Descrizione progetto

Creare un e-commerce stile Amazon per prodotti tecnologici con i seguenti requisiti fondamentali:

- Permette l'accesso a utenti non registrati, dando la possibilità di vedere i vari prodotti sul sito.
- Gli utenti registrati e loggati potranno invece fare acquisti, nel caso un prodotto non sia presente al momento della ricerca potranno decidere se ricevere una notifica nel momento in cui il prodotto torni disponibile, avendo inoltre la possibilità di vedere la propria lista ordini.
- Gli utenti registrati come venditori potranno inserire nuovi prodotti e monitorare come questi stanno andando, decidendo se rifornire i prodotti già messi in vendita.
- Tutti gli utenti potranno cercare tramite caratteristiche quali la marca, il prezzo,il nome del prodotto o la categoria (stampanti, PC, telefonini, etc etc...).

Modelli

users

ho deciso che per modellare la richiesta di diversi users nella stessa app con ruoli divesi ho preferito usare `AbstractUser` così come suggerito da django nella documentazione Using a custom user model when starting a project perché come spiegato

> Some kinds of projects may have authentication requirements for which Django's built-in User model is not always appropriate. For instance, on some sites it makes more sense to use an email address as your identification token instead of a username.

quindi decido di impostare come in molti e-commerce la mail come parametro unique `USERNAME_FIELD = 'email'`, quindi non avremo mai user con stessa email.

ma come spiegato nella guida un app riutilizzabile non dovrebbe mai implementare i propri modelli di user che andrebbero contro la filosofia DRY di Django

> Reusable apps shouldn't implement a custom user model. A project may use many apps, and two reusable apps that implemented a custom user model couldn't be used together. If you need to store per user information in your app, use a ForeignKey or OneToOneField to settings.AUTH_USER_MODEL as described below [. . .]

quindi ho ritenuto opportuno definire tutti i modelli in un applicazione esterna per la sola gestione e definizione degli account.

Inizialmente in `ShopUser` ho creato due fields booleani per modellare il requisito di avere due tipi di utenti, teoricamente ne sarebbe bastato uno essendo mutualmente esclusivi ma in una possibile previsione di un aumento delle tipologie utenti mi è sembrato opportuno non fare uso di Dummy variable ma avere una relazione uno a uno con la rappresentazione del dato. Quindi essendo i due dati mutualmente esclusivi ho fatto un override della funzione`clean()` che permette di fare verifiche sui dati.

Questa soluzione però per quanto inizialmente veloce si è rivelata non ottimale per quanto riguardava la modellazione degli accessi già fornita da Django tramite l'uso dei gruppi, ho quindi in fine deciso di ricorrere all'iscrizione programmatica dei membri, per fare ciò nella view per la creazione di un account inserisco:

```
class SignUpView(CreateView):
    model = settings.AUTH_USER_MODEL
    form_class = CustomUserCreationForm
    template_name = 'registration/signup.html'
    success_url = reverse_lazy('login')

    def form_valid(self, form):
        user = form.save(commit=False)
        user.save()#creo l'utente
        'lo inserisco nel gruppo da lui selezionato'
        account_type = form.cleaned_data.get('account_type')
        if account_type == 'customer':
            group, _ = Group.objects.get_or_create(name='Customers')
        else:
            group, _ = Group.objects.get_or_create(name='Vendors')
        user.groups.add(group)
        login(self.request, user)
        return super().form_valid(form)
```

quindi creo il gruppo se non è ancora mai stato creato e successivamente inserisco l'utente nel gruppo. La scelta del gruppo oltre a una maggiore Granularità è data dal fatto di poter garantire una scalabilità maggiore, creare nuovi gruppi è molto facile e nel caso se ne voglia aggiungere altri non vi è la necessità si fare una migrazione, cosa che, invece, con l'uso dei fields diventa più complesso.

Altre implementazioni dovute dalla scelta di aver usato un `AbstractUser` è che l'account admin potrebbe non funzionare soprattuto perché nel mio caso ho deciso di eliminare il campo `username` ritenendolo inutile ai fini di uno shop online, e come specificato dalla documentazione

> You should also define a custom manager for your user model. If your user model defines username, email, is_staff, is_active, is_superuser, last_login, and date_joined fields the same as Django's default user, you can install Django's UserManager; however, **if your user model defines different fields**, you'll need to define a custom manager that extends BaseUserManager providing two additional methods [. . .]

fatto questo l'errore `django.core.exceptions.FieldError: Unknown field(s) (username) specified for ShopUser` non dovrebbe esserci più.

