Ecommerce Application

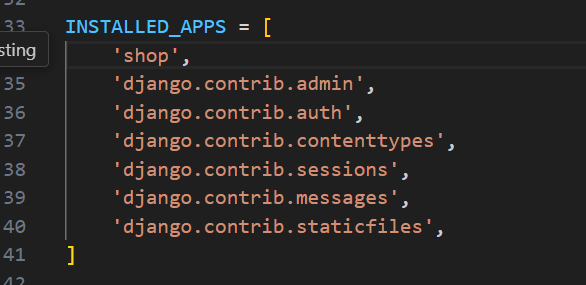
1 Create project working directory ecom

2. inside ecom created virtual enviroment ecomenv

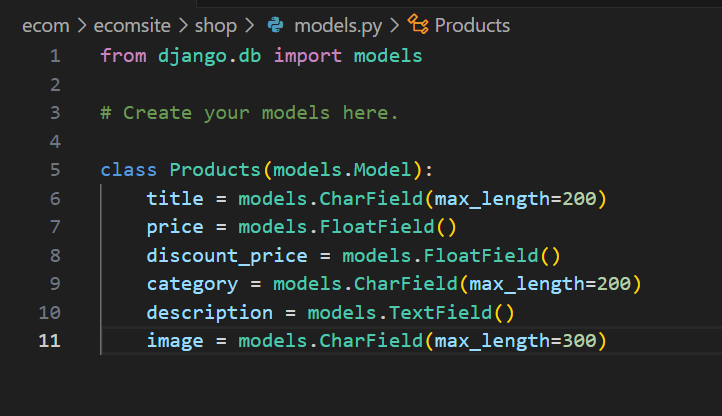
3. inside ecom created django project ecomsite

4. inside the ecomsite create django app shop

5. Go to settings.py file under INSTALLED\_APP, specify the shop app



6. Next created a model for product in the shop > models.py



Note: the image field will be storing the image url, as we will be hosting the actual file on different server

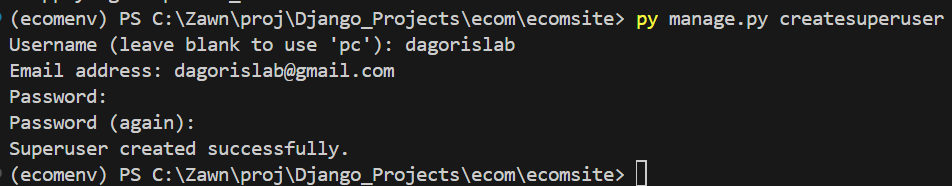
After creating the model, to create the actual database tables, we migrations were made with

Py manage.py makemigrations

Py manage.py migrate

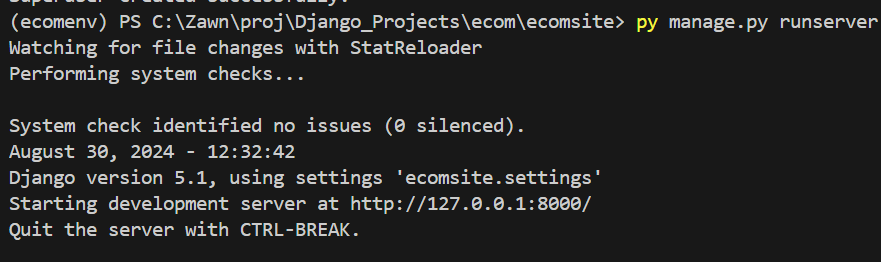
7. Create the superuser using

Py manage.py createsuperuser

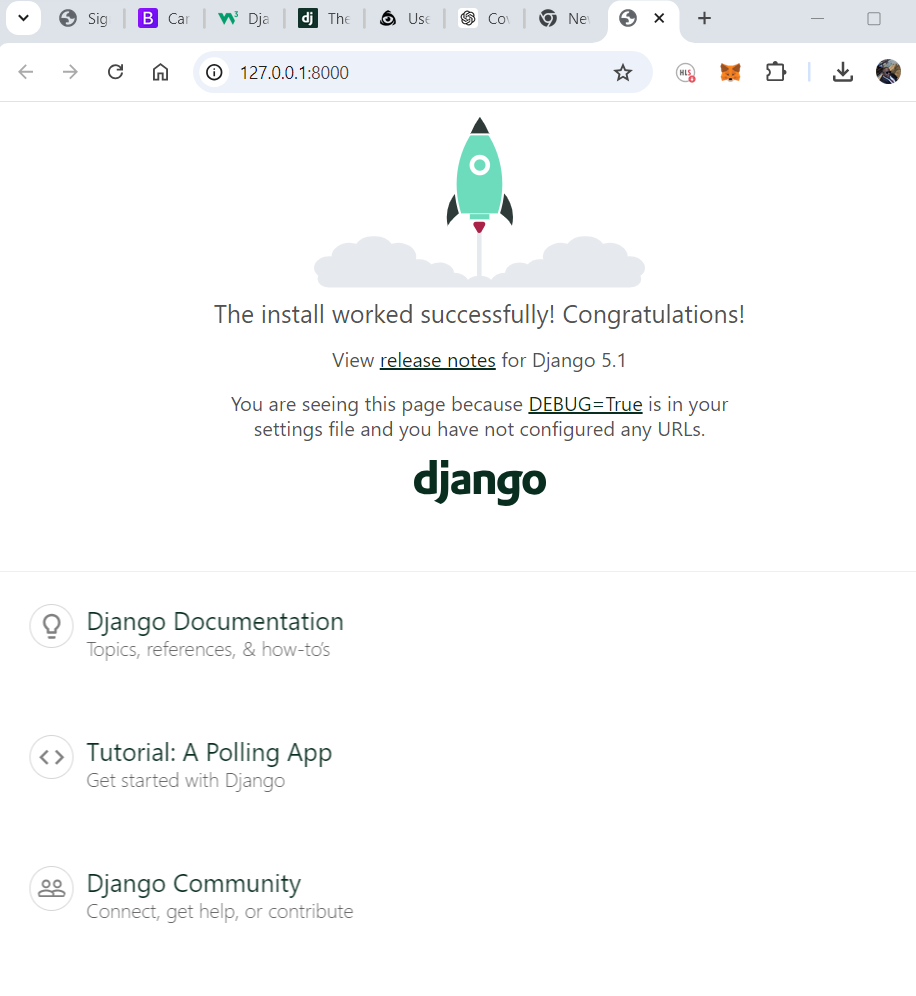


Next login to the admin portal, first start the server

Py manage.py runserver

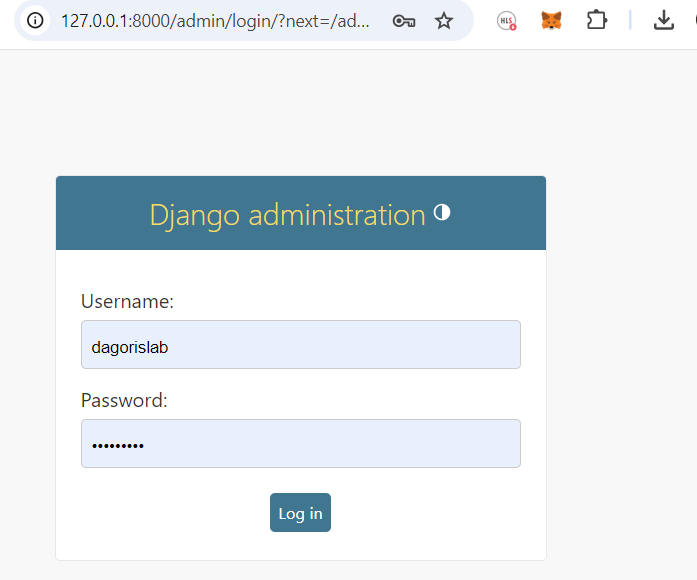


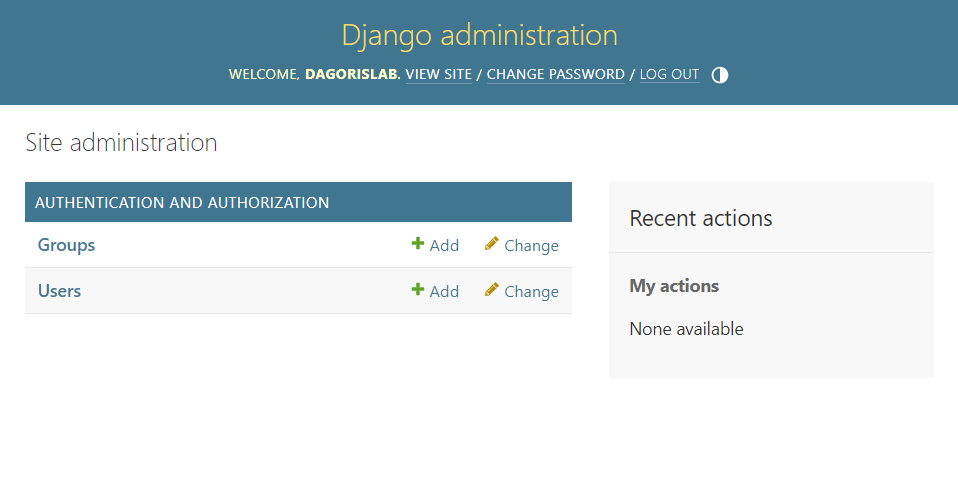
This starts a development server at <http://127.0.0.1:8000/>



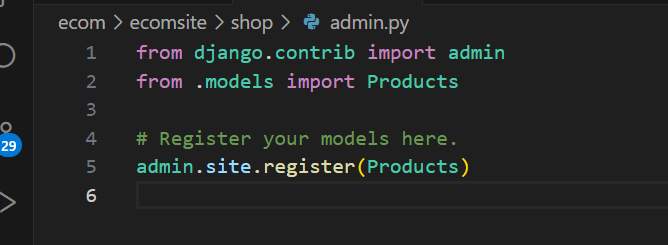
But to access the admin portal, go to

<http://127.0.0.1:8000/admin>

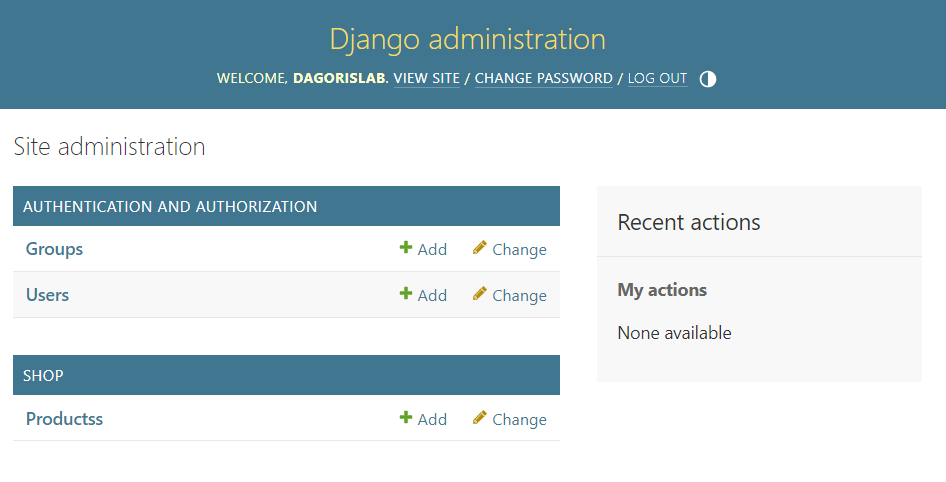




The Created model has not displayed on the admin portal, as it has not been registered, to register the model on the admin portal, do it on shop > admin.py



Now it has shown, note a mistake the model was named in the plural form.

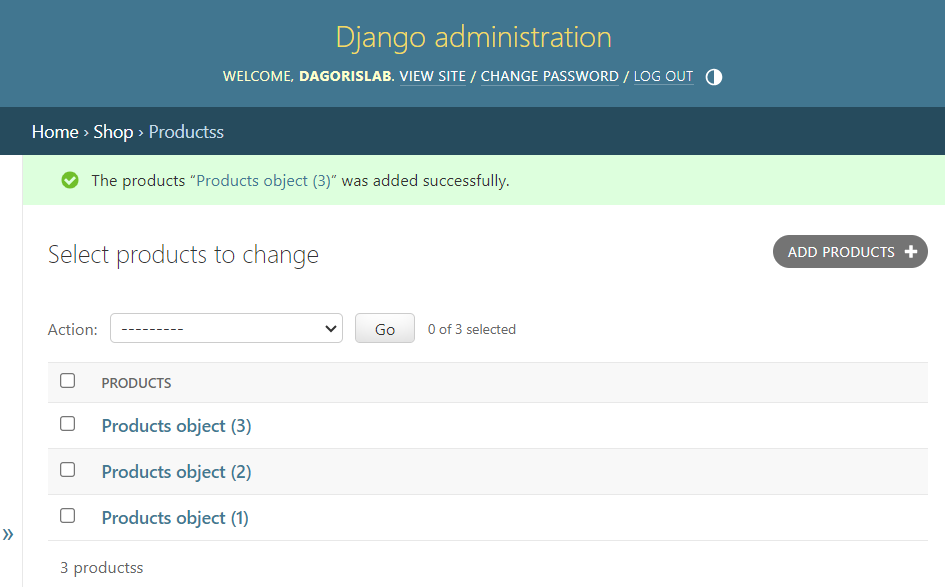


In Django, it is generally better to name your model classes in the **singular** form rather than the plural. This follows Django's convention and Python's standard practice for class naming. Here's why:

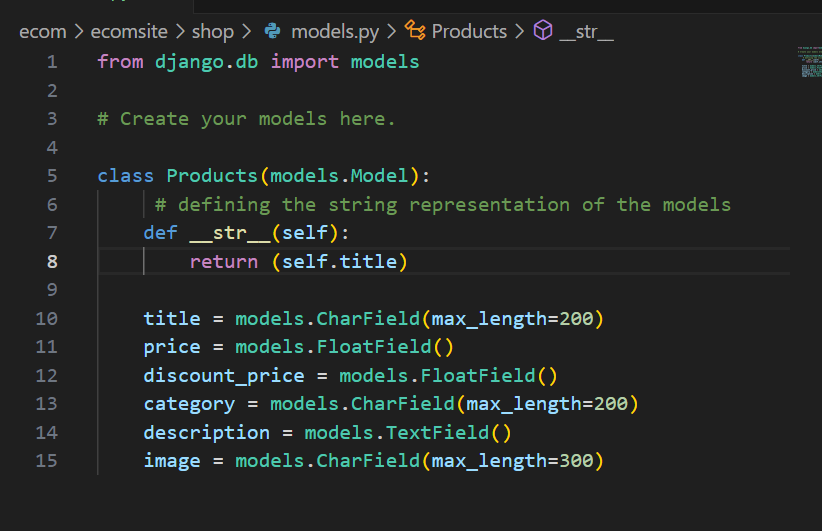
**Reasons to Use Singular Naming for Models:**

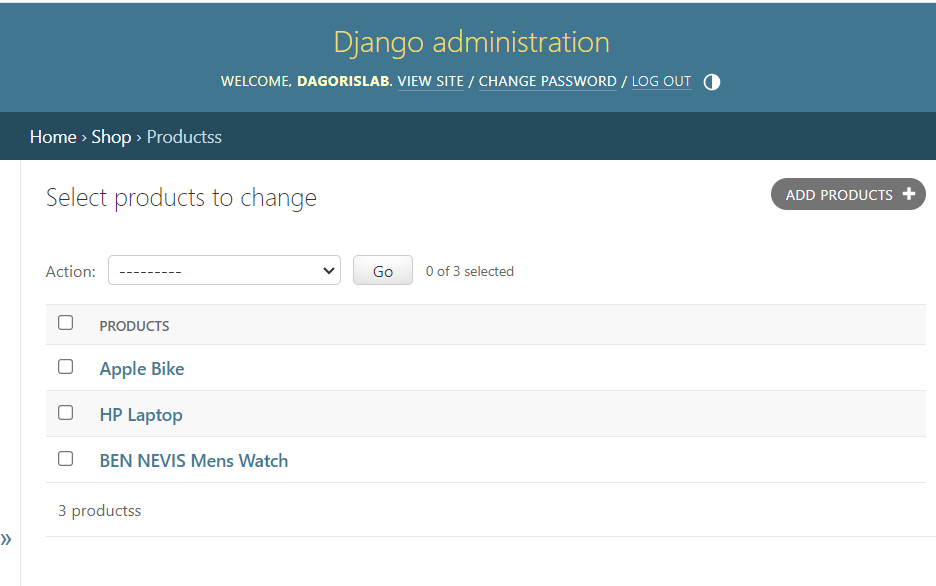
1. **Consistency with Django Conventions:** Django's internal behaviour expects model names to be singular. For example, if you have a model named Movie, Django will automatically use Movies as the default name for the corresponding database table by pluralizing the model name.
2. **Class Naming Conventions:** In Python, class names are typically written in singular form. Since Django models are essentially Python classes, using singular names adheres to this convention.
3. **Readability**: Singular names make the code more readable and intuitive. When you define a model like Movie, it's clear that each instance represents a single movie.
4. **Automatic Pluralization**: Django automatically handles pluralization in many places. For example, in the Django admin, it will display "Movies" if the model is named Movie. This is controlled by Django’s Meta class, where you can also explicitly set verbose\_name\_plural if you need a custom plural form.
5. **Flexibility**: By following this convention, you maintain flexibility and consistency across your project. For example, in querysets, Django will often use the plural form of the model name (movies = Movie.objects.all()), which aligns with the convention.

Back to the project, After adding some products, this is how it displayed



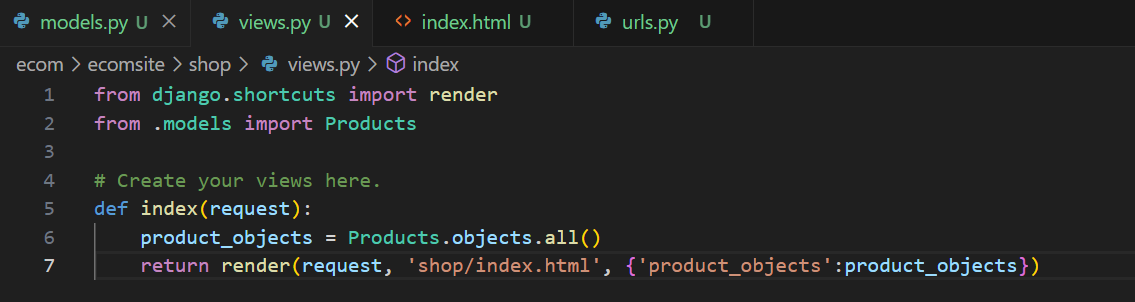
To make it to display the product names, we have to define the string representation of the models as shown below





Next we create a view to display the model inside shop > views.py

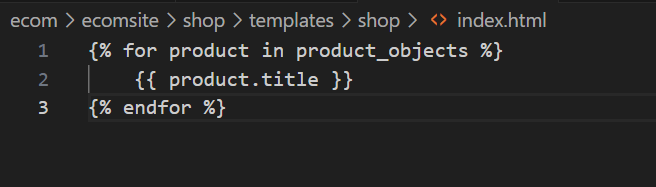
So we will be displaying all the products on the home page, hence, create an index view to do this, you can use any name for the view.



