TASK

Django – eCommerce Application Part 1

Visit our website

Introduction

Now that you have successfully created models and templates and connected them using views, we have to start thinking about limiting our user's access to some of our models and views. You might have experienced this, where you had to log in to a service before you could access certain content on the service. We would like to add similar functionality to our web apps.

Django Authentication

Have you ever wondered what happens after entering your login credentials to your favourite website? Once your finger touches that login button or you click it with your mouse, there is a ton of magic that happens behind the scenes to make sure you are who you say you are. In this task, we will take a look at how web services authenticate users and authorise their access to the service's features. We will be creating users with different roles and giving them permissions that allow us to control their access to our web service. Along with authentication and authorisation, we have to make sure our service can remember a user during the time they will be using our service, otherwise we would need them to log in for every action. After we have made all our changes and added our permissions we will have to perform database migration to save the model changes to our database.

Django authentication:

- provides you with the capability to authenticate and authorise users and their permissions,
- logs users in and out, and
- restricts users' access to views.

Before we have a look at authentication, we first have to introduce the concept of the **user object**.

USER OBJECTS

User objects play a very large role in Django's authentication system. They represent the people who are using your web service and allow you to restrict user access, register user profiles, associate content with specific users, and more.

Some of the primary attributes of user objects are:

- Username
- Password
- Email
- First name
- Last name

Creating users

We can create a user for our service using the `create_user()` function.

In Part 1 - Authentication & Login\AuthLog\grabsomore\views.py

```
from django.contrib.auth.models import User

def register_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        email = request.POST.get('email')
        user = User.objects.create_user(username=username,
password=password)
        user.email = email  # Add the email address
        user.save()  # Save the new user to the database
        login(request, user)
        return HttpResponseRedirect(reverse('grabsomore:welcome'))
    return render(request, 'grabsomore/register.html')
```

In the above example, we have to import our User model from **`django.contrib.auth.models`** first. After importing the User model we can create a user by calling **`User.objects.create_user('username', 'email', 'password')`**. Once we have called **`create_user()`** our user will be created and stored in our database. The **`create_user()`** function also returns the newly created user object for us, allowing us to edit other attributes as seen at the bottom of the previous example, just before the return. Remember to save the changes you have made to your user.

Extra resource

If you are interested in learning more about the **User** model, you can see the **official documentation.**

Note when using user input for creating a user it is important to **verify** this input. You have to consider things such as existing usernames and how users create their passwords. If you have created an account on a web service before, you might recall that there are some requirements when choosing a password, such as a certain minimum length and/or one uppercase and lowercase letter; sometimes they even require a special character. When registering users on your web service, make sure that you implement verification checks that ensure users using your service are using secure passwords.

Admin users

If we want to create an admin user to use Django's built-in admin panel, we have to call the `createsuperuser` command.

In your terminal run the command:

```
python manage.py createsuperuser --username=your_username
--email=example@email.com
```

After running this command you will be prompted to enter a password for your superuser. You can also run the command without `--username` and `--email`, in which case you will be prompted to enter the values.

When you log in as a superuser, you will gain access to Django's admin control panel. This will provide you with a graphical representation of your application's data built from your models, enabling you to perform CRUD operations on your data.

Changing user passwords

To change the password of a user we can use the `set_password()` function.

set_password() is one of many methods available to us through the **User** class. Visit the link to see all the available **methods** of the **User** class.

```
from django.contrib.auth.models import User

def change_user_password(username, new_password):
    user = User.objects.get(username=username)
    user.set_password(new_password)
    user.save()
```

In the above code example, the user model is being imported from `django.contrib.auth.models`. Then, we retrieve a specific user by looking for them using their username. Once we have our user object, we can call our `set_password()` function to change their password. After we have applied our changes, we call the `save()` function to store the changes. This is a very basic implementation for changing a user password. In reality, before changing a user's password, we would probably want to perform some verification checks (such as asking for their previous password and determining whether it is correct before allowing the change to the new password).

You can also change a password through the command line using the command:

```
manage.py changepassword username
```

This will prompt you to enter a password twice, saving the new password if the two entries match.

Django passwords

Along with verifying whether a password is secure enough, we also have to consider the way we are storing our passwords. Let's suppose the security of our service has been breached and the hacker has stolen usernames, emails, and passwords from our database. This could affect our users on more services than just ours. If we have users that use the same password on our service as they do on others, they have effectively been compromised on all the services for which they use that password. This is where hashing and encryption come into play. These two techniques will allow us to store data in a way that makes it very difficult for malicious actors to steal our data, even if they manage to get through our security. Let's look at these two concepts more closely.

Encryption vs. hashing

Encryption is a technique where we hide our information by using a mathematical model to scramble and unscramble our data. When applying a very simple encryption we can change plaintext such as "username" into what is known as ciphertext, that looks something like this: 7GNBbxcdTglBk+Djon8obg==. If you want your encryption to be more secure, you can have the user provide a key that becomes a variable in your encryption model, to help scramble the data even more. If you take these steps, a malicious actor would need to gain access to the data and also have the correct key to decrypt the data.

Hashing allows us to use a mathematical function to convert digital data into an output string with a fixed length. This can be seen as a one-way conversation as it is extremely difficult to convert a hash back into the data from which it originated. When creating a hash from "mypassword" using the md2 hash algorithm, we get the hash "398b7381b88d8697d8c48359152548d0".

Considering these two techniques, we can conclude that encryption will be the preferred option for sensitive user data, and hashing can be applied to our users' passwords. After we encrypt data we can decrypt them as well. This will allow us to store our user information securely while still being able to allow retrieval and viewing by users with the correct permissions. On the other hand, with passwords, we never have to know what the actual password the user has entered is. We can store a hash of the user's password to their profile. When they try to log in, we can hash the password they enter and compare it to the hash stored on their profile. Hiding the user's password from even our application itself creates a very secure layer of protection for passwords that are being stored in our database.

Luckily for us, Django already stores our user passwords as hashes. Nonetheless, it is important to know about these methods, as you will not always have the luxury of working with high-level frameworks such as Django. In such a case, the responsibility will fall on you to make sure that your users' data are stored securely.

