5/9/2024

Tanmaya Kumar Naik

Tan cannon robotics

Django

Shop app explanation

Contents

[*Adding media directory* 2](#_Toc172038181)

[Jinga templating (reusing templates) 2](#_Toc172038182)

[Adding loop to the carousal or sliders 3](#_Toc172038183)

[Adding cart button 5](#_Toc172038184)

[Creating the product page 6](#_Toc172038185)

[Making the contact page work 8](#_Toc172038186)

[Creating Add and Remove Buttons For Items In Cart 10](#_Toc172038187)

[Showing contents in the cart in navbar 13](#_Toc172038188)

[Adding Checkout and ClearCart Buttons in popover cart 14](#_Toc172038189)

[Creating the Checkout Page 15](#_Toc172038190)

[Creating Database Entry For Orders 19](#_Toc172038191)

[Creating Order Tracker For Customers 23](#_Toc172038192)

[Displaying order items on tracker page 27](#_Toc172038193)

[Merging Blog App Into The Shop UI | Python Django Tutorials In Hindi #55 30](#_Toc172038194)

[Creating Product Search Feature In Our E Commerce Website | Python Django Tutorials In Hindi #67 30](#_Toc172038195)

[Django Messages Framework: Showing Messages in Django blog | Python Django Tutorials In Hindi #79 32](#_Toc172038196)

[Django 3 Blog: Authentication & Authorization Tutorial | Python Django Tutorials In Hindi #85 34](#_Toc172038197)

[Creating Users in Django | Python Django Tutorials In Hindi #86 35](#_Toc172038198)

[Conclusion 39](#_Toc172038199)

*This is the 102 videos playlist of CodewithHarry.*

*Admin credentials*: Username: Tanmaya, Password: tancannon@2003

...after the 1st tutorial of Django extra documentation or my journal(what I have learned) is written here

The website requires an internet connection for the bootstrap to work.

# *Adding media directory*

*When any image is added from the admin panel it goes to a separate directory named "media"; i.e. we are saving all the images in a dedicated place within a folder inside named as the* app name (e.g media/shop/images/dustbin.jpg).

-In the "settings.py" of the main project directory(here "mac") add;

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

MEDIA\_URL = '/media/'

-Then import these in the main project directory in urls.py(here "mac");

from django.conf import settings

from django.conf.urls.static import static

and add the url as below(the thing starting with + im talking about);

urlpatterns = [

path('admin/', admin.site.urls),

path('shop/', include('shop.urls')),

path('blog/', include('blog.urls'))

] + static(settings.MEDIA\_URL,document\_root=settings.MEDIA\_ROOT)

# Jinga templating (reusing templates)

to copy the code of a html file to another(or in otherwors to resuse the code) we

use this jinga templating.

say in 'about.html'

{%extends 'shop/base.html' %} #this means we're reusing the contents of base.html in about.html.

also we can add variable blocks in the parent file (here base.html)

e.g;

written in base file (here base.html):

{% block title%}{% endblock %}

written in the reusing file (e.g about.html)

{% block title%} Tancannon Industries {% endblock %}

This replaces the 'Tancannon Industries' at {% block title%}{% endblock %} when reusing and can only be seen in the reusing file i.e about.html.

# Adding loop to the carousal or sliders

Here im fetching the products from the database, so i need to iterate over the product objects fetched from the database.

{% for i in range %}

<li data-target="#demo" data-slide-to="{{i}}" ></li>

{% endfor %}

This is used to to make slides whose numbers is equal to "range".

See in shop/views.py:

products = Products.objects.all() //fetching the objects from the databases.

print(products)

n = len(products)

nSlides = n//4 + ceil((n/4)-(n//4)) //formula to get no. of slides(in carousal) needed to show images.

params = {

'no\_of\_slides': nSlides,

'range': range(nSlides),

'product': products

}

Then;

<div class="container carousel-inner no-padding">

<div class="carousel-item active">

<div class="col-xs-3 col-sm-3 col-md-3">

<div class="card" style="width: 18rem;">

<!--<img src='{% static "shop/test.jpg" %}' class="card-img-top" alt="..."> -->

<img src='/media/{{product.0.image}}' class="card-img-top" alt="...">

<div class="card-body">

<h5 class="card-title">{{product.0.product\_name}}</h5>

<p class="card-text">{{product.0.desc}}</p>

<a href="#" class="btn btn-primary">Add to Cart</a>

</div>

</div>

</div>

* *The 1st product is hardcoded.*

The rest images are loaded into the slides like this using loop;

{% for i in product|slice:"1:"%}

<div class="col-xs-3 col-sm-3 col-md-3">

<div class="card" style="width: 18rem;">

<img src='/media/{{i.image}}' class="card-img-top" alt="...">

<div class="card-body">

<h5 class="card-title">{{i.product\_name}}</h5>

<p class="card-text">{{i.desc}}</p>

<a href="#" class="btn btn-primary">Add To Cart</a>

</div>

</div>

</div>

{% if forloop.counter|divisibleby:3 and forloop.counter > 0 and not forloop.last %}

</div>

<div class="carousel-item">

{% endif %}

{% endfor %}

</div>

# Adding cart button

Here we’ll be able to see the number of different categories of items we need to add to the cart.

In “shop/basic.html”, we have added a “popover button”

<!--manually added popover bottom -->

              <button type="button" class="btn btn-secondary mx-2" id="popcart" data-container="body" data-toggle="popover" data-placement="bottom" data-html="true" data-content="Vivamus

              sagittis lacus vel augue laoreet rutrum faucibus.">

                Cart(<span id="cart">0</span>)

              </button>

“gets popups” in the bottom of the cart button when is pressed.

In “index.html” in the script tag;

if(localStorage.getItem('cart')==null){ //we're saving the cart value in localstoarge of the website.

  var cart = {};

}

else{

  cart = JSON.parse(localStorage.getItem('cart'));

  document.getElementById('cart').innerHTML = Object.keys(cart).length;

}

document.getElementById('cart').innerHTML = Object.keys(cart).length; // This is used to update the element the text or number beside the cart in the cart button text on the nav bar.



The popover is used as below within the script tag in “shop/index.html”.

$('#popcart').popover();

document.getElementById("popcart").setAttribute('data-content','<h5>Cart for your shopping list</h5>');

# Creating the product page

<a href="productview/{{i.id}}"<button id="pr{{i.id}}" class="btn btn-primary cart">Quick view</button></a>

We create a “Quick view” button. It is linked to "productview/{{i.id}}".

This is link is handled in urls.py as (in urlpatterns):

path("productview/<int:myid>",views.productView, name="productView"),

Then as usual it goes to views.py:

def productView(requests,myid):

    product = Products.objects.filter(id=myid)

    print(product)

    params = {'product':product[0]}

    return render(requests,'shop/prodView.html',params) #product is a list of one element

    # return HttpResponse('productview page')

The “i.id” i.e. “product\_id” is send through the link from index.html -> urls.py -> views.py

So, we fetch the product information using the id and pass it to show in the prodView.html for a specific product.

In prodView.html;

{% extends 'shop/basic.html' %}

{% block title%} {{product.product\_name}} - My Awesome Cart{% endblock %}

{% block body %}

<div class="container my-4">

    <div class="row">

    <div class="col-md-4">

    <div class="row">

        <img src="/media/{{product.image}}" width="233px" height="385px">

    </div>

        <div class="row my-2">

            <button class="btn btn-primary mx-3">Buy Now</button>

            <button class="btn btn-primary">Add To Cart</button>

        </div>

    </div>

    <div class="col-md-8">

        <h5>{{product.product\_name}}</h5>

        <p><b>Rs.{{product.price}} </b></p>

        <p>{{product.desc}}</p>

    </div>

    </div>

</div>

{% endblock %}

{% block js %}

<script>

console.log('working');

if(localStorage.getItem('cart') == null){

var cart = {};

}

else

{

cart = JSON.parse(localStorage.getItem('cart'));

document.getElementById('cart').innerHTML = Object.keys(cart).length;

}

$('.cart').click(function(){

console.log('clicked');

var idstr = this.id.toString();

console.log(idstr);

if (cart[idstr] !=undefined){

cart[idstr] = cart[idstr] + 1;

}

else

{

cart[idstr] = 1;

}

console.log(cart);

localStorage.setItem('cart', JSON.stringify(cart));

document.getElementById('cart').innerHTML = Object.keys(cart).length;

});

$('#popcart').popover();

document.getElementById("popcart").setAttribute('data-content', '<h5>Cart for your items in my shopping cart</h5>');

</script>

{% endblock %}

Above the script tag is the showing of the product necessary attributes.

And in the script tag we have just copy pasted the one in index.html, so as to update the cart button on the navbar here also.

# Making the contact page work

shop/contact.html:

{% extends 'shop/basic.html' %}

{% block title %} Contact Us {% endblock %}

{% block body %}

    <div class="container">

        <h3>Contact Us</h3>

        <form action="/shop/contact/" method="post">

            {% csrf\_token %}

            <div class="form-group">

                <label for="name">Name</label>

                <input type="text" class="form-control" id="name" name="name" placeholder="Enter Your Name">

            </div>

            <div class="form-group">

                <label for="name">Email</label>

                <input type="email" class="form-control" id="email" name="email" placeholder="Enter Your Email">

            </div>

            <div class="form-group">

                <label for="name">Phone</label>

                <input type="tel" class="form-control" id="phone" name="phone" placeholder="Enter Your Phone Number">

            </div>

            {% comment %} <div class="form-group"> {% endcomment %}

            <div class="form-group">

                <label for="desc">How May We Help You ?</label>

                <textarea class="form-control" id="desc" name="desc" rows="3"></textarea>

            </div>

            <button type="submit" class="btn btn-success">Sumbit</button>

        </form>

{% endblock %}

Explanation:

<form action="/shop/contact/" method="post">

“action” specifies where to send the form-data when a form is submitted. This attribute overrides the form's action attribute. The form action attribute is only used for inputs/buttons with type="submit".

And set method=”post” i.e to post the form.

The keyword “name” is used to fetch it when the form is posted.

In “views.py” we are fetching the posted data using the values of keyword “name” in input fields.

In views.py:

from .models import Contact #we import the class Contact to use it

def contact(requests):

    # return HttpResponse('contact page')

    if (requests.method=="POST"):

        print(requests)

#using the values of keyword “name” in “contact.html” to store data.

        name=requests.POST.get("name")

        email=requests.POST.get("email")

        phone=requests.POST.get("phone")

        desc=requests.POST.get("desc")

        print(name,email,phone,desc)

#created a instance of class Contact named fetched\_contact

        fetched\_contact = Contact(name=name, email=email, phone=phone, desc=desc)

        fetched\_contact.save() #to save the fetched data in the database.

    return render(requests,'shop/contact.html')

Explanation:

Then to save it on the database we use a class made by us in “models.py” i.e “Class Contact”.

See the above use of instance (a variable named “fetched\_contacts”).

In models.py:

class Contact(models.Model):

    msg\_id = models.AutoField(primary\_key=True)

    name = models.CharField(max\_length=50)

    email = models.CharField(max\_length=70,default="")

    phone = models.CharField(max\_length=70, default="")

    desc = models.CharField(max\_length=500, default="")

    def \_\_str\_\_(self):

        return self.name

Explanation:

This is the class we have created in “shop/models.py”.

The model or class contact need to be registered in “shop/admin.py” before using as below.

