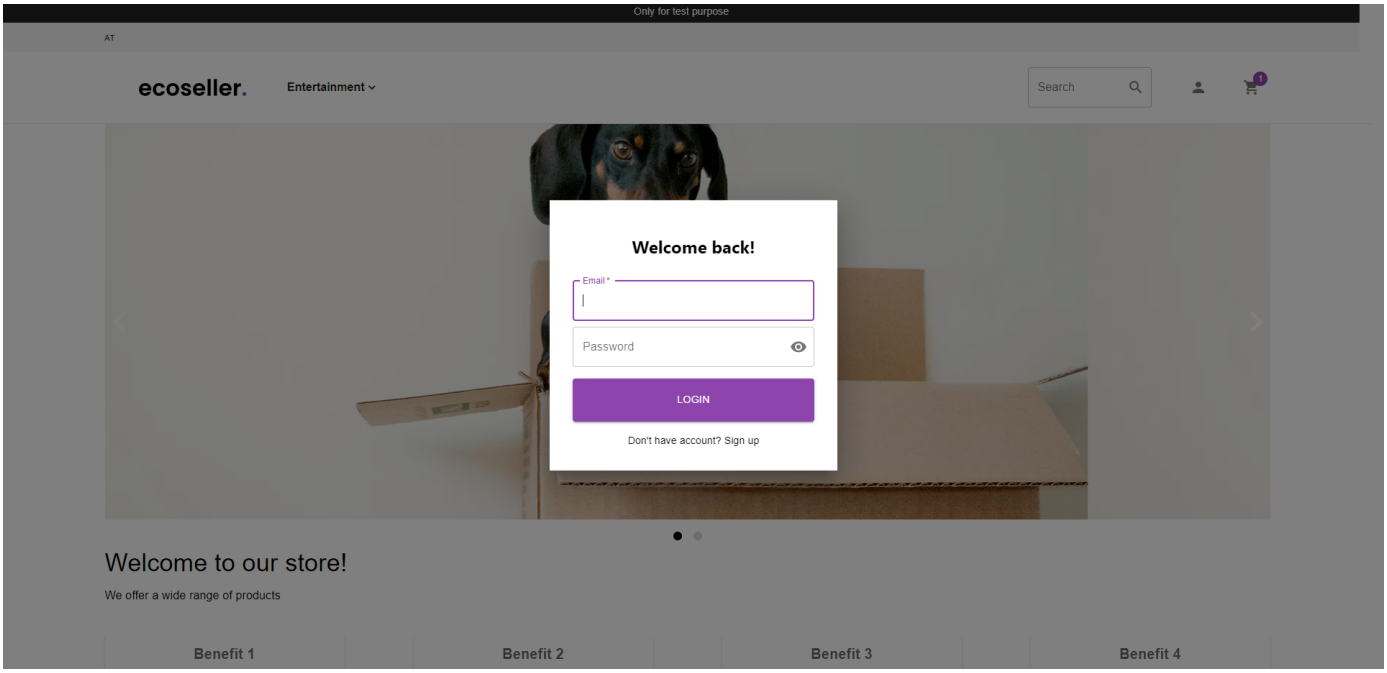
After you complete the order, the following page is displayed and you recieve a confirmation email.