AUTHENTICATION AND LOGIN

Authentication

To authenticate users we can use the **`authenticate()`** function imported from **`django.contrib.auth`**. We can pass a username and password into the functions and authenticate the credentials. Here we can provide a form for our user to enter their username and password. We then pass the user-entered credentials to the authenticate functions for authentication.

In Part 1 - Authentication & Login\AuthLog\grabsomore\views.py

```
def login_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return HttpResponseRedirect(reverse('grabsomore:welcome'))
        else:
            return render(request, 'grabsomore/login.html', {'error':
'Invalid credentials'})
    return render(request, 'grabsomore/login.html')
```

After we have authenticated the user credentials and the authentication is successful, we can use the **`login()`** function imported from the same library as **`authenticate()`** **`django.contrib.auth`**. The **`login()`** function requires two arguments, an HttpRequest, and a user object. Once a user is logged in, their user id gets stored in a session using Django's session framework.

`HttpResponseRedirect` allows us to redirect a user to another URL. Using the reverse function takes the name of a URL as an argument and returns the related url.

Logout

To log a user out, we can simply call the **`logout()`** function imported from **`django.contrib.auth`**. When calling the function, we can pass our **`HttpRequest`** object as an argument and this will clear all the session data from the request. This way, multiple users can use the same browser and not have access to each other's session data as long as everyone logs out.

```
from django.contrib.auth import logout
from django.http import HttpResponseRedirect
from django.urls import reverse

def logout_user(request):
    if request.user is not None:
        logout(request)
        return HttpResponseRedirect(reverse('grabsomore:login'))
```

Restrict views to logged-in users

Now that we know how to log users in and out, how would we go about limiting their view access? One way of going about it is using the **`is_authenticated`** attribute of your user object attached to your HttpRequest. We can use a conditional statement to determine whether the user has been authenticated or not, and redirect them to the appropriate view.

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.urls import reverse

def welcome(request):
    if request.user.is_authenticated:
        return render(request, 'welcome.html')
    else:
        return HttpResponseRedirect(reverse(grabsomore:alterlogin'))
```

In the code above, we determine whether the user has been authenticated; if they are we render the welcome page for them, but if they are not, they will be redirected to the login page.

Another option would be to use the **`login_required()`** **decorator** imported from **`django.contrib.auth.decorators`**. A decorator is a function that takes another function as an argument and adds functionality or augments the function without changing it. Here we can add the **`login_required`** decorator above each view you would like to limit to only logged-in users. In the parenthesis, we can add the **`login_url`** argument to redirect a user who has not logged in to the login page.

```
from django.contrib.auth.decorators import login_required

@login_required(login_url=reverse_lazy('grabsomore:login'))
def welcome(request):
    return render(request, 'grabsomore/welcome.html')
```

USER GROUPS AND PERMISSIONS

If we were building an eCommerce service, we would need to consider different types of users, such as shoppers and sellers. We wouldn't want these users to have the same permissions. Shoppers should be able to add items from different sellers into their carts, check out their items, and leave reviews. Sellers should be able to add and remove inventory, create new product pages, and set prices for their products.

Permissions

Django provides us with four default permissions, namely add, change, delete, and view. By setting these permissions, we gain the ability to limit user access to data. It is important to note that if we assign users these permissions, they are not automatically restricted. We can use the permission as a flag to tell us if a user has a certain permission before we allow the requested action.

In our eCommerce service scenario, let's suppose we named the app eCommerce when we gave the command `python manage.py startapp eCommerce` and created a model called `product` representing the products in a store. The names for our permissions would be as follows:

- add: `eCommerce.add_products`
- change: `eCommerce.change_products`
- delete: `eCommerce.delete_products`
- view: `eCommerce.view_products`

`In Part 2 - User Groups and Permissions\AuthLog\eCommerce\models.py`

```
class Product(models.Model):
    name = models.CharField(max_length=100)         # Product name
    description = models.TextField(blank=True)      # Optional
description
    price = models.DecimalField(max_digits=10, decimal_places=2)  #
Product price
    stock = models.PositiveIntegerField()           # Quantity available

    def __str__(self):
        return self.name

    class Meta:
        permissions = [
            ("add_products", "Can add products"),
            ("change_products", "Can change products"),
            ("delete_products", "Can delete products"),
            ("view_products", "Can view products"),
        ]
```

Let's see how we can limit a user's access with their permissions. We first have to add the permission to the user.

When a new user is registered, we can assign them the necessary permissions.

In Part 2 - User Groups and Permissions\AuthLog\grabsomore\views.py

```
def register_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        email = request.POST.get('email')
        user = User.objects.create_user(username=username, 
password=password, email=email)
        login(request, user)

        try:
            permission = Permission.objects.get(codename='view_products', 
content_type__app_label='eCommerce')
            user.user_permissions.add(permission)
        except Permission.DoesNotExist:
            pass

        user.save()
        return render(request, 'grabsomore/welcome.html')
    return render(request, 'grabsomore/register.html')
```

In our register user function in the above example, we now add permission for our user to view products. When a user wants to view a product page, we can now determine whether they have the permission to take this action before we allow the request.

In Part 2 - User Groups and Permissions\AuthLog\eCommerce\views.py

```
def view_product_page(request):
    user = request.user
    if user.has_perm('eCommerce.view_product') or 
user.has_perm('eCommerce.view_products'):
        if request.method == 'POST':
            product_name = request.POST.get('product')
            if not product_name:
                return render(request, 'eCommerce/product_page.html',
{'error': 'No product name was given.'})
            try:
                product = Product.objects.get(name=product_name)
                return render(request, 'eCommerce/product_page.html',
{'product': product})
            except ObjectDoesNotExist:
                return render(request, 'eCommerce/product_page.html',
{'error': 'Product not found.'})
        return render(request, 'eCommerce/product_page.html')
    return render(request, 'eCommerce/product_page.html', {'error': 'You do
not have permission to view this product.'})
```

Here we use a conditional statement to determine whether our user has 'view' permission. If successful, we render the product page to the user.
We can do the same for our vendors. When we register a vendor, we can permit them to change products, using `user.user_permissions.add()` for them to be able to update their products.

```
user.user_permissions.add('eCommerce.view_product')
```

We can then use another conditional statement within our view to determine whether the requesting user has permission to change a product.