In admin.py:

from .models import Products, Contact

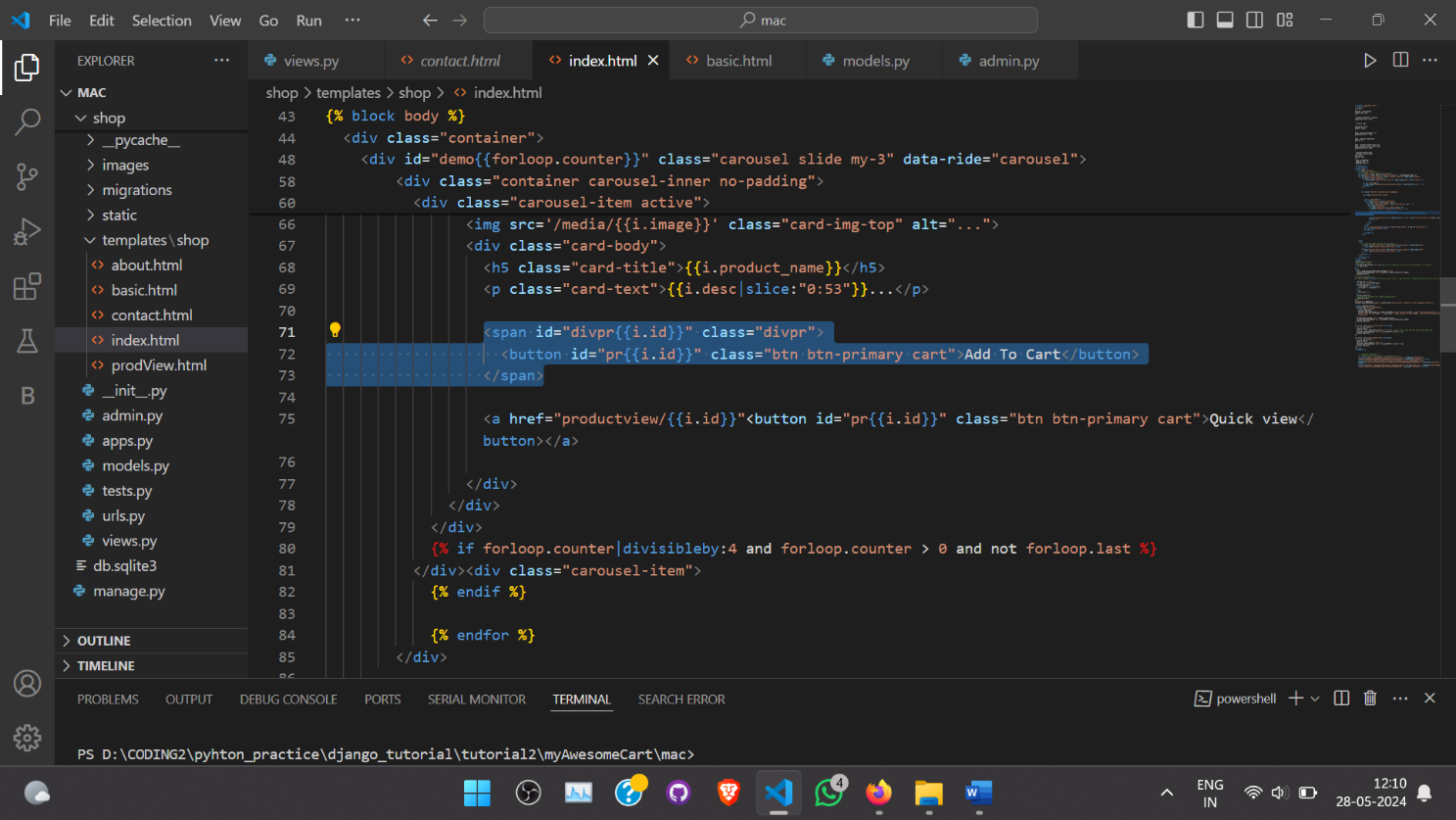
admin.site.register(Products)

admin.site.register(Contact) #this is the registration of class Contact.

# Creating Add and Remove Buttons For Items In Cart

In this tutorial, we will create Add (+) and Remove (-) buttons for the cart of our E-commerce website. First of all, we need to make some changes to the index.html file. So, open the file and make the following changes:

1. Notice the highlighted area of the code in the image given below; we had included a <span> tag before the buttons that we created in one of our previous tutorials. In the future, we will be using this <span> element to target the buttons.



After doing this type the below code in the javascript section of the index.html file:

function updateCart(cart){

  console.log("inside updateCart");

  for (var item in cart){

    console.log(item);

    document.getElementById('div'+item).innerHTML="<button id='minus"+item+"' class='btn btn-primary minus'>-</button> <span id='val"+item+"'> "+cart[item]+" </span> <button id='plus"+item+"' class='btn btn-primary plus'>+</button>";

  }

  localStorage.setItem('cart', JSON.stringify(cart));

  document.getElementById('cart').innerHTML = Object.keys(cart).length;

  console.log(cart);

}

In the above code, we have created a function called updateCart(), which performs the function of showing (+) and (-) button after clicking the Add To Cart button. We are using a for loop to iterate over the items of the cart. After the creation of updateCart function, we need to call this function whenever the user clicks on the "Add To Cart" button. So, type the below code inside the click function that we created earlier.

$('.cart').click(function(){ //setting a event listner to tags (here we're targetting buttons) having class cart

  console.log('clicked');

  var idstr = this.id.toString();

  console.log(idstr);

  if (cart[idstr] != undefined){

    cart[idstr] = cart[idstr] + 1;

  }

  else{

    cart[idstr] = 1;

  }

  console.log(cart);

  //localStorage.setItem('cart',JSON.stringify(cart));

  updateCart(cart);

});

Alright! We are all set. Restart the development server and click on the "Add To Cart" button. You will notice that on click the button, (+) and (-) buttons are displayed. Now, click on (+) and (-) buttons, you will notice that nothing happens. This is because we have not written any code for incrementing and decrementing the number of products. So, type the below code in the index.html file :

$('.divpr').on("click","button.minus",function(){

  a = this.id.slice(7,);

  console.log(a);

  cart['pr'+ a] = Math.max(0,(cart['pr'+ a] -1)); //cause i don't want the item count to go below zero.

  document.getElementById('valpr'+ a).innerHTML = cart['pr'+ a];

  console.log(cart);

});

$('.divpr').on("click","button.plus",function(){

  a = this.id.slice(6,);

  console.log(a);

  cart['pr'+ a] = cart['pr'+ a] + 1;

  document.getElementById('valpr'+ a).innerHTML = cart['pr'+ a];

  console.log(cart);

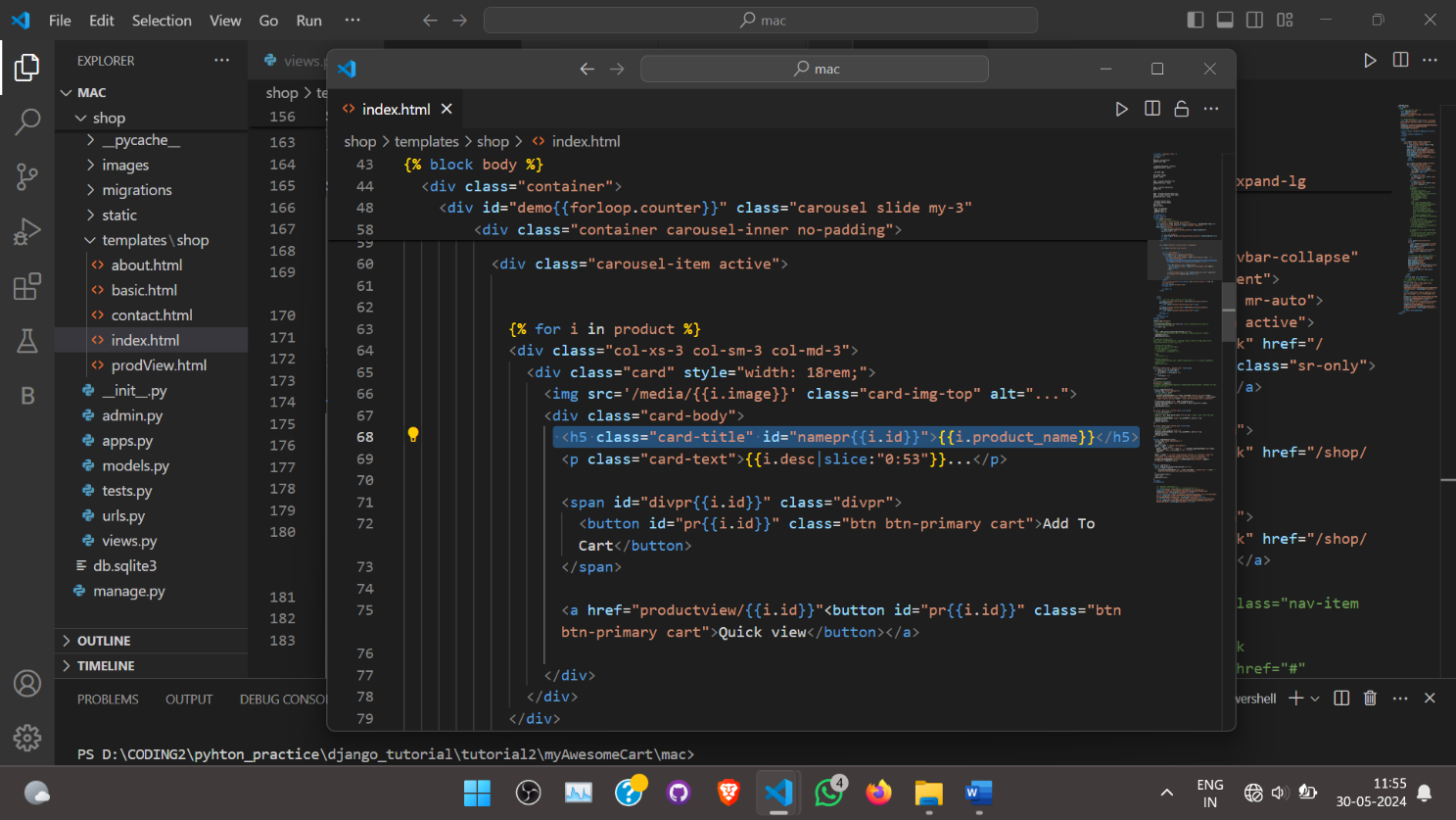
});

In the above code, we've used jquery to select the <span> element with div.pr class. Then, we've used the slice operation to get the id of the elements and in the end, we are changing the inner HTML. Reload the server, and if you have followed the steps correctly, then the (+) and (-) buttons will work correctly. So that's how we can easily create an add and remove button for the cart of our E-commerce website.

# Showing contents in the cart in navbar

In “shop/basis.html”:

We have added a id = ”namepr{{i.id}}” to the highlighted line, so that it can be accessed and shown in the cart contents in the navbar popover.



In “shop/index.html”:

We have written this to access the select and show the title of the products who id is put in the “cart” in the popover in the navbar.

function updatePopover(cart){

  console.log("inside updatePopover");

//we’re making the html into string to render it in the popover

  var popStr = "";

  var count = 1;

  popStr = popStr + "<b>Your Cart</b><br>";

  for (var item in cart){

    popStr = popStr + "<h>"+ count + ". " + document.getElementById('name'+item).innerHTML+ " Qty:"+ cart[item] +"</h>" + "<br>";

    count++;

  }

  popStr = popStr + "<a href='/shop/checkout/'<button id ='checkout' class='btn btn-primary my-1'>Checkout</button></a><button onclick='clearCart()' id ='clearCart' class='btn btn-primary my-1 mx-1'>Clear Cart</button>";

  document.getElementById('popcart').setAttribute('data-content', popStr);

  $('#popcart').popover('show'); }

Explanation:

var popStr = ""; // we’re making the html into string to render it in the popover.

Here we are sending it :

document.getElementById('popcart').setAttribute('data-content', popStr);

$('#popcart').popover('show'); //This just opens the popover automatically.

document.getElementById('name'+item).innerHTML+ : this thing is used to access the heading of those item whose id is in the “cart” variable that we have made.

“item” is ‘pr1’ or ‘pr2’ ..etc. => namepr1 (i.e the id of the heading tag holding the title of product id 1 in the database. In “shop/index.html” id = ”namepr{{i.id}}” that we have written above is used for accessing the content.

# Adding Checkout and ClearCart Buttons in popover cart

This is we have added at last the Checkout and ClearCart Buttons in the

“updatePopover(cart)”.

popStr = popStr + "<a href='/shop/checkout/'<button id ='checkout' class='btn btn-primary my-1'>Checkout</button></a><button onclick='clearCart()' id ='clearCart' class='btn btn-primary my-1 mx-1'>Clear Cart</button>";

The function declaration of ClearCart() is as follows, Checkout button

direct to checkout page shall be made functional later.

function clearCart() {

//get the cart from the localstorage of the web braowser.

  cart = JSON.parse(localStorage.getItem('cart'));

//dynamically take the “Add to cart” button to its original form (i.e when it was not added to the “cart”)

  for (var item in cart) {

      document.getElementById('div' + item).innerHTML = '<button id="' + item + '" class="btn btn-primary cart">Add To Cart</button>'

  }

//Clear the localstorage of the browser.

  localStorage.clear();

//make cart variable empty again.

  cart = {};

  updateCart(cart);

}

# Creating the Checkout Page

In “shop/checkout.html”:

{% extends 'shop/basic.html' %}

{% block title%}Checkout -- My Awesome Cart{% endblock %}

{% block body %}

<div class="container">

    <div class="col my-4">

        <h2> Step 1-  My Awesome Cart Express Checkout - Review Your Cart items </h2>

        <ul class="list-group" id="items">

        </ul>

    </div>

    <div class="col my-4" >

        <h2> Step 2 - Enter Address & Other Details</h2>

            <form id="items" method="post" action="/shop/checkout/">{% csrf\_token %}

                <input type="hidden" name="itemsJson" id="itemsJson">

                <div class="form-row">

                    <div class="form-group col-md-6">

                        <label for="name">Name</label>

                        <input type="name" class="form-control" id="name" name="name" placeholder="Name">

                    </div>

                    <div class="form-group col-md-6">

                        <label for="email">Email</label>

                        <input type="email" class="form-control" id="email" name="email" placeholder="Email">

                    </div>

                </div>

                <div class="form-group">

                    <label for="inputAddress">Address</label>

                    <input type="text" class="form-control" id="inputAddress" name="address1" placeholder="1234 Main St">

                </div>

                <div class="form-group">

                    <label for="inputAddress2">Address Line 2</label>

                    <input type="text" class="form-control" id="inputAddress2" name="address2"  placeholder="Apartment, studio, or floor">

                </div>

                <div class="form-row">

                    <div class="form-group col-md-6">

                        <label for="inputCity">City</label>

                        <input type="text" class="form-control" id="inputCity" name="city">

                    </div>

                    <div class="form-group col-md-4">

                        <label for="inputState">State</label>

                        <input type="text" class="form-control" id="inputAddress" name="state" placeholder="Enter State">

                        </select>

                    </div>

                    <div class="form-group col-md-2">

                        <label for="inputZip">Zip</label>

                        <input type="text" class="form-control" id="inputZip" name="zip\_code">

                    </div>

                </div>

                <div class="form-group">

                    <label for="inputZip">Phone Number</label>

                    <input type="tel" class="form-control" id="inputPhone" name="phone">

                </div>

                <button type="submit" class="btn btn-primary">Place Order</button>

            </form>

    </div>

</div>

{% endblock %}

The above part of the code is the html page that we want to show on the “checkout page”.