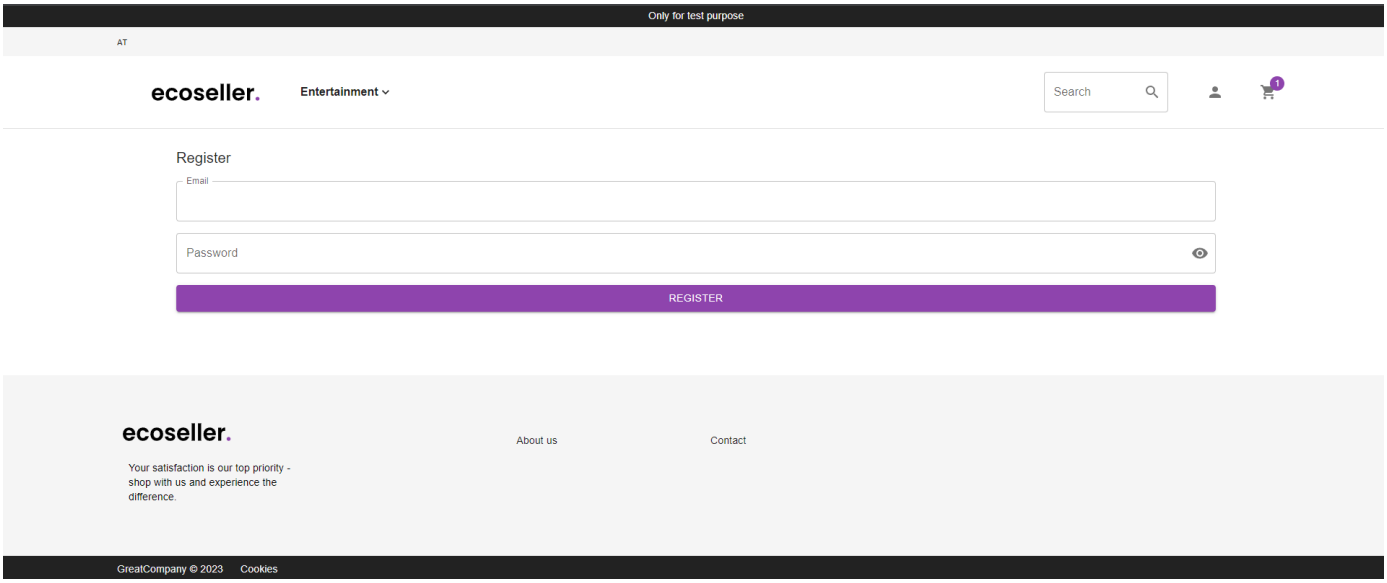

You can proceed to payment using the link displayed.
Note that currently, it doesn't work, because no payments are configured. Go to admin section to see a step-by-step guide on how to configure payments.

# Log-in & Registration

You can log-in using the profile icon in the top menu.




If you wish to create a new account, use the *Sign up* link and you'll be redirected to Registration page.




After you've signed up, you can log in using your credentials.