```
def change_product_price(request):
    user = request.user
    if user.has_perm('eCommerce.change_product') or
user.has_perm('eCommerce.change_products'):
        if request.method == 'POST':
            product_name = request.POST.get('product')
            new_price = request.POST.get('new_price')
            if not product_name or not new_price:
                return render(request, 'eCommerce/change_price.html')
            try:
                product = Product.objects.get(name=product_name)
                product.price = float(new_price)
                product.save()
                return HttpResponseRedirect(reverse('eCommerce:products'))
            except ValueError:
                return render(request, 'eCommerce/change_price.html')
            except ObjectDoesNotExist:
                return render(request, 'eCommerce/change_price.html')
        return render(request, 'eCommerce/change_price.html')
    return render(request, 'eCommerce/change_price.html')
```

Groups

To categorise users into specific roles, we can create groups. We can assign permissions to a group which will in turn assign these permissions to all the users belonging to the group. We can create a group using code, or we can use Django's admin panel to add new groups.

Using code, we have to import the `Group` model from `django.contrib.auth.models`, then call the `get_or_create()` method to create a group.

Open the Django shell and run this

```
from django.contrib.auth.models import Group
Vendors, created = Group.objects.get_or_create(name='Vendors')
```

After we have created our group, we can assign and remove permissions to and from the group using:

```
vendors.permissions.set([permission_list])
vendors.permissions.add(permission, permission, ...)
vendors.permissions.remove(permission, permission, ...)
vendors.permissions.clear()
```

We can now assign our `Vendors` group to accounts registered as vendors. This will automatically apply all the permissions of the group to the user.

```
user.groups.add(vendors)
```

With this knowledge we can change our **`register_user()`** function to add a user to their selected group.

```
def register_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        email = request.POST.get('email')
        user = User.objects.create_user(username=username,
password=password, email=email)
        try:
            vendors_group = Group.objects.get(name='Vendors')
            user.groups.add(vendors_group)
        except Group.DoesNotExist:
            pass
        try:
            permission = Permission.objects.get(codename='view_products',
content_type__app_label='eCommerce')
            user.user_permissions.add(permission)
        except Permission.DoesNotExist:
            pass
        user.save()
        login(request, user)
        return redirect(reverse('grabsomore:welcome'))
    return render(request, 'grabsomore/register.html')
```

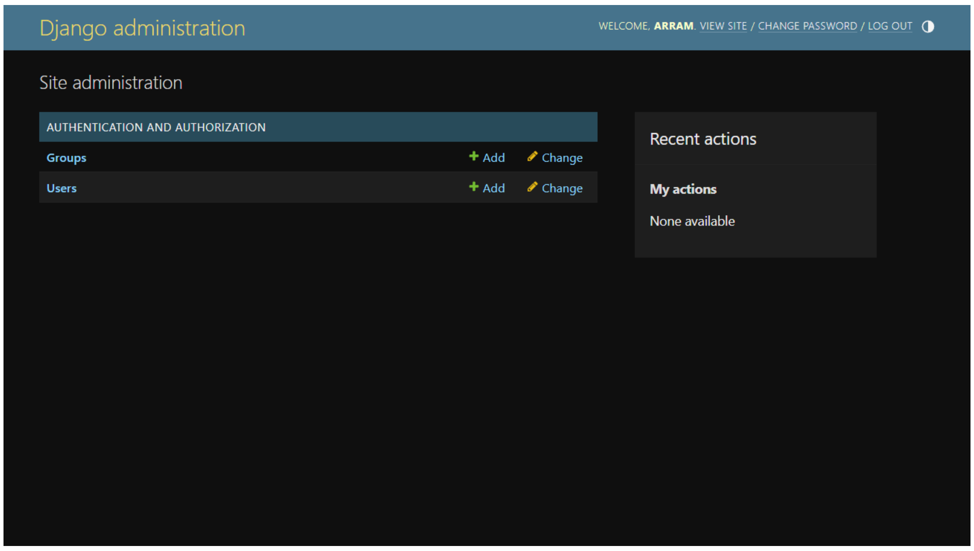
The functions now get some more information from the user, namely the **`account_type`**. This will tell us what type of account is being created, so we can assign the correct group(s) and permissions.

Make sure that you have created and made your migrations

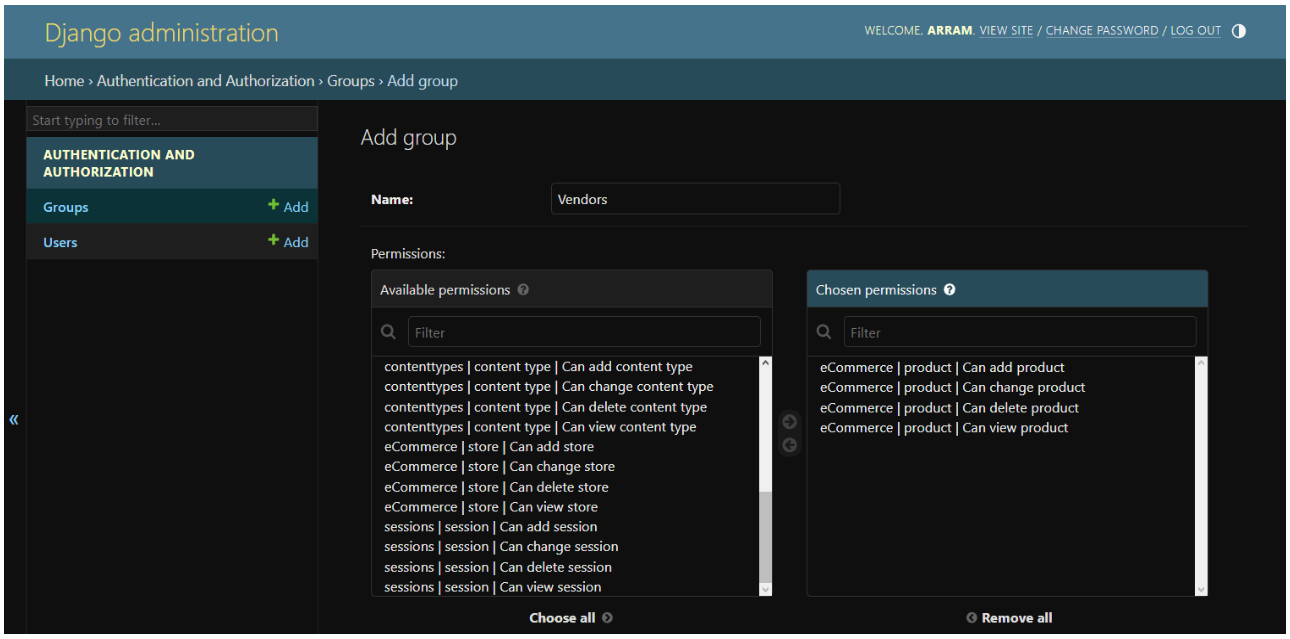
```
python manage.py makemigrations
```