Cart

Uno dei requisiti del progetto è:

> Gli utenti registrati e loggati potranno invece fare acquisti, nel caso un prodotto non sia presente al momento della ricerca potranno decidere se ricevere una notifica nel momento in cui il prodotto torni disponibile, **avendo inoltre la possibilità di vedere la propria lista ordini**.

per modellare l'ultima parte del requisito avevo inizialmente creato un modello in `shop/model.py`:

```
class Order(models.Model):
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
```

```
    customer = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)
    quantity = models.IntegerField()
    ordered_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Order {self.id}"
```

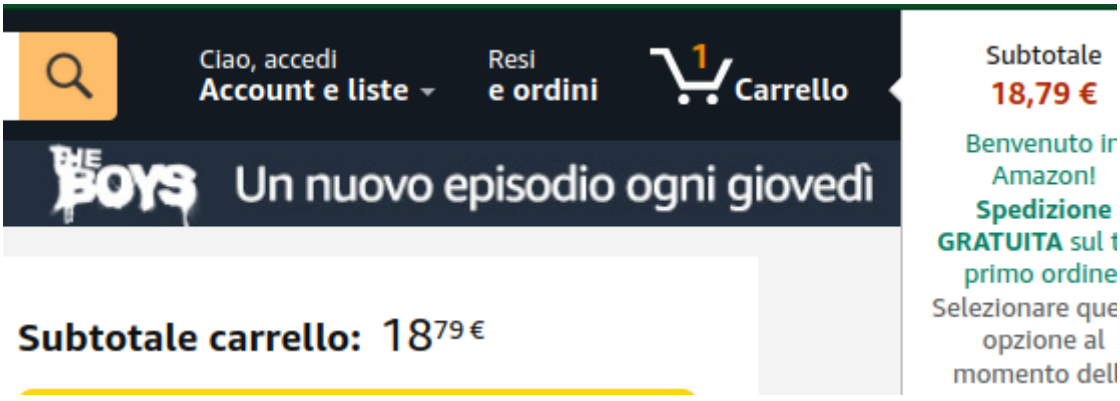
Intuitivamente all'inizio mi sembrò una soluzione ottimale ma mi sono accorto modellando i vari requisiti e le varie view che era una soluzione scadente, il modello teneva conto della quantità solo di un prodotto ma questo non si addice a un moderno ecommerce, quindi decido di implementare una funzione di carrello per creare ordini.

Decido quindi di modellare un carrello ma creando il modello in se mi sono accorto che tecnicamente non fosse una caratteristica della logica di shop ma avrei potuto creare un' applicazione a sè, con i suoi modelli e rispettando di più la logica secondo la quale Django differenzia le applicazioni.

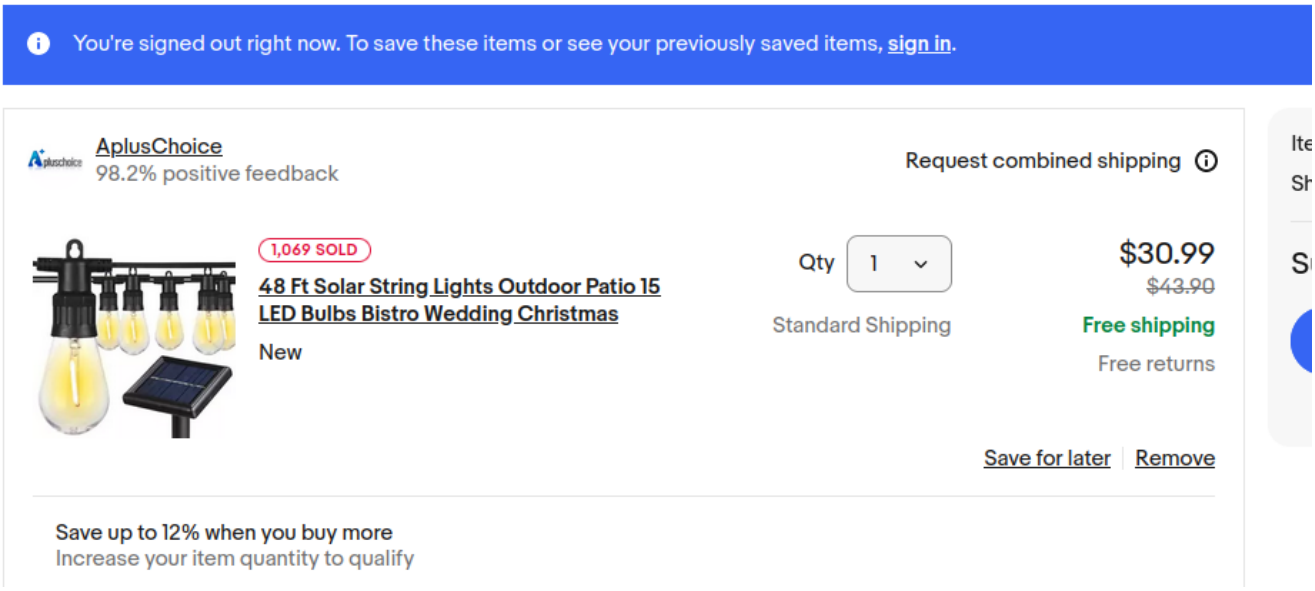
> The term application describes a Python package that provides some set of features. Applications may be reused in various projects.

perché pensandoci una logica di carrello potrebbe essere usata in divese applicazioni sfruttando sempre un entita generale "Prodotto".