The view will be sent to the index.html template in the shop app which has to be created, the file is created at

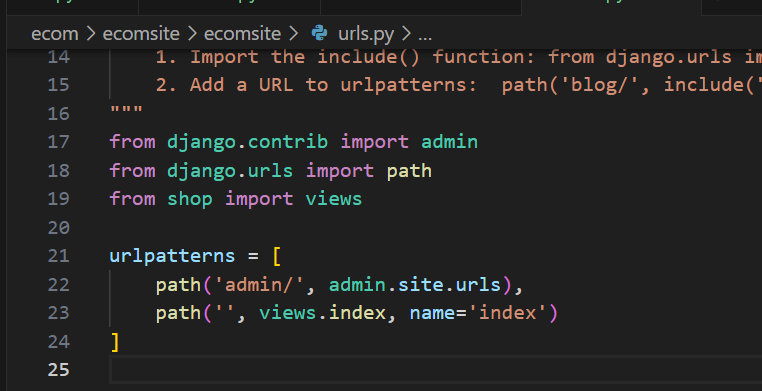
shop>templates>shop>index.html

the **product\_objects** is the context assigned on line 7 of **views.py** above

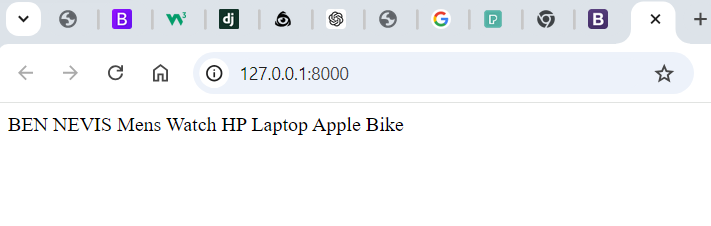


To setup the url pattern for the view, go to ecomsite>ecomsite>url.py

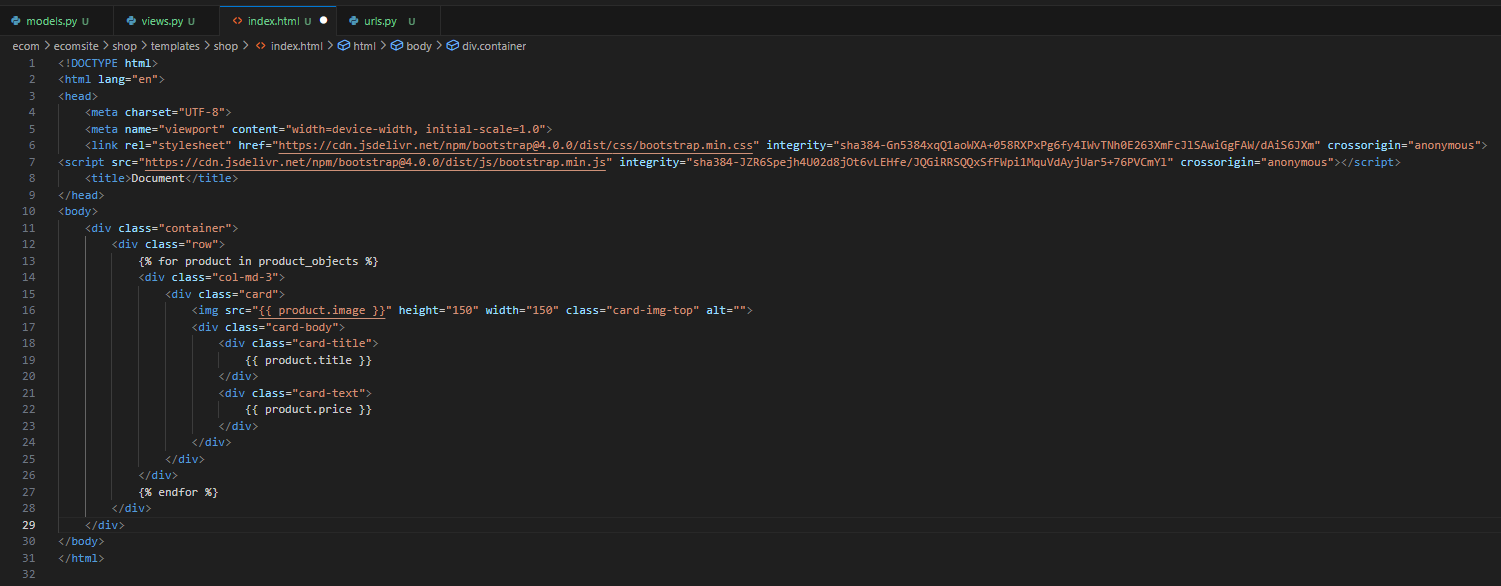
Firstly import the views and specify the url parttern in the urlpatterns list



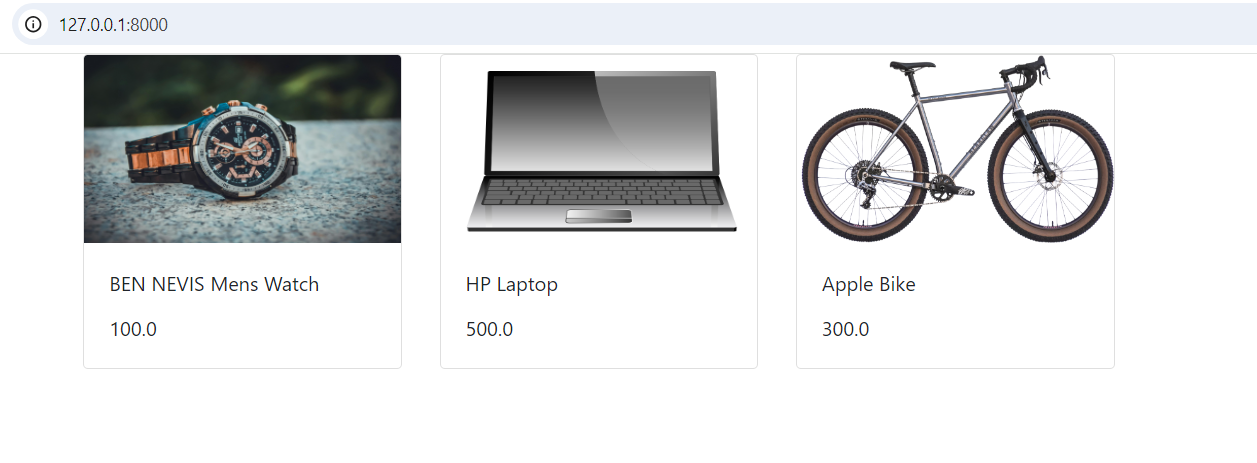
After setting the url and running the server, this is the output



We will be using bootstrap to style the page, after styling the index page we have this



The preview is shown below



**Adding CSS to the Sits**

To add CSS to the website, add a static directory within the app (shop) where the CSS file will be kept.

Create the static file in Shop>static>shop>style.css

Below the bootstrap styles, add the import of the CSS file using

**<link rel="stylesheet" href="{% static 'shop/style.css' %}">**

The **static** tells Django to locate the static directory to find the CSS file specified

At the top of the page, add the following that tells Django to load the static file

{% load static %}

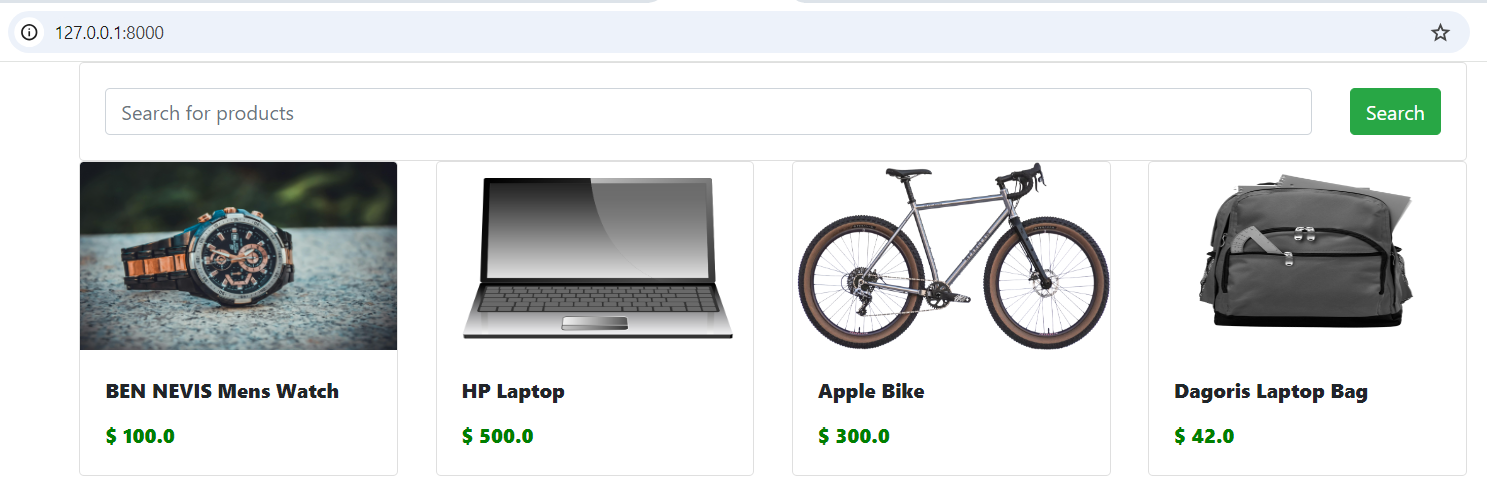
Note: whenever changes are made on the static fields you may need to reload the server to effect the changes

**Adding Search Functionality to the App**

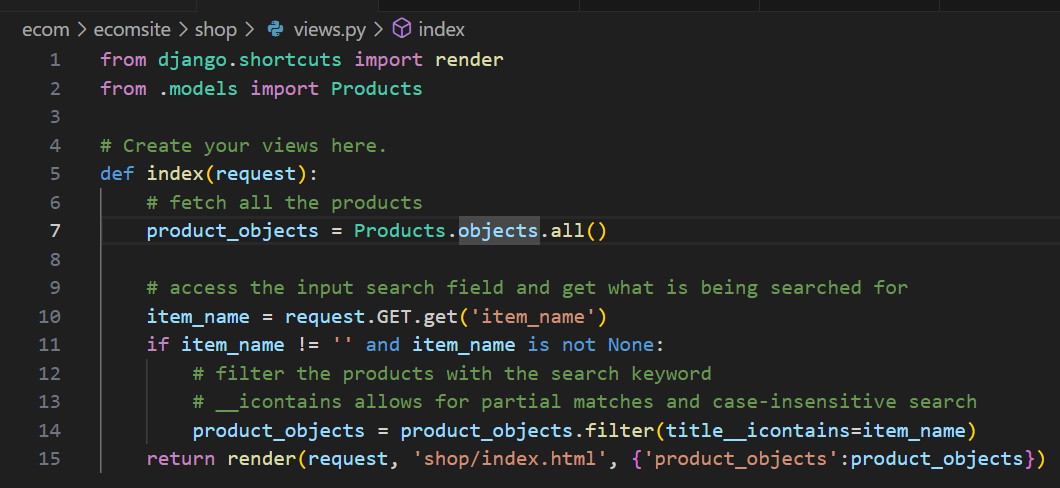
To add a search functionality to the application, firstly, add a search bar to the page with a button



This is the new preview

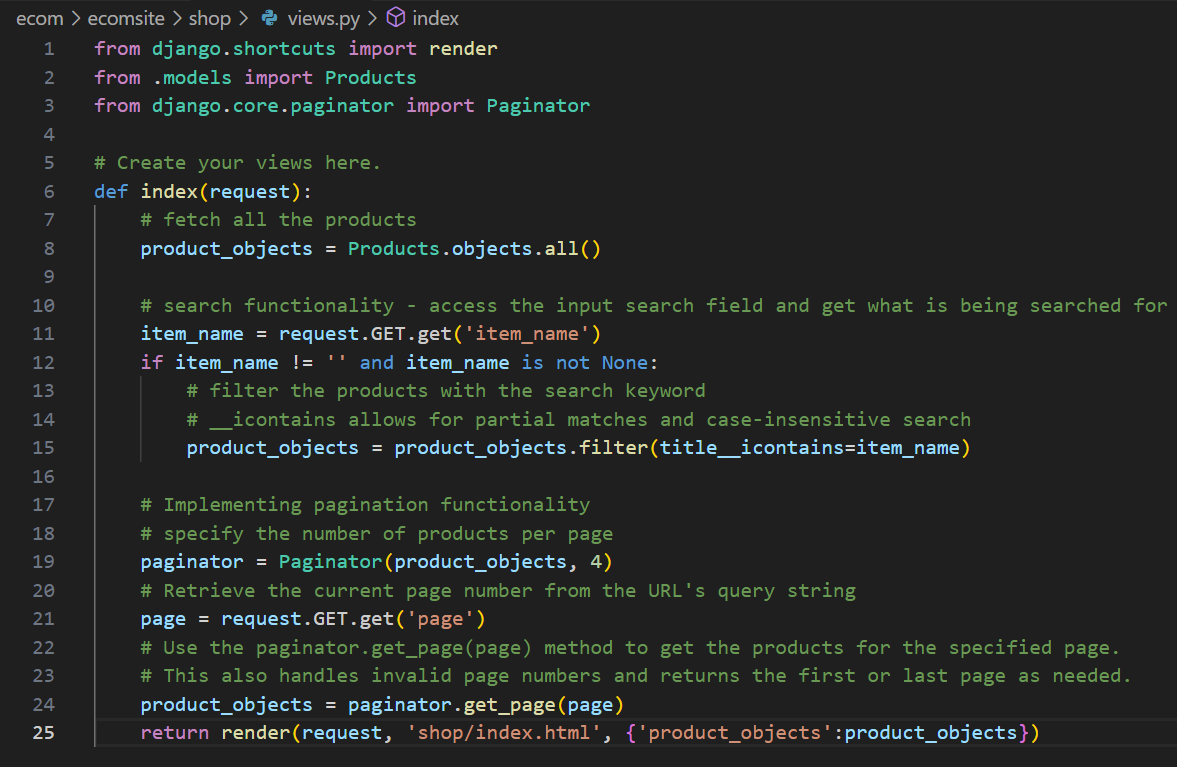


The input search field has a name **item\_name**; this is what will be used to access it. The view was updated to implement the search functionality. The ***\_\_icontains*** allow for partial matches and case-insensitive searches.



**Adding Pagination to the app**

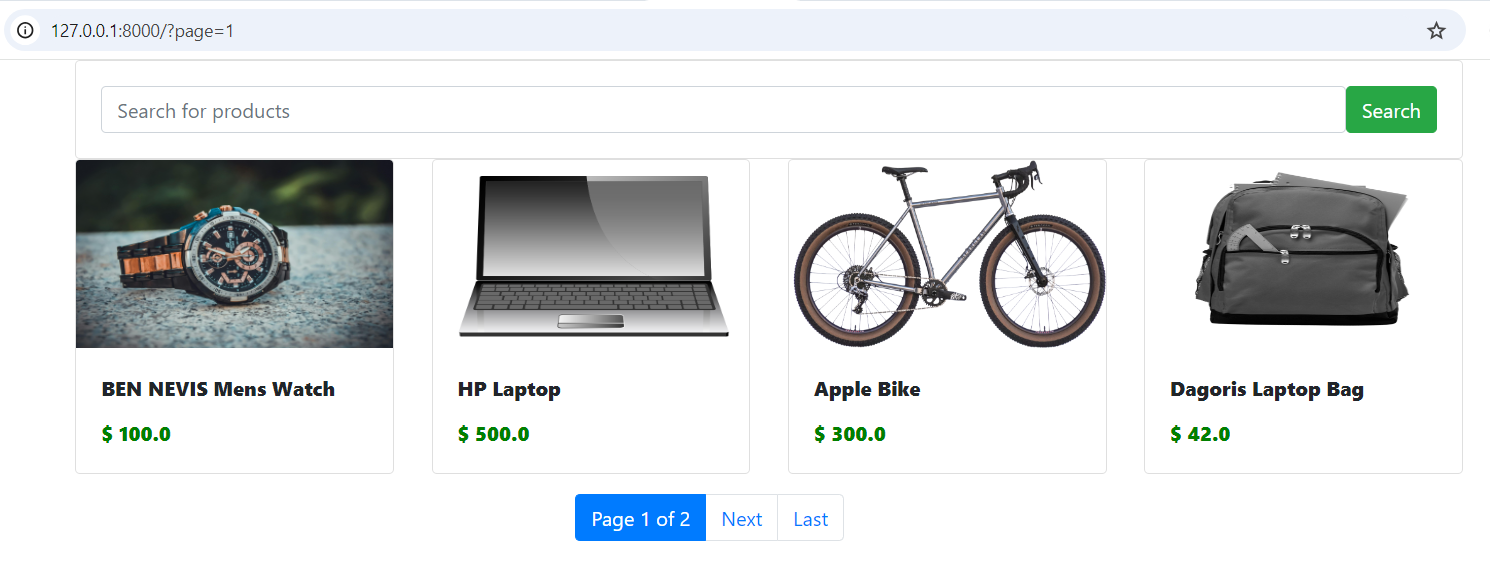
Pagination helps to limit the number of items on a single page, the pagination code has to be below the search code. To implement pagination, firstly you import the ***Paginator***, then we modify the views.py and the index.html pages accordingly.

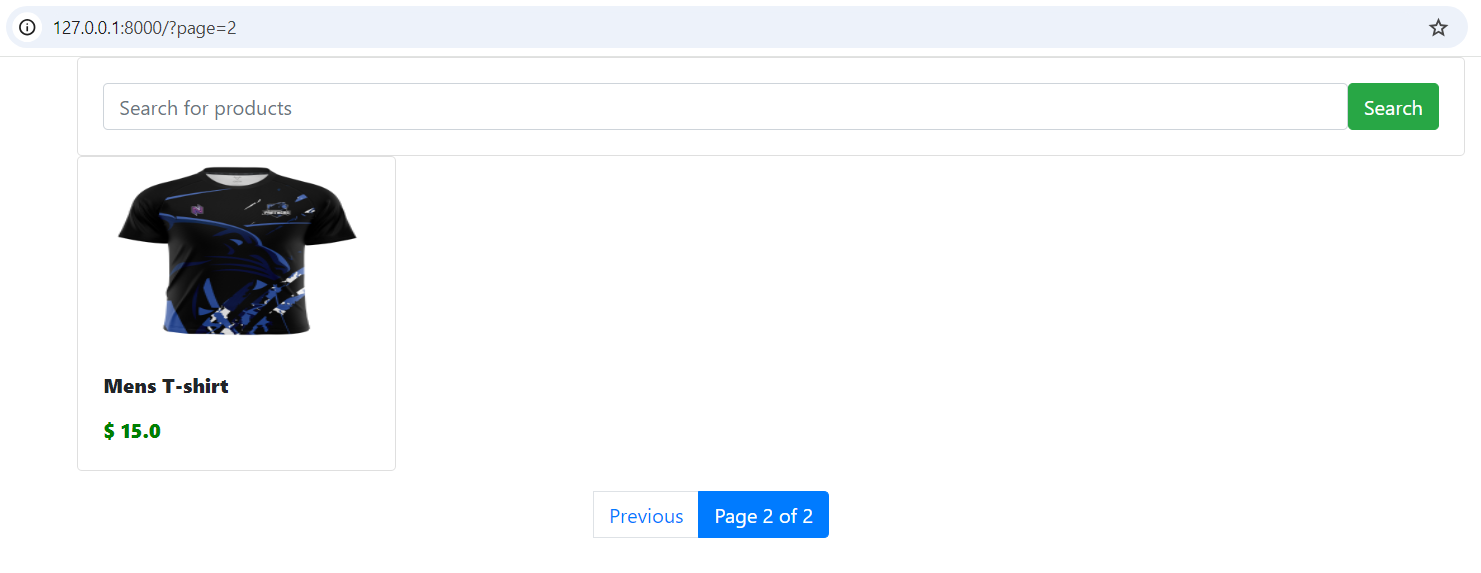




On the index.html template above, the following functionalities were added to the pagination.