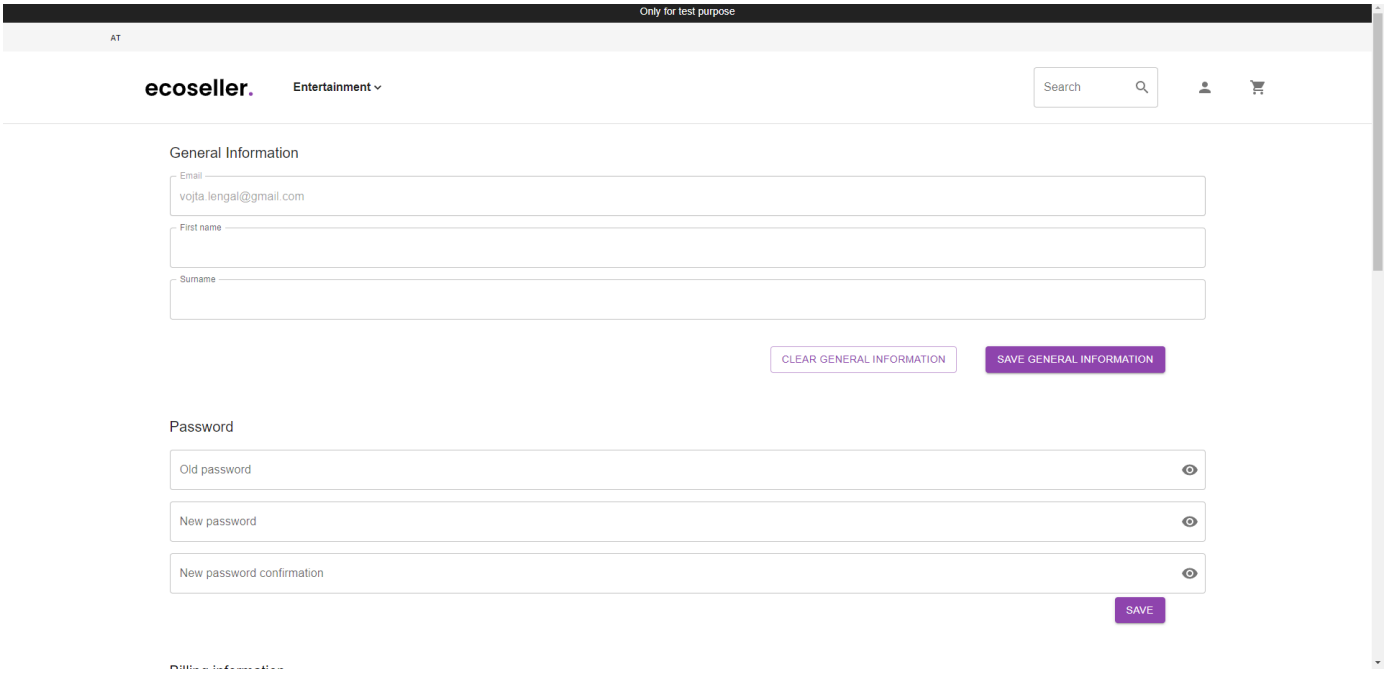
---

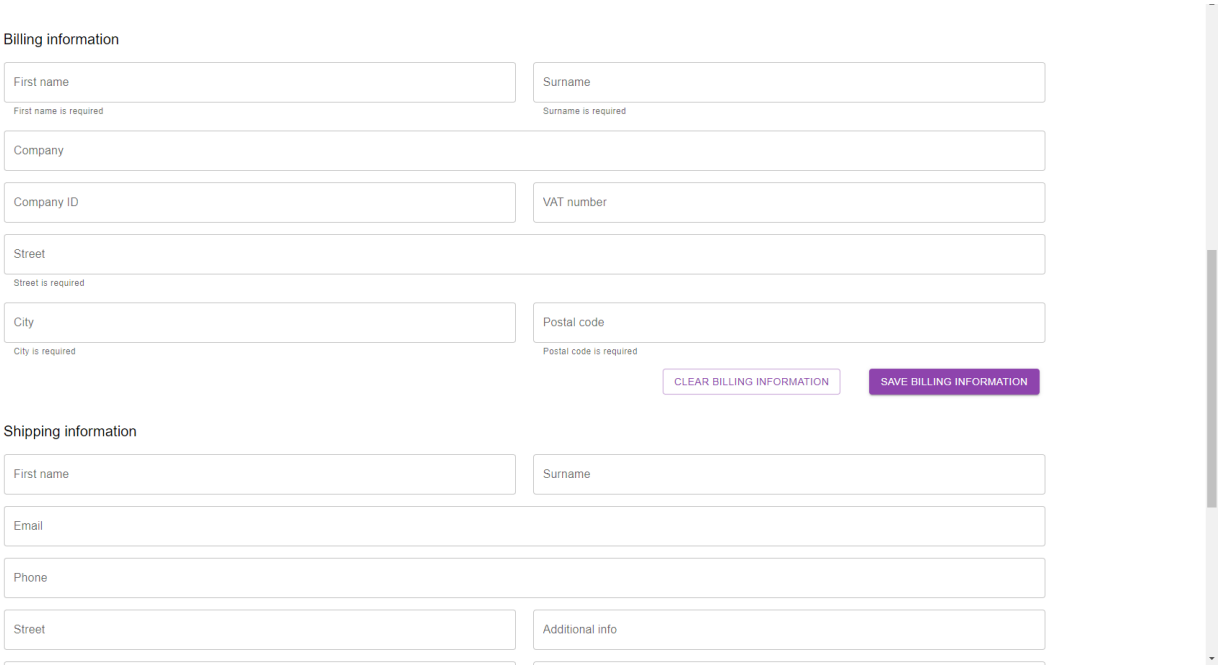
## Note: All of the following actions are available only for logged in users

# User detail

You can navigate to user detail page using the profile icon.

You can change your password here, as well as set the default billing & shipping information

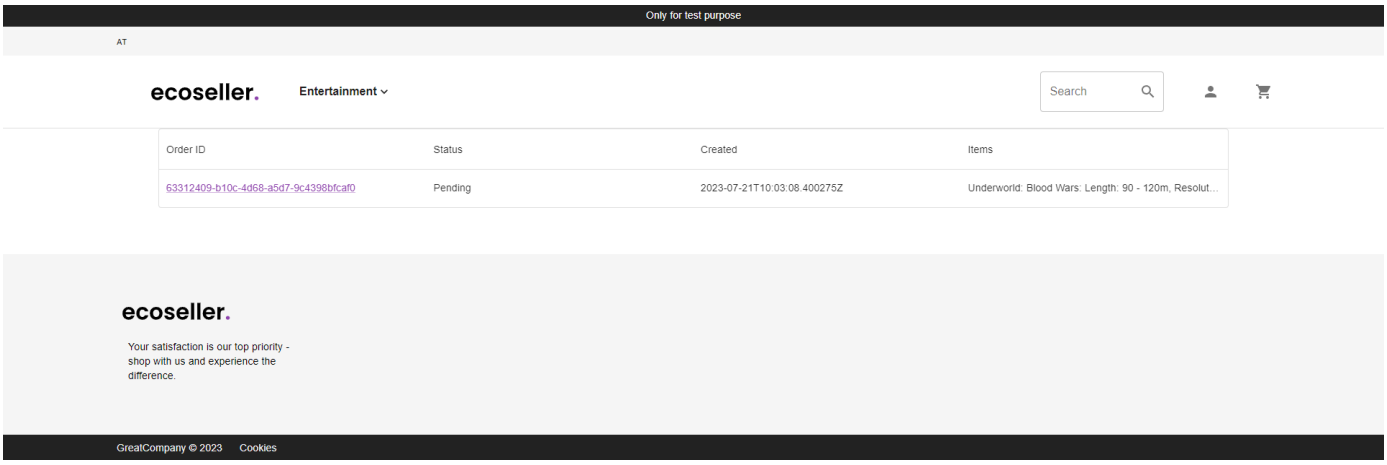





# Order list

You can navigate to order list page using the profile icon and then selecting *Orders*