Altre possibili valutazioni su come implementare e modellare un carrello è quale metodologia usare, inizialmente avevo pensato di usare una sessione per gestire l'inserimento dei prodotti in un carrello, perché molto leggera, si tratta di aggiungere un paio di funzioni, e soprattuto facendo delle ricerche su altri tipi di e-commerce mi è sembrato il modo più comune di implementare, dando la possibilità a tutti gli utenti di creare un carrello ma di poter effettivamente fare il checkout solo se loggati. Di seguito alcuni esempi:



esempio 1: prova d'acquisto su Amazon senza essere loggati



esempio 2: prova d'acquisto su Ebay senza essere loggati

questo ovviamente a livelli commerciali è ottimo un utente anonimo vede i prodotti interessati e li aggiunge al carrello senza dover pensare di registrarsi, cosa che verrà richiesta solo al momento del effettivo acquisto.

Il problema di questa soluzione è, che per quanto leggera ed efficace, non è esente da difetti, il principale tra tutti è che usarla per gli utenti con account comporta che se dovessero pulire la cache o semplicemente perdere la sessione si troverebbero il carrello svuotato. La soluzione a questo problema è stata un'implementazione ibrida tra l'uso della sessione che chiameremo `class SessionCart()` e gli effettivi modelli `CartItem` per la descrizine di un item nel carrello e `Cart` per l'implementazione effettiva di un carrello collegato a un utente.

Questo approccio complica la logica delle view che dovranno sempre fare attenzione a qual è il carrello che stanno mostrando dato che è come se ne avessimo due, uno gestito tramite sessione e uno presente sul database, ma mi è sembrato un tradeoff ottimale per aumentre l'esperienza utente.

Per implementare il tutto uso il tag `CART_SESSION_ID = 'cart'` in modo da avere accesso in tutto il progetto l'id della sessione al carrello, cosa che possiamo ottenere con `cart = self.session.get(settings.CART_SESSION_ID)` per invece assegnare al oggetto una sessione usiamo `self.session = request.session`

> HttpRequest.session: From the SessionMiddleware: A readable and writable, dictionary-like object that represents the current session.

Quindi avendo un ogetto collegato alla sessione potremmo salvare i prodotti nel carrello per fare questo usiamo `add(self, product, quantity=1, set_new_quantity=False)`e`save(self)`; la prima si occupa del aggiungere un prodotto nel carrello, `set_new_quantity` è usato in caso di un update di quantità, banalmente se si volesse diminuire la quantità di un prodotto. La funzione `save(self)` si assicura che l'oggetto sia stato effettivamente modificato perché

> By default, Django only saves to the session database when the session has been modified – that is if any of its dictionary values have been assigned or deleted:

e con `add(...)` andremmo solo a modificare dellle entries del dizionario.

fatto questo possiamo passare alla parte dove implementiamo i modelli che si colleghino al utente che risultenanno molto piccoli e più simili al concetto di un vero carrello

```
class Cart(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE, related_name='cart')

class CartItem(models.Model):
    cart = models.ForeignKey(Cart, on_delete=models.CASCADE,
related_name='items')
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(default=1)
```

gran parte della business logic di questi due modelli non si trova in `view.py` dato che abbiamo creato già un oggetto che implementa gran parte delle funzioni quindi risulteranno funzioni relativamente piccole. L'unica cosa che dovreanno fare e assicurarsi di sincronizare il database con la sessione nel caso l'utente sia autenticato, cosa che otterremo con questa funzione:

```
@login_required
def sync_cart(request):

    session_cart = SessionCart(request)
    cart, created = Cart.objects.get_or_create(user=request.user)

    for item in session_cart:
        cart_item, created = CartItem.objects.get_or_create(cart=cart,
product=item['product'])
        cart_item.quantity = item['quantity']
        cart_item.save()

    session_cart.clear()
    return cart
```

fatto questo il sistema degli ordini cambia leggermente implementando adesso due modelli non correlati al corrello che saranno gli ordini e i singoli prodotti interni agli ordini:

```
class Order(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)

    def __str__(self):
        return f'Order {self.id}'
```

