AKADEMIA HUMANISTYCZNO-EKONOMICZNA W ŁODZI

wydział Informatyki

Kamil Bagieński
Grupa D1
Numer indeksu 155623

Dokumentacja do projektu

Język SQL/System zarządzania bazą danych – Projekt
Aplikacja do zarządzania procesem sprzedaży

Spis treści

Wstęp

Jest to projekt aplikacji webowej w ramach zajęć z Język SQL/System zarządzania bazą danych. Jest to jedno z trzech zadań, które mieliśmy wybrać w ramach projektu. Jego treść:

Aplikacja do Zarządzania Procesem Sprzedaży z Wieloma Tabelami SQL:

- Cel projektu: Stworzenie aplikacji do zarządzania procesem sprzedaży, która wykorzystuje bazę danych SQL do przechowywania i analizy danych.
- Funkcjonalności: Aplikacja umożliwia zarządzanie klientami, produktami, zamówieniami i płatnościami. Wykorzystuje różne rodzaje złączeń (równościowe, nierównościowe) do analizy danych sprzedażowych.

Zbudowałem aplikację na frameworku Django. Obsługuje ona bazę danych w formie ORM, wykorzystując przy tym SQLite w celach jej budowy oraz szablony html w celach frontendowych. W przypadku backendu jest to obiektowy python. Nie będę opisywał samego frameworku, bo jest on dobrze znany, wielu osobom. Jednakże jest to bardzo duży framework, który jest raczej wykorzystywany do większych projektów. Jednakże w tych małych, sprawdza się świetnie jako początek większego projektu. Dzięki temu zadaniu nauczyłem się działania w tym frameworku i jestem teraz nim zafascynowany. W dokumentacji zaprezentuje jego działanie oraz zastosowane przeze mnie rozwiązania. W złączonych plikach daje cały kod do aplikacji. Po ocenieniu mojej pracy, aplikacja będzie dostępna na githubie, w ramach pomocy naukowych dla każdego kto będzie chciał zbudować małą aplikację na tym frameworku. Zmienne w pythonie zostały napisane w języku angielskim dla większej klarowności. Analiza danych jest przedstawiona w chart.js, początkowo miał to obsługiwać moduł dedykowany pod Django, ale niestety nie jest on kompatybilny z najnowszym Pythonem oraz najnowszym Django, przez co musiałem finalnie zrobić wizualizacje danych w chart.js. Projekt ten został przeze mnie przetestowany pod kątem działania na Windows 11 oraz Arch linuxie.

Uruchomienie aplikacji

1. Instalacja Pythona:
   - Odwiedź stronę python.org i pobierz Pythona dla swojego systemu operacyjnego.
   - Zainstaluj Pythona, upewniając się, że dodajesz Pythona do zmiennej środowiskowej PATH.
2. Sprawdzenie Instalacji Pythona i pip:
   - Otwórz terminal i wpisz następujące polecenia, aby sprawdzić wersję Pythona i pip:
   - **python --version**
   - **pip --version**
3. Instalacja Django:
   - W terminalu wpisz polecenie:
   - **pip install django**
4. Przygotowanie Aplikacji Django do Użycia:
   - Odbierz aplikację Django (aplikacja.zip).
   - Rozpakuj pliki w odpowiednim miejscu.
5. Wykonanie Migracji Bazy Danych:
   - Przejdź do katalogu głównego projektu Django w którym jest plik manage.py.
   - Wykonaj migracje bazy danych otwierając terminal w tym katalogu:
   - **python manage.py makemigrations**
   - **python manage.py migrate**
6. Tworzenie Superużytkownika dla Panelu Administracyjnego:
   - Stwórz superużytkownika, aby uzyskać dostęp do panelu administracyjnego:
   - **python manage.py createsuperuser**
7. Uruchomienie Serwera Deweloperskiego:
   - Uruchom serwer deweloperski Django:
   - **python manage.py runserver**
8. Dostęp do Aplikacji:
   - Otwórz przeglądarkę i przejdź do adresu http://localhost:8000/ aby zobaczyć działającą aplikację.
   - Dostęp do panelu administracyjnego jest możliwy pod adresem http://localhost:8000/admin.

Budowa aplikacji

Struktura aplikacji wygląda następująco. Pomijam zbędne fragmenty jego budowy oznaczeniem […] i pokazuje najważniejsze pliki projektu:

```
C:.
|
\---myproject
    |   db.sqlite3
    |   manage.py
    |
    +---myapp
    |   |   admin.py
    |   |   apps.py
    |   |   models.py
    |   |   tests.py
    |   |   urls.py
    |   |   views.py
    |   |   […]
    |   |
    |   +---migrations
    |   |   |   […]
    |   |   \---__pycache__
    |   |           […]
    |   +---templates
    |   |   |   customer_expenses_stats.html
    |   |   |   index.html
    |   |   |   payment_method_expenses_stats.html
    |   |   |   product_sales_stats.html
    |   |   |   sales_data_table.html
    |   |   |   sales_stats.html
    |   |   |
    |   |   \---admin
    |   |           custom_admin_template.html
    |   |
    |   \---__pycache_
    |           […]
    \---myproject
        |   settings.py
        |   urls.py
        |   […]
        \---__pycache__
                […]
```

Katalog Główny Aplikacji (myproject):

Zawiera kluczowe pliki konfiguracyjne i uruchomieniowe aplikacji:

- db.sqlite3: Jest to baza danych SQLite używana przez aplikację.
- manage.py: Skrypt umożliwiający zarządzanie różnymi aspektami aplikacji Django, takimi jak migracje, uruchamianie serwera deweloperskiego i inne.