```
python manage.py migrate
```

To create a new group using Django's Admin panel, you will then have to create a **superuser**. We can type **http://127.0.0.1:8000/admin** in our address bar. This will take us to a login page, to log in as an admin user. Enter your superuser credentials to log in, and you will be presented with a page that looks similar to this:



We can view all our groups by selecting `Groups`, or we can select `Add` to create a new group. When creating a new group, we are presented with this page:

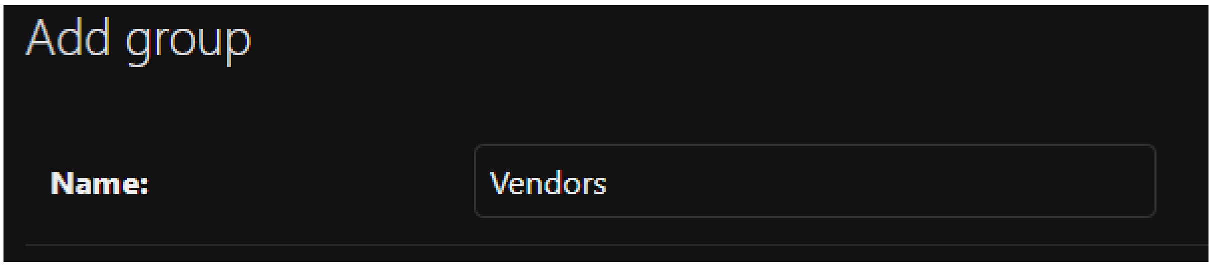

Django administration
WELCOME, ARRAM. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Authentication and Authorization › Groups › Add group
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
Add group
Name:
Permissions:
Available permissions
Filter
Chosen permissions
Filter
Choose all
Remove all

Name: Vendors

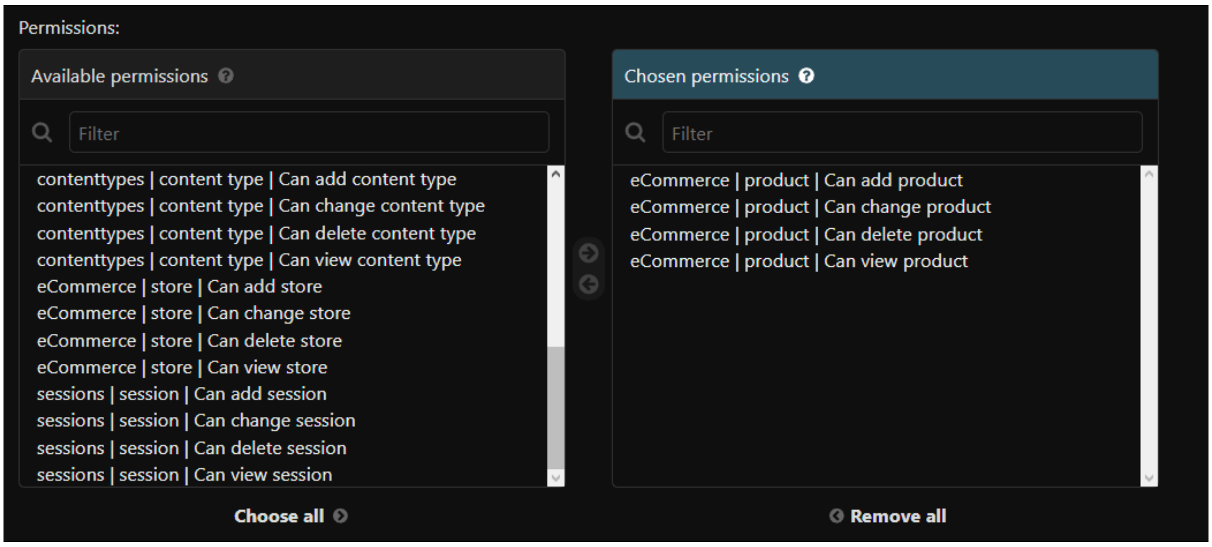
Available permissions:
contenttypes | content type | Can add content type
contenttypes | content type | Can change content type
contenttypes | content type | Can delete content type
contenttypes | content type | Can view content type
eCommerce | store | Can add store
eCommerce | store | Can change store
eCommerce | store | Can delete store
eCommerce | store | Can view store
sessions | session | Can add session
sessions | session | Can change session
sessions | session | Can delete session
sessions | session | Can view session

Chosen permissions:
eCommerce | product | Can add product
eCommerce | product | Can change product
eCommerce | product | Can delete product
eCommerce | product | Can view product

Here we can give our new group a name and assign the necessary permissions.


Add group

Name: Vendors


Permissions:

Available permissions

contenttypes | content type | Can add content type
contenttypes | content type | Can change content type
contenttypes | content type | Can delete content type
contenttypes | content type | Can view content type
eCommerce | store | Can add store
eCommerce | store | Can change store
eCommerce | store | Can delete store
eCommerce | store | Can view store
sessions | session | Can add session
sessions | session | Can change session
sessions | session | Can delete session
sessions | session | Can view session

Chosen permissions

eCommerce | product | Can add product
eCommerce | product | Can change product
eCommerce | product | Can delete product
eCommerce | product | Can view product

DATABASE MIGRATION

When working with users and groups, it is important to keep the MTV architecture in mind. Users and groups are both models that work similarly to the models you create. When we make changes to our models, we have to make sure we migrate our database for the changes to occur.

Until now we have been using a basic SQLite database in your project. As our eCommerce service grows, it would make more sense to have our database run on a different server than our eCommerce service. This will prevent our data handling and eCommerce services from affecting each other's efficiency.

For this project, we will be connecting our MariaDB server to our project. Note that there are other services we could have used instead, such as PostgreSQL, Oracle, and MySQL.

In the settings.py file of our project, we have to make a few changes to the `DATABASE` environment variable.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DatabaseName',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

We change the `ENGINE` from sqlite to mysql to connect to our MariaDB server. `NAME` refers to the name of our database. `USER` and `PASSWORD` are the username and password you use on your MariaDB server. When testing and running your database in your local environment you can add localhost as the `HOST` or you can add the server address for your database and the server port.

When we make changes to our models, we have to run the following command in the terminal:

```
python manage.py makemigrations appname
```

This will create a migrations file, containing the changes to our models, that will be applied the next time we migrate our database. We can then call the following command to migrate the database:

```
python manage.py migrate
```

Once this is done, all our migrations should be applied and our models should be working as intended.

SESSIONS

Now that we have our different users with their roles and permissions, we can have a look at how we can determine the state of our service and track some data.

Sessions are a mechanism we can use to enable data to persist across multiple requests and responses to and from a user. They give us a way for us to bypass the statelessness of HTTP, which is often necessary; for example, for web applications, it is beneficial to store some state data for things such as authentication, personalisation, or any other user-specific data.

To use sessions within our Django projects, we have to first ensure that we add the session middleware to the settings.py file of our project.

In **settings.py**, go to the `MIDDLEWARE` list and ensure it contains `django.contrib.sessions.middleware.SessionMiddleware`. Note that if you have created your project using `django-admin startproject` the session, middleware should be added automatically.

We have already made use of sessions when we logged users in and out. When we call the Django `login()` function we pass the request and the user to the function. The function then takes the user's ID and saves it in the session. This way we know when a user is logged in because they will have a user ID inside of their session.

It is important to keep in mind what `login` and `logout` do with the session. Any data inside an anonymous session before a user has logged in will be retained once the user logs in. However, when we call the `logout` function it will clear all the data from the current session, thus we have to retrieve the data we would like to keep from the session before we call the `logout` function.

Here is a basic example of how the login functions would use a session object.

```
def login_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)

        if user is not None:
            login(request, user)
            request.session['user_id'] = user.id
            request.session['username'] = user.username
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Your username and password didn't match.")
    return render(request, 'grabsomore/login.html')
```

We can see the function gets the username from the request object and retrieves the user from the database. . If we want to check whether a user is logged in, we simply have a look in the session to see if the user's ID is present.

A user might add items from multiple stores to their cart, or place items in their cart, leave and do something else, and then come back and continue adding items to their cart. To handle this case, we need a way to remember which items were in the cart. This is where Django's session framework will come in handy.

Sessions will allow us to store data on the server side and abstract the sending and receiving of cookies. When the user makes a request, a cookie will be attached to the request containing a session ID for the user's session. Now that we have a session and a way of accessing a session with an ID, we can create a dictionary or a list of the items the user had in their cart, and add the data to our session. When a user comes back to their cart, their session ID can be used to retrieve their cart items from the session data, and we can rebuild their cart with the items they had.

Let's suppose a user adds an item to their shopping cart. We can create a function that will add an item to the user's cart for us inside the session. We can then call this function inside our view, and pass the request object to it as an argument.

```python
from .models import Product

def add_item_to_cart(request):
    item = request.POST.get('item')
    quantity = request.POST.get('quantity')
    if not item or not quantity:
        return redirect('eCommerce/cart_page')
    try:
        quantity = int(quantity)
        if quantity < 1:
            quantity = 1
    except ValueError:
        quantity = 1

    cart = request.session.get('cart', {})
    if item in cart:
        cart[item] += quantity
    else:
        cart[item] = quantity

    request.session['cart'] = cart
    request.session.modified = True
    return redirect('eCommerce/cart_page.html')
```

Here, we get the session from the request object as well as the product name and quantity. We then check to see if there is a 'cart' in the session. If the cart exists we just add the new item to the cart, otherwise we create the cart in the session and add the new item to the cart. We then mark the session as modified to ensure it saves.

Now that we have all the items of a user stored in their session, they can freely navigate our service and go to different vendors and add all their items. When we need to retrieve our items we can use the data in the session to determine which items the user had in their cart.

```python
def retrieve_products(request):
    products = []
    session = request.session
    if 'cart' in session:
        for name, quantity in session['cart'].items():
            try:
                product = Product.objects.get(name=name)
                products.append({'product': product, 'quantity': quantity})
            except Product.DoesNotExist:
                pass
    return products
```

Here we built a function to retrieve the product from the cart for us. We create a list in which to store all the products, then we retrieve the current session. We first check to see whether there is a cart inside the session; if there is no cart we will return an empty list, whereas if there is a cart we will retrieve the necessary products from the database and create a list of the products with their quantities. We can then pass the list with all our products as context data to the HTML template we are rendering as seen in the code snippet below.

```
def show_user_cart(request):
    cart = retreive_products(request)
    return render(request, 'main_cart.html', {'cart':cart})
```

Now we can use the context data in our template to render each product in a table.

```
{% block content %}
<h2>Your cart:</h2>
<table>
    <tr>
        <th>Product name</th>
        <th>Quantity</th>
        <th>Price</th>
      </tr>
    {% for product in cart.items() %}
    <tr>
        <td>{{ product[0].name }}</td>
        <td>{{ product[1] }}</td>
        <td>{{ product[0].price }}</td>
    </tr>
    {% endfor %}
</table>
```

```
{% endblock content %}
```

Session expiration

Django sessions also allow us to choose when a session expires. We can select a time in seconds, choose a specific date and time, or have the session expire when the user closes their browser. We can do this by using the `set_expiry()` method from our session object.

Let's say we want a session to only last 10 minutes. We can use the following code:

```
request.session.set_expiry(600)
```

If we want the session to expire when the user closes their browser we can add a zero as the parameter.