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE,
related_name='items')
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(default=1)
    status = models.CharField(max_length=10, choices=StatusEnum.choices,
default=StatusEnum.PENDING)


    def __str__(self):
        return f'OrderItem {self.id}'
```

Gli stati del singolo item sono giustificati dal fatto che un venditore potrebbe essere in vacanza o meomentaneamente impossibilitato dal accedere al magazino quindi, voler confermare un ordine personalemente, cosa che potrà fare tramite la sua view peronalizzata. Una volta confermato l'ordine viene usato il modello 'Sale' per modellare il passaggio da ordine a ogetto effettivamente venduto, questo permette una maggiore usabilità per quanto riguarda controlli vari e da una maggiore granularità sulla gestione degli ordini per possibili implementazione future.

Ratings e Review

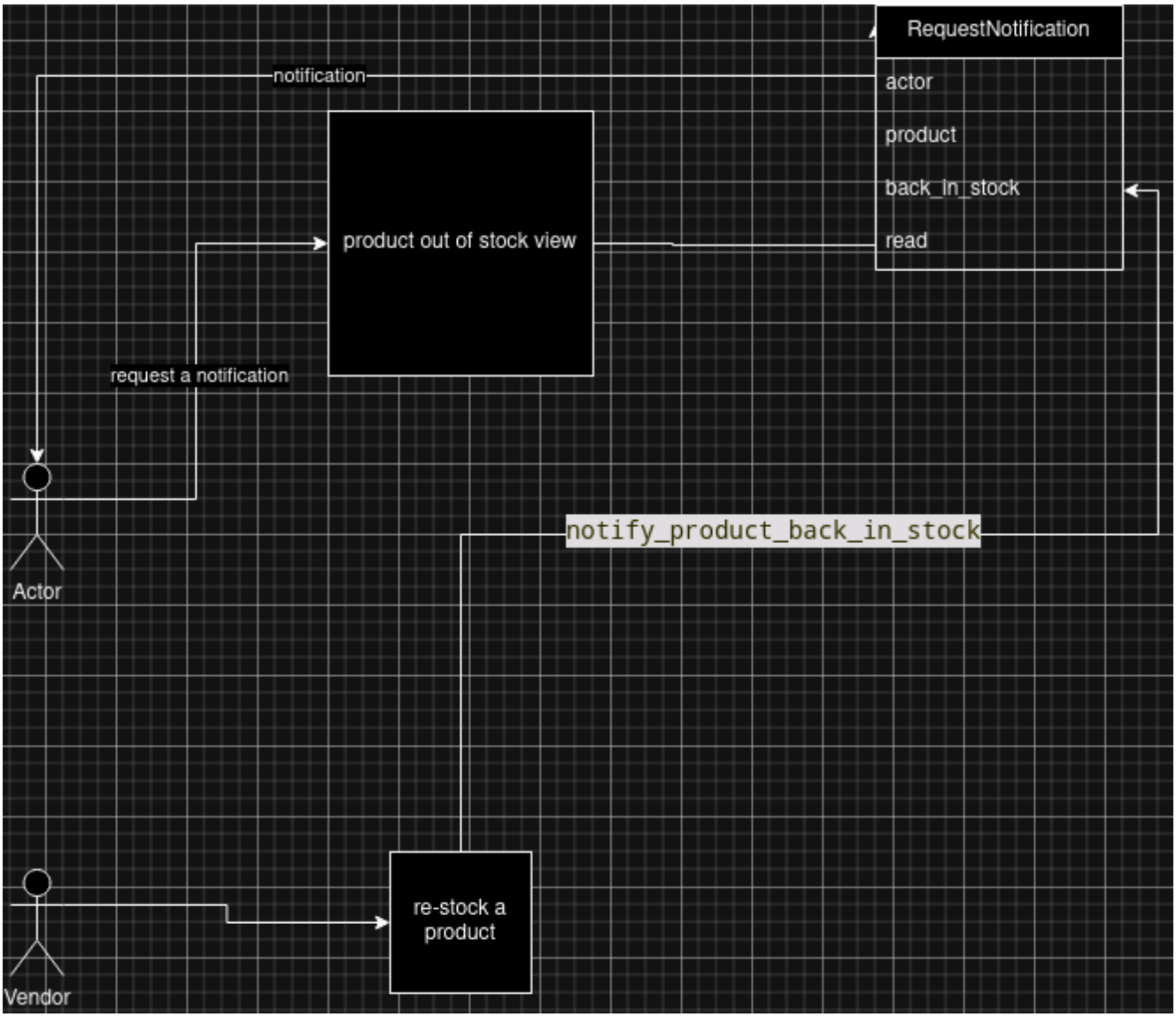
Altra caratteristica classica dei negozzi online è la possibilità di inserire review del prodotto dando in oltre una valutazione numerica, questo nel progetto è modellato tramite un altro modello `Review` che da la possibilità di inserire un valore numerico (rating) e un commento.

Notification system

> Gli utenti registrati e loggati potranno invece fare acquisti, **nel caso un prodotto non sia presente al momento della ricerca potranno decidere se ricevere una notifica nel momento in cui il prodotto torni disponibile**, avendo inoltre la possibilità di vedere la propria lista ordini.

Tornando invece al requisito precedentemente scritto va modellata la parte per le notifiche.

per fare questo è possibile usare `django.Signals`, la mia soluzione è stata creare due file `signals.py` e `handlers.py` uso il primo per dichiarare i segnali usati nel applicazione e il secondo per l'effettiva gestione del segnale e delle procedure. Lo schiema di funzionamento è come rappresentato in schema_1:



schema_1: schema funzionamento notifiche

Quindi un cliente della piattaforma che vuole comprare un oggetto non presente nel magazino, quindi gli viene offerta la possibilità di riciedere una notifica, questo crea una nuova entry nella tabella di RequestNotification. Quando il vendor deciderà di rimettere in magazino il prodotto viene eseguita la funzione