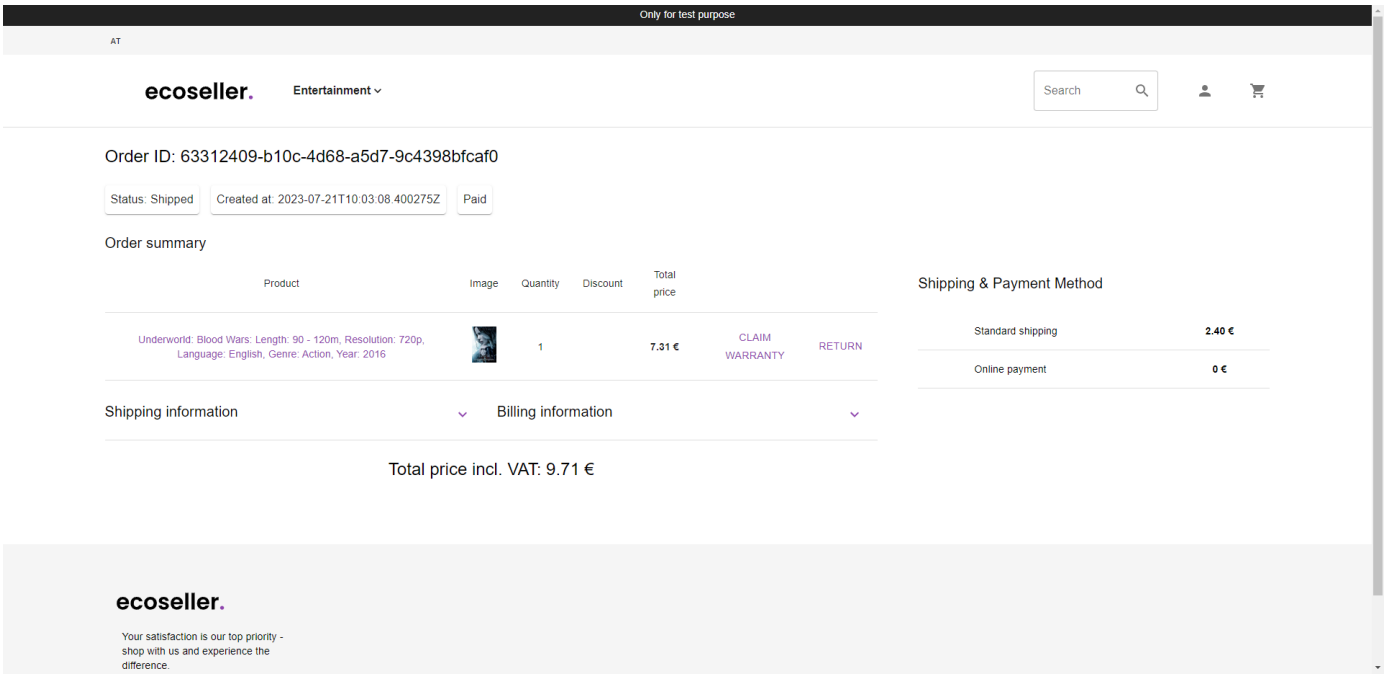
List of your orders is displayed




After you click on the Order ID, you'll be redirected to order detail page.