The below is the required javascript required to fetch and display necessary data (cart).

{% block js %}

<script>

    //getting the "cart" values

    if (localStorage.getItem('cart') == null) {

        var cart = {};

    } else {

        cart = JSON.parse(localStorage.getItem('cart'));

    }

    console.log(cart);

    //using the values with the "cart".

    var sum = 0; //it keep the count of quantity of items in cart

    if ($.isEmptyObject(cart)) {

        //if object is empty

        mystr = `<p>Your cart is empty, please add some items to your cart before checking out!</p>`

        $('#items').append(mystr);

    } else {

        for (item in cart) {

            let name = cart[item][1];

            let qty = cart[item][0];

            sum = sum + qty;

//ES6 notation to use variable within html

            mystr = `<li class="list-group-item d-flex justify-content-between align-items-center">

                        ${name}

                        <span class="badge badge-primary badge-pill">${qty}</span>

                    </li>`

            $('#items').append(mystr); //we can append one by one html lines to tag id "items" like this

        }

    }

    //updating the cart items count on the nav bar

    document.getElementById('cart').innerHTML = sum;

    //setting the tag of id "itemsJson" to "cart" value

    $('#itemsJson').val(JSON.stringify(cart));

    //alert notification that the order is placed when the place order button is pressed, "thanks" becomes True in "shop/views.py"

    {% if thank %}

        alert('Thanks for ordering with us. Your order ID is {{id}}. Use it to track your order using our order tracker');

        //after placing the order just empty the localStorage

        localStorage.clear();

        document.location = "/shop";

{% endif %}

</script>

{% endblock %}

Illustration:

$('#items').append(mystr); //we can append one by one html lines to tag id "items" like this

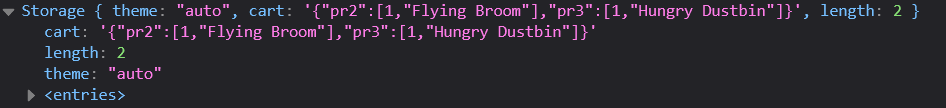


        for (item in cart) {

            let name = cart[item][1];

            let qty = cart[item][0];

The “cart” was changed to something like this and is used like this everywhere whenever “cart” is used in other part of the website too.



# Creating Database Entry For Orders

In “shop/models.py” add this:

class Orders(models.Model):

    readonly\_fields = ('order\_id',)

    order\_id = models.AutoField(primary\_key=True)

    items\_json = models.CharField(max\_length=5000)

    name = models.CharField(max\_length=111)

    email = models.CharField(max\_length=111)

    address = models.CharField(max\_length=111)

    city = models.CharField(max\_length=111)

    state = models.CharField(max\_length=111)

    zip\_code = models.CharField(max\_length=111)

    phone = models.CharField(max\_length=111)

    def \_\_str\_\_(self):

        return self.name

After this, we need to register this model, so open the admin.py file and type the below code:

from django.contrib import admin

from .models import Orders

admin.site.register(Orders)

After this, we need to store the above changes in the database, so migrate the changes by typing the below commands one by one :

python manage.py makemigrations

python manage.py migrate

Now, we need to handle the orders in the views.py file. So, open the views.py file and type the below code:

def checkout(requests):

    # return HttpResponse('checkout page')

    if (requests.method=="POST"):

        # print(requests)

        items\_json=requests.POST.get("itemsJson",'')

        name=requests.POST.get("name",'')

        email=requests.POST.get("email",'')

        address=requests.POST.get("address1",'') + " " + requests.POST.get("address2",'')

        city=requests.POST.get("city",'')

        state=requests.POST.get("state",'')

        zip\_code=requests.POST.get("zip\_code",'')

        phone=requests.POST.get("phone")

        # print(items\_json,name,email,address,city,state,zip\_code,phone)

        order = Orders(items\_json=items\_json,name=name,email=email,address=address,city=city,state=state,zip\_code=zip\_code,phone=phone)

        order.save()

        #"thank is used to check if order was placed, it's made true to notify the js on the checkout.html page to show order placed notification"

        thank = True

        id = order.order\_id

        #This to save order tracking update in the database

        update = OrderUpdate(order\_id = id, update\_desc = "The order has been placed")

        update.save()

        return render(requests, 'shop/checkout.html', {'thank':thank, 'id': id})

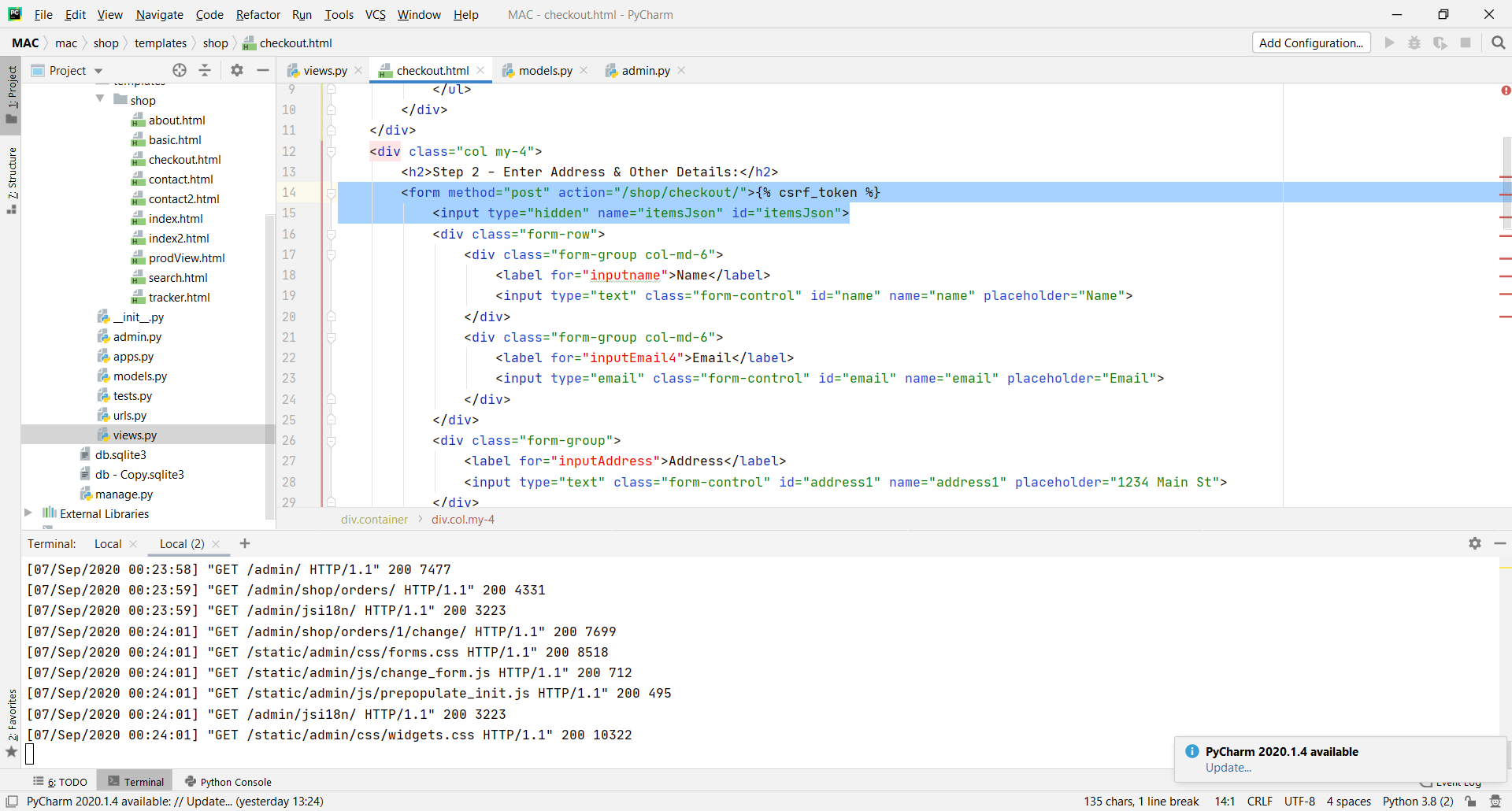
    return render(requests, 'shop/checkout.html')

After this, we need to define the HTTP method and CSRF token in the form of the checkout.html file. So, open the checkout.html file and type the below code:

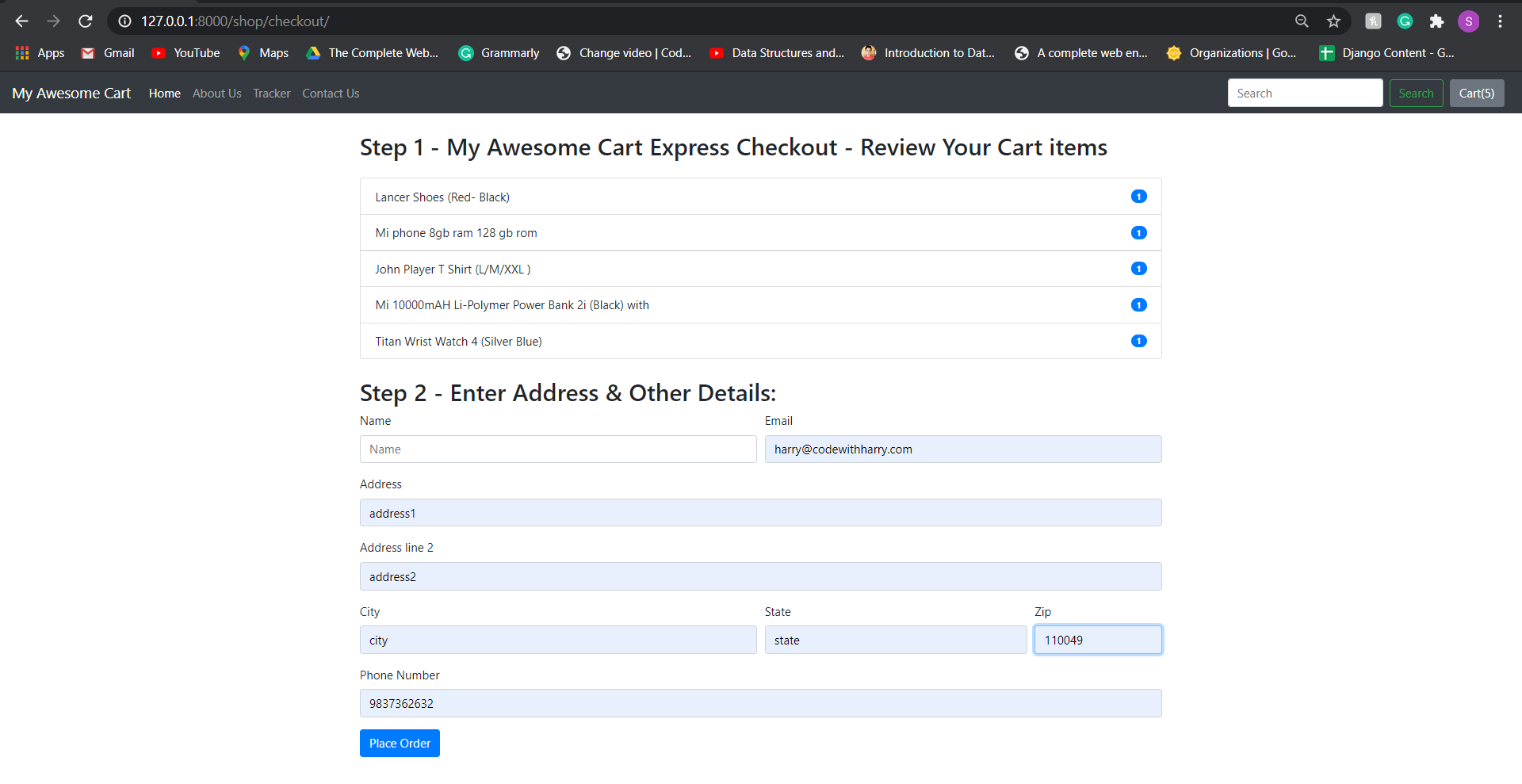
<form method="post" action="/shop/checkout/">{% csrf\_token %}