```
@receiver(product_restocked)
def notify_product_back_in_stock(product, **kwargs):

NotificationRequest.objects.filter(product=product).update(back_in_stock=Tr
ue)
```

che prende tutte le entries che hanno il prodotto uguale al prodotto cambiato e le aggiorna settanto `back_in_stock=True`.

Per la gestione delle notifiche volevo evitare che un singolo utente possa chiedere un numero indefinito di notifiche per un singolo elemento, così ho fatto in modo che una `NotificationRequest` richiede l'unicità

della coppia user-prodotto per fare questo si sfrutta la classe meta dei modelli e il campo `unique_together`

```
class NotificationRequest(models.Model):
    ...
    ...
    class Meta:
            verbose_name_plural = "notifications"
            unique_together = (('product', 'user'),)
```

questo però potrebbe portare a un problema, un utente che ha chiesto una notifica su un prodotto per la seconda volta potrebbe non riuscire. Come sistema per evitare questa eveninza ho implemetato che se una `NotificationRequest` viene segnata come letta viene successivamente eliminata; questo evita di occupare spazio inutile per notifiche già lette e ovviamente permette di richiedere nuovamente una notifica.

Altra implementazione da fare per evitare il server sollevi un eccezione e vada in crash in caso un utente provi a chiedere più volte una notifica allo stesso elemento è usare le eccezioni fornite da django, in questo caso database Exeption nel mio caso usate come segue nella view di notification request:

```
try:
            NotificationRequest.objects.create(product=product,
user=request.user)
            messages.success(request, 'You have successfully requested a
notification for this product.')
        except IntegrityError:
            messages.error(request, 'You have already requested a
notification for this product.')
```

Sales

Non essendo previsto un vero sistema di pagamento, se non tramite bonifico quindi esterno al sito stesso il modello `Sale` rappresenta una spedizione effettuata, dato che una volta effettuata la spedizione vine gestita da un corriere che avrà le sue logiche. Ho ceato questo modello anche se Definitivamente non strettamente utile per motivi pratici, è più facile fare delle ricerche senza dover filtrare l'orderItem che presenta il proprio stato in confirmed, inoltre mi è stato molto utile per la creazione dei grafici:

```
data = [Sale.objects.filter(sold_item__product=product)
                .aggregate(total_quantity=Sum(F('sold_item__quantity')))
['total_quantity'] or 0
                for product in products]
```

l'uso di questo modello permette, inoltre, una maggiore rappresentazione dei vari possibili dati ed è sicuramente utile per piani futuri.

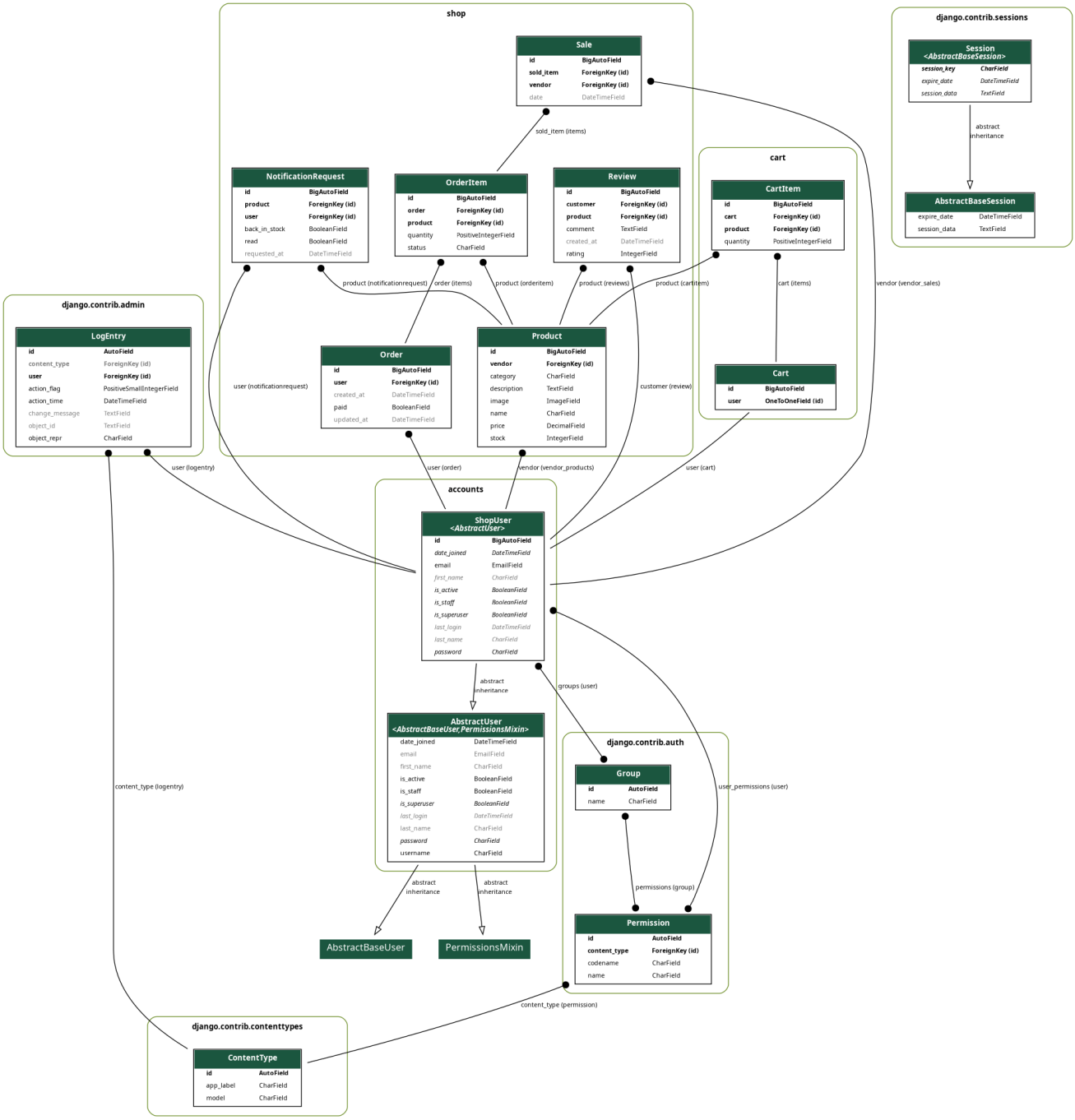
overview finale sui modelli

Per fornire un'overview sul progetto e i vari modelli appena discussi è possibile usare il comando