```
request.session.set_expiry(0)
```

To have the session expire on a certain date we can create a datetime object and use it for the argument in `set_expiry()`.

In Part 2 - User Groups and Permissions\AuthLog\grabsomore\views.py

```
from datetime import datetime

def login_user(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)

            # Set session expiry to 30 Dec 2025
            exp_date = datetime(2025, 12, 30)
            request.session.set_expiry(exp_date)
```

This will set our session to expire on 30 December 2025. We set session expiry in the login function because that's when the session is created and active. This is the earliest point in the program run time where you can control how long the user stays logged in.

HANDLING USERS FORGETTING PASSWORDS

Emails

Django contains an `EmailMessage` class that will link with our email back end and send emails for us. Using these objects we can send a user an email with a link they can use to reset their password if they have forgotten it. To send an email we simply have to build an email object with our data such as the subject, body, sender email, and receiver email. After building our object, we can call the `send()` method to send our email.

In Part 2 - User Groups and Permissions\AuthLog\grabsomore\views.py

```
from django.core.mail import EmailMessage
```

```
def build_email(user, reset_url):
    subject = "Password Reset"
    user_email = user.email
    domain_email = "example@domain.com"
    body = f"Hi {user.username},\nHere is your link to reset your
password: {reset_url}"
    email = EmailMessage(subject, body, domain_email, [user_email])
    return email
```

We can build any type of email we would like to send to our users using the `EmailMessage` object.

When we want to allow a user to reset their password, there are a few more steps we have to take to make sure we do things securely and effectively. If the user cannot provide us with their password, they have to provide us with some information about their account. In most cases, this will be an email address. Using the user's email address, we can locate them in the database and build a URL for them to use to reset their password. We also need to keep in mind that the URL should not last forever, i.e., we want it to time out after a certain period for security. Knowing this, we would need to generate some sort of token that we assign to our user that they can use to reset their password. Let's see how this can be done.

The first thing we need is a token model that will keep track of all our tokens. We can give our tokens a user, a token value, an expiry date, and a flag telling us whether the token has been used or not.

In Part 2 - User Groups and Permissions\AuthLog\grabsomore\models.py

```
class ResetToken(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)
    token = models.CharField(max_length=500)
    expiry_date = models.DateTimeField()
    used = models.BooleanField(default=False)
```

Here is our model. We connect the token to a user using a foreign key. Then we add the other fields as required.

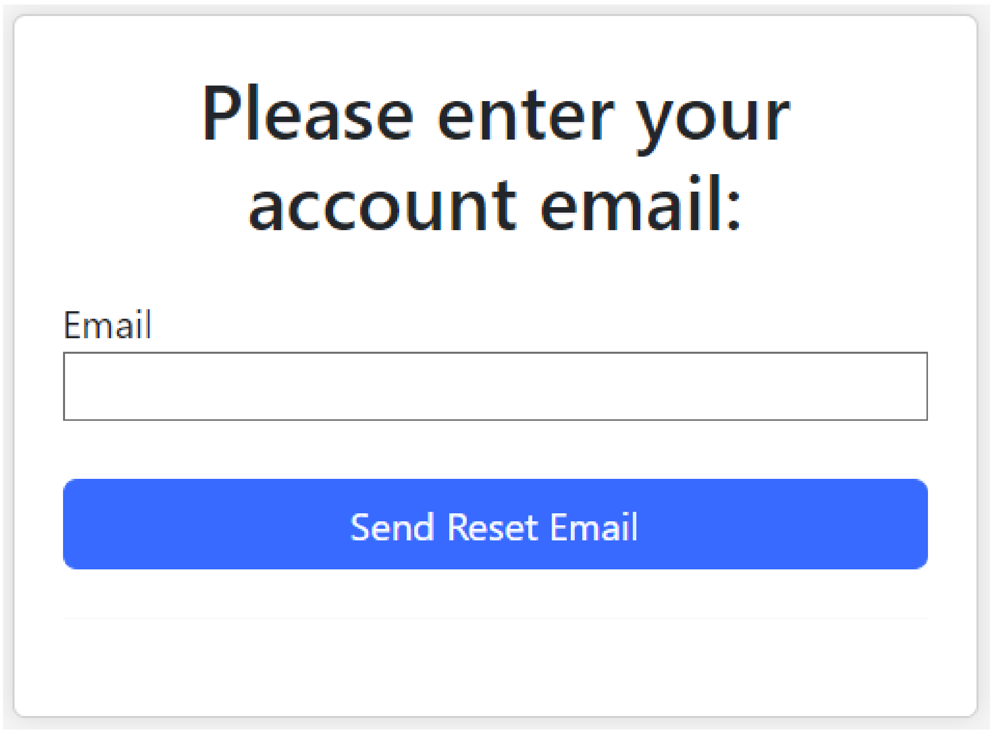
Now that we have a model to store our tokens, we can create a function to generate tokens and create a reset URL.

```python
import secrets
from datetime import datetime, timedelta
from hashlib import sha1


def generate_reset_url(user):
    domain = "http://127.0.0.1:8000/"
    app_name = "grabmore"
    url = f"{domain}{app_name}/reset_password/"
    token = str(secrets.token_urlsafe(16))
    expiry_date = datetime.now() + timedelta(minutes=5)
    reset_token = ResetToken.objects.create(user=user,
token=sha1(token.encode()).hexdigest(), expiry_date=expiry_date)
    url += f"{token}/"
    return url
```

To generate a token we need to import the secrets module from Python. Using “secrets” we can generate a URL-safe token. We then get the current data and time and set the expiry “date” to be five minutes after the user has requested the URL. Next, we create our ResetToken object. Finally, we attach the token to the URL and return the URL to the view which will send our email.

When a user forgets their password, we can present them with a page to enter their email address.



When the user clicks on “Send Reset Email” we can initiate the following code:

```
def send_password_reset(request):
    user_email = request.POST.get('email')
    user = User.objects.get(email=user_email)
    url = generate_reset_url(user)
    email = build_email(user, url)
    email.send()
    return HttpResponseRedirect(reverse(grabsomore:login'))
```

We grab their email from the form and locate the user in our database. Once we have our user object, we can pass it as an argument to the `generate_reset_url()` function we have created. After generating our URL we can pass our user and the URL to our `build_email()` function and, finally, send our email to the user.

Setting up email back end

For our `EmailMessage` object’s `send()` method to work we have to connect our project to an email back end. If you do not want to send real emails you can use the console or file back end. Using the console back end you can add the following environment variable to your settings.py file.

