
Sylwester Walczak
Nowoczesne
Django
Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/nowdja
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-9348-6

Printed in Poland.

Spis treści

8. Złóżmy zamówienie!

W tym rozdziale stworzymy logikę odpowiedzialną za przyjmowanie zamówień oraz procesy poboczne. Już niewiele dzieli nas od rozpoczęcia pracy nad podziałem monolitu.

Tworzenie modelu

Pracę rozpoczniemy od omówienia, czego właściwie potrzebujemy od tego modułu. Najważniejsza funkcja to oczywiście widok CRUD dla naszego modelu. Jednak zanim zaczniemy tworzyć widok, musimy stworzyć model. Jak myślisz, jakie informacje należy umieścić w modelu? Na pewno przydałaby się wiadomość o tym, co jest zamawiane oraz jaki jest status danego zamówienia. Do tego warto by było wiedzieć, czyje jest dane zamówienie, więc musimy zidentyfikować klienta z konkretnym zamówieniem. Dobrze będzie również sporządzać informacje o dacie, a przede wszystkim godzinie składania zamówień, aby w przyszłości móc szacować te dane. Informacja o osobie tworzącej zamówienie także wydaje się przydana. Zastanówmy się teraz, jakie procesy powinny zachodzić podczas składania takiej informacji.

Na pewno potrzebne jest zarezerwowanie zasobów, czyli podczas składania zamówienia jakaś ilość składników powinna zostać usunięta z naszego magazynu. Musimy również przewidzieć sytuację, w której ktoś zamówi swoją kawę i z jakiegoś powodu zrezygnuje z zakupu jeszcze przed rozpoczęciem procesu produkcji. Dla magazynu oznacza to, że zasoby nie zostały wykorzystane.

Kolejny proces to przekazanie aktualnej listy zamówień wraz z ich statusem do załogi oraz mniej szczegółowej listy np. do odpowiedniego wyświetlacza w kawiarni.

Musimy też umożliwić każdemu pracownikowi edycję danego zamówienia. Nie zawsze będzie chodzić o zmianę przedmiotów zamówienia, a np. o jego status.

Dobrze, skoro wiemy, czego potrzebujemy, weźmy się do roboty! Jak zawsze, zacznijmy od wpisania odpowiedniej komendy w terminalu, by utworzyć nowe aplikacje. Ja wybrałem nazwę `purchase`.

```
> docker exec -it djangomicro_mono_django_1 python manage.py startapp purchase
```

Następnie musimy zmienić właściciela nowo utworzonego katalogu:

```
> sudo chown -R $USER:$USER purchase/
```

Kolejny krok to dodanie aplikacji do listy zainstalowanych aplikacji w całym programie poprzez edycję pliku *settings.py* w katalogu *djangomono*:

```
#djangomono/settings.py
INSTALLED_APPS = [
    ...
    #own app
    'authx',
    'supplier',
    'story',
    'menu',
    'purchase'
]
```

Musimy też umieścić naszą nową aplikację w API. Edytuj plik *api.py* w tym samym katalogu:

```
#djangomono/api.py
from django.urls import path
from django.urls.conf import include

urlpatterns = [
    path('authx/', include('authx.urls')),
    path('supplier/', include('supplier.urls')),
    path('story/', include('story.urls')),
    path('menu/', include('menu.urls')),
    path('purchase/', include('purchase.urls')),
]
```

Aby uniknąć wyświetlenia komunikatu o błędzie, dodaj plik *urls.py* w katalogu *purchase* i umieść w nim poniższy kod:

```
#purchase/urls.py
from django.urls.conf import include
from django.urls import re_path

from rest_framework.routers import DefaultRouter
router = DefaultRouter()


urlpatterns = [
    re_path(r'', include(router.urls)),
]
```

Nasza nowo utworzona aplikacja została dodana. Następie musimy utworzyć model. W tym celu w pliku *models.py* umieść poniższy kod:

```
#purchase/models.py
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.contrib.auth import get_user_model

User = get_user_model()


class PurchaseOrder(models.Model):
    STATUS_CHOICE = (
        (1, _("New")),  #Nowe
        (2, _("Pending")),  #W przygotowaniu
        (3, _("Ready")),  #Gotowe
```

```
        (4, _("Retrieved")),  #Odebrane
    )
    items = models.ManyToManyField("menu.MenuItem")
    status = models.PositiveIntegerField(
        choices=STATUS_CHOICE,
        default=1
    )
    order_number = models.CharField(max_length=50)
    client_name = models.CharField(max_length=50, blank=True, null=True)
    created_date = models.DateTimeField(auto_now_add=True)
    update_date = models.DateTimeField(auto_now=True)
    created_by = models.ForeignKey(
        User,
        on_delete=models.SET_NULL,
        blank=True,
        null=True
    )

    def __str__(self):
        return f"{self.order_number} – {self.get_status_display()}"
```