```
./manage.py graph_models -a -g --dot -o view_tech_shop.dot
 dot -Tpng view_tech_shop.dot -oview_tech_shop.png
```

insttallando però prima l'opportuna applicazione `django-extension`

il risultato finale è quanto segue in *schema_2* con i vari modelli raggruppati per applicazione



schema_2: overview sul progetto con tutti i modelli e applicazioni create e usate

Aspetti tecnici e altre scelte progettuali

raccomandation system

Anche se molto elementri sono presenti dei Raccomandation system, uno nella main page che raccomanda a tutti gli utenti gli oggetti organizzati per la quantità di vendite totalizzati

```
products = (Product.objects

.annotate(total_sales=Sum(F('orderitem__quantity')))
                            .order_by('-total_sales'))
```

per fare questo uso una F( ) expression, operazioni sul database che permettono di non caricare tutti i dati del database in memoria, in questo caso recupero da `OrderItem` la quantità, le sommo e le annoto per ogni Prodotto per poi ordinarle in modo discendente per la quantità appena creata.

Altro elementare raccomandation system è implementato e se l'utente è autenticato va a controllare la categoria di prodotto più ordinate dal utente

```
most_purchased_category = (OrderItem.objects
                                    .filter(order__user=request.user)
                                    .values('product__category')

.annotate(total=Count('product__category'))
                                    .order_by('-total')
                                    .first())
```

per fare questo, una volta selezionati tutti i suoi ordini creo un query dictionary con `values()` dove selezioniamo le categorie dei prodotti, conta le occorrenze di tutte le categorie, le ordina e prende solo la prima

fatto questo possiamo effettimvamente prendere i più venduti di quella categoria con una query simile a quella precedentemente usata per i prodotti più venduti.

```
for_you_products = (Product.objects
                                .filter(category=category)

.annotate(total_sold=Sum(F('orderitem__quantity')))
                                .order_by('-total_sold')[:10])
```

Altro elemnto di raccomandation system è nella view di dettaglio del prodotto dove vengono suggeriti altri prodotti con la stessa categoria, ovviamente escludendo il prodotto stesso.

```
related_products = (Product.objects
                                .filter(category=product.category)
                                .exclude(pk=pk)

.annotate(total_sold=Sum(F('orderitem__quantity')))
                                .order_by('-total_sold')[:4])
```

Strumenti di ricerca e filtraggio

In tutte le pagine vi è un barra di ricerca, l'implementazione backend è basata sul uso dei Q() object utili per l'uso di query complesse Dalla documentazione

> Keyword argument queries – in filter(), etc. – are “AND”ed together. If you need to execute more complex queries (for example, queries with OR statements), you can use Q objects. che è quello che ho trovato più utile per dellle ricerche su più parametri