```
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

This will send all emails to your console where you can view them. If you would like to use the file back end, you can do the following.

```
EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = "/emails/"
```

In the `EMAIL_BACKEND` variable, we change `console` to `filebased` and we also add an extra environmental variable to set the path determining where we want to store our file.

To send real emails using an SMTP server such as Gmail, we can change our back end environment variable.

```
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_USE_TLS = True
EMAIL_PORT = 587
EMAIL_HOST_USER = 'appemail@gmail.com'
EMAIL_HOST_PASSWORD = 'your password'
```

We also have to create and set a few other variables. Our host is the SMTP server to which we want to connect. In this case, we will use the Gmail SMTP server. We set `EMAIL_USE_TLS` to `True` to create a secure connection and set the port to “587”. `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD` refer to the Gmail email and password you will be using.

It is considered bad practice to store your email credentials within your app as such it best to make use of an .env to store your details securely.

Dynamic URLs and password reset

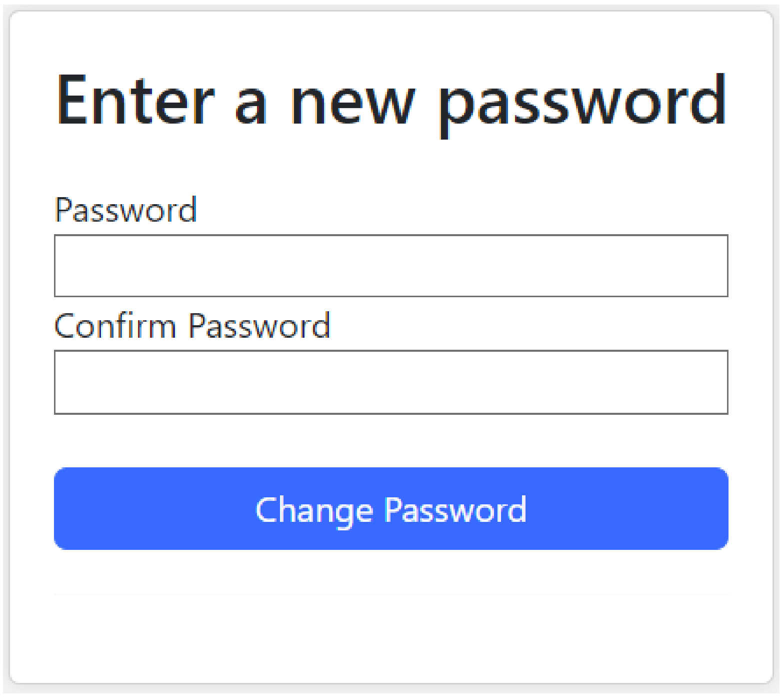
We have to add a dynamic URL in our urls.py file that will be used for our reset URL with tokens. We can add the following to our URL patterns:

```
path('reset_password/<str:token>/', reset_user_password,
name='password_reset')
```

When a user clicks on a reset URL it will contain a token after `reset_password/`; this token will be stored in a variable called `token` that we can use in our view to determine whether the user has a valid token.

```python
def reset_user_password(request, token):
    try:
        sha1(token.encode()).hexdigest()
        user_token =
ResetToken.objects.get(token=sha1(token.encode()).hexdigest())
        if user_token.expiry_date.replace(tzinfo=None) < datetime.now():
            user_token.delete()
        request.session['user'] = user_token.user.username
        request.session['token'] = token
    except:
        user_token = None
    return render(request, 'password_reset.html', {'token':user_token})
```

In the example above, you can see the extra parameter variable called `token`. This is where the token from our URL has been stored. The function hashes the token and locates it in the database. It then determines whether the token is expired or not. If the token is expired it will be deleted, whereas if it is not, both the token and the username connected to the token will be added to the session. The user will then be presented with a page to enter a new password, as follows:



We initiate the following view after the user clicks on “Change Password”.

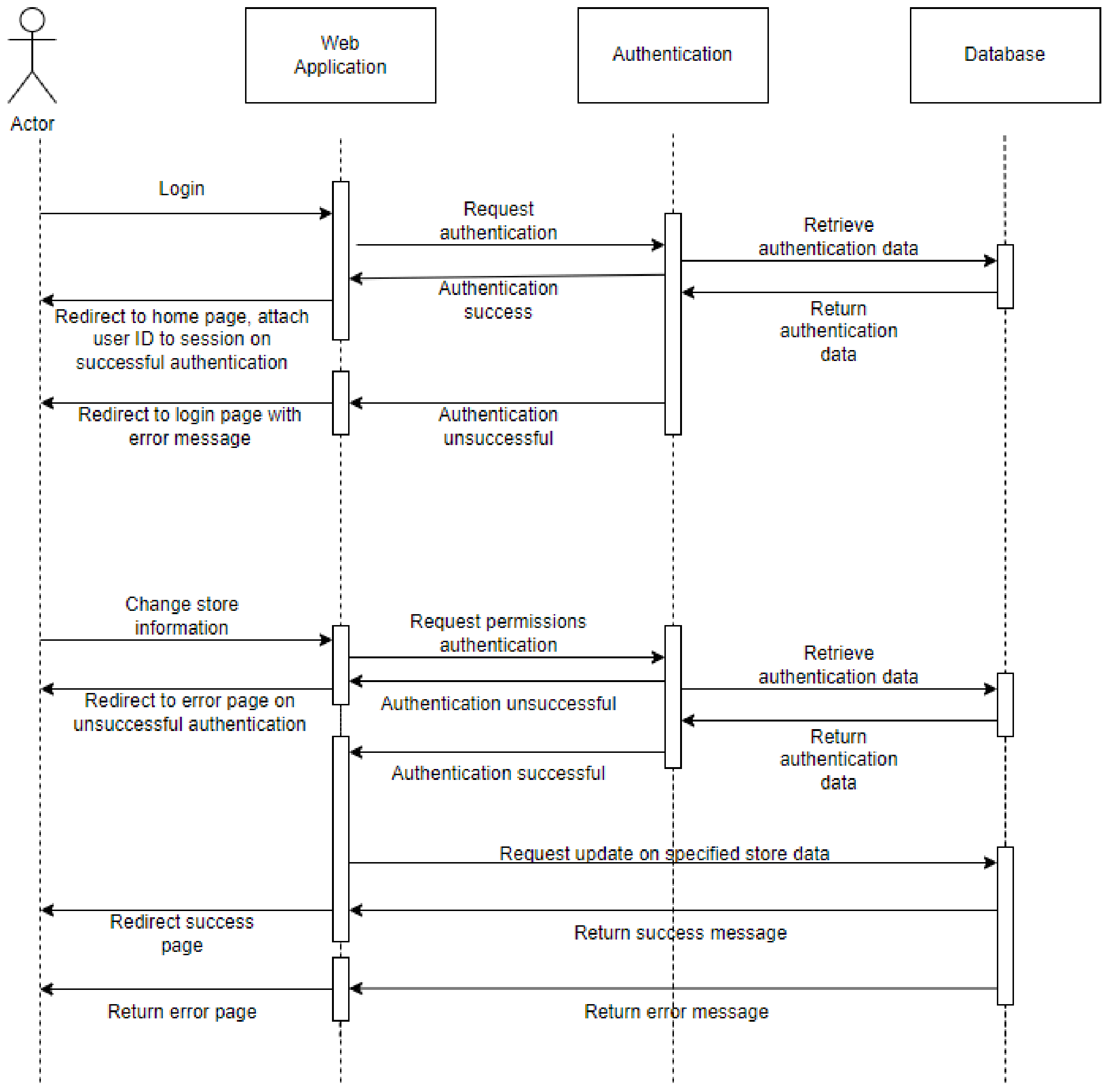
```
def reset_password(request):
    username = request.session['user']
    token = request.session['token']
    password = request.POST.get('password')
    password_conf = request.POST.get('password_conf')
    if password == password_conf:
        change_user_password(username, password)

ResetToken.objects.get(token=sha1(token.encode()).hexdigest()).delete()
        return HttpResponseRedirect(reverse(grabsomore:login'))
    else:
        return HttpResponseRedirect(reverse(grabsomore:password_reset'))
```