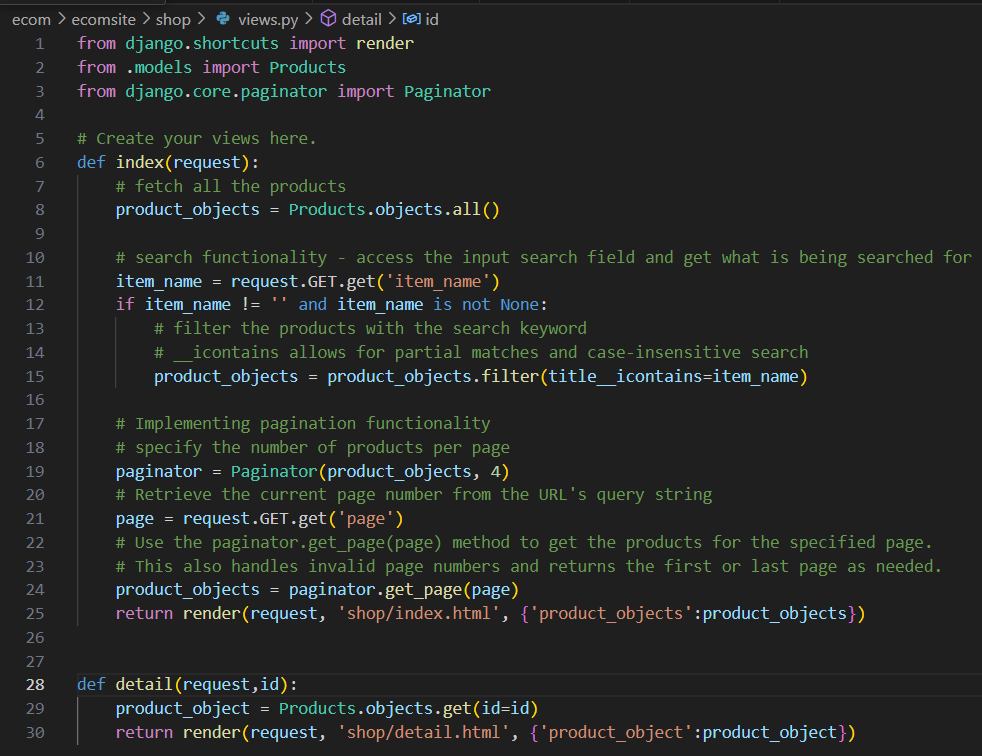
* First Link: Ensure the "First" page link doesn’t show if the current page is the first page. Shown only if the current page is greater than 2.
* Next Link: Always shown if there is a next page.
* Current page: Show "Current of Total" pages for the current page.
* Last Link: Show 'Last' link if the current page is not the last page

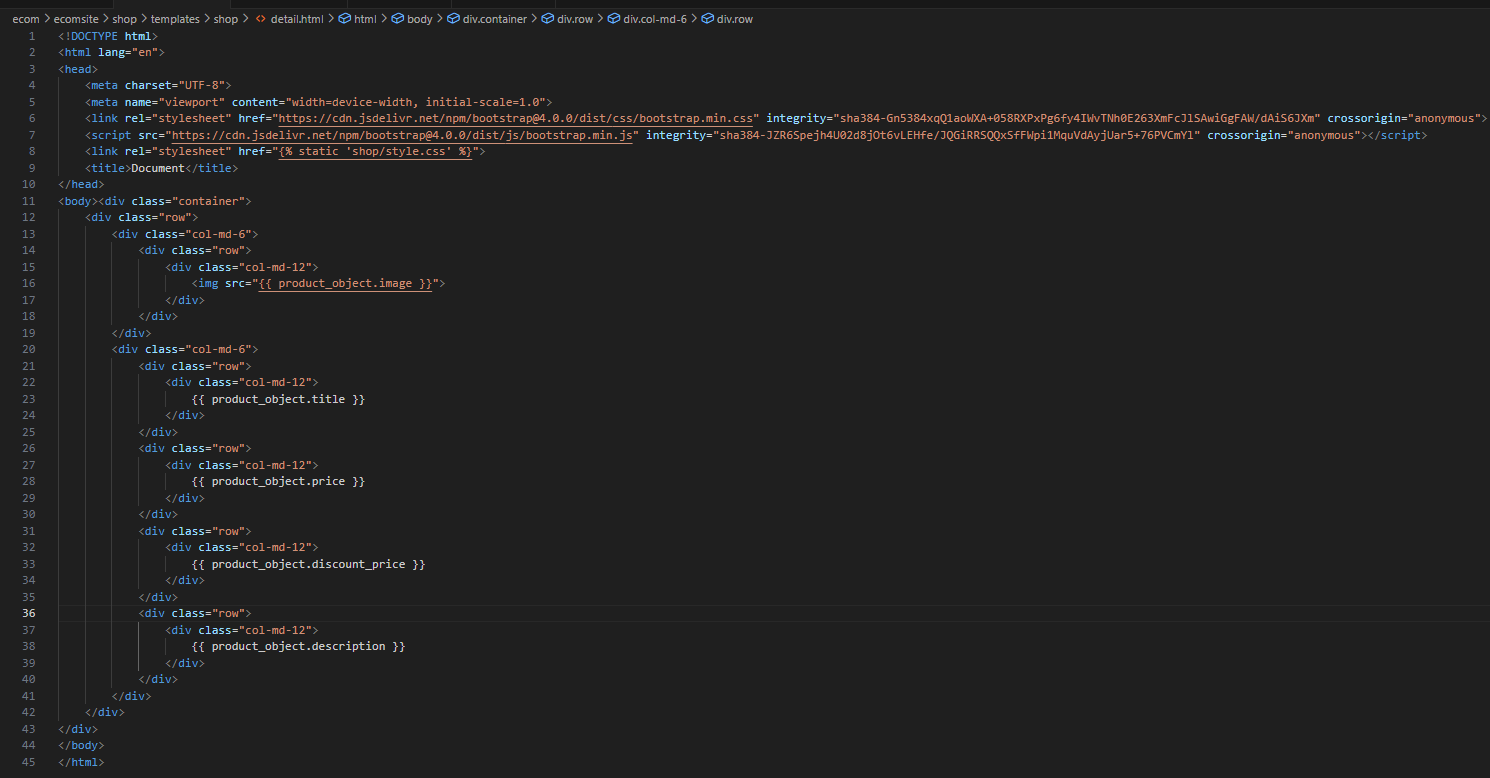




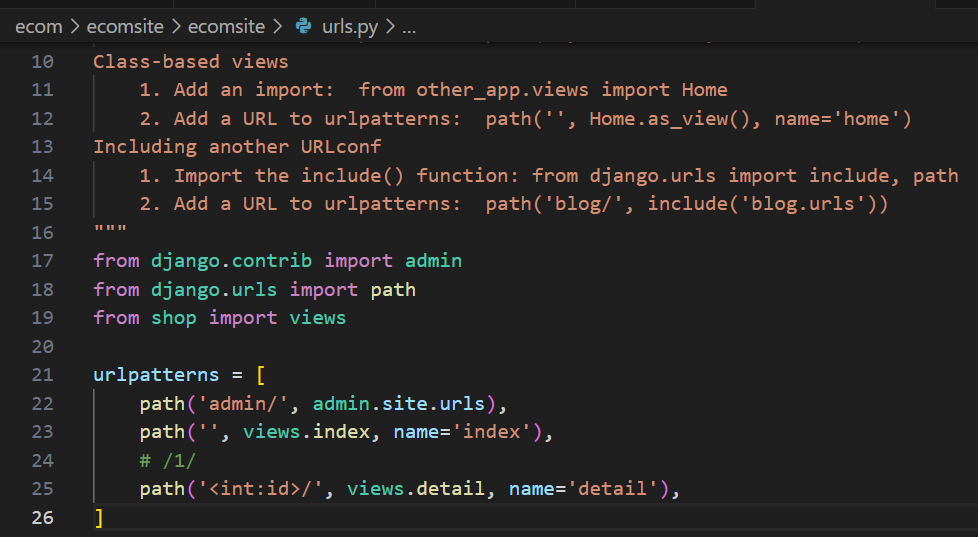
**Create Detail View for Products**

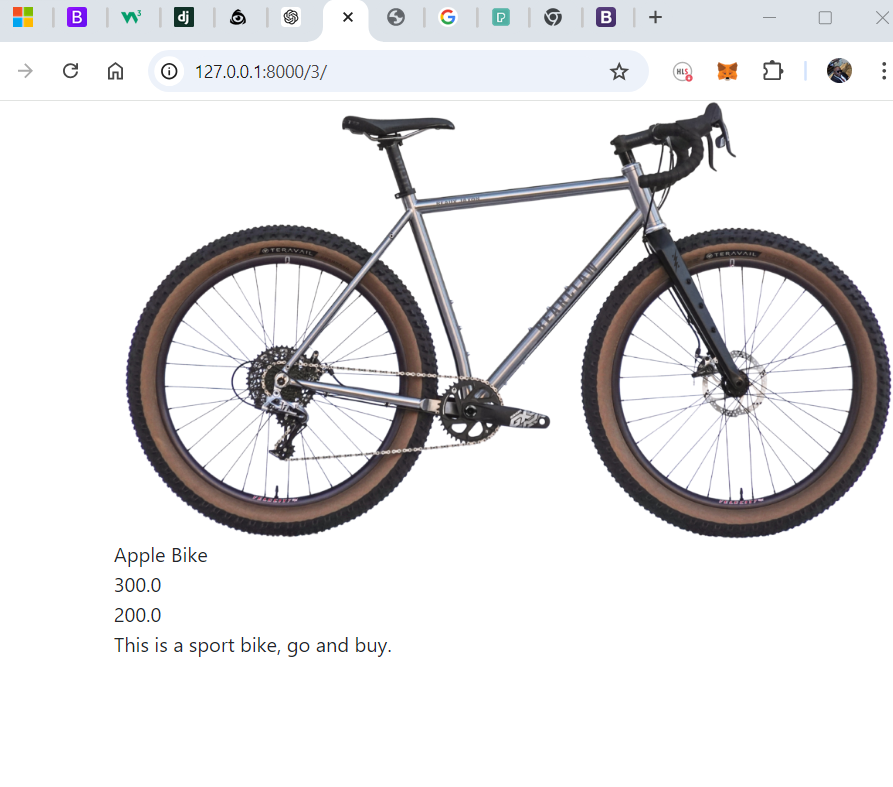
To achieve this, we create a view for the detail page in views.py, and template where the view will be displayed, let's call it detail.html



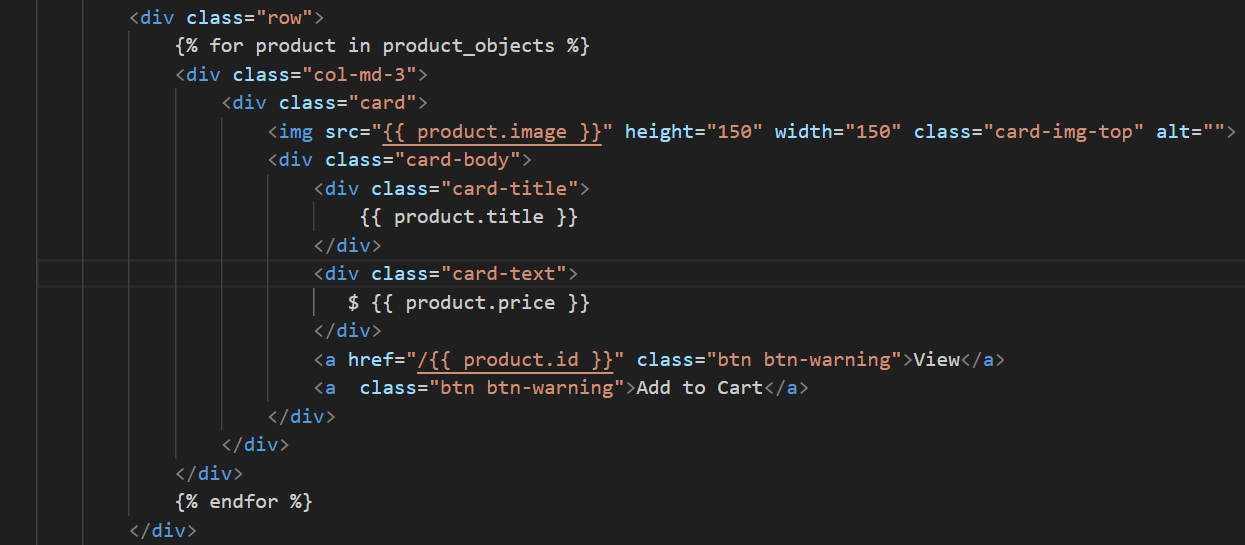


After creating the detail views and template, we need to set the URL pattern for the detail view





After that modified index or the product page to have two buttons to view and add products to cart.



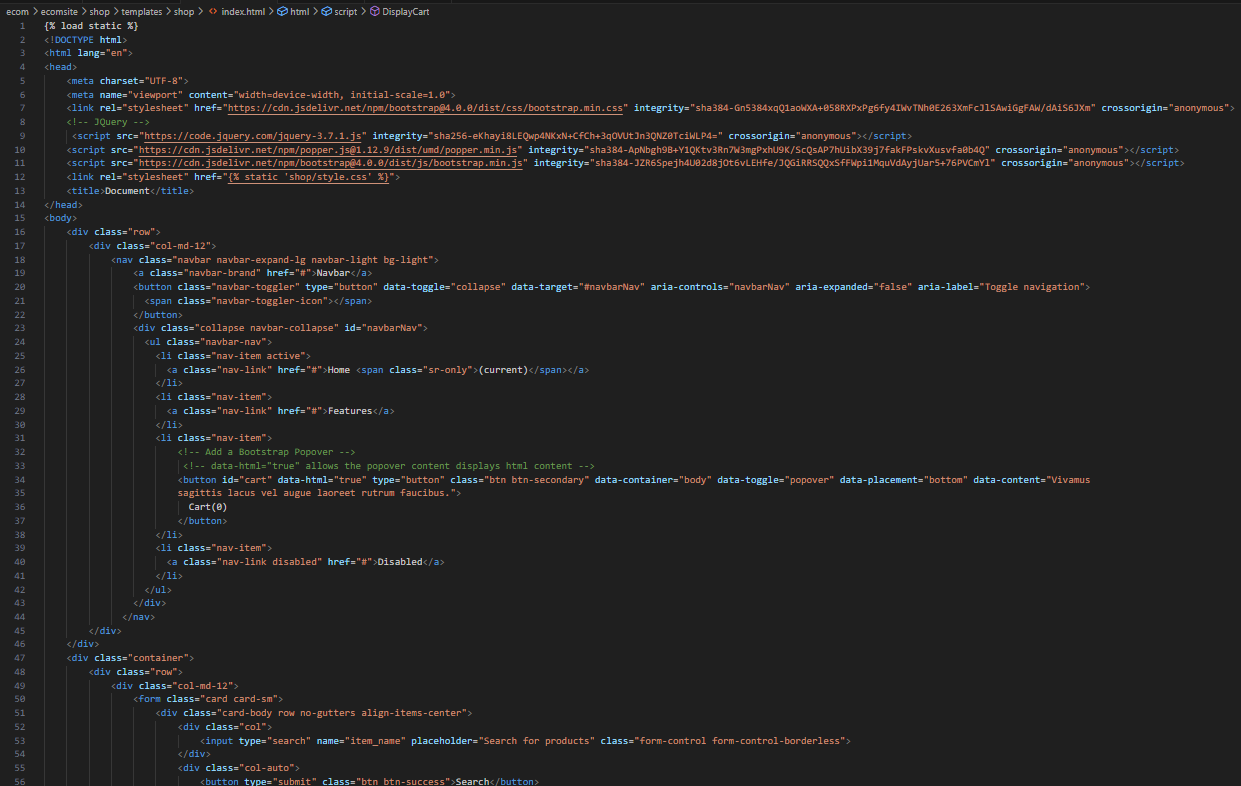
**Create Add To Cart Functionality**

We will use JQuery to help implement this functionality, and the cart details will be stored in the local storage using JavaScript. The cart functionality is been implemented on the index or product list page. We will also be displaying the number of items in the cart in the navigation. Navigation from Bootstrap was copied and used. Also, we will be displaying the cart in the navigation as a popover, the popover was implemented using Bootstrap JS popover components.

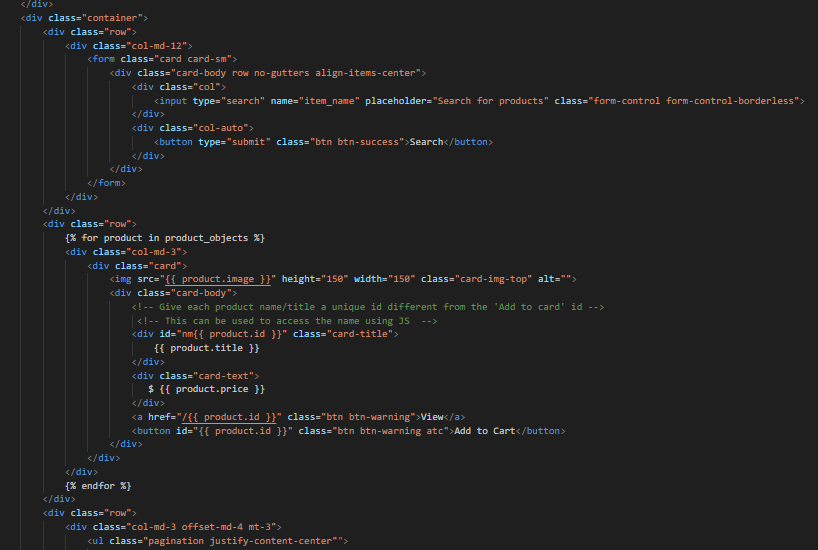
The index.html is give below, it uses JS to listen for a click event on the Add to cart button and displays it on the navigation, it is first stored on the browser localstorage



The above cord is the jS code, also below is the same page showing navbar added, adjustment made by adding a bootstrap popover component and the ID set, the popover was populated using the above JS code



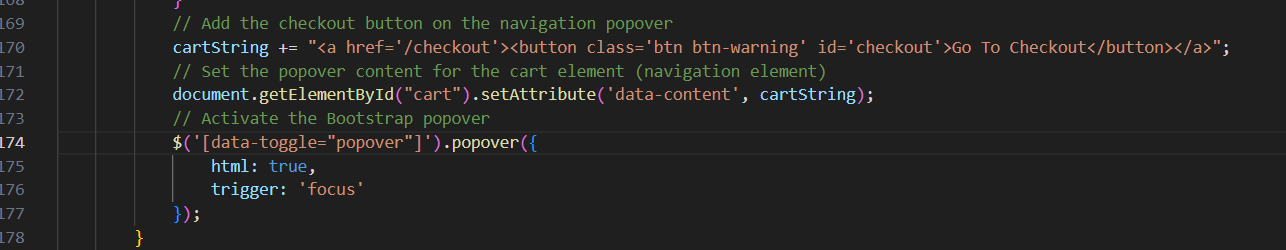
The below code is still the Index page showing the modification made by assigning a dynamic id to each product Add to Cart button and to each product title or name. The Id for the product title was modified so as to be different from that of the add to Cart button.