<input type="hidden" name="itemsJson" id="itemsJson">

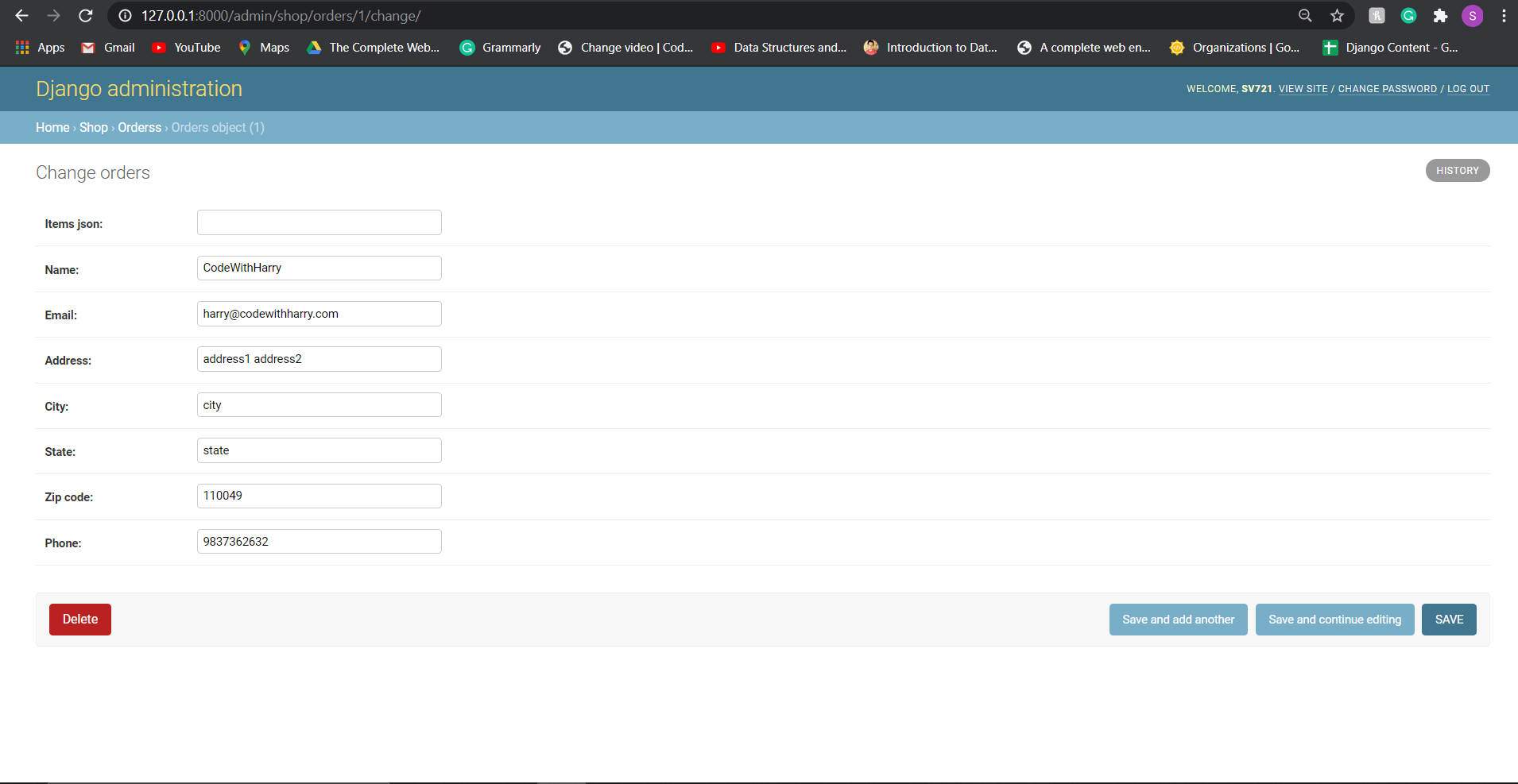
Notice the highlighted code in the below image and make the required changes by yourself. Also, do not forget to give a unique id and name to every form group.



Now, restart the development server, add items in the cart, and head to the checkout page. Fill in all the required details and click on the "Place Order" button. In the below image, you can see that I've added some items and filled in all the required form details.



Now, head over to the Django admin panel and click on the Orders. You will notice that all details filled in by you while placing the order are sent to the admin pane. You can also find the same in the below image :



In the above image, you can see that all the details filled while placing the order are displayed except items JSON. To fix this issue, we need to write a javascript function, so open the checkout.html file and type the below code:

 //setting the tag of id "itemsJson" to "cart" value

    $('#itemsJson').val(JSON.stringify(cart));

    //alert notification that the order is placed when the place order button is pressed, "thanks" becomes True in "shop/views.py"

    {% if thank %}

        alert('Thanks for ordering with us. Your order ID is {{id}}. Use it to track your order using our order tracker');

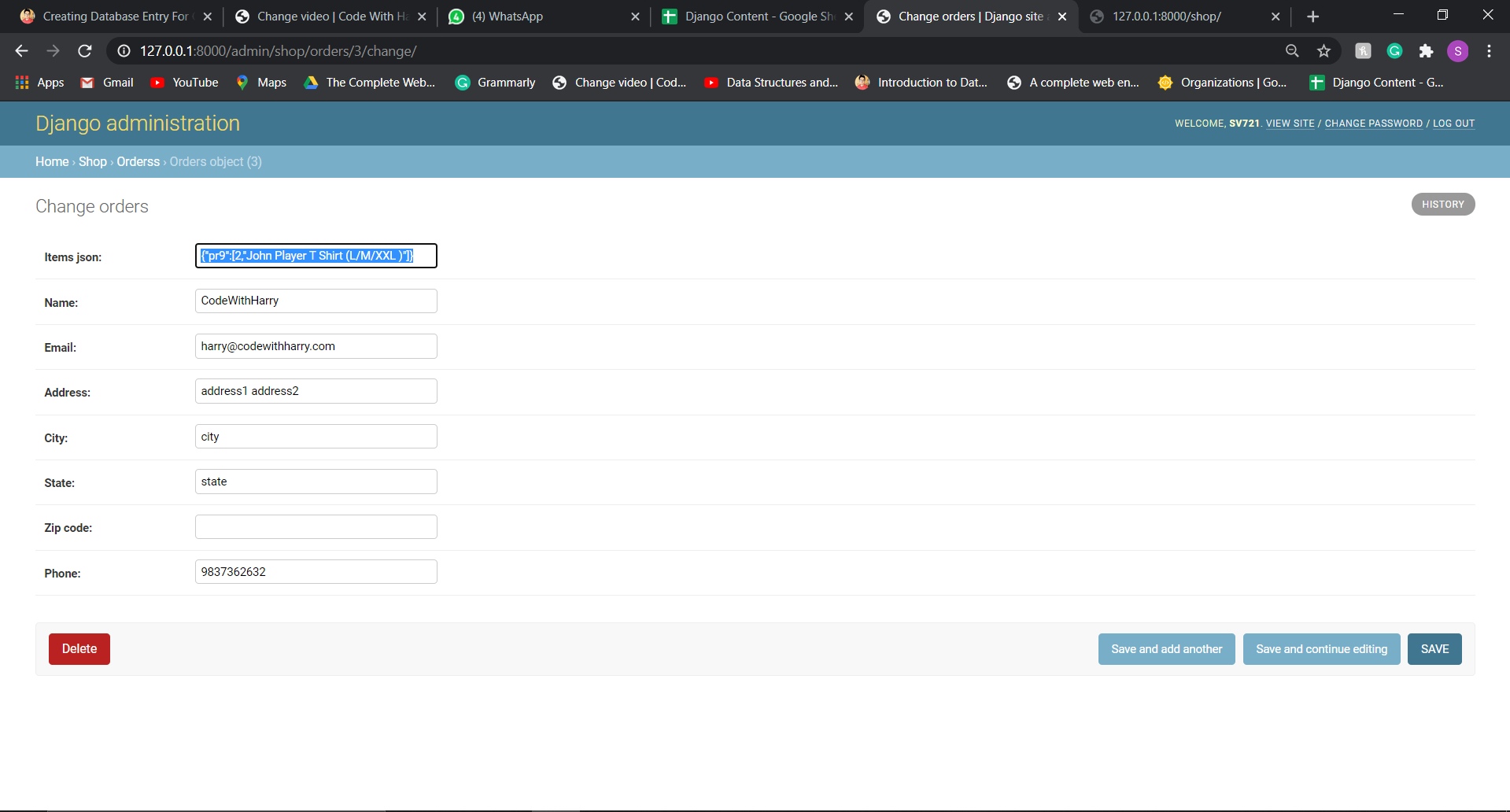
        //after placing the order just empty the localStorage

        localStorage.clear();

        document.location = "/shop";

{% endif %}

With the help of the above code, we will show an alert to the user if the order is placed successfully. Now, restart the development server and check if everything is working correctly or not. In the below image, you can see that the name and quantity(JSON items) are now getting displayed on the admin panel.



# Creating Order Tracker For Customers

we will create a new model to track the orders. So, get into the shop/models.py file and type the below code.

class OrderUpdate(models.Model):

    update\_id = models.AutoField(primary\_key = True)

    order\_id = models.IntegerField(default="")

    update\_desc = models.CharField(max\_length=5000)

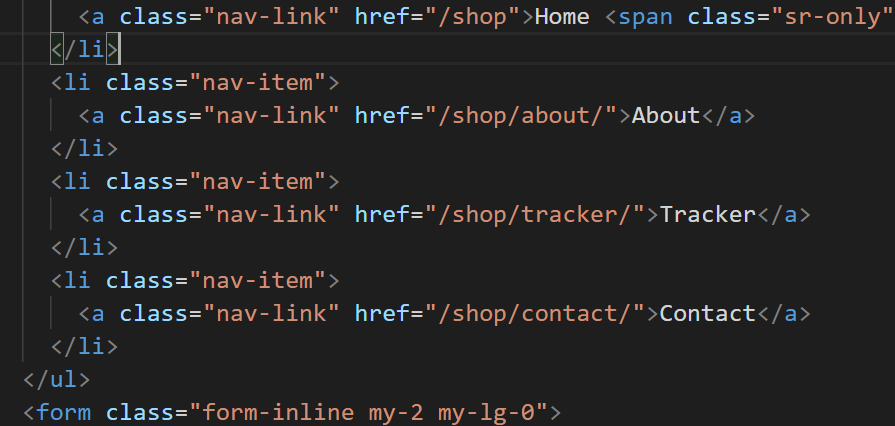
    timestamp = models.DateField(auto\_now\_add=True)

    def \_\_str\_\_(self):

        return self.update\_desc[0:12] + "..."

Do not forget to register the model and migrate the changes in the database. After migrating the model, open the views.py file of the shop app and write the following code in the checkout function:

Now add a “tracker” to the navbar in “shop/basic.html”



Add the below code to “shop/views.py” in class Order():

#This to save order tracking update in the database

        update = OrderUpdate(order\_id = id, update\_desc = "The order has been placed")

        update.save()

It adds initial order status in the database.

In “shop/views.py”:

def tracker(requests):

    if requests.method=="POST":

        orderId = requests.POST.get('orderId', '')

        email = requests.POST.get('email', '')

        # print(orderId,email)

        try:

            order = Orders.objects.filter(order\_id=orderId, email=email) #returns a list

            print(f"{order}-order")

            if len(order)>0:

                update = OrderUpdate.objects.filter(order\_id=orderId)

                updates = []

                for item in update:

                    updates.append({'text': item.update\_desc, 'time': item.timestamp})

                    response = json.dumps(updates, default=str)

                return HttpResponse(response)

            else:

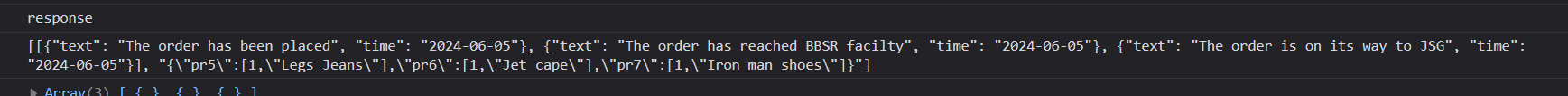
                return HttpResponse('{}')

        except Exception as e:

            return HttpResponse('{}')

This sends a response to “tracker.html”, when we submit the form in tracker.html.

This is how the response looks like. Ignore the items part it was added later in this journal.