Zaimportowaliśmy odpowiednie modele i utworzyliśmy klasę `PurchaseOrder`, która będzie zawierała informacje o zamówieniu. Następnie stworzyliśmy krotkę o nazwie `STATUS_CHOICE`, przetrzymującą statusy zamówienia. Wykorzystamy ją za chwilę. Pole `items` to pole „wiele do wielu" i możemy dzięki niemu zamawiać wszystkie przedmioty z menu. Pole `status` jest odpowiedzialne za przetrzymywanie informacji o aktualnym statusie zamówienia. To właśnie pole jest ograniczone do wyboru kliku elementów z wcześniej utworzonej krotki `STATUS_CHOICE`. Domyślnie każde zamówienie dostaje status `New`. Następne dwa pola, `order_number` oraz `client_name`, służą do trzymania informacji o kliencie. Pierwsze pole będzie generować się automatycznie podczas tworzenia zamówienia, natomiast drugie służy do opcjonalnego zapisania imienia klienta w sytuacji, gdy polityka firmy będzie wymagała wywoływania klienta za pomocą jego imienia. Pola `created_date` oraz `update_date` służą do przetrzymywania informacji o utworzeniu zamówienia oraz dacie jego aktualizacji. Data dodaje się automatycznie w momencie stworzenia zamówienia, zaś drugie pole aktualizuje się za każdym razem w momencie edycji danego zamówienia. Ostatnie pole, `created_by`, reprezentuje użytkownika, który utworzył zamówienie. Gdyby właściciel kawiarni zdecydował się na ustawienie kiosku do przyjmowania zamówień, mógłby stworzyć użytkownika *kiosk*, dzięki któremu klienci mogliby składać swoje zamówienia.

Ostatecznie za pomocą ciągu tekstowego nadpisaliśmy reprezentację klasy numerem zamówienia wraz z jego statusem.

Jak pewnie zauważyłeś, nasz system posiada pewną wadę. Mianowicie nie umożliwia wyboru więcej niż jednego produktu z tej samej kategorii w ramach jednego zamówienia. Jak rozwiązałbyś ten problem? Drobna podpowiedź: już się zmierzyliśmy z takim problemem w jednym z poprzednich rozdziałów.

Jak już wiesz, teraz musimy przeprowadzić migracje naszego nowo utworzonego modelu. W tym celu w terminalu wykonaj poniższe polecenia:

```
> docker exec -it djangomicro_mono_django_1 python manage.py makemigrations
Migrations for 'purchase':
  purchase/migrations/0001_initial.py
    - Create model PurchaseOrder
> docker exec -it djangomicro_mono_django_1 python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, authx, contenttypes, menu, purchase, sessions,
  ↪story, supplier
Running migrations:
  Applying purchase.0001_initial... OK
```

Następnie w pliku *admin.py* dodaj model tak, byśmy bez posiadania API mogli utworzyć wstępnie jakieś obiekty.

```
#purchase/admin.py
from django.contrib import admin
from . models import PurchaseOrder

admin.site.register(PurchaseOrder)
```

Po dodaniu kilku zamówień za pomocą panelu administracyjnego jesteśmy gotowi, by przejść do tworzenia widoków dla naszego modelu. Potem dodamy dwa procesy poboczne, które umożliwią nam nadanie numeru podczas składania zamówienia oraz edycję stanów magazynowych w zależności od tego, jaką akcję wykonamy.