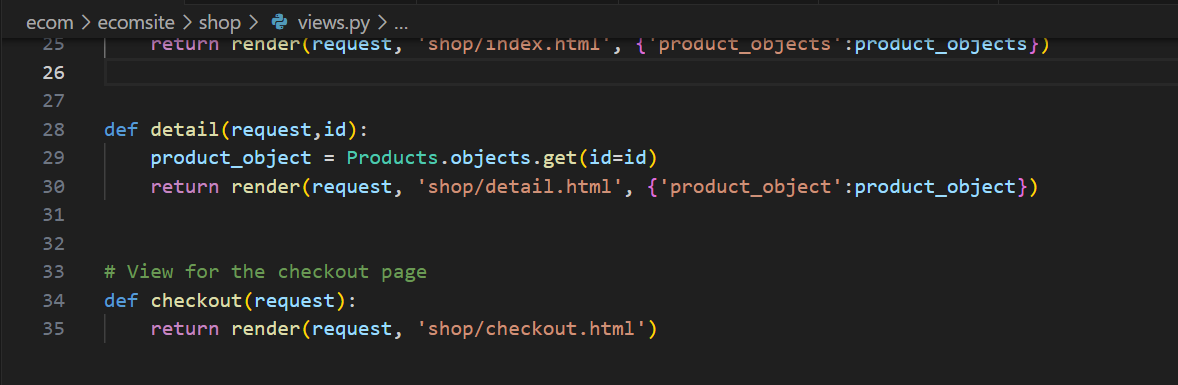
**Add a Go to Cart button on the Popover**

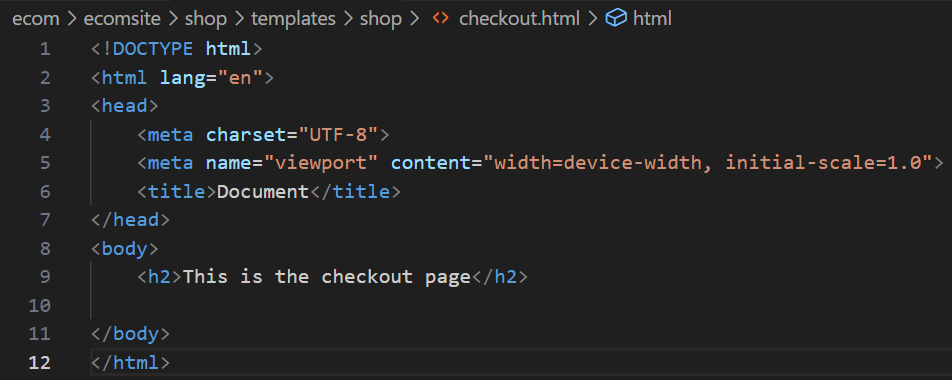
To create the checkout functionality:

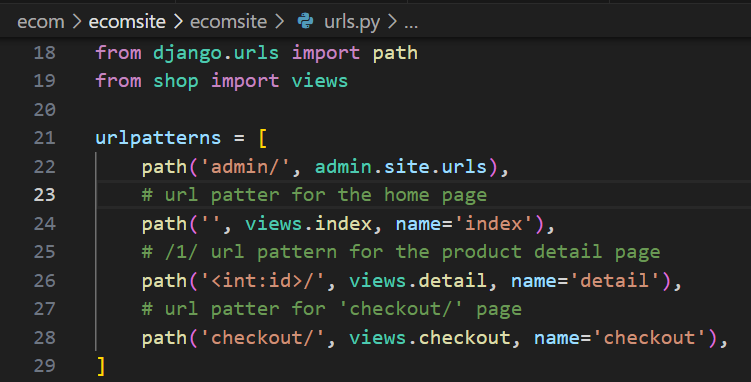
A button was added on the popover, this was done in the JS code on the index.html page, below is the modification made. As you can see the button is linked to a checkout link which has not been created.



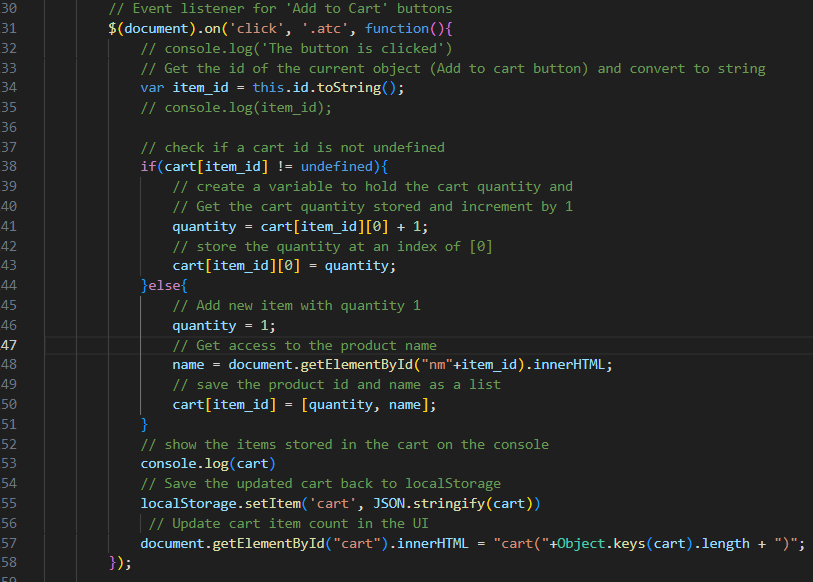
Next, we create the view for the checkout and the corresponding template. Below is the view created for the checkout page and the corresponding checkout.html page, and then, following shows modifications done on the url.py page







Next is to format the checkout page, a bootstrap list group is also used in the design. Firstly the index.html was modified to allow storing both the product ID and name in localstorage, as this will enable for easy access of the product name on the checkout page. The modification made on the index.html template is shown below.



After formatting the checkout template, figure shown below; it has JS script codes that retrieve the product cart information from the local storage and display it on the screen. The page also contains an HTML form to collect the customer's information. This information is retrieved by modifying the views.py > checkout. For the data to be stored in the database, a model has to be created to store this information.



Figure: Checkout.html. Note that the csrf token is added whenever form data is to be submitted. This secures the form from attackers.

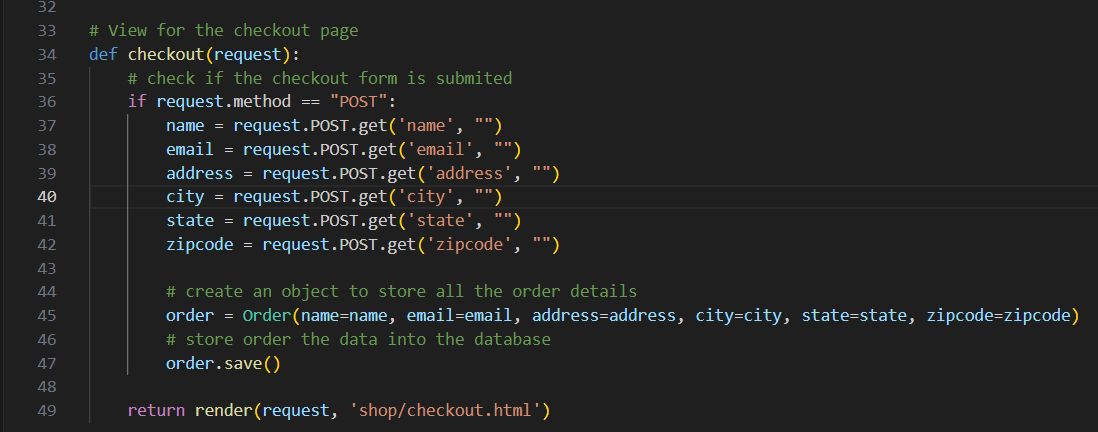


Figure: views.py, modified to accept and save customer information to the database table ‘Order’.

After this, a model was created to store the aforementioned customer order details

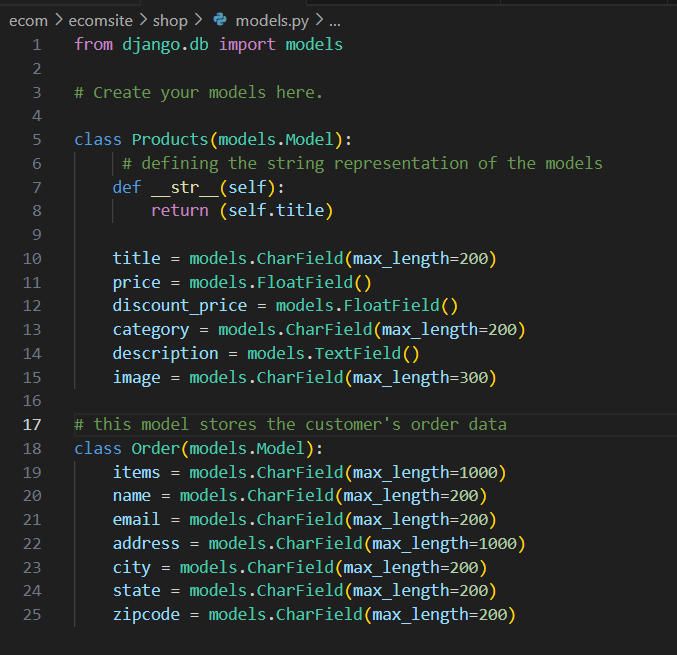
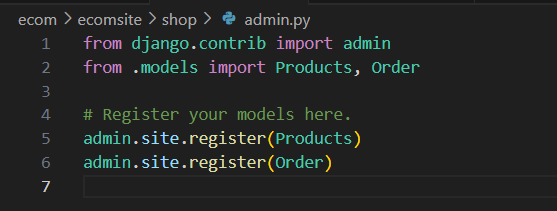


Figure: modified models.py file to contain models to store customer’s order details. After doing this, migrations were made for the model, this creates the order table in the backend. This admin.py file was also modified.



Up until now, we could capture the customer order details and save them in the database, but not the ordered item. This is the next task.

To be able to capture the ordered items in the list group and save it to the database,

Create a hidden input field in the checkout.html page at the top of the checkout form and used JS to determine its value

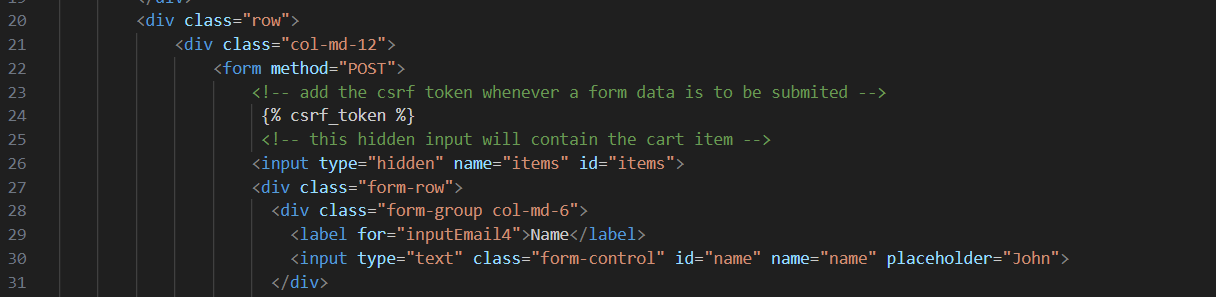


Figure: checkout.html hidden input added

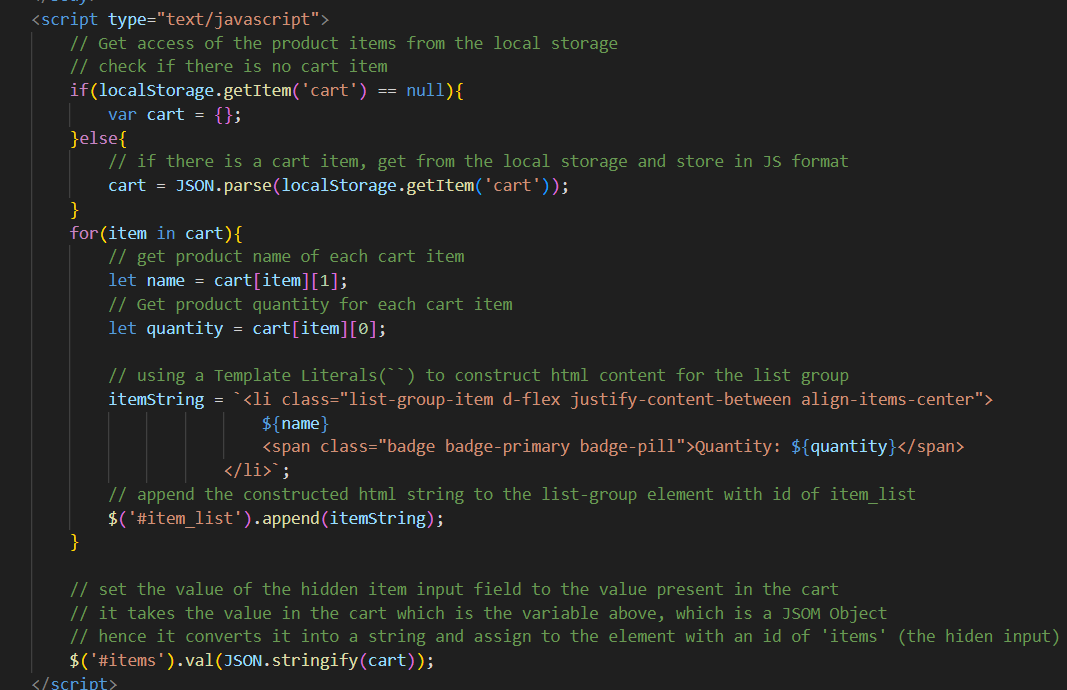


Figure: checkout.html JS added to automatically determine the value of the hidden field.

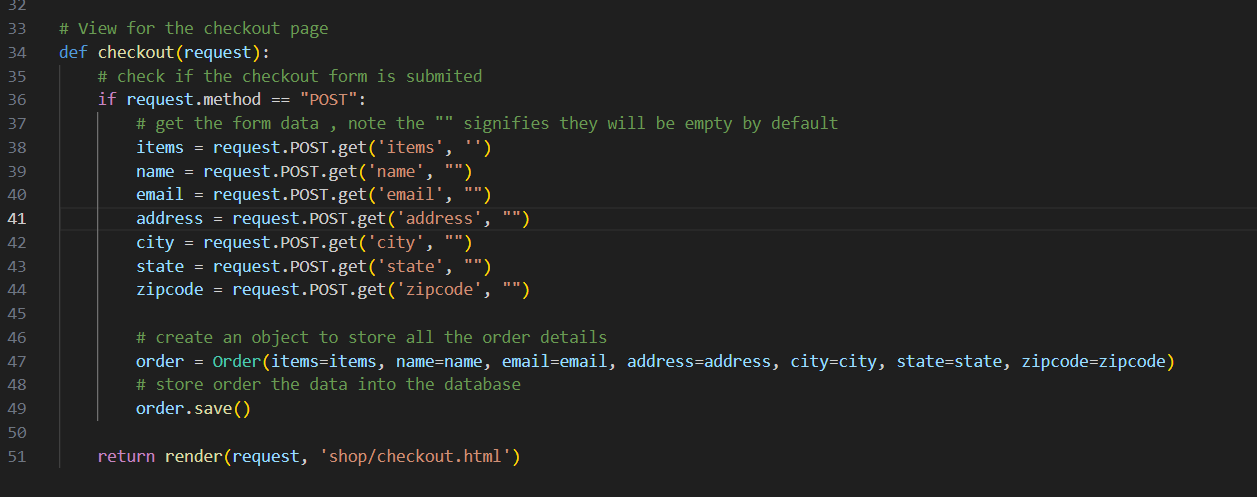


Figure: views.py, checkout view modified to accommodate the product items from the hidden input on the checkout page.