In “shop/tracker.html”:

{% extends 'shop/basic.html' %}

{% block title %} My Awesome Cart Tracker{% endblock %}

{% block body %}

<div class="container">

    <div class="col my-4">

        <h2> Enter Your Order Id and Email address to track your order </h2>

           <form method="post" action="#" id="trackerForm">{% csrf\_token %}

            <div class="form-row">

                <div class="form-group col-md-6">

                    <label for="inputname">Order Id</label>

                    <input type="text" class="form-control" id="orderId" name="orderId" placeholder="Orer Id">

                </div>

                <div class="form-group col-md-6">

                    <label for="inputEmail4">Email</label>

                    <input type="email" class="form-control" id="email" name="email" placeholder="Email">

                </div>

                 <button type="submit" class="btn btn-primary">Track Order</button>

            </div>

    </div>

    <div class="col my-4">

        <h2>Your Order Status :</h2>

        <div class="my-4">

            <ul class="list-group" id="items">

                Enter your Email ID and order ID to track your order delivery.

            </ul>

        </div>

    </div>

</div>

{% endblock %}

{%block js%}

<script>

    //This is used to fetch the data without reloding the webpage

    $('#trackerForm').submit(function(event) {

        $('#items').empty();

        //sending data in form of JSON

        var formData = {

            'orderId': $('input[name=orderId]').val(), //to get value of input tags

            'email': $('input[name=email]').val(),

            'csrfmiddlewaretoken': $('input[name=csrfmiddlewaretoken]').val()

        };

        //With Ajax, web applications can send and retrieve data from a server asynchronously without interfering with the display and behaviour of the existing page.

        $.ajax({

                type: 'POST',

                url: '/shop/tracker/',

                data: formData,

                encode: true

            })

            //response is the data returned as Httpsrespnse from views.py from class order().

            .done(function(response){

                console.log(response)

                updates = JSON.parse(response);

                if (updates.length > 0 & updates != {}) {

                    for (i = 0; i < updates.length; i++) {

                        let text = updates[i]['text'];

                        let time = updates[i]['time'];

                        mystr = `<li class="list-group-item d-flex justify-content-between align-items-center">

                        ${text}

                        <span class="badge badge-primary badge-pill">${time}</span>

                    </li>`

                        $('#items').append(mystr);

                    }

                } else {

                    mystr = `<li class="list-group-item d-flex justify-content-between align-items-center">

                        Sorry, We are not able to fetch this order id and email. Make sure to type correct order Id and email</li>`

                    $('#items').append(mystr);

                }

            });

        //prevents default behaviour of the form i.e reloade webpage, etc.

        event.preventDefault();

    });

    </script>

{%endblock%}

# Displaying order items on tracker page

In “shop/views.py”:

We have made some changes to def tracker(requests): ,

        try:

            order = Orders.objects.filter(order\_id=orderId, email=email) #returns a list

            print(f"{order}-order")

if len(order)>0:

                update = OrderUpdate.objects.filter(order\_id=orderId)

                updates = []

                for item in update:

                    updates.append({'text': item.update\_desc, 'time': item.timestamp})

                response = json.dumps([updates,order[0].items\_json], default=str)

                return HttpResponse(response)

we are also passing the “items\_json” attribute as a response to “tracker.html”.

It is used there to show the items name and quantity.

In “shop/tracker.html”:

<div class="col my-4">

        <h2>Your Order Status :</h2>

        <div class="my-4">

            <ul class="list-group" id="items">

                Enter your Email ID and order ID to track your order delivery.

            </ul>

        </div>

        <h2>Your Orderered itmes :</h2>

        <div class="my-4">

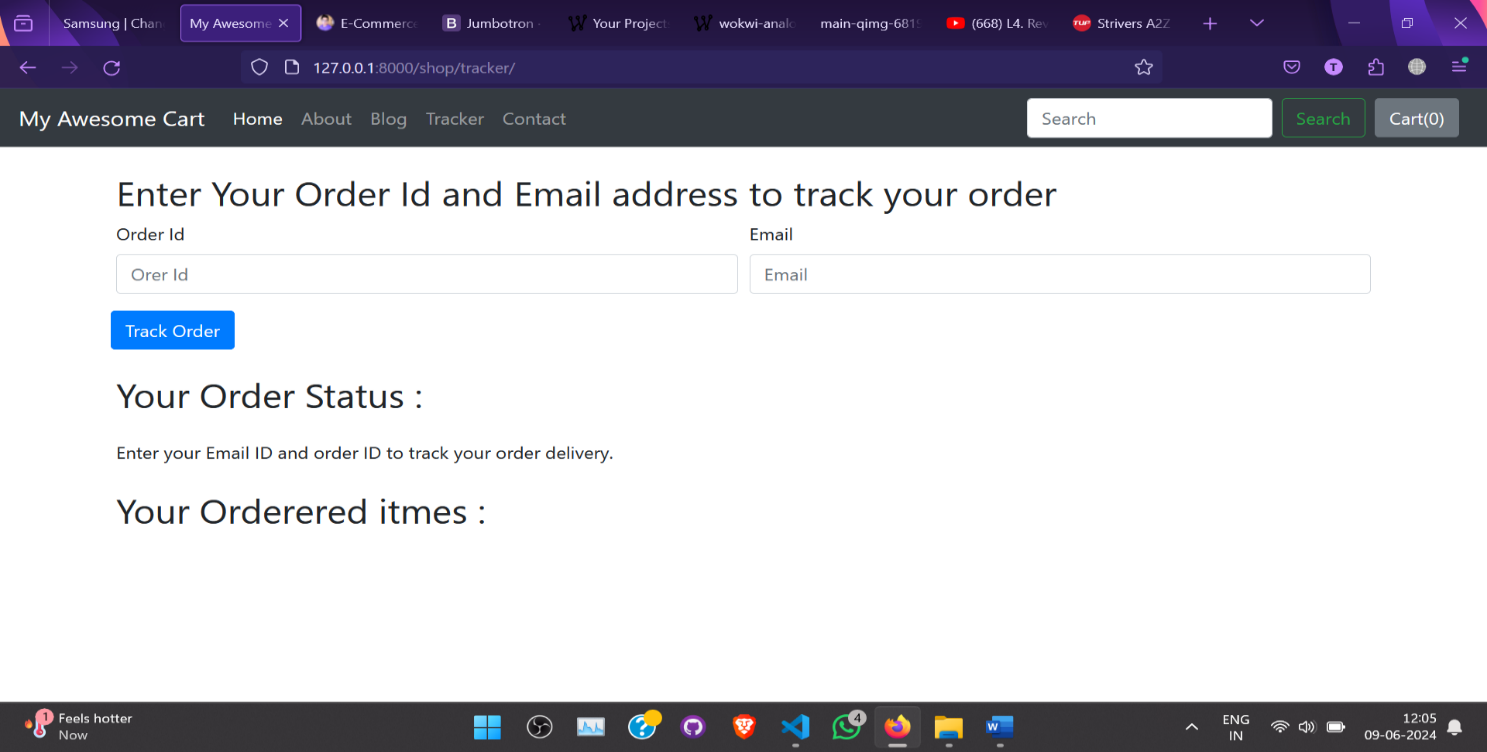
            <ul class="list-group" id="citems">

            </ul>

        </div>

    </div>

The above part is where we show it.



This is how it looks when empty.

We have added the following code in .Ajax (the one talked about earlier):

                //To show also the ordered items

                if (response!={}){

                    orderedItems = JSON.parse(response[1]) //need to parse again as response[1] is itself a string.

                    if (orderedItems != undefined)

                    console.log(orderedItems)

                    for (item in orderedItems) {

                            let name = orderedItems[item][1];

                            let qty = orderedItems[item][0];

                            mystr = `<li class="list-group-item d-flex justify-content-between align-items-center">

                            ${name}

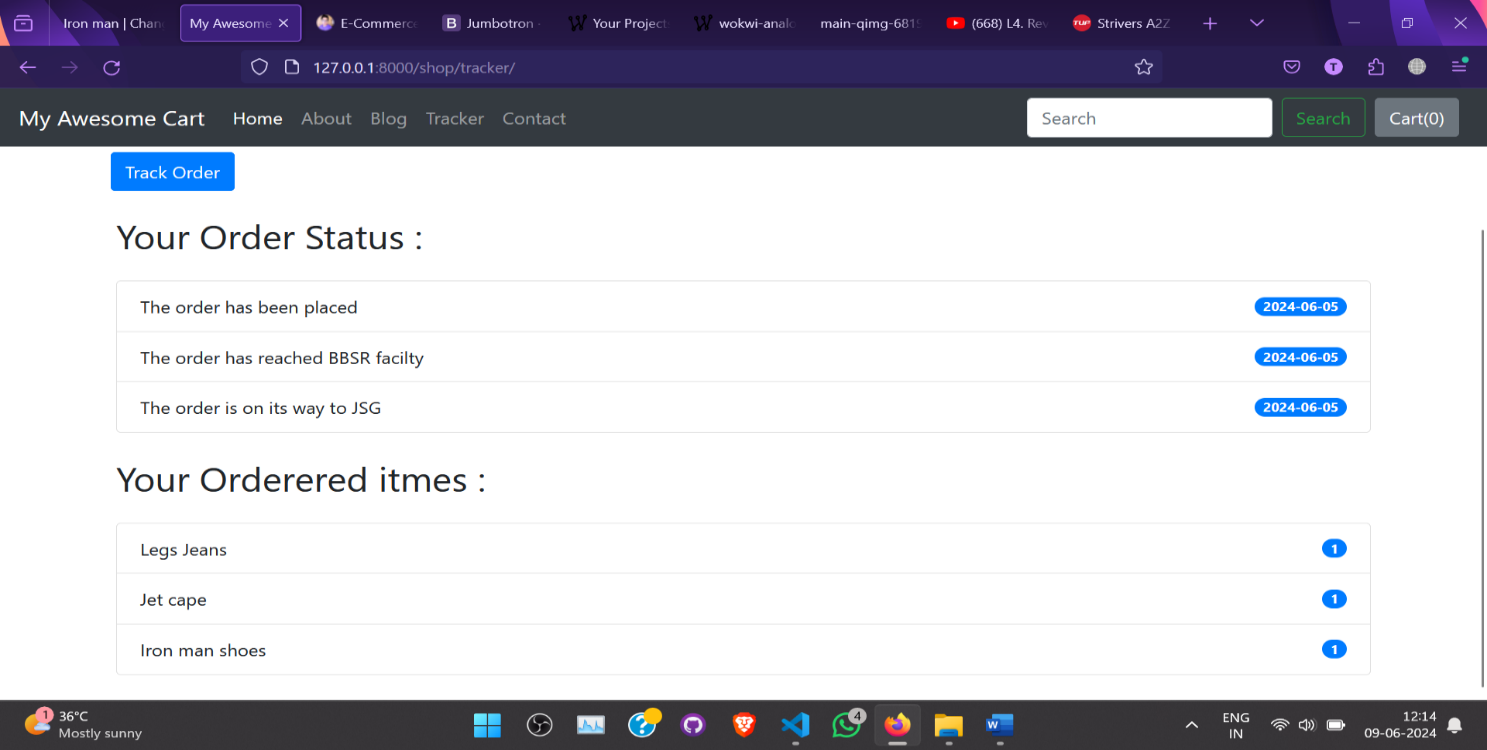
                            <span class="badge badge-primary badge-pill">${qty}</span>

                        </li>`

                            $('#citems').append(mystr); // to append mystr to tag id “citems”

                    }

                }



This is how it looks.

# Merging Blog App Into The Shop UI | Python Django Tutorials In Hindi #55

In project folder “mac/templates/index.html”:

<!DOCTYPE html>

<html lang="en">

  <head>

    <script> window.location="/shop";</script> <!--to redirect to the shop app without rendering the below code!!-->

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<script> window.location="/shop";</script> to redirect to the shop app without rendering the code of the base url i.e <http://127.0.0.1:8000/> !!-->

# Creating Product Search Feature In Our E Commerce Website | Python Django Tutorials In Hindi #67

In “shop/views.py”:

#to check the words present in query and items fetched from the database one by one

def searchMatch(query, item):

    if (query in item.desc.lower() or query in item.product\_name.lower() or query in item.category.lower()):

        return True

def search(requests):

    query =  requests.GET.get('search')

    # print(f"query- {query}")

    allProds = []

    catprods = Products.objects.values('category', 'id')

    # print(f'catprods:{catprods}')

    cats = {item['category'] for item in catprods} #stores various categories

    # print(cats)

    for cat in cats:

        prodtemp = Products.objects.filter(category=cat) #category wise filter product, returns a list of objects

        prod = [item for item in prodtemp if (searchMatch(query, item))] #checking if the item exists in the query or words made or put by the user in the search.

        n = len(prod)

        if (n != 0):

            print(f"n-{n}")