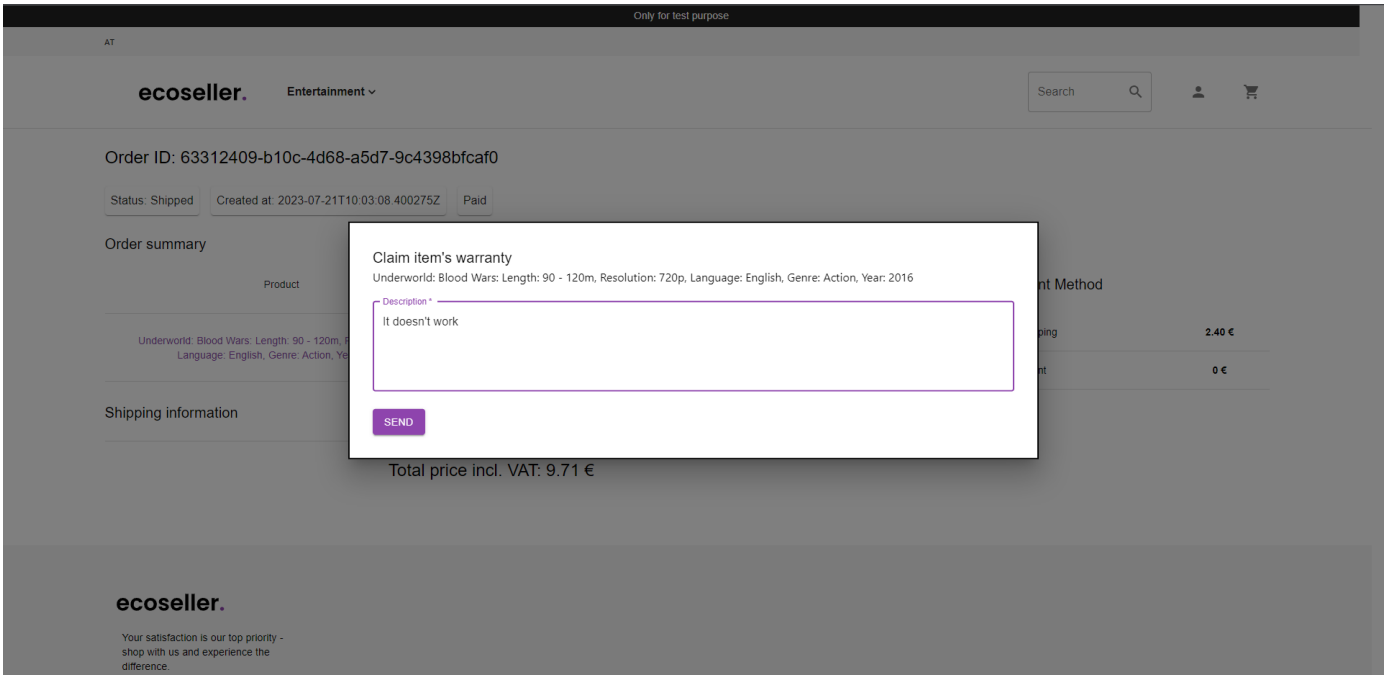
# Order detail page

This page shows detailed info about the given order.

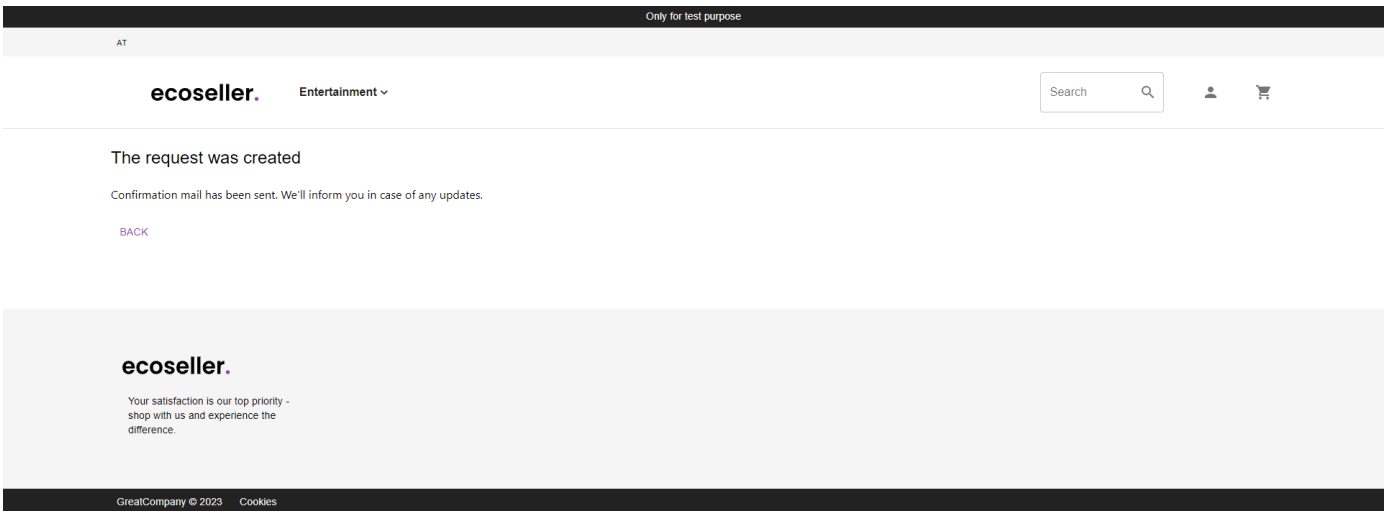



# Warranty claim / return request

For each item, you can create a request for returning or claim its warranty using the *Return*, resp. *Claim warranty* buttons.
Then, a form is displayed.



After you submit the form, the confirmation page is displayed and an email is sent.




You’re notified about any updates using the email as well.

ecoseller

ecoseller

# Contribution

To contribute to ecoseller, please stick to the following rules:

# GIT Workflow

When working on a project, it is important to have a workflow that is easy to understand and follow. This document describes the workflow that we use at ecoseller.