Next, the checkout page will be designed to include the price of each item and the total. Hence, we must capture each product's ID from the local storage to calculate the total cost. Therefore, the index or products page has to be modified to include an ID for the price element. The ID is added below for the price

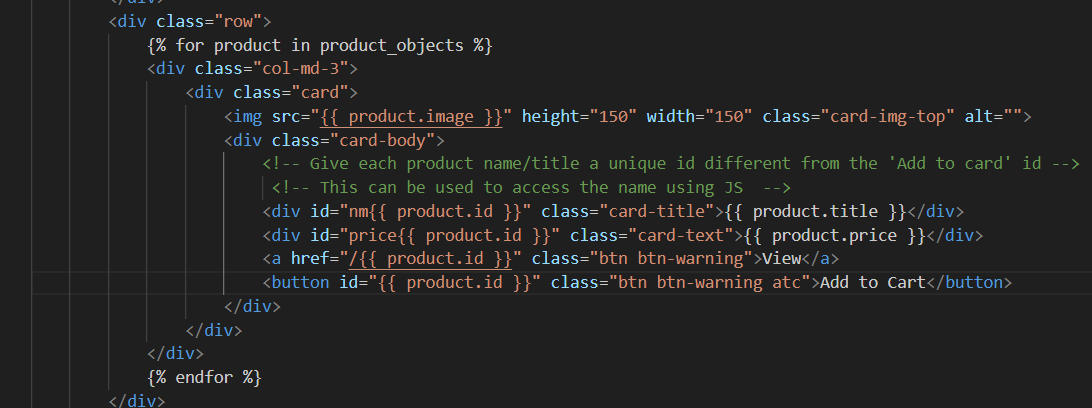


Figure: index.html

Now, we need to store the product price in the local storage. This has been implemented below

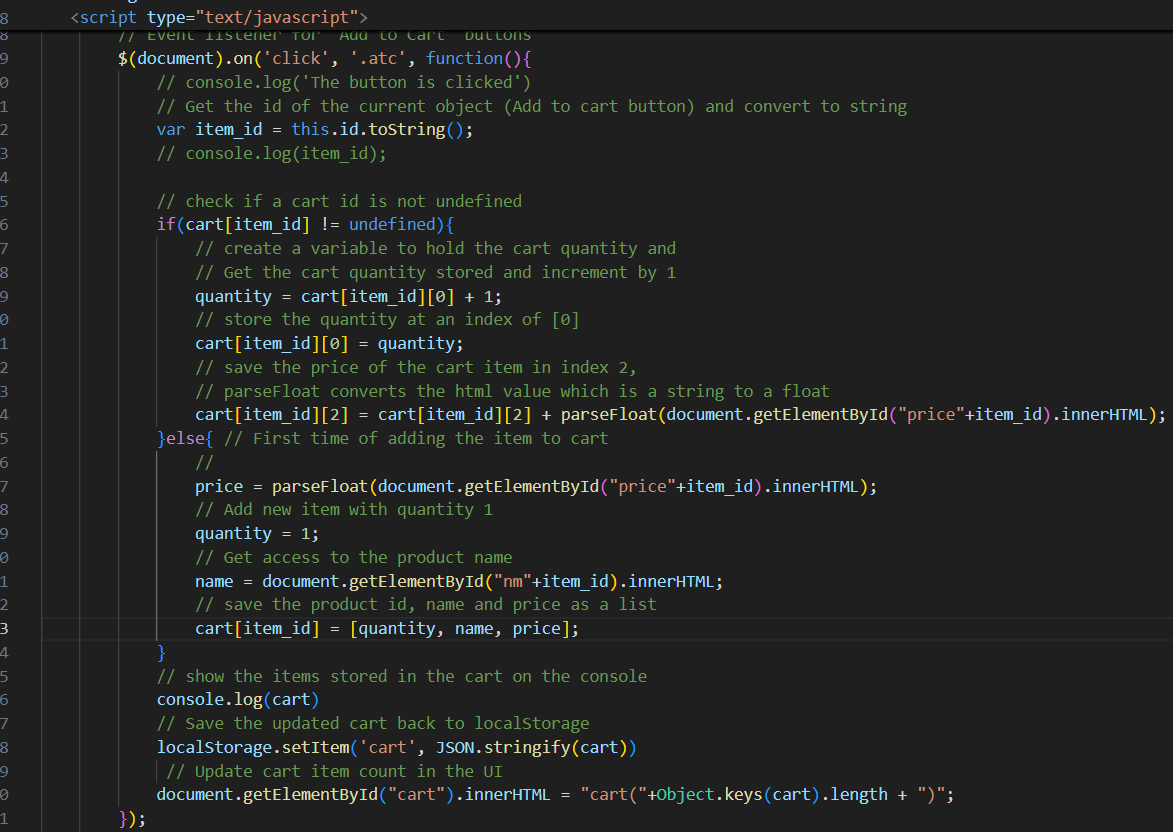
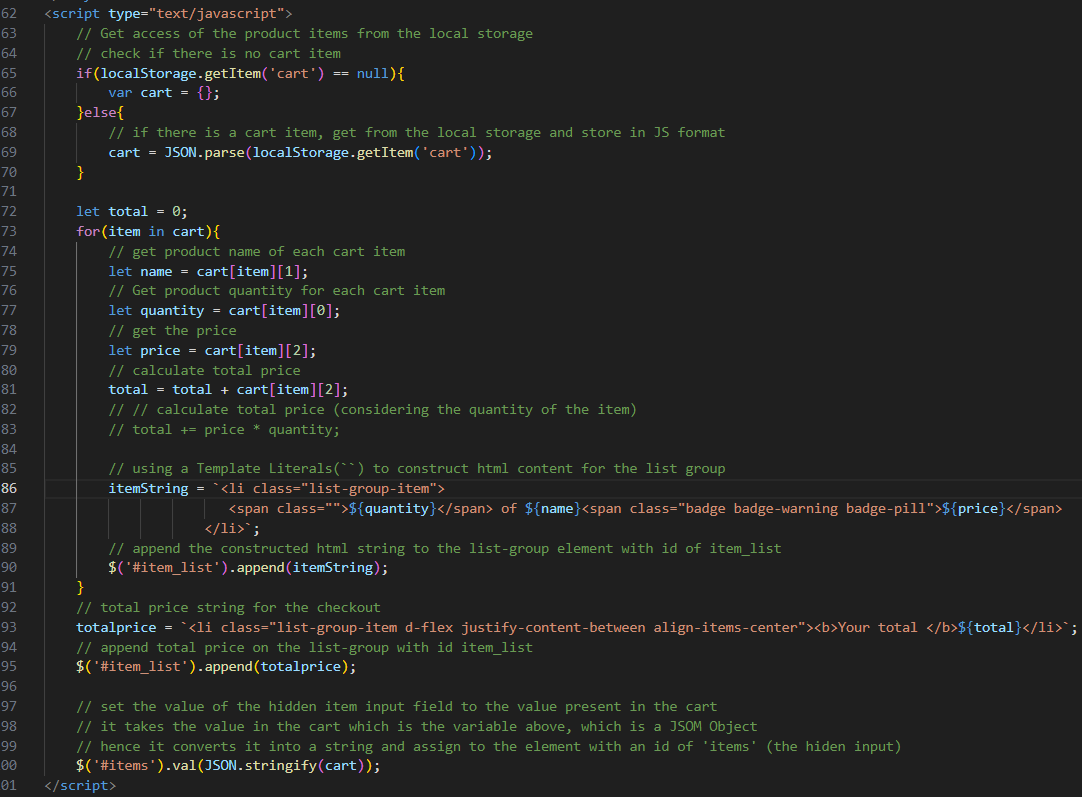
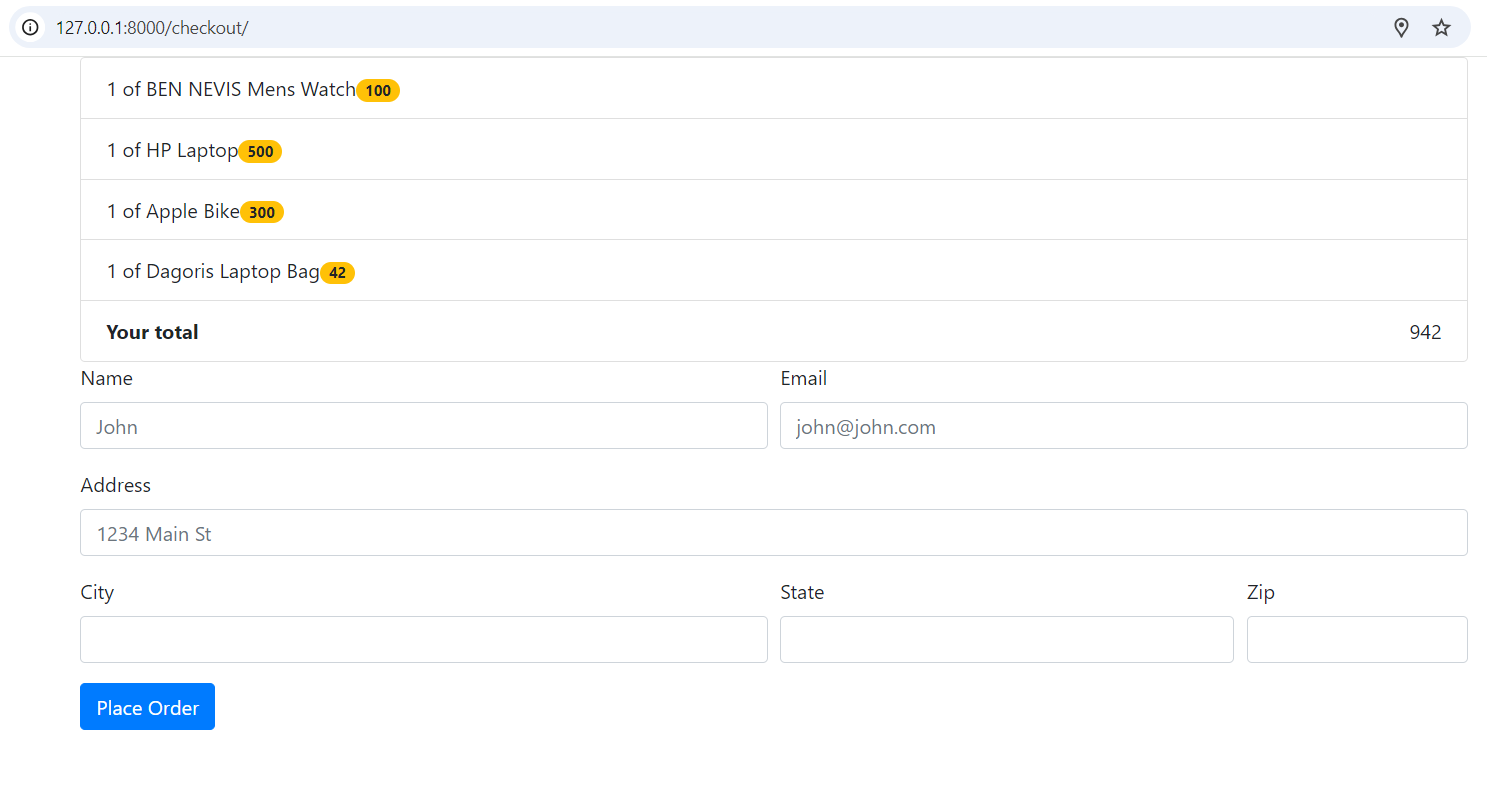
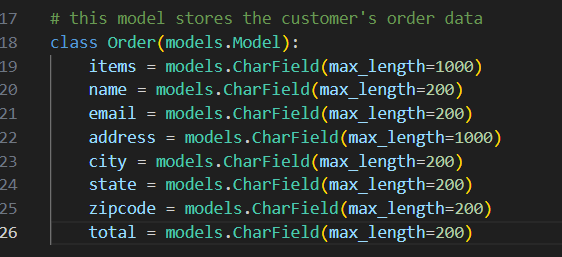


Figure: index.html Adjust JS to accommodate the product (or total price per item depending on number of times the add to cart button is clicked) in the cart and store it in the local storage. We have to populate the total price on the checkout page, this achieved below.

Figure: checkout.html modified JS to show total price on page and little formatting, he output is shown below



Next, we need to store the total price in the database. To this end, a field was added to the ‘Order’ model, and migrations were made as changes were made to the database. The field is stored as a character, but it can also be stored as a float.



Durng migration a defult value of one was set for the total price field and this will be used for previous orders.

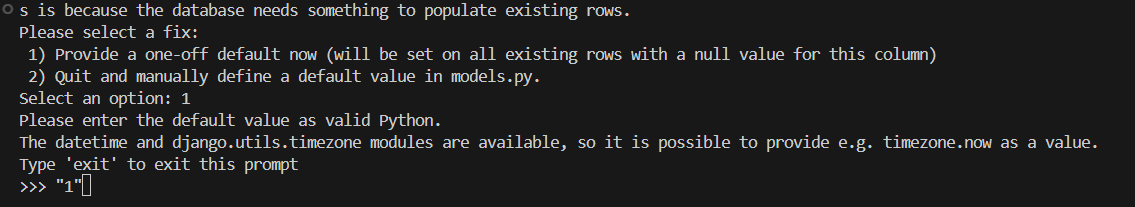


Figure Terminal during migration, option 1 was selected and default value of one was used.

So, to store the total price in the database, we have to create a read-only form input whose value (total price) will be assigned using javascript so it can be sent when the form is submitted.

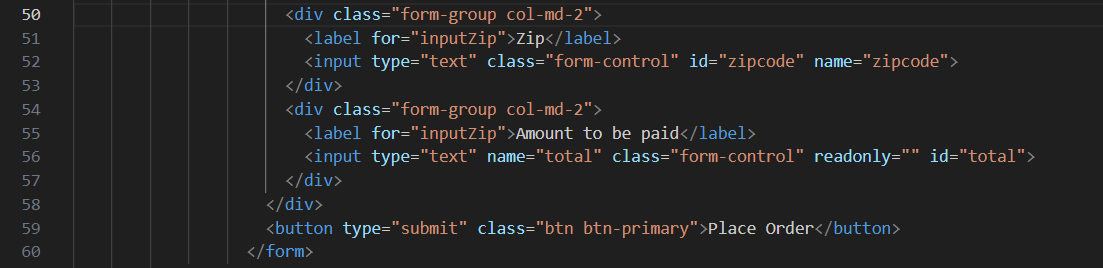


Figure: checkout.html modified form to contain read-only input for total price

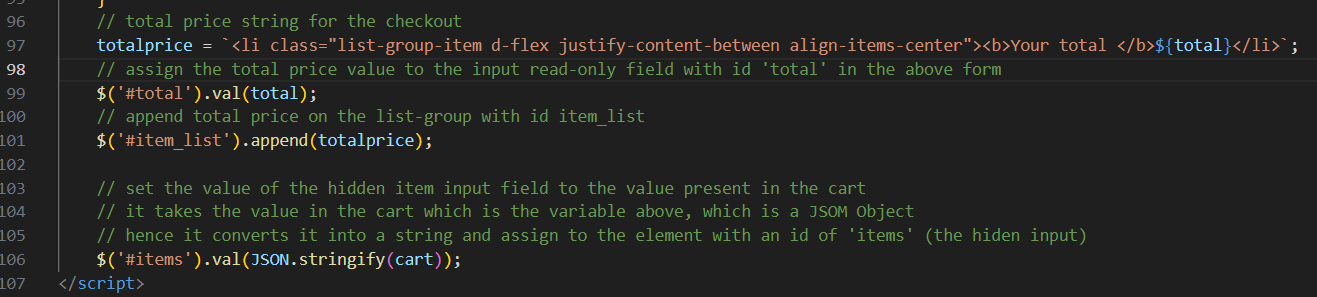


Figure: checkout.html modified JS code to assign the total price value to the input read-only field with id 'total' in the above form.

Next, the views.py has to be modified so it can save the total price to the database and its given below.

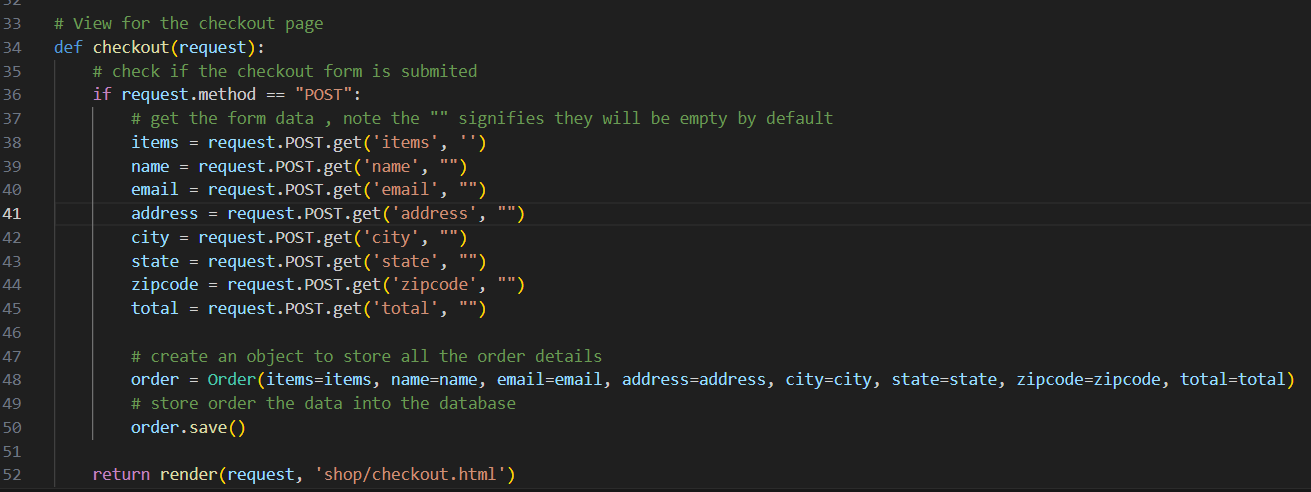


Figure: views.py modified checkout views to include the ‘total’ price

**Fix the Add to Cart**

The add to cart functionality has an issue that, say you add a product from page 1, it adds it to the popover, if you go to a different page the popover starts malfunctioning. To fix it in the JS function DisplayCart(cart) instead of fetching items directly from the html object, its better to fetch it from the cartObject. This is because the items has already been added into the local storage with the event listener click event ***$(document).on('click', '.atc', function(),*** function. The cartObject is the cart stored in the local storage. Below is the modification