```
class SearchResultsView(View):
    template_name = 'shop/search_results.html'

    def get(self, request):
        form = SearchForm(request.GET)
        query = request.GET.get('query')
        results = []

        if query:
            results = Product.objects.filter(
                Q(name__icontains=query)
                | Q(description__icontains=query)
                | Q(category__icontains=query)
                ).distinct()
        context={'form': form,
                 'query': query,
                 'results': results}

        return render(request, self.template_name, context)
```

viene ricercata la query data dal utente su contemporaneamente:

- name
- description
- category

per evitare duplicati (essendo tutte in OR) ho applicato `distinct()` al risultato finale.

Ho preferito questo approccio piuttosto che uno basato su un form perché, a mio parere, rimuove molta della complessità dal utente, se l'utente vuole una specifica di prodotto la trovrà cercando.

Tra le altre tecniche di ricerca offerte ci sono i filtri offerti nella main page; permettono di filtrare usando la categoria e organizzare secondo prezzo decrescente.

Grafici

per soddisfare il requisito:

> Gli utenti registrati come venditori potranno inserire nuovi prodotti e monitorare come questi stanno andando, decidendo se rifornire i prodotti già messi in vendita.

Ho deciso di rappresentare in dei grafici a barre i prodotti e rappresentare la quantità di prodotti venduti affiancata alla quantita di richieste attive di re-stock, in modo che il venditore abbia la possibilità di decidere la quantità di prodotti da mettere in re-stock

tecnologie utilizate

- Per la gestione delle immaggini è stato necessario usare Pillow
- Ajax è stato usato per rendere le pagine più dinamiche e reattive, è stato usato nel implementazione del contatore per il notification system:

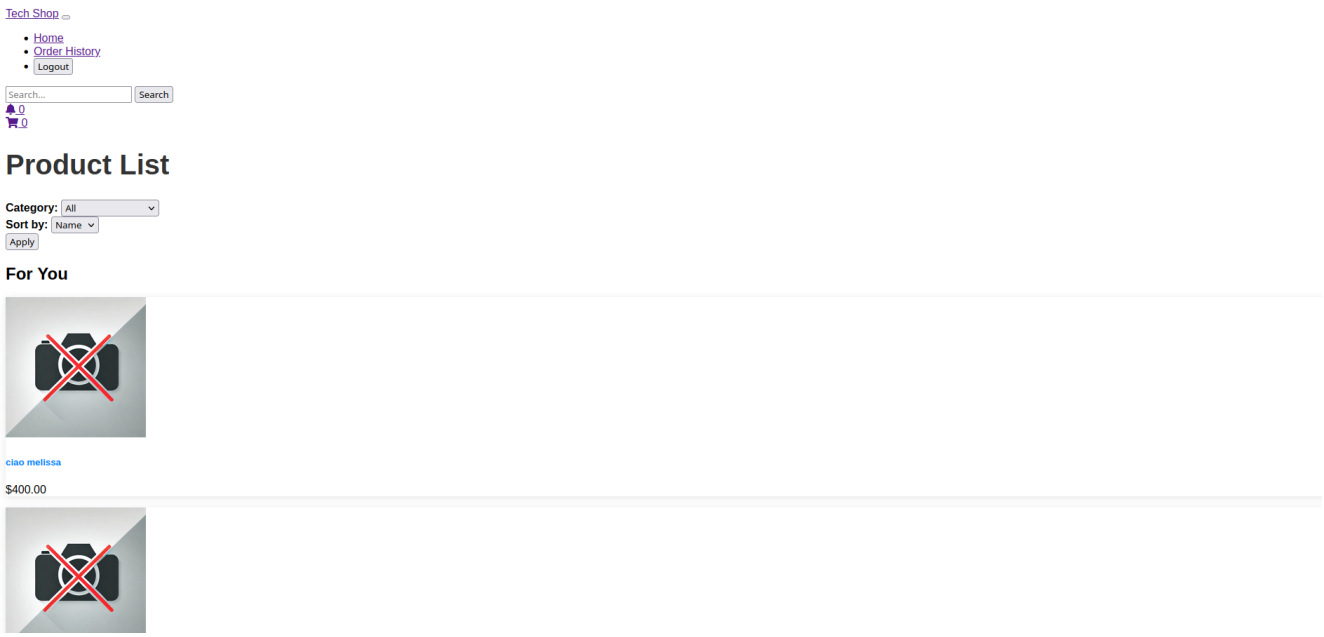
```
$(document).ready(function() {
            function updateNotificationCount() {
                $.ajax({
                    url: '{% url "unread_notification_count" %}',
                    method: 'GET',
                    success: function(data) {
                        $('#notification-count').text(data.unread_count);
                    }
                });
            }


            updateNotificationCount();

            setInterval(updateNotificationCount, 6000);
        });
```

o per la rimozione in modo dinamico senza dover ricorrere al refresh delle notifiche una volta lette:

```
    $(document).ready(function() {
        function deleteReadNotifications() {
            $.ajax({
                url: '{% url "delete_read_notifications" %}',
                method: 'POST',
                headers: {'X-CSRFToken': '{{ csrf_token }}'},
                success: function(response) {

                    if(response.success) {

                        response.removed_ids.forEach(function(id) {
                            $("#notification-" + id).remove();
                        });
                    } else {
                        console.error("Failed to delete read
notifications");
                    }
                },
                error: function(xhr, status, error) {
                    console.error("Error occurred: " + status + ", " +
error);
```