## Working on a new feature/bug

1. Update your master branch - on `master` branch run the following:

```
git pull origin master
```

2. Create a new branch and checkout to it
   - follow naming convetion for branch names `T-<task-number>`

```
git checkout -b T-<task-number>
```

3. Work on a given feature/bug. Commit your work often (you do not need to push these commits to remote branch)
   - follow naming convention for commits

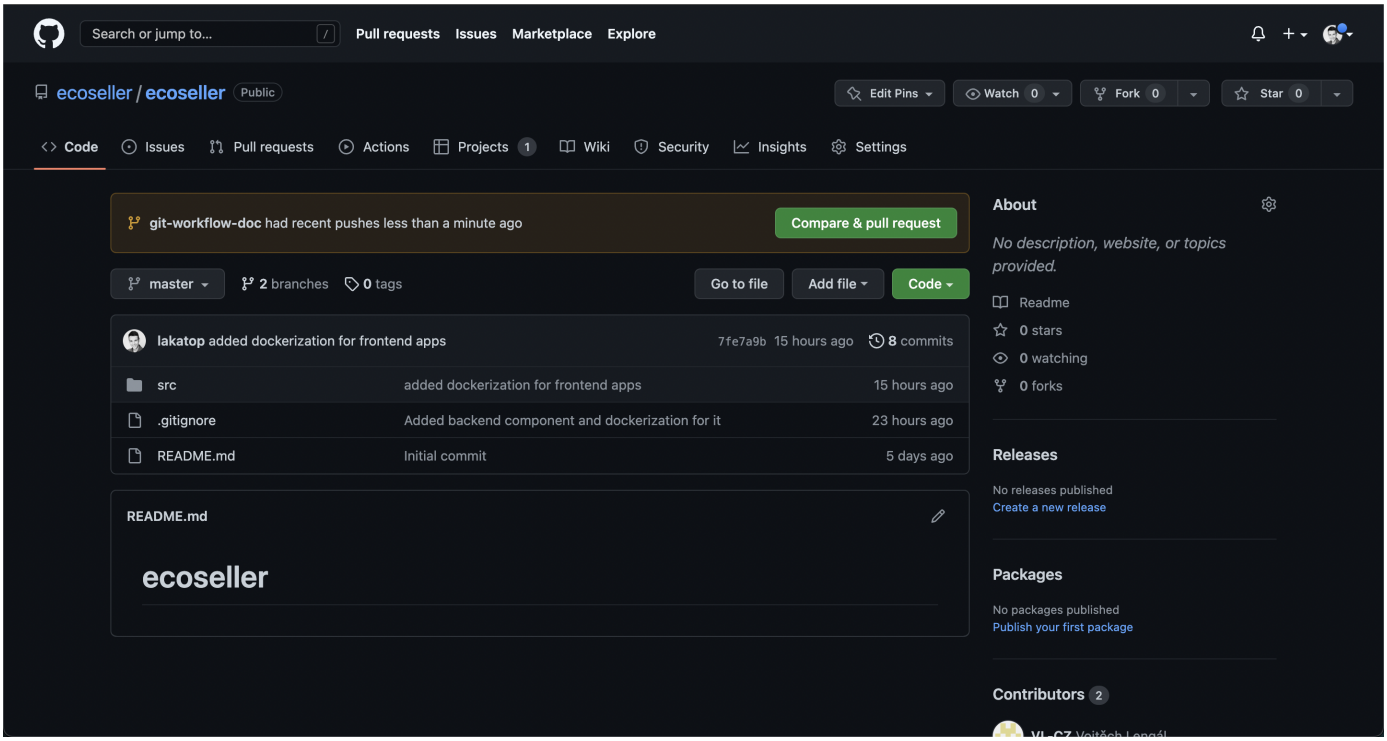
```
[T-<task-numbe>] <short-description>
```

```
git add .
git commit -m "[T-<task-number>] <short-description>"
```