Tworzenie widoków

Jak już wiesz, naszą pracę zaczniemy od stworzenia serializerów potrzebnych do obsługi widoków tej aplikacji. W tym celu w folderze *purchase* utwórz plik *serializers.py* i umieść w nim następujący kod:

```
#purchase/serializers.py
from menu.serializers import CashierMenuItemSerializer
from rest_framework.serializers import (
ModelSerializer,

PrimaryKeyRelatedField,
CharField
)

from .models import PurchaseOrder
from menu.models import MenuItem


class ListPurchaseOrderSerializer(ModelSerializer):
    status = CharField(source='get_status_display')
    class Meta:
        model = PurchaseOrder
        fields = ["status", "order_number"]


class PurchaseOrderSerializer(ModelSerializer):
    items = CashierMenuItemSerializer(many=True)
```

```
    class Meta:
        model = PurchaseOrder
        fields = '__all__'


class CreatePurchaseOrderSerializer(PurchaseOrderSerializer):
    items = PrimaryKeyRelatedField(
        many=True,
        queryset=MenuItem.objects.all()
    )
```

Utworzyliśmy trzy klasy do obsługi zamówień. Klasa `ListPurchaseOrderSerializer` dziedziczy po `ModelSerializer`. Jako model wybraliśmy nowo utworzony `Purcha ↪seOrder`, a z racji tego, że ten widok będzie dostępny nawet dla niezalogowanych użytkowników, wyświetlamy tylko dwa pola: `status` oraz `order_number`. Te dane są niezbędne, aby dany klient mógł śledzić status swojego zamówienia za pomocą ekranu w kawiarni. Jak widzisz, pole `status` jest polem wyboru, którego reprezentacją jest numer. Musimy nieco dostosować to pole — dlatego użyliśmy klasy `Char ↪Field` dostępniej w Django REST framework. Jako argument `source` przekazujemy `get_status_display`, co, jak pamiętasz z poprzednich rozdziałów, umożliwiło nam pobranie nazwy przypisanej do danego numeru. Ograniczanie pól odbywa się poprzez przekazanie tych, które chcemy wyświetlić za pomocą listy.

Serializer `PurchaseOrderSerializer` pełni dwie funkcje. Pierwsza z nich to wykorzystanie w widokach przeznaczonych dla pracowników kawiarni, by mogli przeglądać, edytować oraz tworzyć zamówienia. Jak widzisz, w celu wyświetlenia przedmiotów w zamówieniu używamy utworzonej wcześniej klasy `CashierMenu ↪ItemSerializer`, dzięki czemu pracownicy będą mieli również dostęp do składu danej pozycji z menu. `CreatePurchaseOrderSerializer` będzie potrzebny podczas tworzenia oraz edycji naszego zamówienia. Wystarczy jedynie przesłanie numerów `id` przedmiotów, które zamawiający będzie chciał kupić.

Skoro serializery są gotowe, weźmy się za tworzenie widoków. Jak na początku ustaliliśmy, aby niepotrzebnie nie skakać po plikach, widok oraz zestaw widoków umieścimy w jednym pliku o nazwie *viewSets.py*. Utwórz ten plik w katalogu *purchase* oraz dodaj w nim poniższy kod:

```
#purchase/viewsets.py
from rest_framework.viewsets import ModelViewSet
from rest_framework.generics import ListAPIView
from rest_framework.permissions import AllowAny

from authx.permissions import IsCashierUser
from .serializers import (
    CreatePurchaseOrderSerializer,
    PurchaseOrderSerializer,
    ListPurchaseOrderSerializer
)

from .models import PurchaseOrder


class PurchaseListView(ListAPIView):
```

```
    queryset = PurchaseOrder.objects.all()
    serializer_class = ListPurchaseOrderSerializer
    permission_classes = [AllowAny]


class PurchseViewSet(ModelViewSet):
    queryset = PurchaseOrder.objects.all()
    permission_classes = [IsCashierUser]

    def get_serializer_class(self):
        if self.action in ['update', 'partial_update', 'create']:
            return CreatePurchaseOrderSerializer
        else:
            return PurchaseOrderSerializer
```

Utworzyliśmy dwa widoki. Pierwszy z nich, `PurchaseListView`, służy do pobierania listy dla niezalogowanych użytkowników, dlatego jako `permission_classes` użyliśmy `AllowAny`. `serializer_class` to `ListPurchaseOrderSerializer`, dzięki czemu możemy wyświetlać tylko numer wraz ze statusem. Druga klasa, czyli `Purchase ↪ViewSet`, odpowiada za zestaw widoków dla obsługi kawiarni. W tym przypadku `permission_classes` to `IsCashierUser` i tylko zalogowani użytkownicy będą mogli pobierać zasoby z tego widoku. Nadpisaliśmy metodę `get_serializer_class` tak, aby podczas tworzenia oraz edycji zamówienia jako reprezentacja `items` używany był serializer `CreatePurchaseOrderSerializer`, co pozwoli nam jedynie przesyłać numery `id` wybranych przedmiotów.