```
                }
            });
        }


        deleteReadNotifications();


        setInterval(deleteReadNotifications, 600);
    });
```

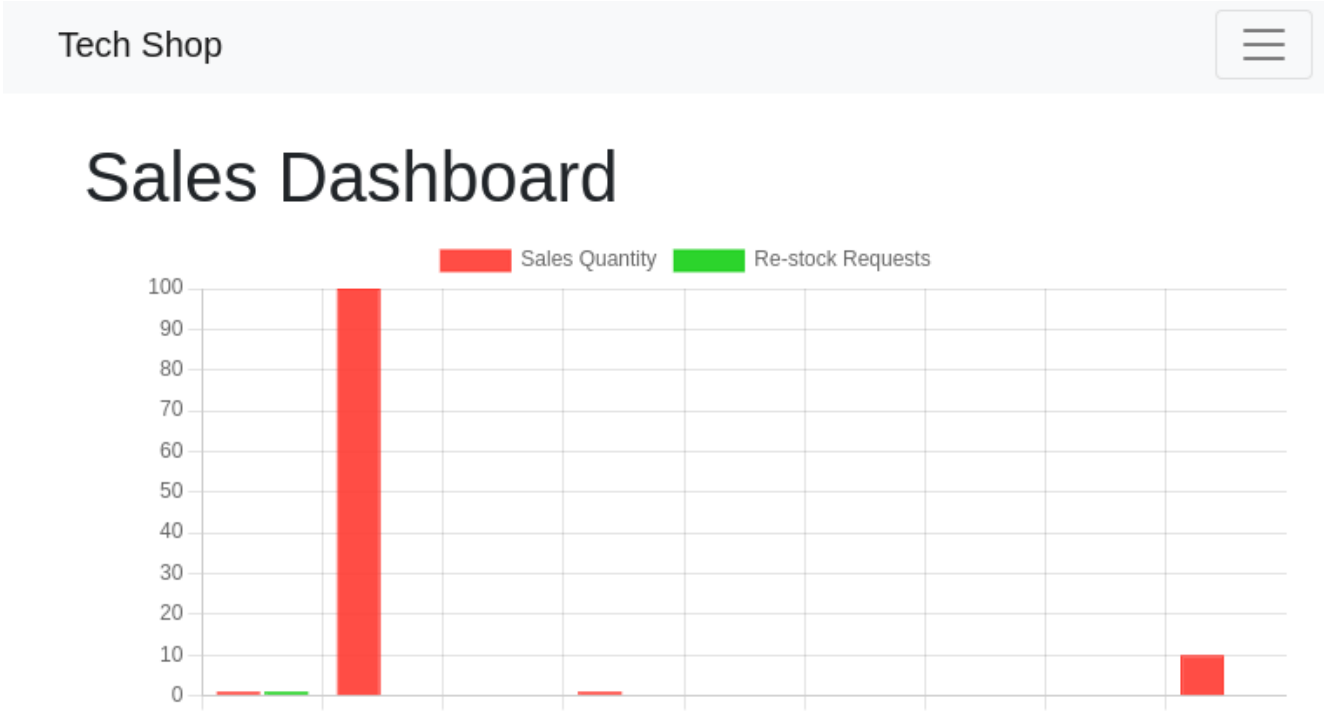
- Bootstrap per rendere più accattivante lo stile:



---

esempio_3: prova della homepage disattivando Bootsrap

- chartJs per la creazione dei grafici



esempio_4: l'uso di chartJs nel sito techshop

Test

I test effettuati sono 10 5 sugli account e 5 nel applicazione shop per testare il carrello

- test_signup_view_get:
    - semplice etst per ssicurarmi che la pagina log-in funzioni
- test_signup_view_post_customer:
    - mi assicuro che il processo di creazione account per il customer funzioni e che venga aggiunto nel gruppo corretto
- test_signup_view_post_vendor
    - similmente al test precedente mi assicuro che il processo di creazione account per il vendors funzioni e che venga aggiunto nel gruppo corretto
- test_login_view_get
    - test per la pagina di login
- test_login_view_post
    - test per corretta redirezione
- test_access_view_as_authenticated_customer
    - mi assicuro avvengano i corretti redirect
- test_access_view_as_authenticated_vendor

    - mi assicuro avvengano i corretti redirect
- test_access_view_as_unauthenticated_user
    - mi assicuro avvengano i corretti redirect
- test_create_order_with_sufficient_stock
    - mi assicuro che funzioni correttamente la creazione di un ordine e che il risultato del prodotto in stock venga alterato correttamente
- test_create_order_with_insufficient_stock
    - mi assicuro che ritorni un ordine con i messaggi di errore dovuti e che il prodotto in stock non venga alterato