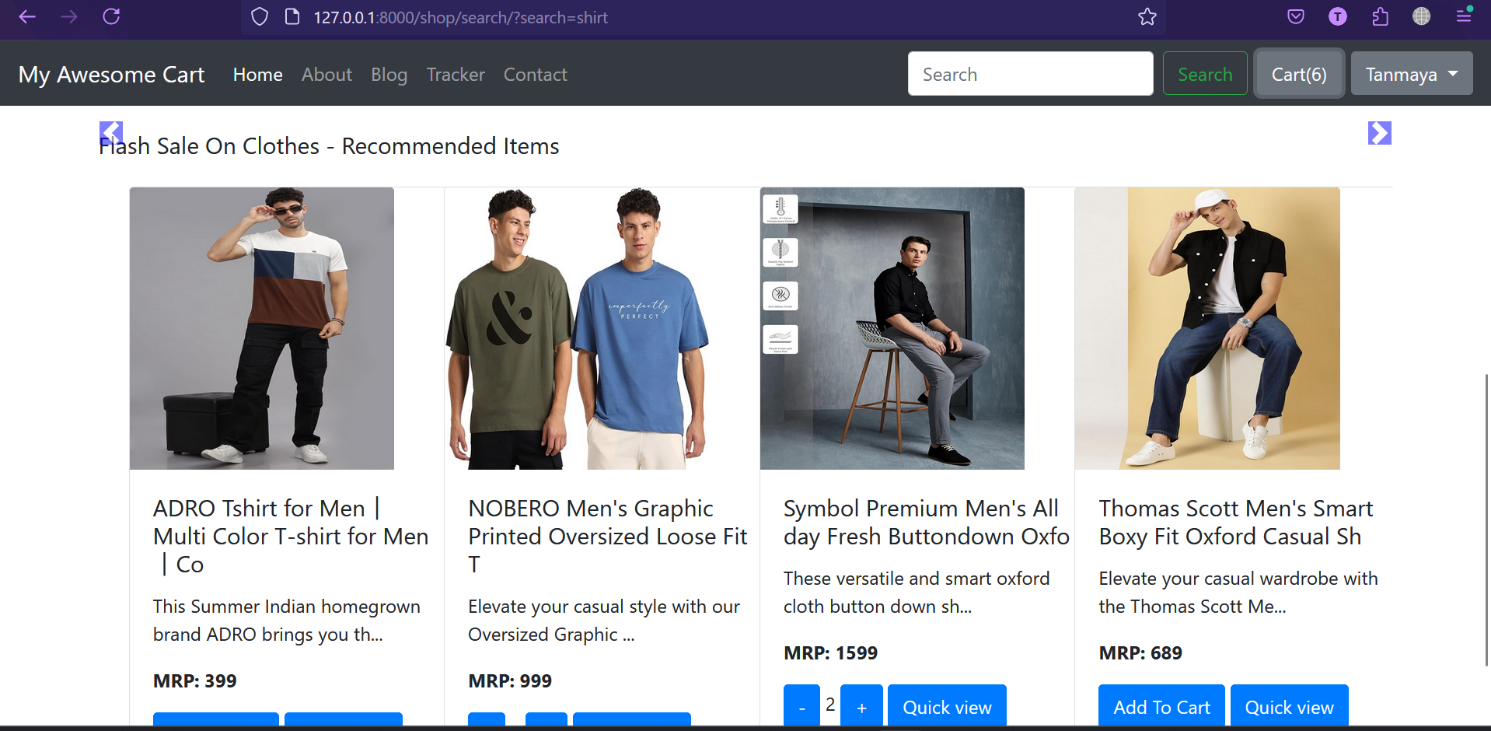
            nSlides = n // 4 + ceil((n / 4) - (n // 4))

            allProds.append([prod, range(1, nSlides), nSlides])

    params = {'allProds':allProds}

    return render(requests,"shop/search.html",params)

For example I have searched for ‘shirt’ and the following is the result:



The search bar gets empty after the search.

# Django Messages Framework: Showing Messages in Django blog | Python Django Tutorials In Hindi #79

**Step 1**: Import the message framework by typing the below code in the settings.py file:

#maine import kiya hai, use karna tha

import os

from django.contrib.messages import constants as messages # to use inbuilt message functions of Django in jinga templates

**Step 2:** Set the below message tag to display an error message:

#this is added to give a tag name to the message type to be used in the template from bootstrap as "error" is not a tag name to be found in bootstrap

MESSAGE\_TAGS = {

    messages.ERROR:'danger'

}

Now, we will use the Bootstrap dismissible alert to display an alert message So, open the “shop/base.html” file and type the below code:

{%if messages%}

      {% for message in messages %}

      <div class="alert alert-{{ message.tags }} alert-dismissible fade show"

        role="alert">

        <strong>Message : </strong> {{ message }}

        <button type="button" class="close" data-dismiss="alert"

          aria-label="Close">

          <span aria-hidden="true">&times;</span>

        </button>

      </div>

      {% endfor %}

  {%endif%}

One of the use instances can be seen in “shop/views.py”:

def handelLogin(request):

    if request.method=="POST":

        # Get the post parameters

        loginusername=request.POST.get('loginusername')

        loginpassword=request.POST.get('loginpassword')

        user=authenticate(username= loginusername, password= loginpassword)

        if user is not None:

            login(request, user)

            messages.success(request, "Successfully Logged In")

            return redirect("/shop")

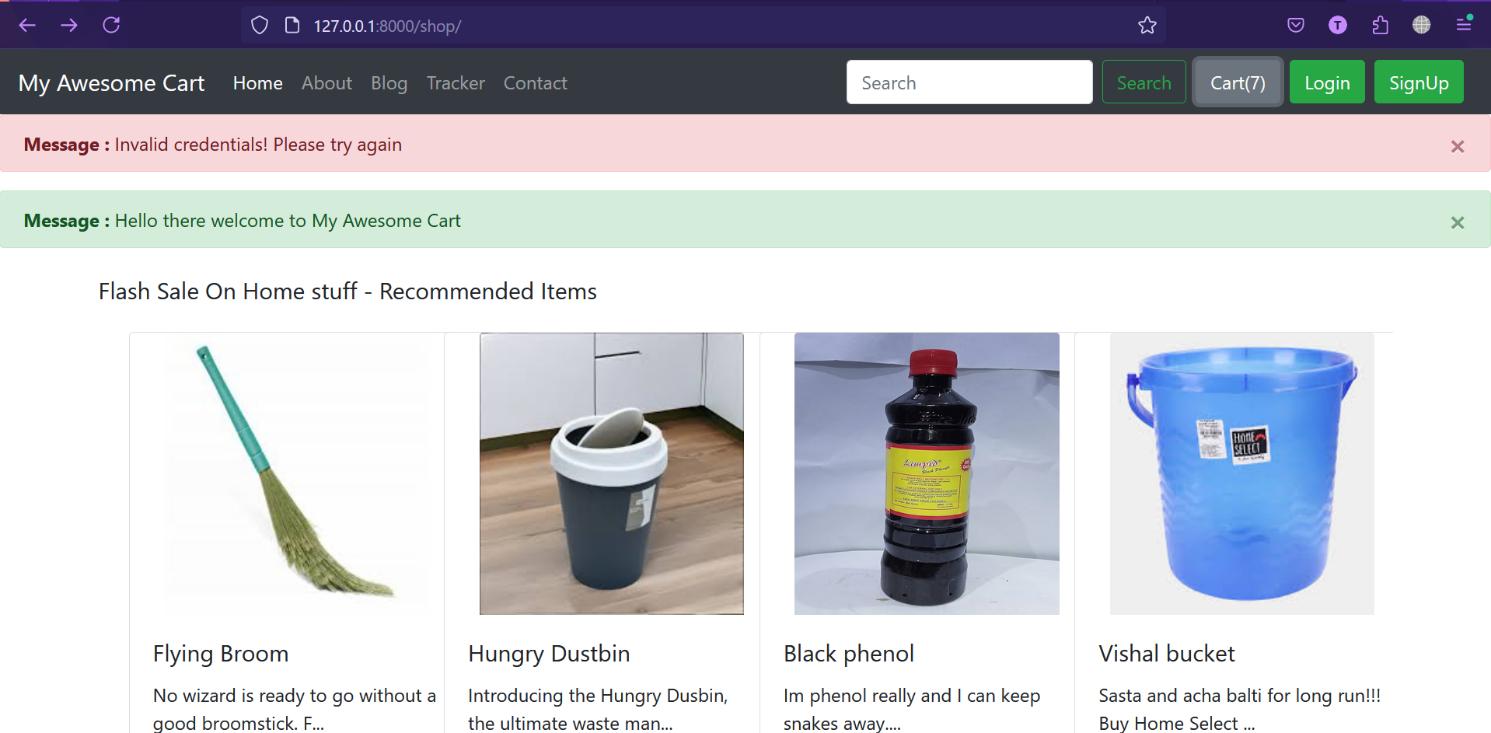
        else:

            messages.error(request, "Invalid credentials! Please try again")

            return redirect("/shop")

    return HttpResponse("404- Not found")

messages.tag\_name()



Red message is the “.error” and green message is the “.success”.

# Django 3 Blog: Authentication & Authorization Tutorial | Python Django Tutorials In Hindi #85

Why Use Django User Authentication:

* It's better to use the existing built-in boilerplate than to reinvent the wheel.
* We will use the Django authentication system to create users who can sign up and comment on the shop and blog app both.

(blog app was created later in the project)

* The Django authentication system handles both authentication and authorization.

Difference Between Authentication And Authorization:

* Usually, the term authentication refers to both the task, but there is a slight difference between these two terms.
* Authentication verifies a user who they claim to be, and authorization determines what an authenticated user is allowed to do.

What has the Django authentication system got for us?

The auth system consists of:

* Users
* **Permissions:** Binary(yes/no) flags designation whether a user may perform a certain task.
* **Groups:** A generic way of applying labels and permissions to more than one user.
* A configurable password hashing system.
* Forms and view tool for logging in users or restricting content.
* A pluggable backend system.

What Is Not Included?

1. Django's authentication system aims to be very generic and doesn't provide some features commonly found in web authentication systems.
2. Solutions for some of these common problems have been in third-party packages:

* Password strength checking
* Throttling of login attempts
* Authentication against the third party
* Object-level permissions

# Creating Users in Django | Python Django Tutorials In Hindi #86

In “shop/basic.html”, we have created a signup form:

<!-- Button to  trigger SignUp modal -->

<button type="button" class="btn btn-success mr-2" data-toggle="modal"data-target="#signupModal">

        SignUp

</button>

<!-- SignUp Modal -->

<div class="modal fade" id="signupModal" tabindex="-1"

    aria-labelledby="signupModal" aria-hidden="true">

    <div class="modal-dialog">

      <div class="modal-content">

        <div class="modal-header">

          <h5 class="modal-title" id="signupModalTitle">SignUp Here</h5>

          <button type="button" class="close" data-dismiss="modal"

            aria-label="Close">

            <span aria-hidden="true">&times;</span>

          </button>

        </div>

        <div class="modal-body">

          <form action="/shop/signup/" method='post'>

            {% csrf\_token %}

            <div class="form-group">

              <label for="username">Username</label>

              <input type="text" class="form-control" id="username"

                name="username" placeholder="Choose a unique username">

            </div>

            <div class="form-group">

              <label for="fname">First Name</label>

              <input type="text" class="form-control" id="fname" name="fname"

                placeholder="Enter Your First Name">

            </div>

            <div class="form-group">

              <label for="lname">Last Name</label>

              <input type="text" class="form-control" id="lname" name="lname"

                placeholder="Enter Your Last Name">

            </div>

            <div class="form-group">

              <label for="email">Email address</label>

              <input type="email" class="form-control" id="email" name="email"

                placeholder="name@example.com">

            </div>

            <div class="form-group">

              <label for="pass1">Choose a password</label>

              <input type="password" class="form-control" id="pass1"

                name="pass1" placeholder="Choose Your Password">

            </div>

            <div class="form-group">

              <label for="pass2">Confirm Password</label>

              <input type="password" class="form-control" id="pass2"

                name="pass2" placeholder="Enter your password again">

            </div>

            <button type="submit" class="btn btn-primary">Submit</button>

          </form>

        </div>

      </div>

    </div>

  </div>

The form action is handeled by handelSignup() in “shop/views.py”:

First of all, we're checking if the user has made a post request to make a new account on the blog. After this, we're storing all the inputs from the user in different variables. We've used the create\_user() function of Django to create a new user. create\_user() takes three arguments:

* username
* email
* password

In the end, we're storing the newly created user with the help of myuser.save(). If the user is created successfully, we will display a success message by using Django's message framework, and the user will be redirected to the home page. Define the path given below in the urls.py file of the home app:

from django.contrib.auth.models import User

from django.shortcuts import redirect

def handelSignup(requests):

    if requests.method=="POST":

        # Get the post parameters

        username=requests.POST.get('username')

        email=requests.POST.get('email')

        fname=requests.POST.get('fname')

        lname=requests.POST.get('lname')

        pass1=requests.POST.get('pass1')

        pass2=requests.POST.get('pass2')

        # check for errorneous input

        # if len(username)<10:

        #     messages.warning(requests, " Your user name must be under 10 characters")

        #     return redirect('/blog')

###some checks to let the user know the format that we want to him to enter###

        if not username.isalnum():

            messages.warning(requests, " User name should only contain letters and numbers")

            return redirect('/shop')

        if (pass1!= pass2):

            messages.warning(requests, " Passwords do not match")

            return redirect('/shop')