Świetnie, widoki gotowe! Pozostało jedynie przygotować *url*, dlatego otwórz plik *urls.py* w katalogu *purchase* i umieść w nim poniższy kod:

```
#purchase/urls.py
from django.urls.conf import include, path
from django.urls import re_path

from rest_framework.routers import DefaultRouter
from .viewsets import PurchaseListView, PurchseViewSet

router = DefaultRouter()

router.register(
    r'purchase',
    PurchseViewSet,
    basename="purchase"
)

urlpatterns = [
    re_path(r'', include(router.urls)),
    path('purchase-list/',
         PurchaseListView.as_view(), name='purchase_list')
]
```

Gotowe! Moduł odpowiedzialny za dodawanie oraz odczytywanie zamówień jest prawie skończony. Pozostało dodać logikę odpowiedzialną za tworzenie numerów zamówień oraz edycję zapasów magazynowych. Na koniec rozdziału przeprowadzimy testy nowo stworzonych widoków.

Akcje dodatkowe

Pracę zaczniemy od stworzenia logiki odpowiedzialnej za generowanie numeru zamówienia. Właściciel kawiarni chciał, aby numer zawierał datę złożenia zamówienia oraz numer oznaczający kolejność złożenia zamówienia danego dnia. Jak wiesz, pole `order_number` musimy przekazać obowiązkowo podczas tworzenia obiektu w bazie, a serializer wymaga od nas podania tego parametru podczas składania zamówienia.

Moglibyśmy powierzyć proces generowania tego numeru aplikacji frontendowej, ta jednak nie będzie miała wiedzy o tym, że na przykład dwóch klientów rozpoczęło w tym samym czasie proces zamawiania: jeden za pośrednictwem kiosku, natomiast drugi przy kasie. Zakładając, że aplikacja frontendowa tworzy dany numer podczas składania zamówienia przy jednoczesnym zatwierdzeniu, istnieje możliwość utworzenia dwóch takich samych numerów, co jest niedopuszczalne. Co zrobić w takiej sytuacji?

Jeżeli wgłębisz się w dokumentację (*https://www.django-rest-framework.org/api-guide/serializers/#advanced-serializer-usage*), zobaczysz, że **Django REST framework** umożliwia manipulowanie danymi w serializerze za pomocą metod `.to_internal_value(self, dane)` oraz `.to_representation(self, instance)`. Metoda `to_representation` pozwala nam na modyfikacje reprezentacji zasobu.

Dla nas jednak pomocna okazuje się metoda `to_internal_value`, która pobiera niezwalidowane jeszcze dane wejściowe i udostępnia je jako `serializer.validated_data`. Takie dane są już wykorzystywane w metodach `create()` oraz `update()`. Brzmi to jak ratunek w obecnej sytuacji! W takim razie przejdź do pliku *purchase/serializers.py* i edytuj klasę `CreatePurchaseOrderSerializer`, a także metodę `to_internal_value` zgodnie z kodem poniżej:

```
#purchase/serializers.py
from datetime import datetime
from menu.serializers import CashierMenuItemSerializer
...
class CreatePurchaseOrderSerializer(PurchaseOrderSerializer):
    def to_internal_value(self, data):
        today = datetime.now()
        conunt_today = PurchaseOrder.objects.filter(
            created_date__date=today).count() + 1
        data['order_number'] = f"{today.strftime('%d%m%y')}_{conunt_today}"

        return super(CreatePurchaseOrderSerializer, self).to_internal_value(data)

    items = PrimaryKeyRelatedField(
        many=True,
        queryset=MenuItem.objects.all()
    )
```

Co właśnie zrobiliśmy? Jedyne, co musieliśmy zrobić, to edytować dane przekazywane do walidacji, dlatego całą logikę możemy zostawić bez zmian, a jako `return`