Katalog Aplikacji (myapp):

- Jest to katalog głównej aplikacji w projekcie Django. Składa się z następujących plików i katalogów:
- admin.py: Służy do konfiguracji panelu administracyjnego Django, szczególnie w zakresie prezentacji modeli danych.
- apps.py: Zawiera konfigurację samej aplikacji.
- models.py: Zawiera definicje modeli danych, które aplikacja wykorzystuje.
- tests.py: Umożliwia tworzenie testów jednostkowych dla aplikacji.
- urls.py: Definiuje wzorce URL, które aplikacja obsługuje.
- views.py: Zawiera logikę odpowiedzialną za obsługę żądań HTTP.
- migrations: Katalog ten zawiera pliki migracji bazy danych, które są wykorzystywane do zarządzania schematem bazy danych.
- templates: W tym katalogu znajdują się pliki HTML używane jako szablony dla widoków aplikacji. Wyróżniają się tu pliki takie jak customer_expenses_stats.html, index.html, payment_method_expenses_stats.html, product_sales_stats.html, sales_data_table.html, sales_stats.html. Dodatkowo, katalog admin zawiera niestandardowy szablon custom_admin_template.html dla panelu administracyjnego.
- __pycache__: Zawiera skompilowane pliki bajtowe Pythona, które przyspieszają ładowanie aplikacji.

Katalog Konfiguracyjny Projektu (myproject/myproject):

Zawiera pliki konfiguracyjne dla całego projektu Django:

- settings.py: Jest to główny plik konfiguracyjny projektu, zawierający ustawienia takie jak konfiguracja bazy danych, zainstalowane aplikacje, middleware, szablony panelu admina itd.
- urls.py: Definiuje wzorce URL na poziomie całego projektu, wskazuje gdzie szukać URLS aplikacji.
- __pycache__: Podobnie jak w aplikacji, zawiera skompilowane pliki bajtowe.

W tej strukturze każdy element ma swoje specyficzne zadanie. Pliki w katalogu myapp są odpowiedzialne za różne aspekty funkcjonowania aplikacji, od modelowania danych, przez logikę biznesową, po prezentację i interakcję z użytkownikiem. Pliki w głównym katalogu myproject odpowiadają za ogólną konfigurację i zarządzanie całym projektem Django.

Modele bazy

Moja aplikacja Django zawiera następujące modele bazy danych ORM, które są opisane w pliku models.py:

Model Customer:

- id: Unikalny identyfikator typu UUID, będący kluczem głównym.
- name: Pole tekstowe przechowujące nazwę klienta (maksymalnie 255 znaków).
- email: Pole e-mailowe, które jest unikalne dla każdego klienta.

```python
class Customer(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
```

Model Product:

- name: Pole tekstowe przechowujące nazwę produktu (maksymalnie 255 znaków).
- price: Pole zmiennoprzecinkowe przechowujące cenę produktu.

```python
class Payment(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE)
    amount = models.FloatField()
```

Model Order:

- id: Unikalny identyfikator typu UUID, będący kluczem głównym.
- customer: Klucz obcy powiązany z modelem Customer, określający klienta składającego zamówienie.
- product: Klucz obcy powiązany z modelem Product, określający zamawiany produkt.
- quantity: Pole całkowitoliczbowe określające ilość zamawianego produktu.
- payment_method: Pole tekstowe z określonymi wyborami (karta, pobranie, PayPal) określające metodę płatności.
- amount: Pole zmiennoprzecinkowe przechowujące kwotę zamówienia.
- order_date: Pole daty i czasu, automatycznie ustawiane na czas dodania zamówienia.
- product_name: Pole tekstowe przechowujące nazwę produktu; może być puste.
- Metoda save: Zmodyfikowana metoda zapisu, która ustawia product_name na nazwę powiązanego produktu.

```python
class Order(models.Model):
    PAYMENT_METHODS = (
        ('card', 'Karta'),
        ('cod', 'Pobranie'),
        ('paypal', 'PayPal'),
    )

    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.IntegerField()
    payment_method = models.CharField(max_length=20, choices=PAYMENT_METHODS)
    amount = models.FloatField()
    order_date = models.DateTimeField(auto_now_add=True)
    product_name = models.CharField(max_length=255, blank=True, null=True)
    def save(self, *args, **kwargs):
        if self.product:
            self.product_name = self.product.name
        super(Order, self).save(*args, **kwargs)
```

Model Payment:

- order: Klucz obcy powiązany z modelem Order.
- amount: Pole zmiennoprzecinkowe przechowujące kwotę płatności.

```python
class Payment(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE)
    amount = models.FloatField()
```

Każdy z tych modeli jest reprezentacją struktury danych w mojej aplikacji Django, gdzie Customer i Product są podstawowymi encjami, a Order i Payment są powiązane z procesem zamówienia i płatności. Model Order posiada dodatkową logikę w metodzie save, co jest typowym podejściem w Django do dostosowywania zachowania modeli bazy ORM.