        # Create the user

        myuser = User.objects.create\_user(username, email, pass1)

        myuser.first\_name= fname

        myuser.last\_name= lname

        myuser.save()

        messages.success(requests, " Your Account has been successfully created")

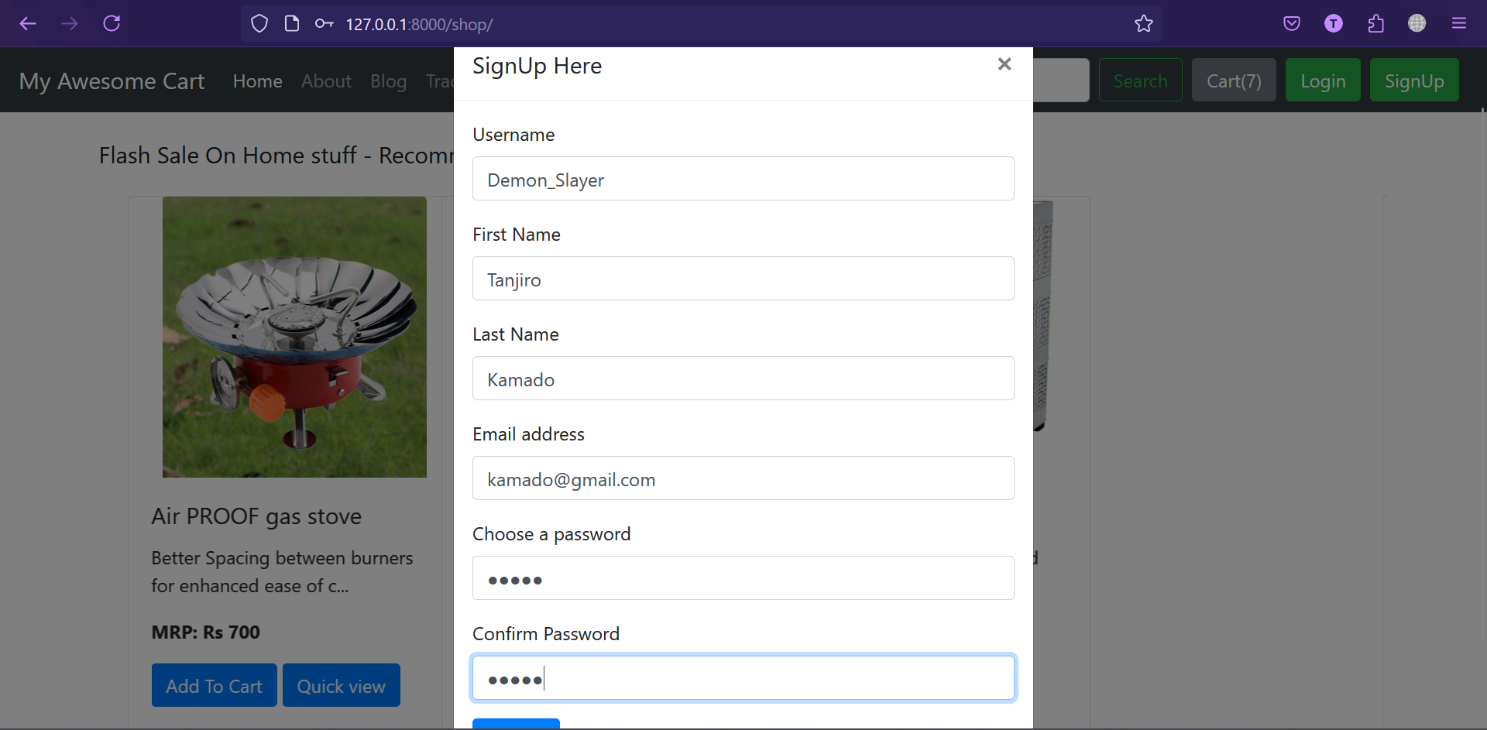
        return redirect('/shop')

    else:

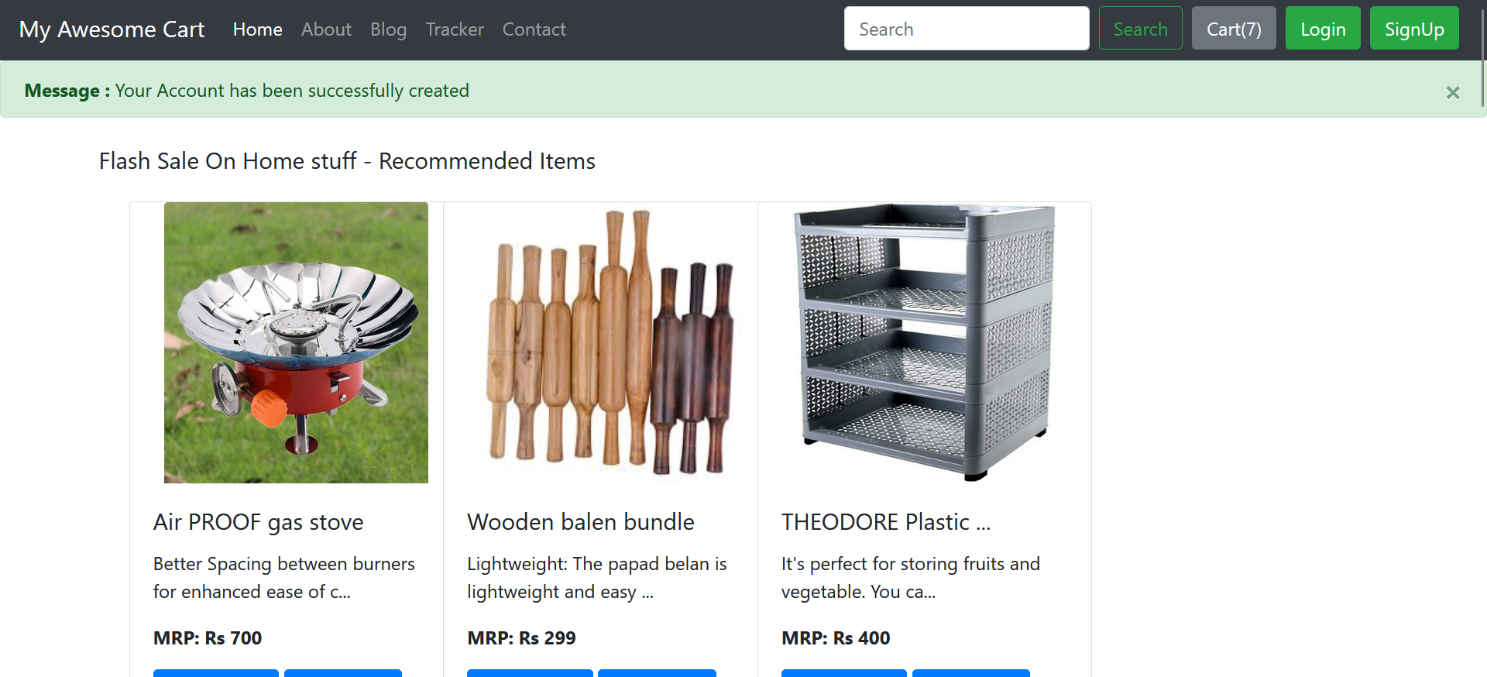
        return HttpResponse("404 - Not found")

Define the path given below in the urls.py file of the home app:

path('signup/',views.handelSignup,name='signup')

Demonstration: Input:

Output:



# Creating login and logout functionalities

Add this in “shop/urls.py”:

path('login', views.handeLogin, name="handleLogin"),

path('logout', views.handelLogout, name="handleLogout")

In “shop/templates/basic.html”, I have added the login form:

<!-- Button to  trigger Login modal -->

<button type="button" class="btn btn-success mr-2" data-toggle="modal" data-target="#loginModal">

        Login

</button>

<!-- Login Modal -->

  <div class="modal fade" id="loginModal" tabindex="-1"

    aria-labelledby="loginModal" aria-hidden="true">

    <div class="modal-dialog">

      <div class="modal-content">

        <div class="modal-header">

          <h5 class="modal-title" id="loginModalTitle">Login Here</h5>

          <button type="button" class="close" data-dismiss="modal"

            aria-label="Close">

            <span aria-hidden="true">&times;</span>

          </button>

        </div>

        <div class="modal-body">

          <form method="post" action="/shop/login/">

            {% csrf\_token %}

            <div class="form-group">

              <label for="username">Username</label>

              <input type="text" class="form-control" id="loginusername"

                name="loginusername" placeholder="Choose a unique username">

            </div>

            <div class="form-group">

              <label for="pass">Enter your password </label>

              <input type="password" class="form-control" id="loginpassword" name="loginpassword"

                placeholder="Enter your password ">

            </div>

            <button type="submit" class="btn btn-primary">Submit</button>

          </form>

        </div>

Now in “shop/views.py” I handel the post request:

from django.contrib.auth.models import User

from django.shortcuts import redirect

from django.contrib.auth import authenticate, login, logout

def handelLogin(request):

    if request.method=="POST":

        # Get the post parameters

        loginusername=request.POST.get('loginusername')

        loginpassword=request.POST.get('loginpassword')

        user=authenticate(username= loginusername, password= loginpassword)

        if user is not None:

            login(request, user)

            messages.success(request, "Successfully Logged In")

            return redirect("/shop")

        else:

            messages.error(request, "Invalid credentials! Please try again")

            return redirect("/shop")

    return HttpResponse("404- Not found")

we've used the authenticate() function of Django; authenticate() function takes two keyword arguments :

1. username
2. password

If the username and password are correct, it returns a user object for the given username otherwise, it returns None. After this, we've used an if loop to check whether the user is None or not. If the user is None, it means the provided credentials are incorrect, and we will display an error message. If the user is not None, then we are using the login() function of Django. The login() function also takes two arguments:

1. request: A HttpResponse object.
2. user: User object returned by the authenticate() function.

As soon as the login function executes, Django's session framework saves the user's id in the session.

The following in “shop/views.py” is for log out a user:

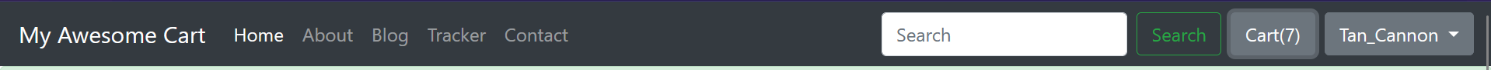
def handelLogout(request):

    logout(request)

    messages.success(request, "Successfully logged out")

    return redirect('/shop')

In “shop/basic.html” this is shown in the nav bar to logout:



{% if user.is\_authenticated %} <!—to check if a user has logged in->

      <div class="dropdown">

        {% comment %} <button class="btn btn-secondary dropdown-toggle" type="button" data-bs-toggle="dropdown" aria-expanded="false">

          {{user.username}}

        </button> {% endcomment %}

        <a class="btn btn-secondary dropdown-toggle" href="/shop/logout/" role="button" data-bs-toggle="dropdown" aria-expanded="false">

          {{user.username}}

        </a>

        <ul class="dropdown-menu">

          <li><a class="dropdown-item" href="/shop/logout/">Logout</a></li>

        </ul>

      </div>

# Creating comment functionality in Blog app

## First create the comment model in “blog/models.py”:

from django.contrib.auth.models import User

from django.utils.timezone import now

class BlogComment(models.Model):

    #comment unique id

    comment\_id = models.AutoField(primary\_key=True)

    #what is the comment

    comment = models.TextField()

    #who wrote

    user = models.ForeignKey(User,on\_delete=models.CASCADE)

    #on which post the comment is made

    post = models.ForeignKey(Blogpost,on\_delete=models.CASCADE)

    #to which comment its a reply or is it a 1st comment i.e null=True

    parent=models.ForeignKey('self',on\_delete=models.CASCADE,

null=True )

    #when comment was posted

    timestamp = models.DateTimeField(default=now)

    def \_\_str\_\_(self):

        return self.comment[0:13] + "..." + "by" + " " +

self.user.username

We use foreign\_key to define a many-to-one relationship in Django. In the case of the blog app, a comment will be associated with a user and a blog post. ForeignKey takes two arguments:

1. The first argument is the class to which the foreign key is pointing.
2. The second argument is on\_delete. This method is used to adopt a behaviour when the reference object is deleted. Django requires us to tell it what should be done with model instances that depend on the other model instance. When a user deletes a blog, we want to delete all the comments associated with the blog post. Here, the deletion of the comments depends upon the deletion of the blog post. So, when the reference object is deleted, then all the objects that have references to it will also be deleted.

After creating the above model, make the migrations by typing the below code:

python manage.py makemigrations

Now, type the below command to write the changes in the database:

python manage.py migrate

In Django, we use makemigrations command to save the changes in a file. This command does not make any change in the database. As soon as we execute the migrate command, then only the changes are applied in the database. After this, we need to register the model, so open the admin.py file of the blog app and type the below code:

from django.contrib import admin

from blog.models import Post, BlogComment

admin.site.register((Post, BlogComment))

## Create the API to post comments:

The function that we are writing within “blog/views.py” can be said as an API.