zapisujemy wywołanie metody `to_internal_value` pochodzącej z klasy `CreatePur ↪chaseOrderSerializer`, czyli nie edytujemy samego procesu zachodzącego w tej metodzie. Do obiektu `data` przekazaliśmy wartość dla pola `order_number`, którą wygenerowaliśmy, pobierając wszystkie utworzone obiekty `PurchaseOrder` powiększone o jeden. Z biblioteki *datatime* pobraliśmy `datatime`, następnie, po wywołaniu metody `now()`, otrzymaliśmy bieżącą datę. Za pomocą Django ORM i zapisu `created_date__date` możemy przefiltrować nasz `queryset` tylko poprzez datę, bez uwzględniania godziny. Dzięki temu dostaniemy obiekty z całego dnia. Wykorzystując formatowania ciągów tekstowych, przekazaliśmy sformatowaną datę metodą `strftime`, która umożliwi nam modyfikowanie daty w dowolny sposób. Jako argument przekazaliśmy `%d%m%y'`, gdzie `%d` to dzień, `%m` miesiąc, a `%y` rok. Wszystkie kody dla tej metody możesz znaleźć pod tym linkiem: *https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes*.

Przykładowym numerem może być `260322_14`, co reprezentuje czternaste zamówienie w dniu 22 marca 2022 roku.

Aby dokończyć proces, musimy jeszcze edytować stan magazynowy podczas składania zamówienia przez naszego klienta.

Walidacja ilości składników wydaje się prostym zadaniem, jednak tak nie jest. Musimy uwzględnić nie tylko usuwanie zasobów w sytuacji, gdy ktoś złoży zamówienie, ale również musimy obsłużyć moment, w którym ktoś zedytuje zamówienie, czyli usunie jakiś obiekt lub po prostu anuluje zamówienie. Dodatkowo musimy uwzględnić przypadek, gdy zamówienie będzie wymagało pobrania większej ilości zasobów, niż zawiera magazyn.

Na początek możemy się zabezpieczyć poprzez zmianę pola `quantity` w modelu `Ingredient`. W tym celu przejdź do pliku *story/models.py* oraz zmień poniższe pole:

#story/models.py
```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Ingredient(models.Model):
    UNIT_CHOICE = (
        (1,  _('ml')),
        (2, _('g')),
    )
    name = models.CharField(max_length=50, null=False, blank=False)
    sku_nubmer = models.CharField(max_length=50, null=False, blank=False)
    quantity = models.PositiveIntegerField(default=0)
    supplier = models.ForeignKey(
        'supplier.Supplier',
        on_delete=models.SET_NULL,
        blank=True,
        null=True
    )
    unit = models.PositiveSmallIntegerField(
        choices=UNIT_CHOICE,
        default=1,
```

```
    )

    @property
    def label(self):
        return f'{self.name} — {self.quantity} {self.get_unit_display()}'

    class Meta:
        ordering = ('name', )

    def __str__(self):
        return self.label
```

Dzięki polu `PositiveIntegerField` nie będziemy mogli dodać do bazy obiektu z wartością ujemną. Teraz musimy przeprowadzić migracje. Jednak zanim to zrobimy, upewnij się, że nie ma w tym modelu żadnych obiektów z ujemnymi wartościami. W tym celu wejdź do panelu administracyjnego i zmień wartość, jeżeli jakaś jest ujemna.

Następnie wykonaj poniższą komendę:

```
> docker exec -it djangomicro_mono_django_1  python manage.py makemigrations
Migrations for 'story':
story/migrations/0002_alter_ingredient_quantity.py
- Alter field quantity on ingredient

> docker exec -it djangomicro_mono_django_1  python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, authx, contenttypes, menu, purchase, sessions,
  ↪story, supplier
Running migrations:
  Applying story.0002_alter_ingredient_quantity... OK
```

Kolejny krok to wybranie momentu, w którym dokona się walidacja. Zdecydowałam, że dobrym punktem będzie sygnał z Django.

Czym są sygnały? Django wykorzystuje wzorzec projektowy *Obserwator* do implementacji tego rozwiązania. W mechanizmie sygnałów biorą udział nadawca (`sender`) oraz odbiorca (`receiver`). Ich role są dokładnie powiązane z nazwą, tzn. jeden obiera sygnał, drugi go wysyła. Django posiada kilka użytecznych sygnałów, jednak nas interesuje jeden konkretny: `m2m_changed` (*https://docs.djangoproject.com/en/dev/ref/signals/#m2m-changed*). Dlaczego ten? Ponieważ klasa `Purchase` przechowuje przedmioty zamówienia w relacji „wiele do wielu". Moglibyśmy pokusić się o sygnał `pre_save`, ale on podczas tworzenia instancji nie ma dostępu do pola `ManyToMany`.