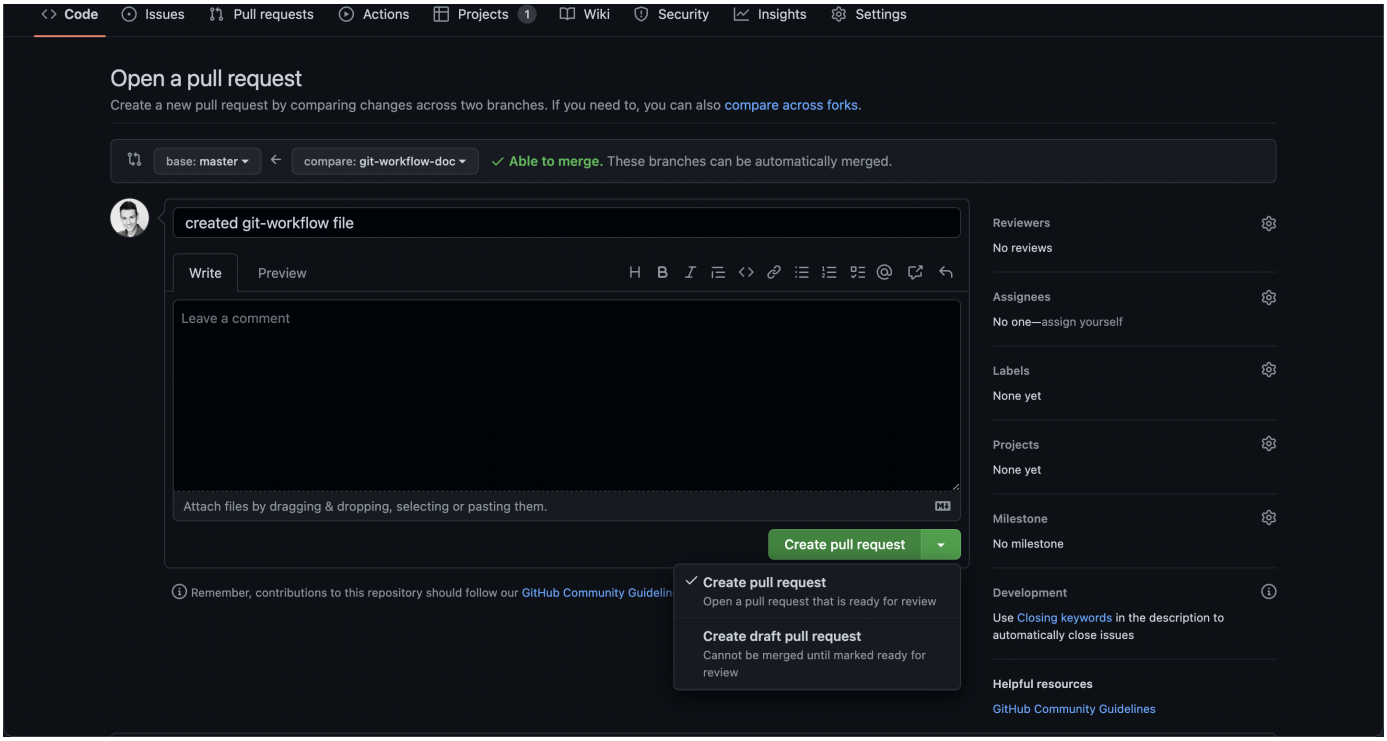
4. Push changes to remote branch

```
git push origin T-<task-number>
```

- First push to remote branch will create pull request - go to the project github page and click on *Compare & pull request* button




- If you are not done with the feature/bug yet, mark this pull request as draft




1. After feature/bug is done, fill in nice description, mark pull request as ready and send it to review
2. When a feature/bug branch is ready to be merged into a `master` branch, do the following:
    1. Update your master branch to the latest state

        ```
        git checkout master
        git pull origin master
        ```

    2. Checkout to feature/bug branch

        ```
        git checkout T-<task-number>
        ```

    3. Rebase feature/bug branch on top of `master` branch and squash commits into one
        - This can be achieved via interactive rebase ( `-i` option) which will bring up editor where you can squash all commits to the first one (let `pick` option for the first commit and on all the follwoing commits use `s` option)

    - After squashing, do not forget about naming convention of representative commit `[T-<task-numbe>] <short-description>`

      ```
      git rebase -i master
      ```

    - Fix all potential conflicts while rebasing
4. Checkout to `master` branch and merge feature/task branch

   ```
   git checkout master
   git merge T-<task-number>
   ```

# Linear history

Applying this workflow keeps the git history of a project **linear**. That is good for the following reasons:

- Easier to read
  - History is more clear
  - No useless commits and new merge commits
- Easier reverting and cherry-picking
- Git bisect

# What to do when

This section describes some common made mistakes and how to fix them.

- **Forgot to create a new branch** You already made some changes and forgot to switch to new branch. At this point its easy fix, just add what you have done so far and then switch to new branch:

  ```
  git add .
  git checkout -b T-<task-number>
  ```

# Useful commands to remember

- `git log` - shows commits log
  - `--oneline` option
- `git status` - status on current branch
- `git fetch origin` - fetch remote branches
- `git reset HEAD~<number>` - moves `HEAD` pointer `<number>` commits behind
  - `--hard` option - discards local changes
  - `--soft` option - keeps local changes

# Continuous integration

We use Github actions for CI.
There are multiple jobs set up (1 for each project component + action for docker compose), which automatically run on every commit to `master` branch and pull request update.

# What to do if CI jobs fail

See the error in Github Action detail.

If it's a linter/formatter error, see the section of corresponding component.