So in there we have added this:

def postComment(requests):

    if requests.method == "POST":

        #what is the comment

        comment=requests.POST.get("comment")

        print(comment)

        #who has made the comment

        user = requests.user

        #in which post the comment has been made

        postId = requests.POST.get("postId")

        #in which comment is this comment has been made as it can be reply to another comment

parentId = requests.POST.get("parentId")

        parent = BlogComment.objects.get(comment\_id = parentId)

        post = Blogpost.objects.get(post\_id= postId)

        if (parentId != ""): #i.e it a comment to another comment (a

reply)

comment = BlogComment( comment = comment, user = user,

post = post, parent = parent)

            comment.save()

            messages.success(requests, "Your reply has been posted

successfully")

        else: #i.e it is a parent comment

            comment = BlogComment( comment = comment, user = user,

post = post)

            comment.save()

            messages.success(requests, "Your comment has been posted

successfully")

    return redirect(f"/blog/blogpost/{post.post\_id}")

The template that uses this above function is as below in “blog/templates/blogpost.html”:

The user can only comment if he/she is logged in.

{% if user.is\_authenticated %}

        <div class="container px-0">

          <h2>Comments ({{comments.count}})</h2>

          <form action="/blog/postcomment/" method="post">

          {% csrf\_token %}

          <input type="text" name="comment" placeholder="Enter comment

here">

          <input type="hidden" name="postId" value="{{post.post\_id}}">

          <input type="hidden" name="parentId" value="">

          <input type="submit" value="Submit">

         </form>

         {% else %}

         <span class="badge badge-secondary"> Please login to comment

</span>

         {% endif %}

## Create or modify the blogpost to render comments and also replies too:

def blogpost(requests,id):

    post = Blogpost.objects.filter(post\_id = id)[0] #returns a list so using [0] to access elment

    # print(post)

    total\_posts = Blogpost.objects.count()

    comments = BlogComment.objects.filter(post=post, parent=None)

    replies = BlogComment.objects.filter(post=post).exclude(parent=None)

    replyDict = {}

    for reply in replies:

        if reply.parent.comment\_id not in replyDict.keys():

            replyDict[reply.parent.comment\_id] = [reply]

        else:

            replyDict[reply.parent.comment\_id].append(reply)

    # print(replyDict)

    params = {'total\_posts': total\_posts,'post':post, 'comments':comments, 'replyDict':replyDict}

    # print(total\_posts)

    return render(requests,'blog/blogpost.html', params)

In the above code, we've created two variables :

* comments: To store the comments.
* replies: To store the posted replies.

Then, we've created an empty dictionary named replyDict. After this, we are using a for loop to iterate over the replies variable. In the end, we are rendering the replies to the template by using the context dictionary. We will now use the replyDict dictionary in the template to display the replies on the blog post. But, inside the replyDict dictionary, replies are stored in the form of key-value pairs where the key is the parent number, and the value is the reply, and we only want to display the value and not the keys. So, we need to create a custom template filter to filter the values from the replyDict dictionary. To create a custom template filter, create a new directory named templatetags inside the blog directory. Create the following files inside the templatetags directory :

* \_\_init\_\_.py
* extras.py

Type the below code in the extras.py file:

from django import template

register=template.Library()

@register.filter(name='get\_val')

def get\_val(dict, key):

    return dict.get(key)

get\_val is the name of the filter that we will use to filter out the replies. Now, we need to load this template filter in the blog post template. So, type the below code in the blogPost.html file :

{% load extras %}

After this, we need to iterate over the replies so that we can display them on the blog post. We will perform this task with the help of a for loop. Type the below code in the blogPost.html after the submit button of the reply form:

         <!--rendering comments-->

          {% for comment in comments%}

          <div class="row my-3 mx-0">

            <div class="col-md-1">

<!--you can add the commenter pic here-->

              <img class="rounded mx-auto d-block w-100 p-2"

src="/media/shop/images/wooden\_chappal.jpg"  alt="user">

            </div>

            <div class="col-md-11 ">

              <b> {{comment.user.username}} </b> at

{{comment.timestamp}}

              <div>   {{comment.comment}} </div>

              <!--rendering replies-->

              <div class="reply">

              {% if user.is\_authenticated %}

                <button class="btn my-2 sm-2 btn-primary"

type="button" data-toggle="collapse" data-

target="#replyBox{{comment.comment\_id}}" aria-

expanded="false" aria-

controls="replyBox{{comment.comment\_id}}">

                    Reply

                </button>

                <div class="collapse my-2"

id="replyBox{{comment.comment\_id}}">

                  <div class="card card-body my-2">

                    <form action="/blog/postcomment/" method="post">

                        {% csrf\_token %}

                        <div class="form-group">

                            <label for="comment">Post a reply </label>

                            <input type="text" class="form-control"

name="comment" placeholder="Enter

comment here">

                            <input type="hidden" name="parentId"

value="{{comment.comment\_id}}">

                        </div>

                        <input type="hidden" name="postId"

value="{{post.post\_id}}">

                        <button type="submit" class="btn btn-

primary">Submit</button>

                    </form>

                  </div>

                </div>

              {% else %}

                <button class="btn btn-sm btn-secondary disabled my-2"

type="button" data-toggle="collapse" data-

target="#replyBox{{comment.sno}}" aria-

expanded="false" aria-

controls="replyBox{{comment.sno}}">

                    Login to reply

                </button>

              {% endif %}

              </div>

              <div class="replies">

                {% for reply in replyDict|get\_val:comment.comment\_id

%}

                    <div class="row my-3 mx-0">

                      <div class="col-md-1  ">

<!--you can add the commenter pic here-->

                        <img class="rounded mx-auto d-block w-100 p-2"

src="/media/shop/images/wooden\_chappal.jpg"

alt="user">

                      </div>

                      <div class="col-md-11 ">

                        <b> {{reply.user.username}} </b> at

{{reply.timestamp}}

                        <div>   {{reply.comment}} </div>

                      </div>

                    </div>

                {% endfor %}

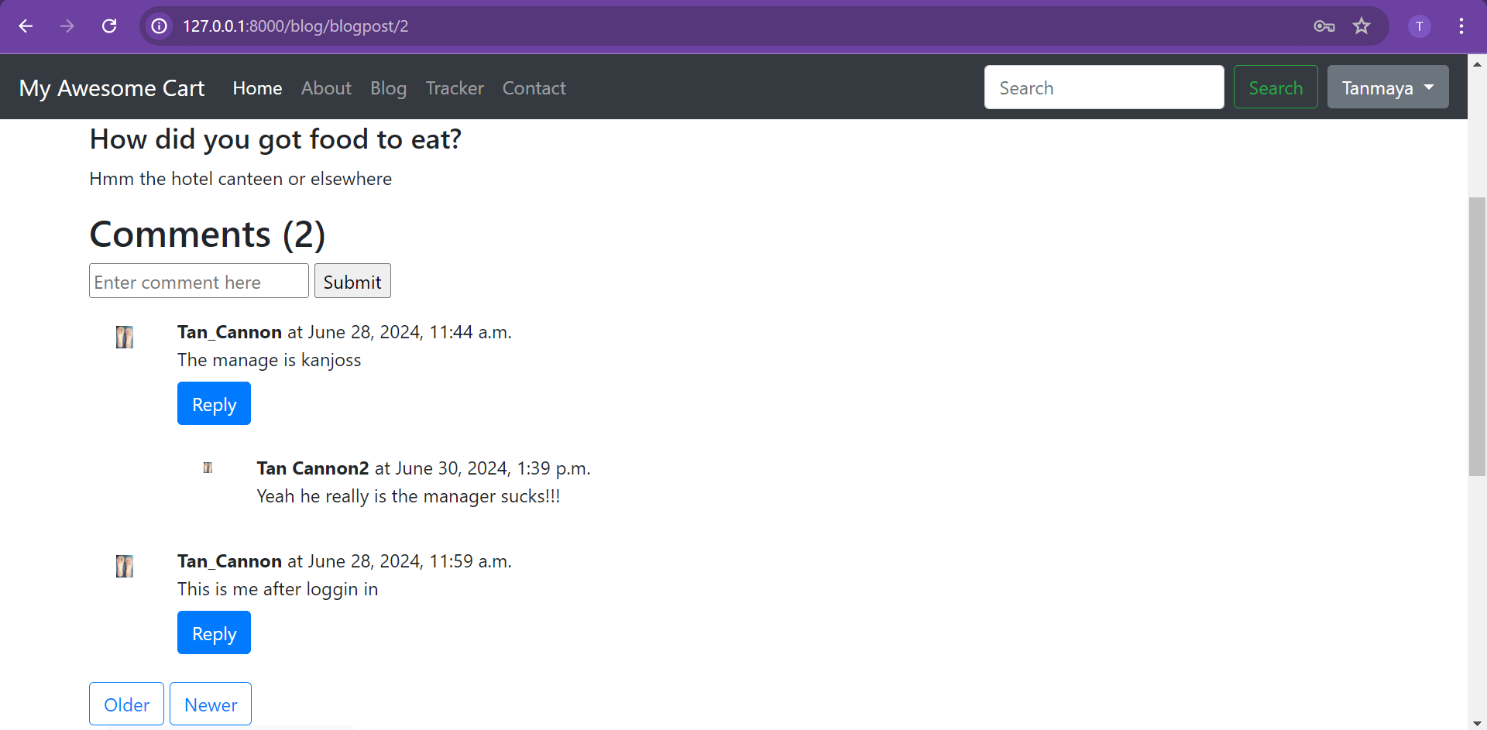
              </div>

            </div>

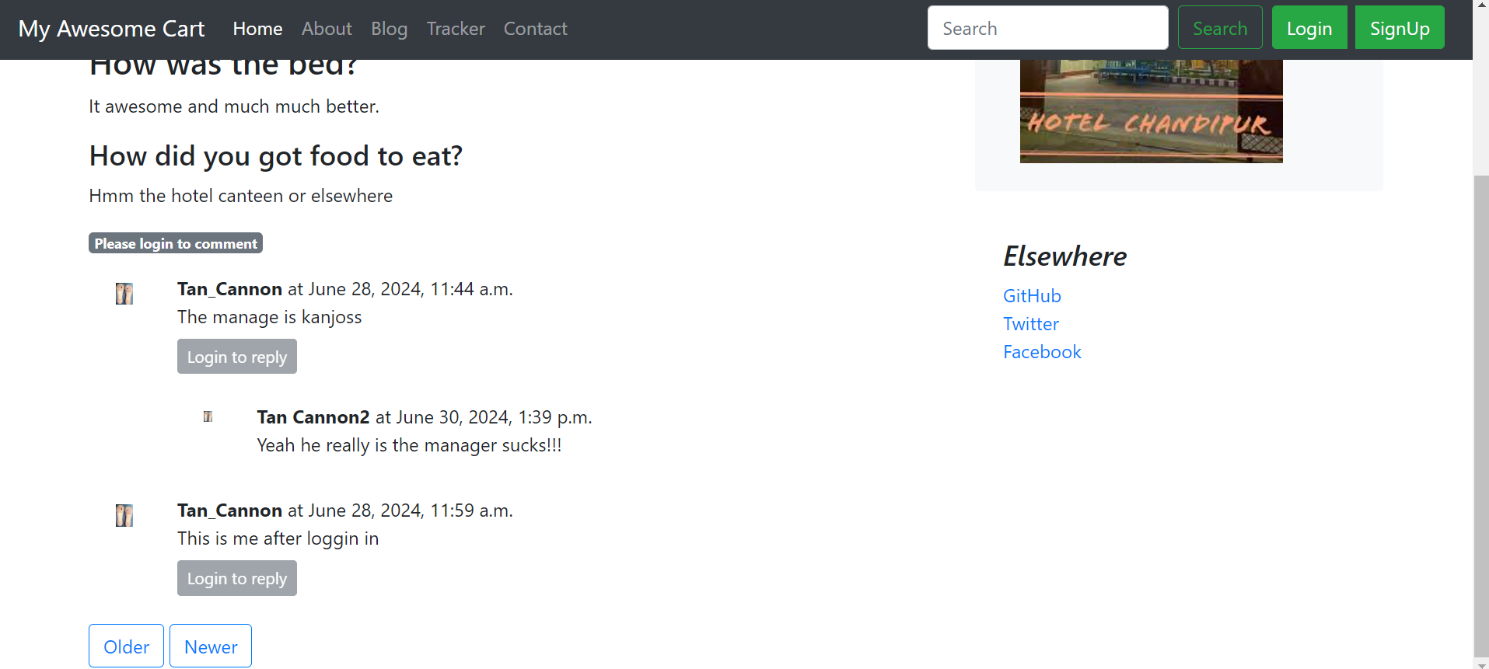
        </div>

        {% endfor %}

Results: The user is logged in here. So he can comment and reply.



The user is not logged in so he cannot comment nor can reply.



# Conclusion

How to create models?

1. Create the model class in models.py.
2. Register that model in admin.py.
3. Then migrate the model into the database. (1. python manage.py makemigrations, 2. python manage.py migrate)