Sygnał jest wysyłany, gdy zostanie zmienione pole `ManyToManyField`, więc nie jest to sygnał modelu. Jednak jak zostało napisane w dokumentacji, jest on uzupełnieniem sygnałów z modelu, dlatego jest zadeklarowany w tym samym miejscu co pozostałe.

Ten sygnał zwraca kilka argumentów, lecz dla nas są istotne tylko dwa z nich: `pk_set` oraz `action`.

Pierwszy — `pk_set`, w zależności od parametru `action`, przesyła klucze usuniętych lub dodanych obiektów do relacji, co jest dla nas niezwykle istotne, aby poprawnie liczyć tylko wartości tych instancji, które zostały w jakiś sposób edytowane. Drugi argument to `action`, który zwraca jeden z sześciu kluczy w zależności od wykonanej akcji. Dla nas istotne będą akcje `post_remove` i `post_add`, dzięki którym będziemy wiedzieli dokładnie, kiedy uruchomić logikę usuwania lub dodawania ilości przedmiotów.

Dobrze, skoro wiemy, co chcemy osiągnąć, bierzmy się do pracy! Przejdź do pliku *purchase/models.py* i wprowadź poniższe zmiany:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.contrib.auth import get_user_model

from django.db.models.signals import m2m_changed
from django.db.models import F
from menu.models import MenuItem, Component
from story.models import Ingredient

User = get_user_model()


class PurchaseOrder(models.Model):
...


def change_story(instance, **kwargs):
    action = kwargs.get('action')
    pk_set = kwargs.get('pk_set')

    all_quantity = dict()
    obj = list()

    for id in pk_set:
        item = MenuItem.objects.get(pk=id)
        for component in item.ingredients.all():
            if component.pk in all_quantity:
                all_quantity[component.pk] += component.quantity
            else:
                all_quantity[component.pk] = component.quantity

        for key in all_quantity.keys():
            all_res = all_quantity[key]
            new_obj = Component.objects.get(pk=key).ingredient
            if action == 'post_add':
                new_obj.quantity = F('quantity') - all_res
            if action == 'post_remove':
                new_obj.quantity = F('quantity') + all_res
            obj.append(new_obj)

    Ingredient.objects.bulk_update(obj, ['quantity'])


m2m_changed.connect(change_story, sender=PurchaseOrder.items.through)
```

Stworzyliśmy funkcję `change_story`, następnie przypisaliśmy ją do sygnału `m2m_changed`, którego nadawcą jest `PurchaseOrder.items.through`. Dopisek `through` to atrybut dający nam dostęp do klasy `ManyToMany` dla danego pola.

Na samym początku naszej funkcji pobraliśmy z `**kwargs` dwie wartości: `action` i `pk_set`, a także stworzyliśmy słownik `all_quantity`, który przetrzymuje sumy wszystkich składników, oraz `obj` reprezentujący listę obiektów do edycji.

Następnie iterowaliśmy po numerach uzyskanych z wartości `pk_set`. Podczas iteracji za pomocą `id` z `pk_set` został znaleziony obiekt z klasy `MenuItem`, dla którego pobraliśmy wszystkie składniki. Te posłużyły nam do kolejnej iteracji, dzięki której umieściliśmy w słowniku ilość danego składnika, a jako klucz wykorzystaliśmy numer `id` danego obiektu z modelu `Component`.

Uzyskaliśmy w ten sposób sumę konkretnych składników ze wszystkich pozycji w menu. Następnie, iterując po tych sumach, odnaleźliśmy dany składnik i za pomocą wyrażenia `F` zedytowaliśmy ilość danego składnika. Wyrażanie `F()` bez wyciągania danych z bazy jest w stanie przeprowadzić operację dodawania bądź odejmowania, ponieważ generuje wyrażenie `SQL`, które opisuje daną operację na poziomie bazy danych.

Innym zabiegiem wpływającym na wydajność takiego rozwiązania jest użycie metody `bulk_udpate`, która przyjmuje jako argumenty obiekty do edycji oraz pola do edycji. Ta metoda przyjmuje jeszcze jeden opcjonalny argument, `batch_size`, który określa wielkość wygenerowanego zapytania `SQL`. Ogromną zaletą tej metody jest fakt, że wszystkie operacje przeprowadza za pomocą jednego zapytania do bazy danych.

Następnie, aby uniknąć błędów wywołanych tym, że chcemy użyć większej ilości składników, niż jest dostępna, musimy nadpisać serializer:

```
#purchase/serializers.py
from datetime import datetime

from rest_framework.serializers import ModelSerializer, PrimaryKeyRelatedField,
↪CharField, ValidationError
from django.utils.translation import ugettext_lazy as _

from menu.serializers import CashierMenuItemSerializer
from menu.models import MenuItem, Component
from .models import PurchaseOrder
...
class CreatePurchaseOrderSerializer(PurchaseOrderSerializer):
    items = PrimaryKeyRelatedField(
        many=True,
        queryset=MenuItem.objects.all()
    )

    def validate(self, data):

        all_quantity = {}
```

```
        for item in data['items']:
            for component in item.ingredients.all():
                if component.pk in all_quantity:
                    all_quantity[component.pk] += component.quantity
                else:
                    all_quantity[component.pk] = component.quantity

        for key in all_quantity.keys():
            new_obj = Component.objects.get(pk=key).ingredient
            all_res = all_quantity[key]
            if all_res > new_obj.quantity:
                raise ValidationError(_("No ingredients"))

        return super(CreatePurchaseOrderSerializer, self).validate(data)
```

Nadpisaliśmy metodę `validate`, uwieczniając w niej kod podobny do tego, który jest umieszczony w naszej funkcji podpiętej pod sygnał `m2m_changed`. Różnica polega na sprawdzeniu, czy dana wartość nie przekracza dostępnej ilości. W sytuacji, gdy wartość przekracza dostępną ilość, zwracamy błąd za pomocą klasy `Valida` ↪`tionError`, której podczas inicjalizacji przekazaliśmy wiadomość, jaką należy zwrócić w żądaniu. Razem z odpowiedzią przyjdzie kod `400`, co oznacza `bad request`.

Podsumowanie

Sporo pracy za nami, napisaliśmy już praktycznie całą logikę odpowiedzialną za składanie zamówień oraz ich przetrzymywanie w bazie danych. Aby ukończyć ten proces, pozostało nam zająć się automatycznym pobieraniem użytkownika, który złożył zamówienie, do pola `created_by`. Następnie musimy obsłużyć logikę anulowania zamówienia. Na koniec pracy z tym modułem zostanie nam obsłużenie listy zamówień wraz z ich statusami, z wykluczeniem tych, które zostały zrealizowane bądź anulowane.

Książka Sylwestra Walczaka jest poświęcona Django, wolnej, otwartoźródłowej platformie programistycznej służącej do tworzenia aplikacji internetowych. Autor udowadnia, że wbrew obiegowym opiniom Django jako framework nadaje się do budowania mikroserwisów — autonomicznych usług, które współpracują ze sobą, tworząc na przykład serwis internetowy. Autor odwołuje się przy tym do API potrzebnego do obsługi kawiarni. Jest to więc swego rodzaju instrukcja, która przeprowadza przez projektowanie, programowanie i wdrożenie systemu do obsługi kawiarni, składającego się między innymi z takich aplikacji jak magazyn, menu, obsługa klientów i kolejkowanie utworzonych zamówień.

To podręcznik przeznaczony dla programistów; do zrozumienia poruszonych w nim zagadnień wymagana jest znajomość języka Python i komunikacji webowej. Lektura kolejnych rozdziałów pozwoli Ci na tworzenie bibliotek i napisanie wtyczki do Django odpowiadającej za autoryzację. Poznasz świat mikrousług, w tym ich zalety i wady. Przy okazji zapoznasz się z kilkoma narzędziami przydatnymi nie tylko w świecie architektury rozproszonej. Zatem — kawa i do dzieła?

Dzięki książce:

- **Dowiesz się, co to jest konteneryzacja, Docker, REST API i webhooki**
- **Utworzysz własny system autoryzacji**
- **Napiszesz system rozproszony**
- **Dynamicznie przepiszesz serializery do widoków**
- **Przygotujesz dokumentację API**
- **Opracujesz komunikację między usługami**

Sylwester Walczak

Entuzjasta nowych technologii i programowania. Full-stack deweloper, obecnie rozwija aplikację umożliwiającą pracę z danymi przestrzennymi. W działalności programistycznej stosuje regułę DRY i przywiązuje wagę do jakości kodu. Programowanie to również jego hobby, dlatego zajmuje się nim także w wolnych chwilach, ucząc się nowych rzeczy. Kiedy chce odświeżyć umysł, sięga po książki i kawę.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9348-6

9 788328 393486

Cena: 59,00 zł