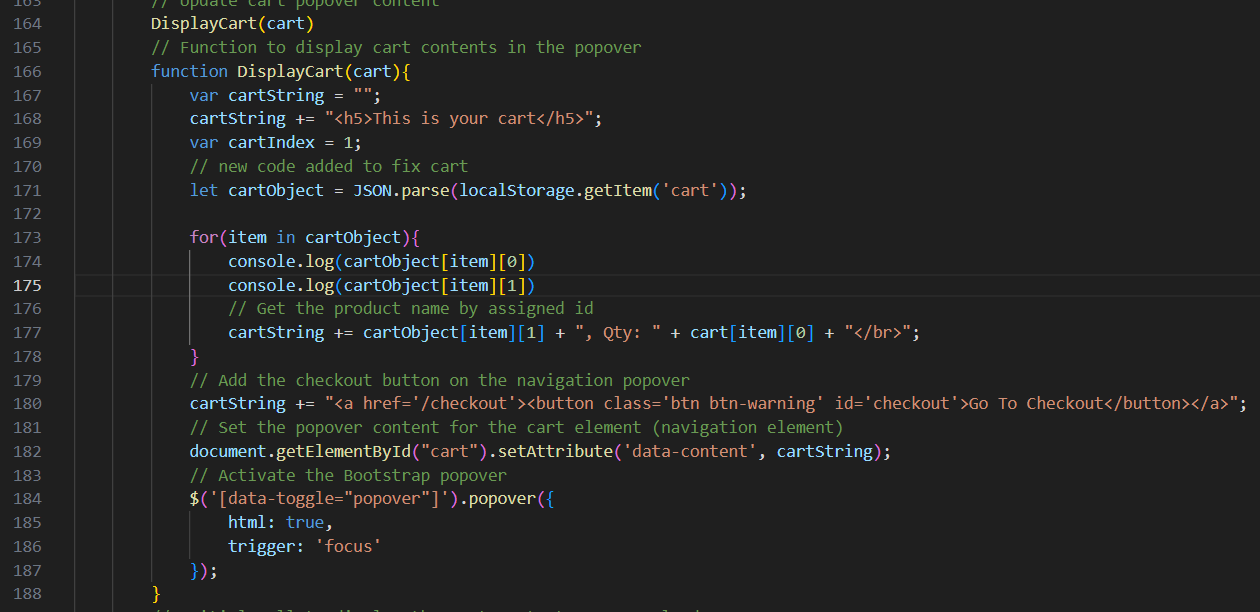
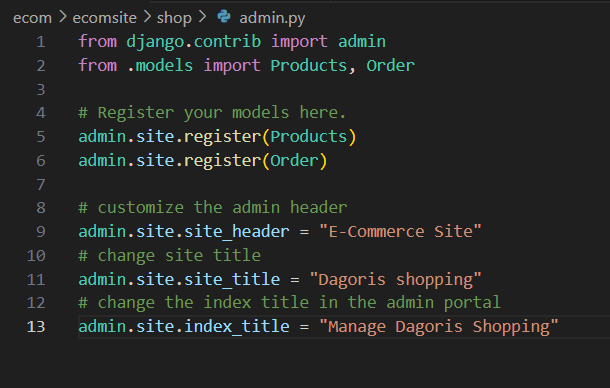
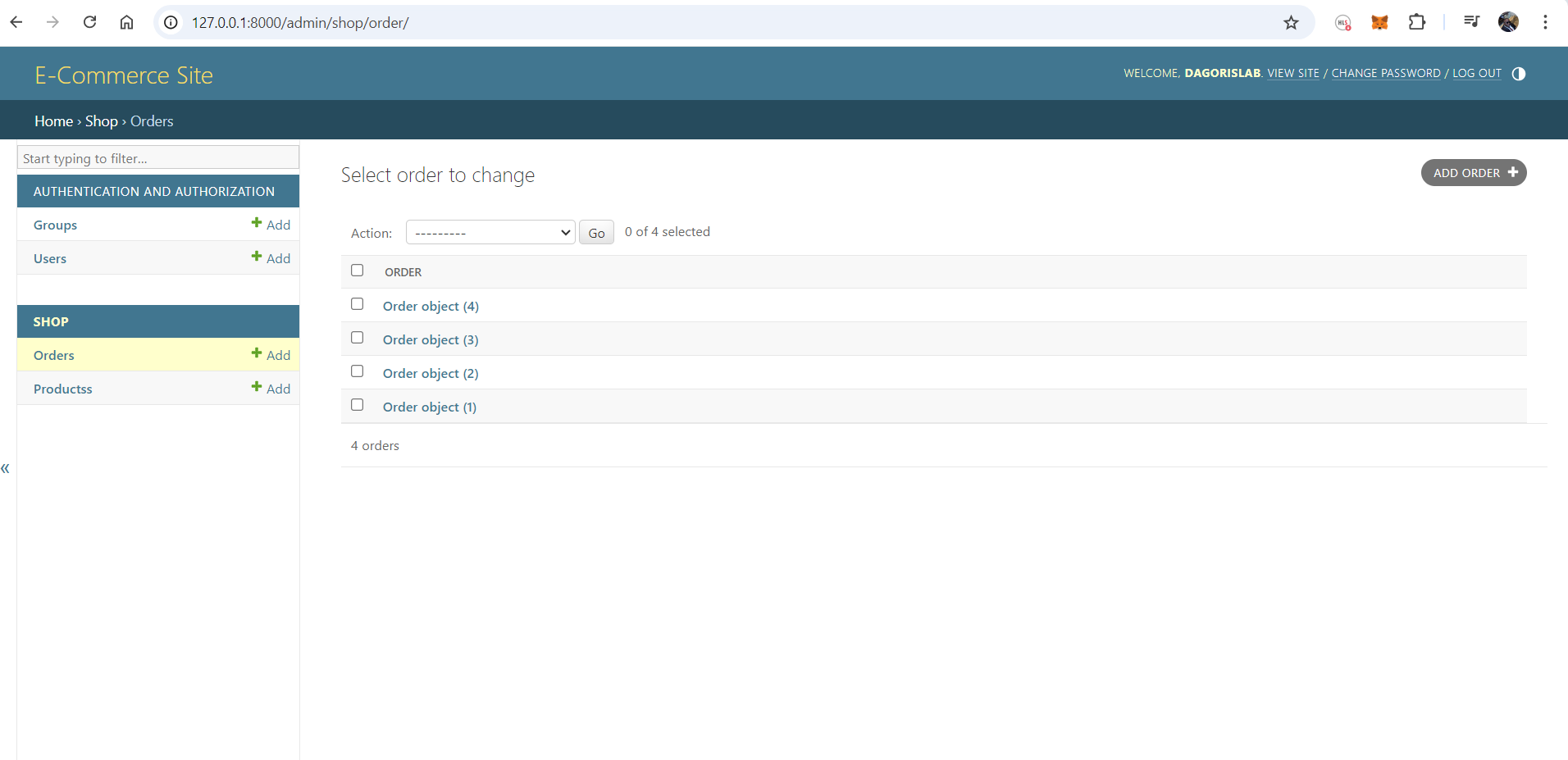


Figure: index.html Fixed ‘Add to Cart’ issue

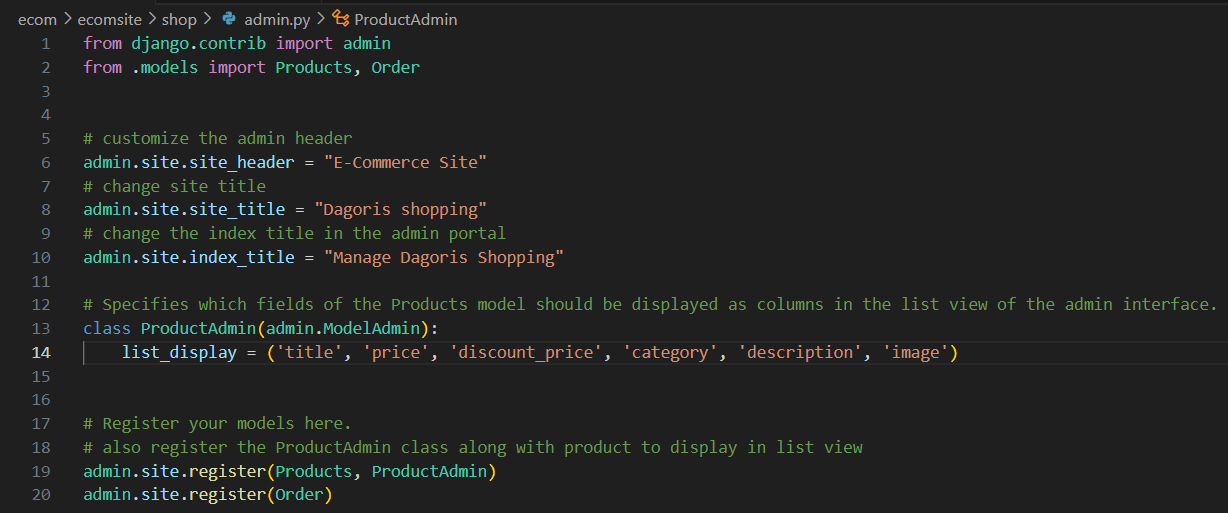
**Customising the Django Admin Panel**

In customising the Django admin panel, you need to go to the admin.py

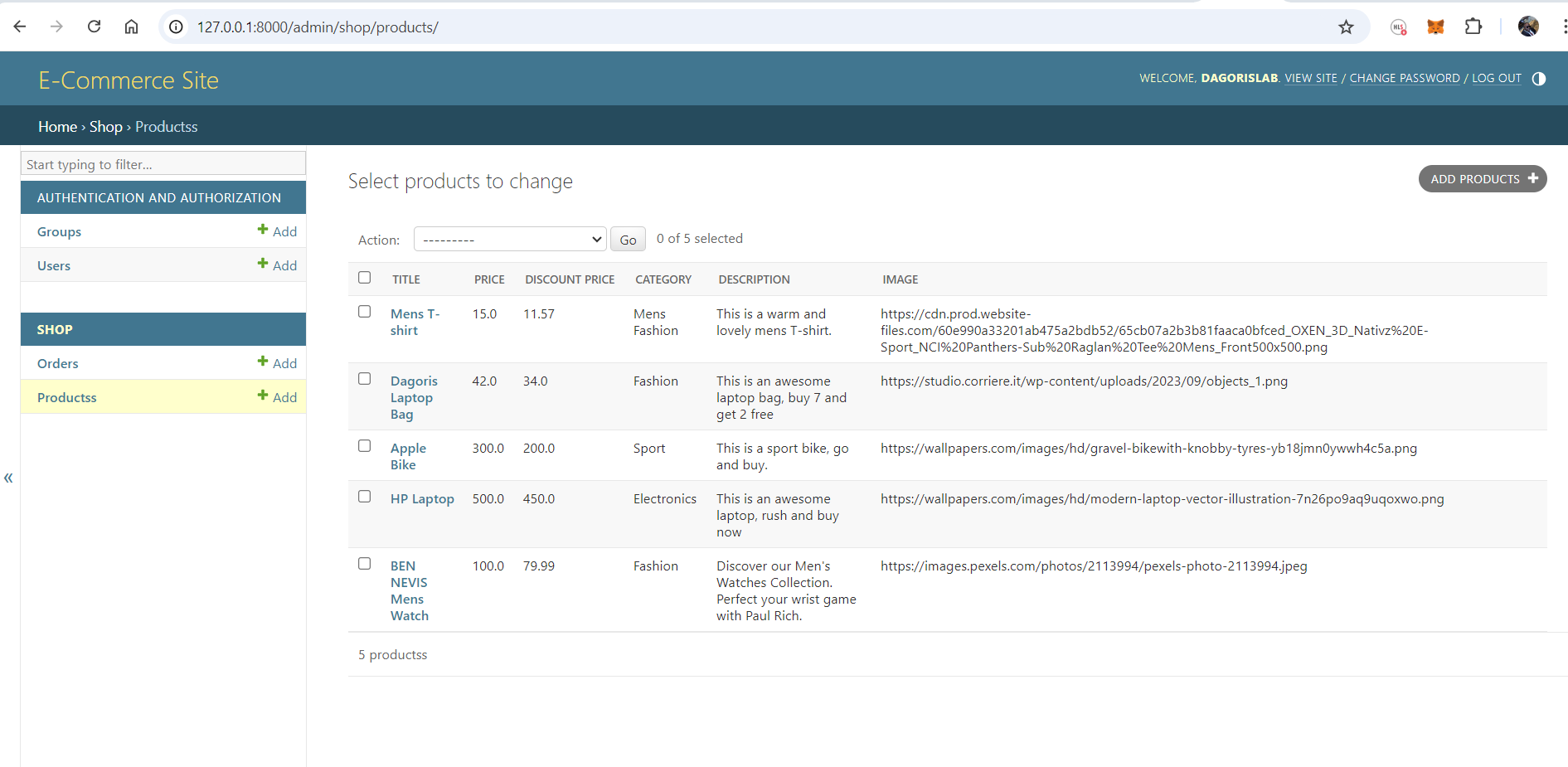




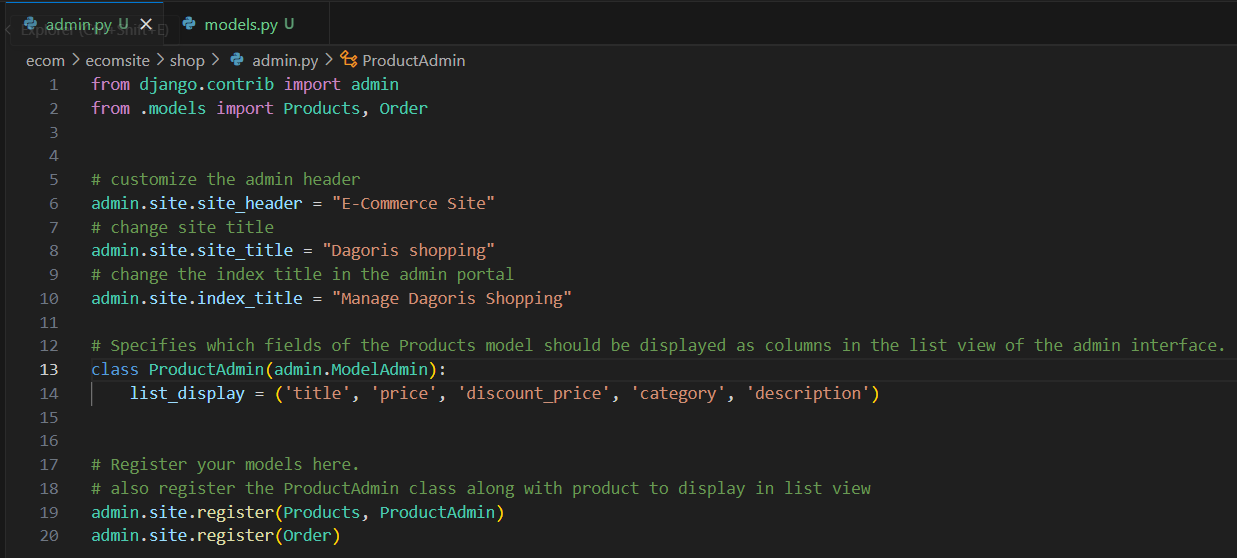
To specifies which fields of the Products model should be displayed as columns in the list view of the admin interface. This is done in the admin.py file as shown below

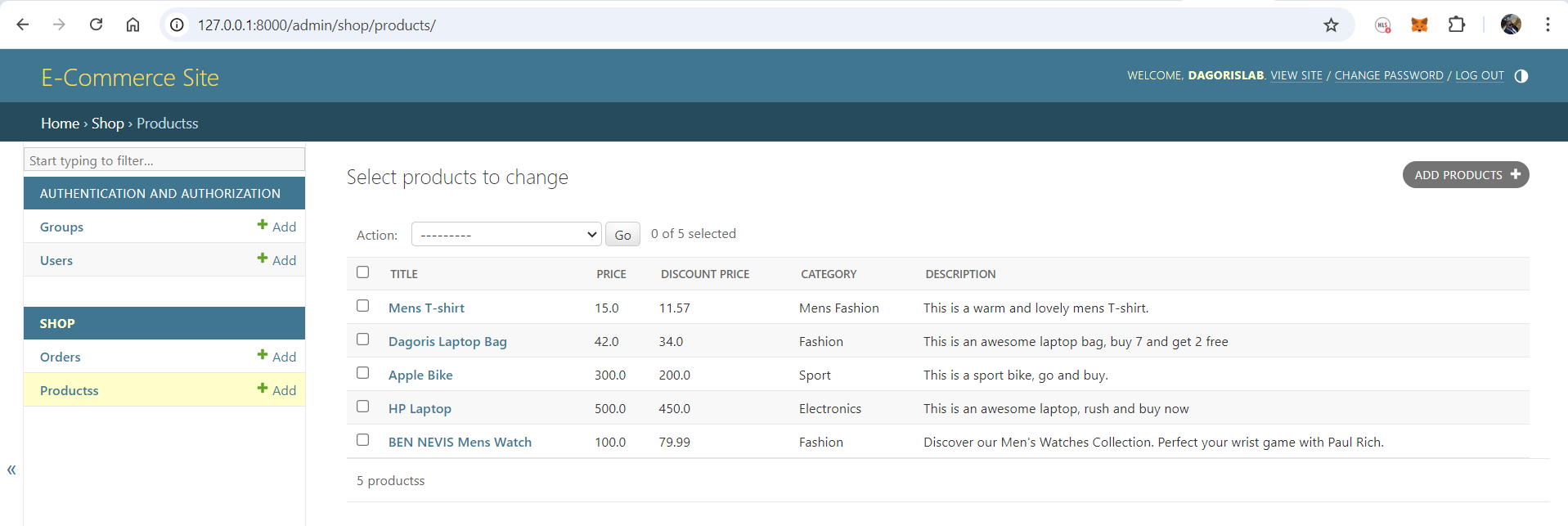


This is the output



You can remove a field, example removing the image field





**Adding a Search box on the Admin Portal**

To make the title field searchable on the admin column, add

***search\_fields = ('title',)***

Remember to leave the comma at the end. This displays a search box on the admin portal that can be used to search for product titles or names. You can add multiple items, but be sure to leave a comma by the end of the last field.



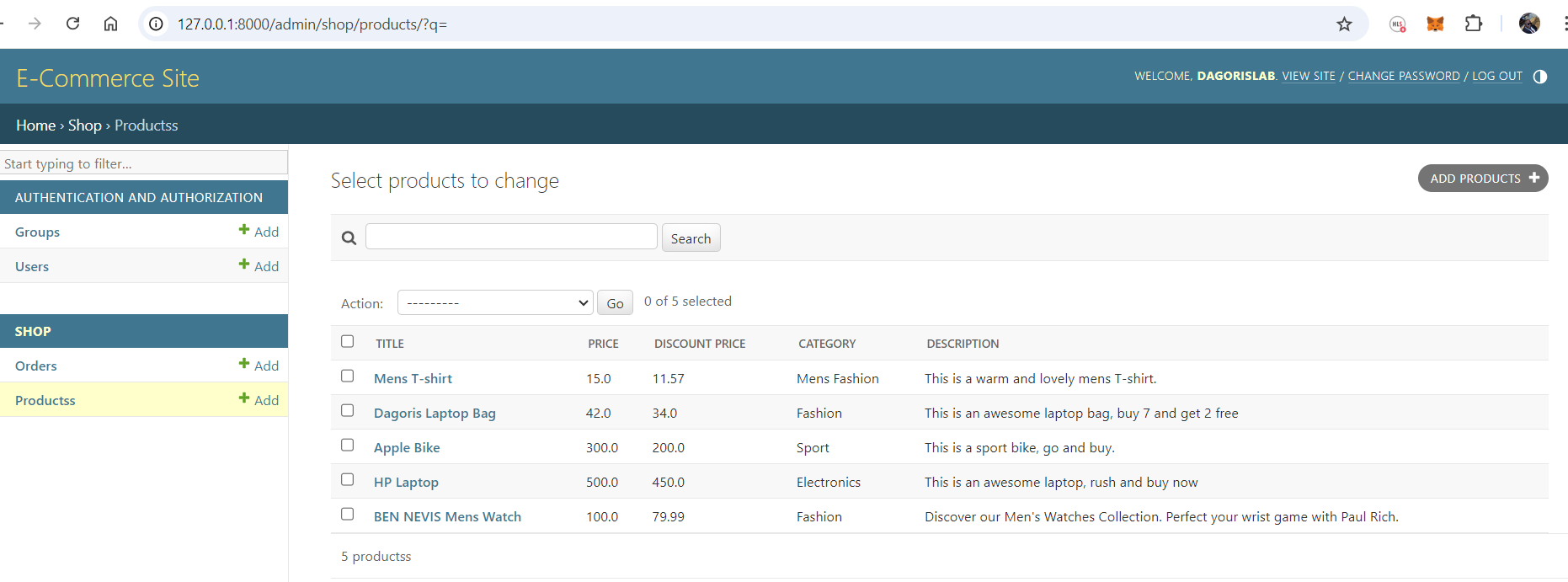


Figure: Admin portal with search box to search the title field

The search is modified for multi-field, now the title, category and description are all searchable fields.