We grab the username and token from the session and the passwords entered from the form. We then compare the two passwords that user has entered. If they match, we will proceed to call our `change_user_password()` function from earlier to change the user's password. We will also delete the token to ensure that it cannot be used again. If they do not match, we will reload the page.

PROGRAM SEQUENCE

To better understand how our project's components will be working together we can visualise this with a sequence diagram.


Actor
Web Application
Authentication
Database
Login
Request authentication
Retrieve authentication data
Authentication success
Return authentication data
Redirect to home page, attach user ID to session on successful authentication
Redirect to login page with error message
Authentication unsuccessful
Change store information
Request permissions authentication
Retrieve authentication data
Redirect to error page on unsuccessful authentication
Authentication unsuccessful
Return authentication data
Authentication successful
Request update on specified store data
Redirect success page
Return success message
Return error page
Return error message

The above sequence diagram displays how our systems layers will interact with each other when a user requests to log in or change store information. The first sequence shows how the layers interact during login. The web app takes the credentials entered by the user and requests authentication from our authentication layer. Our authentication layer will then retrieve the necessary data from the database to compare against the user's entered credentials. If the user is successfully authenticated they are redirected to the home page and their user ID is added to the current session. If their authentication is unsuccessful they will be redirected back to the login page.

The second sequence shows the interaction between layers when a user requests to change information about a store. The web app will do the same as before and request authentication but this time we want to determine if our user has the correct permissions to make the request. The authentication layer will once again retrieve the authentication data from the database to perform authentication. If authentication of permissions is successful, the web app will request an update on the store data from the database layer. If the information is successfully updated, the database layer will return a success message telling the web app to redirect the user to the success page. If the information is not successfully updated the web app will receive an error message to return an error page for the user.

Practical task 1

In this task, you will build an eCommerce web app that will allow users to become vendors or buyers.

Before jumping into programming, it is prudent to plan the project first.

1. Identify the requirements of your system. Try thinking of the different users that will be using your system and how they will be interacting with this system.
2. Plan the layout of your system's user interface to provide a good user experience.
3. Layout how access to your program will be controlled and how you will secure your data.
4. You have to plan for failure as well. Consider your system and determine how the system should react or recover when unexpected behaviour occurs.
5. You can add all your planning into a folder named **Planning** inside your submission folder.

Follow these steps:

Create a new Django project using `django-admin startproject project_name`.
Create a new app using `python manage.py startapp appname` that will represent an eCommerce application.
Add the following functionality to your application.

1. Users should be able to register and log in as vendors or buyers.
    a. **Vendors**:

        - Stores: Vendors can create stores and should also be able to view, edit, and remove their stores.
        - Products: Vendors should be able to add, remove, edit, and view products in their stores as well.
    b. **Buyers**
        - Products: can view products from different stores, add them to their carts, and checkout.
        - Reviews: Buyers should also be able to leave reviews.
2. Remember to use sessions to keep track of a user's cart while they are browsing your store.
3. When a buyer checks out, your applications should remove the products from their respective carts, build an invoice of the items in the cart, and send it to the user's email..;
4. **Reviews**: **verified** and **unverified**. If a user has purchased a product and leaves a review for the product, it should be seen as a verified review. If they have not purchased a product item, they can still leave a review about that product, but it should be treated as an unverified review.
5. Allow users to recover a forgotten password and reset it by sending them an email with a password reset URL. Remember that the URL needs to expire for security. You will have to generate and keep track of tokens to manage this.
6. Use MariaDB or another relational database engine of your choice for your `DATABASE` back end, and perform database migration.
7. Use authentication and implement permissions to prevent users from accessing specific views and models based on their login status and permissions.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Practical task 2

There are some concepts we have not covered in this task. To get some practice researching and understanding technical concepts in the way you will need to do when working as a professional developer, please research the topics listed below in order to answer the questions about them. Note that you absolutely must not copy and paste, or even quote from a source – we want you to research these concepts until you actually understand them well enough to explain them in your own words.

1. Do some research on the Python `requests` module. Explain what it is, and how it is used to make HTTP requests.
2. Document your independent findings on the JSON and XML data formats and what they are used for. List at least four advantages and four disadvantages of each data format.
3. Provide a brief explanation of what a RESTful API is, how it works and what it is used for. List at least four advantages and four disadvantages of RESTful APIs.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.