- backend
- dashboard
- storefront
- recommender

Follow the instructions for linting / formatting.

After everything works locally, commit and push the changes, CI jobs will start automatically.

---

# Backend

## `black`

We're using black code formatter.

Run

```
black ./core
```

to format source files (you need to have virtual env activated).

## `flake8`

We're using flake8 linter.

Run

```
flake8 ./core
```

to check for errors and warnings. If there are any errors, you need to fix them manually.

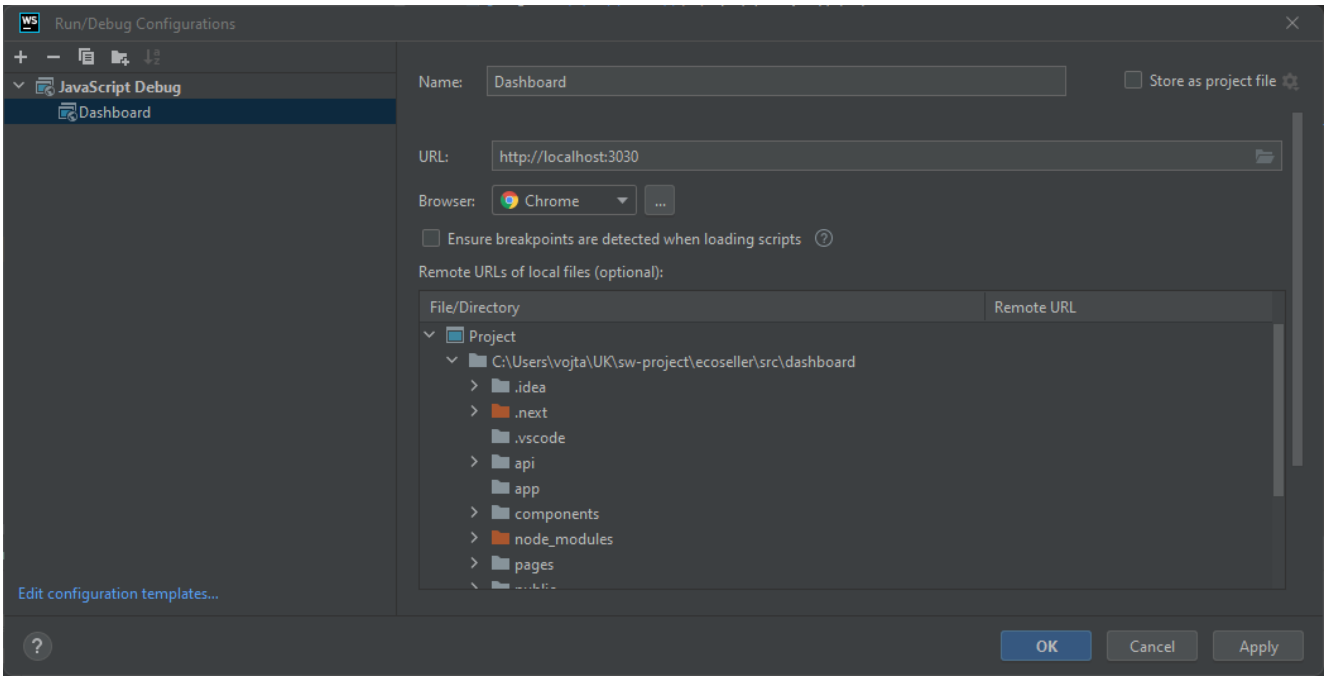
---

# Dashboard

# Debugging

## Webstorm

1. Run dashboard app using `docker compose` or locally
2. Open dashboard folder in Webstorm
3. Select Run/debug configuration > Add New > Select *JavaScript Debug*
4. Use the URL, where the dashboard is running (see the example below)




5. Then, run debuging using *Debug* button in menu.
   (Also note that this way, you'll be able to debug client-side code only - therefore not e.g. `getServerSideProps` method)

## `eslint`

We use ESLint integrated in Next.js for linting

Run

```
npm run lint
```

to check for warnings.

If there are any warnings, you can fix them automatically (if possible) by running:

```
npm run lint -- --fix
```

### `prettier`

We use prettier code formatter.

Run

```
npm run format
```

to format source code files.

---

# Storefront

### `eslint`

We use ESLint integrated in Next.js for linting

Run

```
npm run lint
```

to check for warnings.

If there are any warnings, you can fix them automatically (if possible) by running:

```
npm run lint -- --fix
```

### prettier

We use prettier code formatter.

Run

```
npm run format
```

to format source code files.

---

## Recommender

### black

We're using black code formatter.

Run

```
black ./src
```

to format source files (you need to have virtual env activated).

### flake8

We're using flake8 linter.

Run

```
flake8 ./src
```

to check for errors and warnings. If there are any errors, you need to fix them manually.