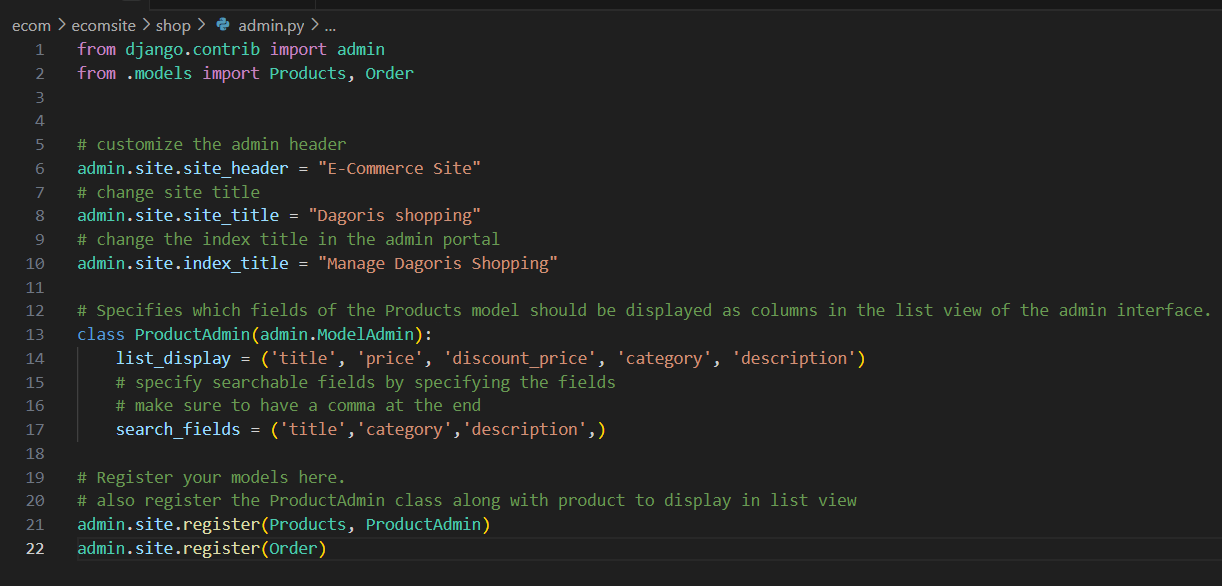
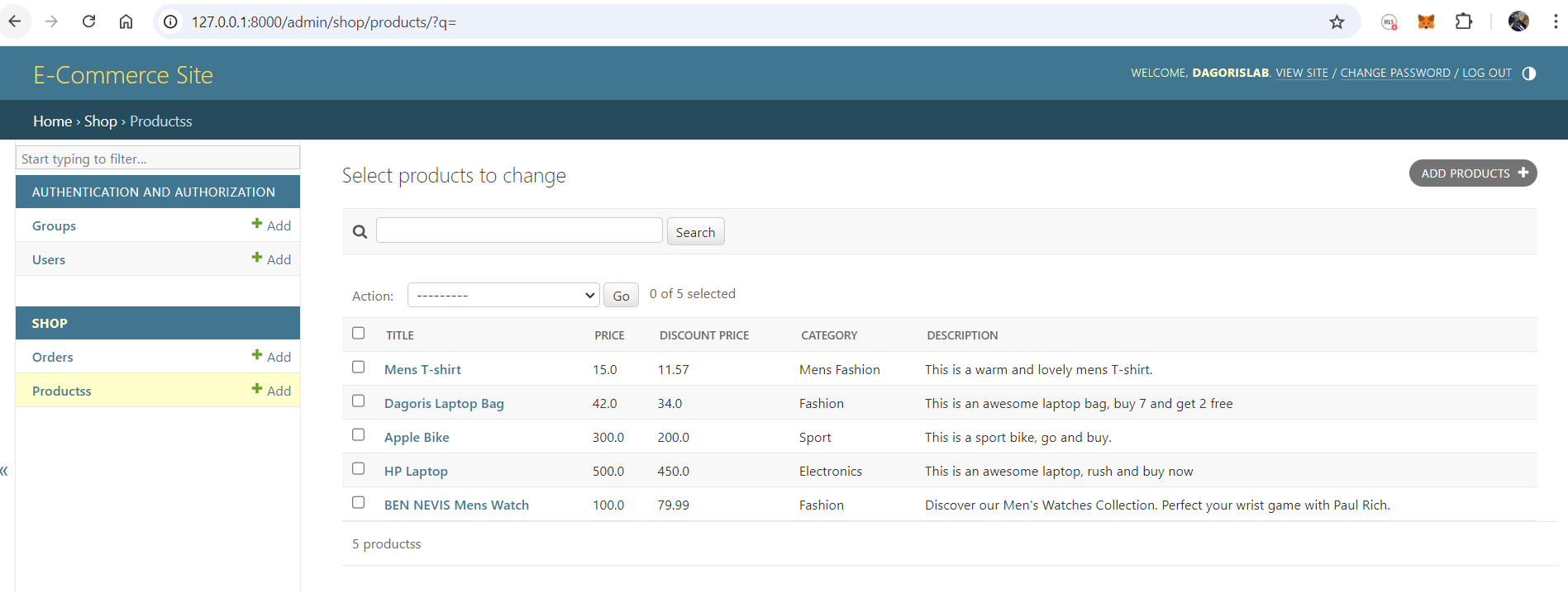


Figure: admin.py made the title, category and description a searchable field in the admin portal. The search can be done on one search field.



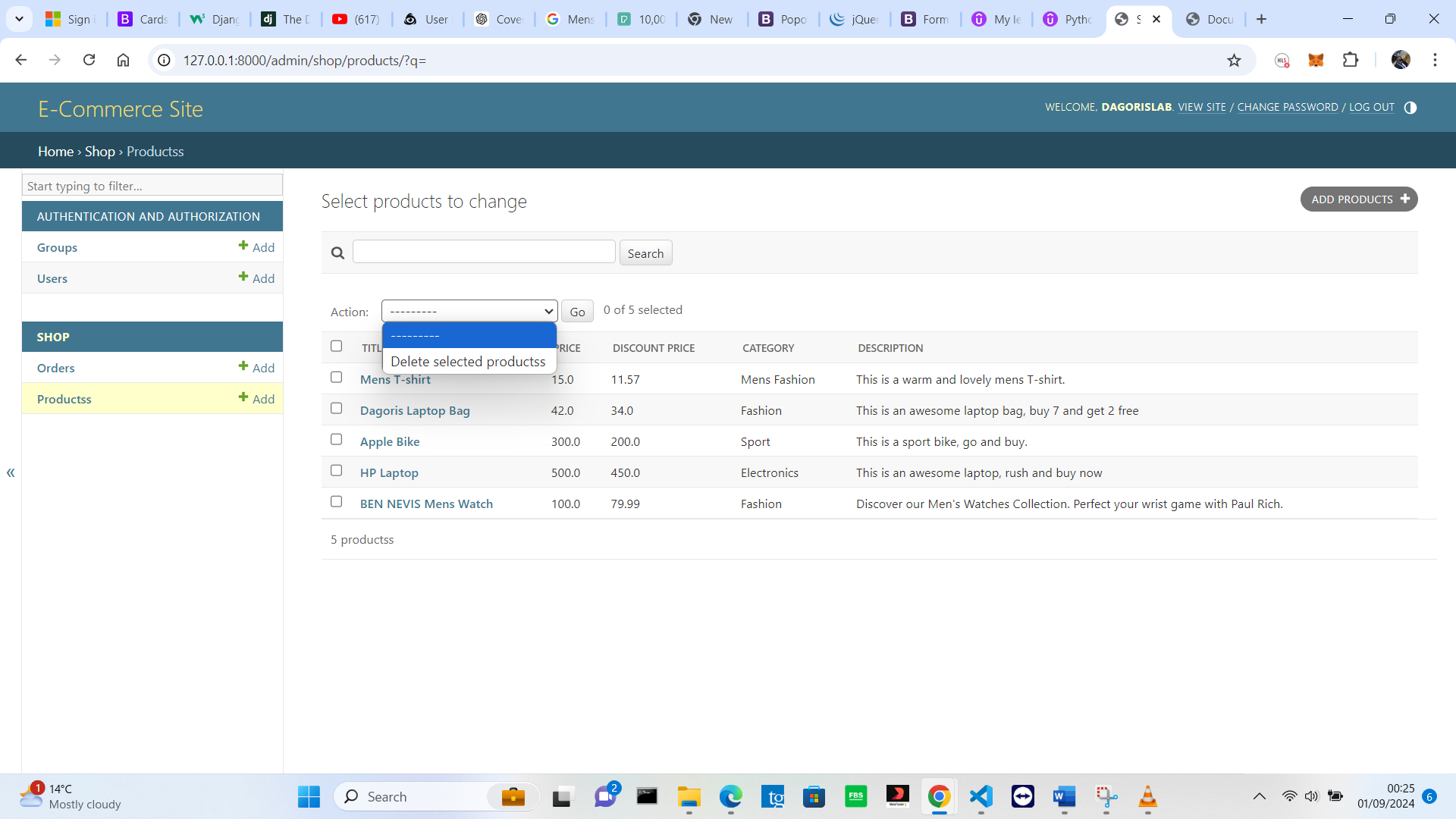
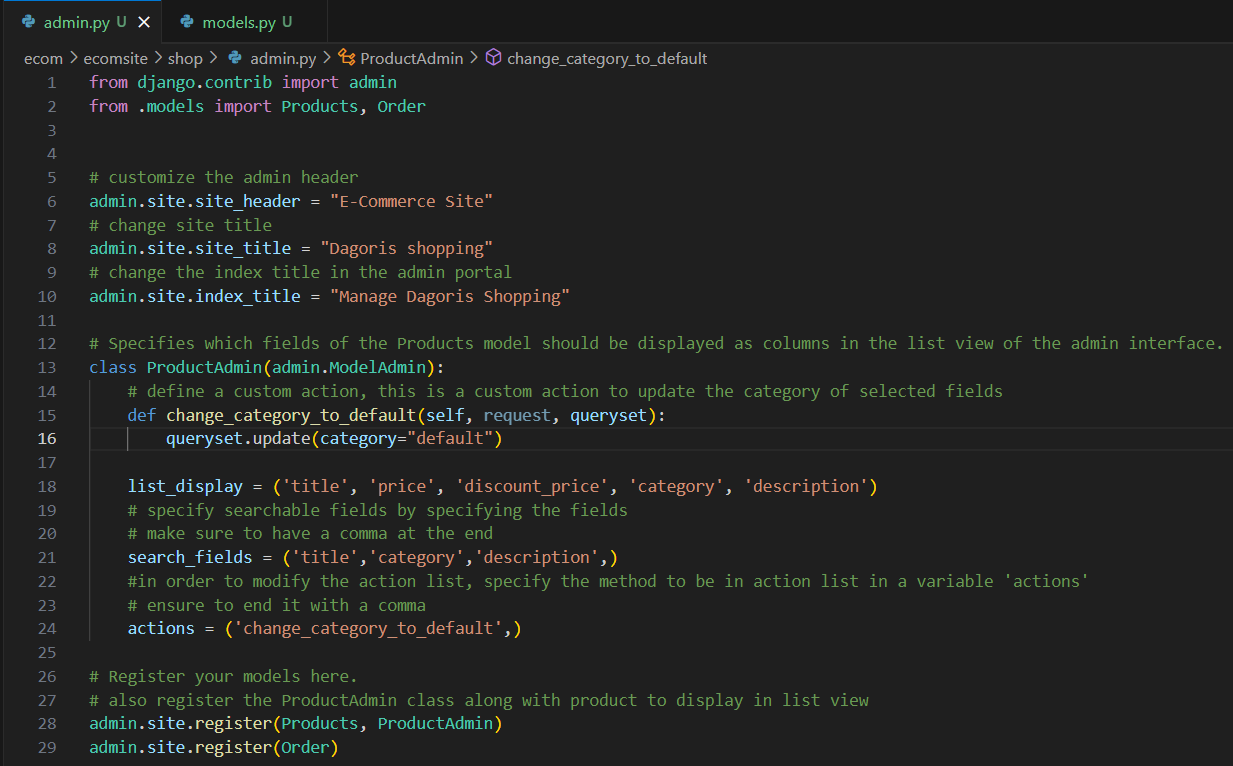


Figure: admin portal page showing the action field

**Setting a custom action field in the admin portal**



Admin.py showing a custom action field method created to change selected category to default

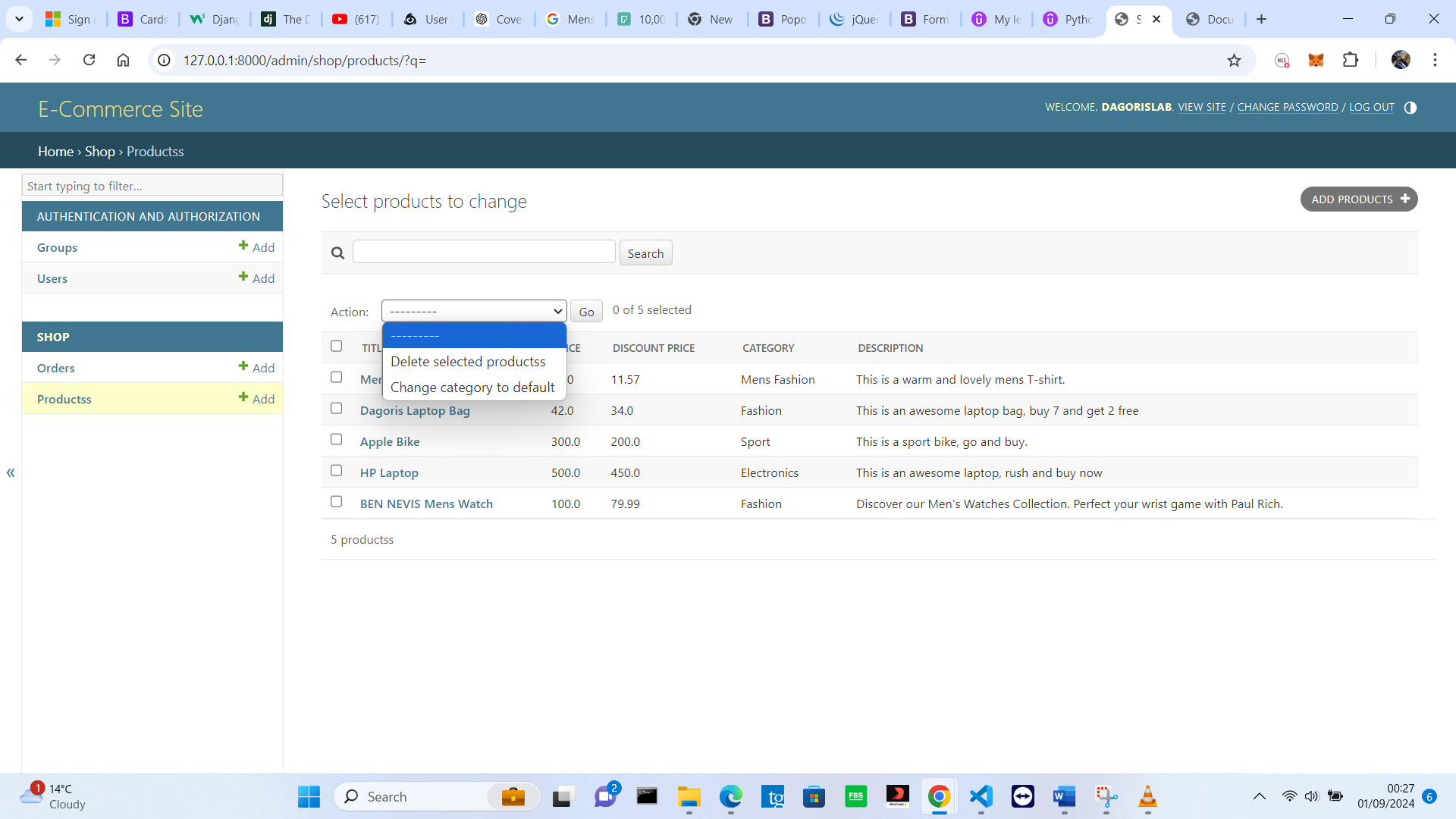


Figure: admin portal page showing a modified action field. The displayed action field method name can be modified

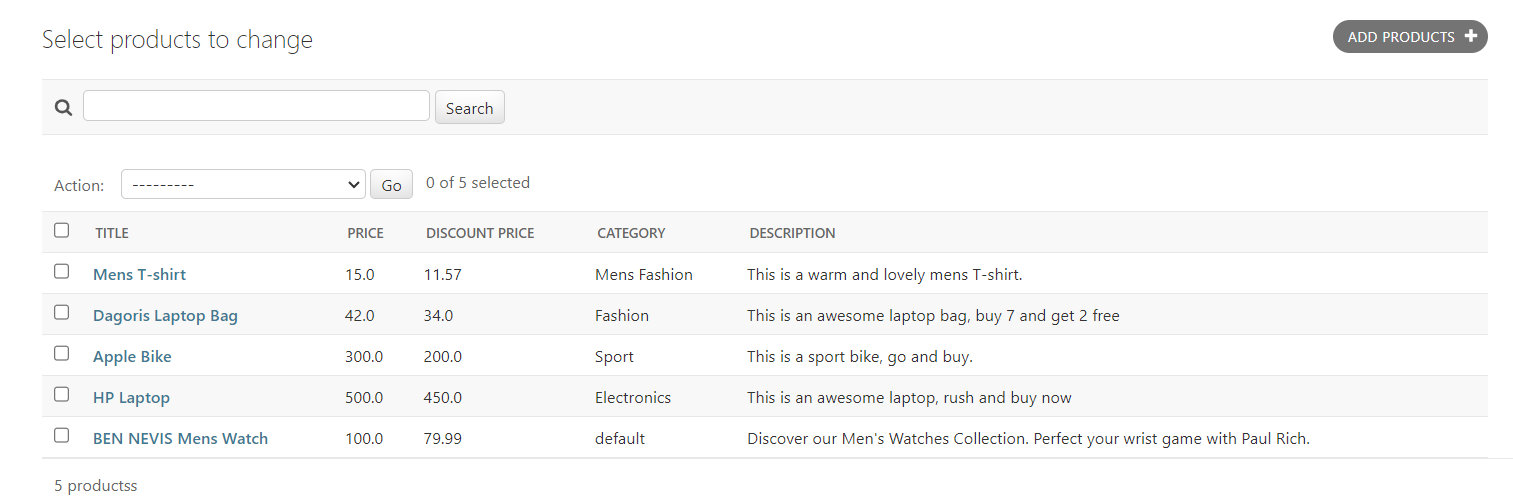
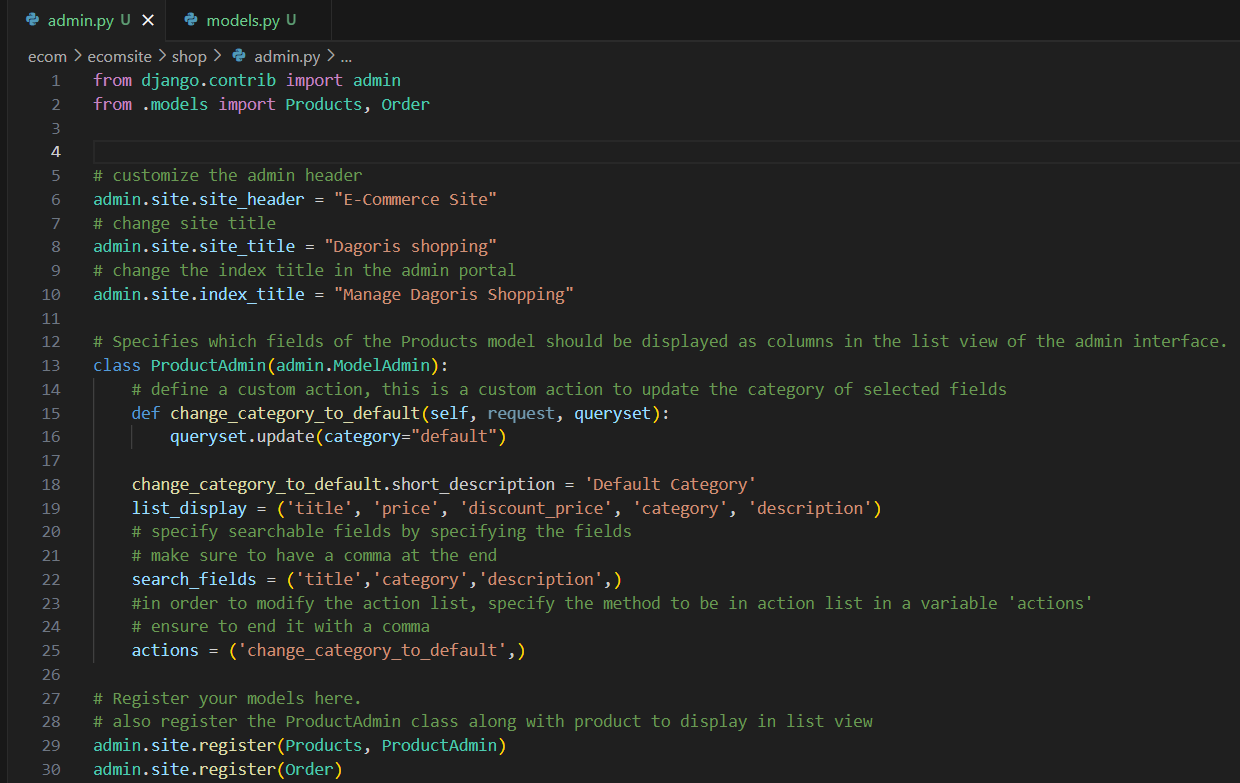
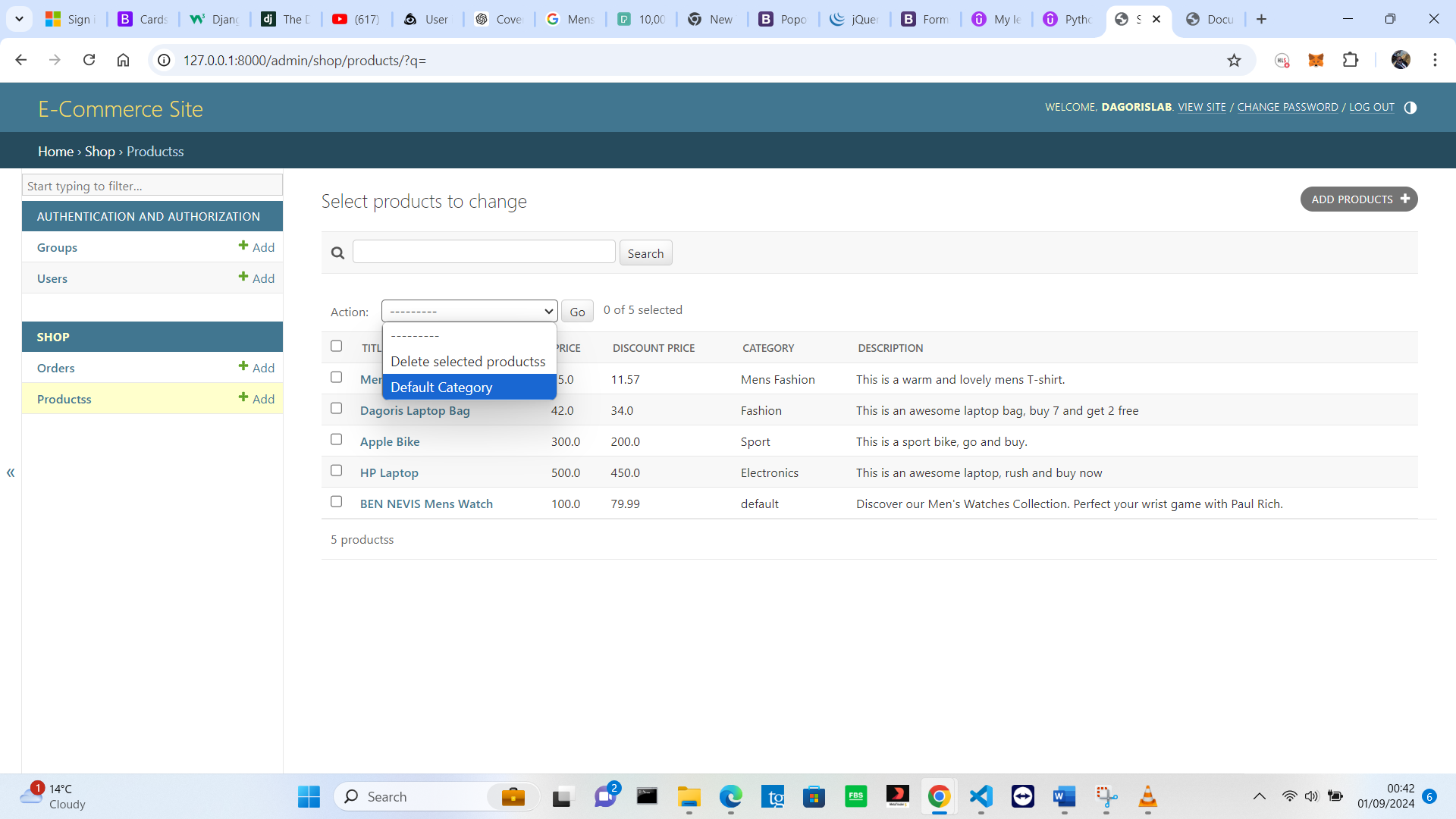


Figure: The admin portal shows a custom action method applied to the ‘BEN NEVIS Mens Watch’, and its category was changed to default.

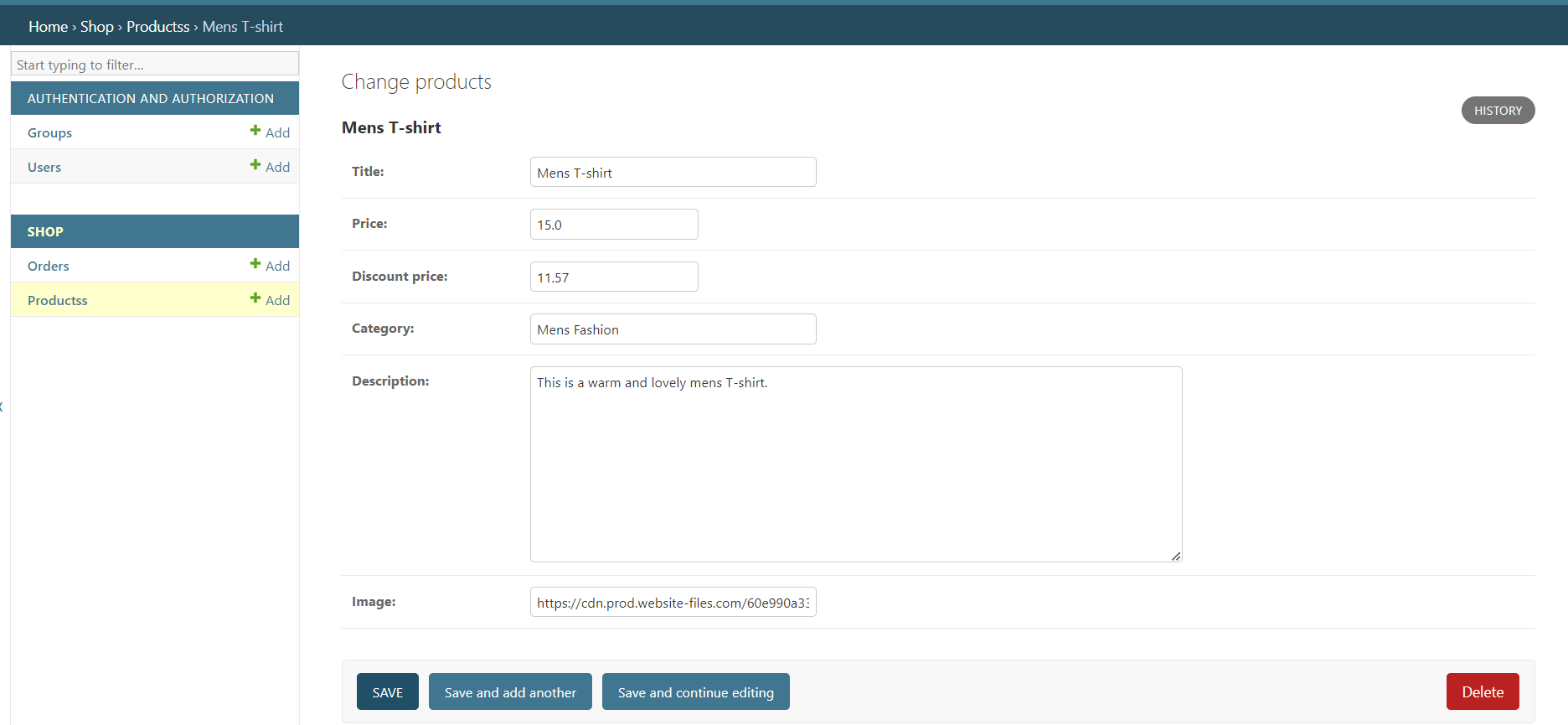
To change the display name of the custom action method, the cod below was added to the admin.py as shown below

***change\_category\_to\_default.short\_description = 'Default Category'***





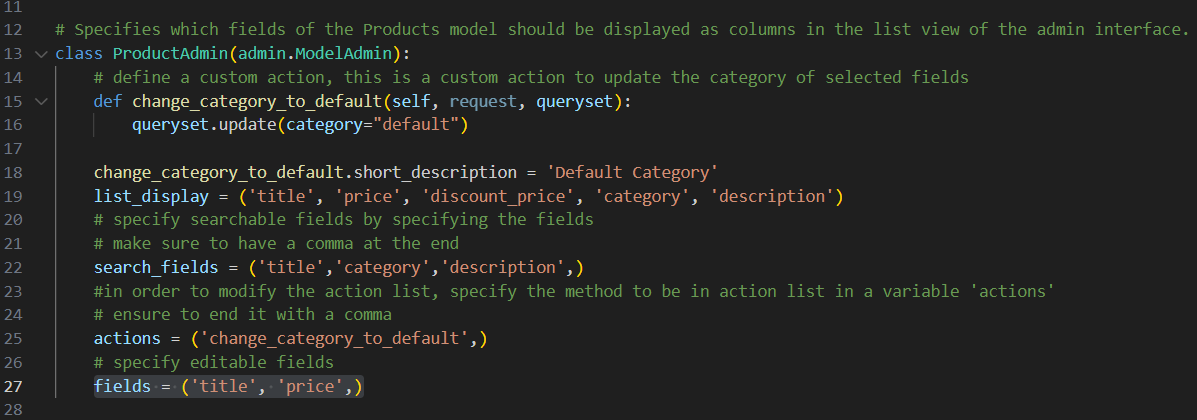
**To make Field Editable in the admin portal**



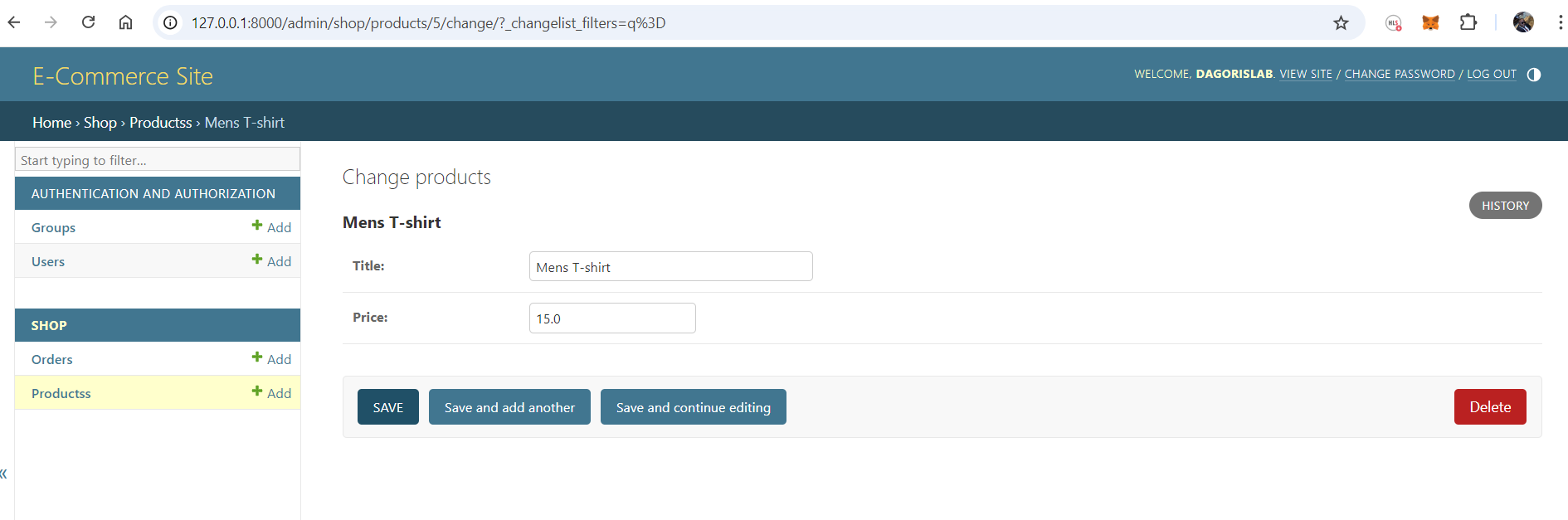
Admin portal showing normal view of the product update page in the admin portal. It is showing all the fields and they are all editable.

But to limit the number of editable fields and show only those field, modify the admin.py file with the code below. The variable must be called fields.

***fields = ('title', 'price',)***



Admin.py field, making the title and price the only editable field in the product update page in the admin portal. The result is shown below.



**Make Fields editable on the list page**

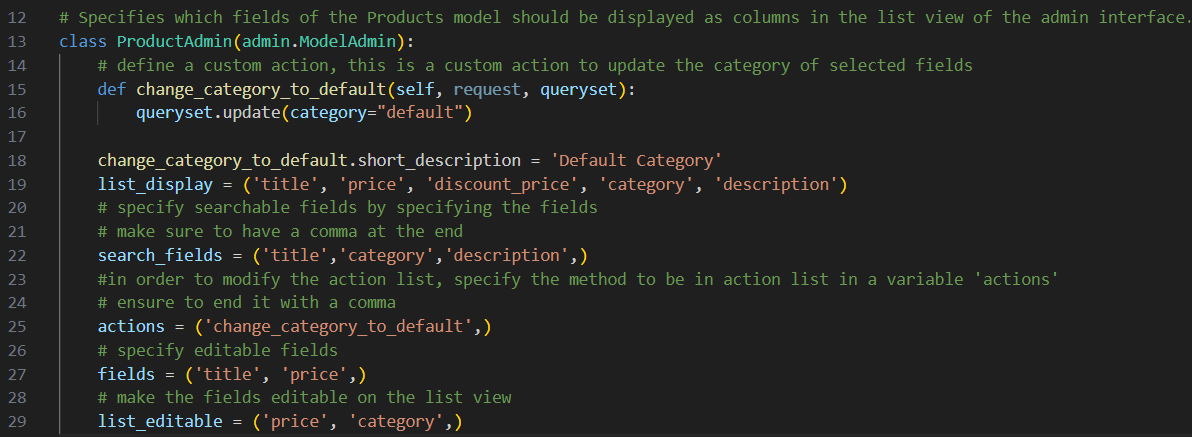


Figure: Admin.py making the fields editable on the list view in the admin portal. The above codes make the price and category column editable on the list view, and the output is shown below.

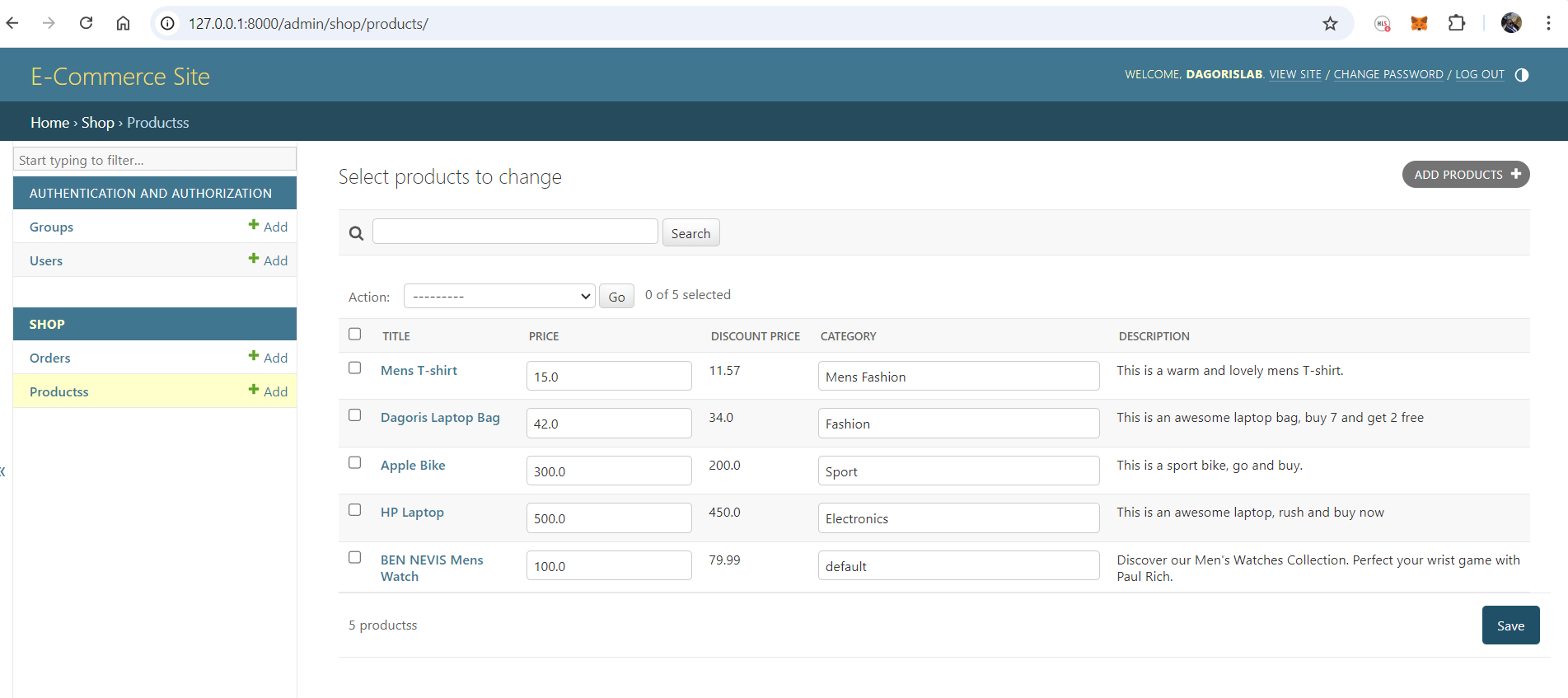


Figure: admin portal showing the price and category field as editable, and it also displays a save button