Nazwa produktu jest zapisywana dwukrotnie, wynika to z budowy mojego views oraz wybranego przeze mnie automatycznego pola pobierania daty zamówienia z zegara serwera. W innym przypadku chart.js nie chciał pobierać danych bezpośrednio z tabeli product, do dynamicznego wybierania nazw w formularzach. Wybrałem takie rozwiązanie gdyż było ono najprostsze do osiągnięcia celu.

Wykorzystanie bazy danych i widoki Django

W pliku views.py aplikacji Django zdefiniowano następujące widoki i funkcje pomocnicze.

Widoki główne:

Funkcja get_products:

Ta funkcja pobiera wszystkie produkty z bazy danych, używając modelu Product. Zwraca listę słowników, w których każdy słownik zawiera id, name i price produktu. Jest to przykład funkcji pomocniczej, która może być wykorzystana w różnych widokach do uzyskiwania informacji o produktach.

```
def get_products():
    products = Product.objects.all().values('id', 'name', 'price')
    return products
```

Widok index:

Główny widok aplikacji. Wywołuje funkcję get_products do pobrania danych o produktach, a następnie renderuje szablon index.html, przekazując te dane. Jest to podstawowy przykład wykorzystania szablonu do wyświetlania danych w Django.

```
def index(request):
    products = get_products()
    return render(request, 'index.html', {'products': products})
```

Funkcja add_customer:

Obecnie jest to pusta funkcja (zawiera tylko pass), ale nie została jeszcze zrealizowana, ale jest ona pozostawiona w przypadku rozbudowania aplikacji o funkcje rejestracji konta. Funkcja ta miała służyć do dodawania nowego klienta do bazy danych, w przypadku właśnie rejestracji klienta. Jednakże jest to proces, który nie jest tematem tego projektu

Widok add_customer_route:

Ten widok obsługuje żądania POST, które są wysyłane, gdy użytkownik chce dodać nowego klienta. Funkcja pobiera dane (name i email) z żądania i używa ich do wywołania funkcji add_customer, ale też jest wywoływana przez inne pliki aplikacji przez co dodaje klienta do bazy. Następnie przekierowuje użytkownika z powrotem do widoku głównego.

```python
def add_customer_route(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        add_customer(name, email)

    return redirect('index')
```

Widok add_order_route:

Podobnie jak add_customer_route, ten widok obsługuje tworzenie nowych zamówień poprzez żądania POST. Pobiera dane z formularza (identyfikatory klienta i produktu, ilość) i używa ich do utworzenia nowego obiektu Order. Po utworzeniu zamówienia, widok przekierowuje użytkownika do strony głównej.

```python
def add_order_route(request):
    if request.method == 'POST':
        customer_id = int(request.POST.get('customer_id'))
        product_id = int(request.POST.get('product_id'))
        quantity = int(request.POST.get('quantity'))

        print("Dane formularza:", customer_id, product_id, quantity)
# Dodane do debugowania

        try:
            order = Order.objects.create(customer_id=customer_id, product_id=
product_id, quantity=quantity)
            order_id = order.id
            success_message = f"Zamówienie nr {order_id} zostało dodane pomyślnie."
            print("Zamówienie utworzone:", order_id)  # Dodane do debugowania
        except Exception as e:
            print(f"Błąd przy tworzeniu zamówienia: {e}")  # Dodane do debugowania
            success_message = None

        products = get_products()
        context = {'products': products, 'success_message': success_message}
        print("Kontekst wysyłany do szablonu:", context)  # Dodane do debugowania

        return render(request, 'index.html', context)
    print("Nie znaleziono danych formularza")  # Dodane do debugowania

    return redirect('index')
```

Widoki takie jak add_customer_route, add_order_route, create_order i add_payment_route obsługują różne aspekty interakcji użytkownika z aplikacją, takie jak dodawanie klientów, tworzenie zamówień i dodawanie płatności. Są to typowe przykłady widoków w aplikacjach Django, które obsługują logikę biznesową i interakcje z bazą danych na podstawie danych wejściowych od użytkownika.

Widok create_order:

Ten widok obsługuje proces tworzenia zamówienia w bardziej złożony sposób:

- Najpierw sprawdza, czy żądanie jest typu POST.
- Pobiera dane z formularza: name, email, product_id, quantity i payment_method.
- Używa tych danych do stworzenia lub pobrania (jeśli już istnieje) obiektu Customer o podanej nazwie i adresie e-mail.
- Następnie pobiera obiekt Product na podstawie product_id.
- Oblicza całkowitą kwotę zamówienia, mnożąc cenę produktu przez ilość.
- Tworzy nowe zamówienie (Order) z pobranymi danymi, w tym wybraną metodą płatności.
- Po utworzeniu zamówienia wyświetla użytkownikowi komunikat o sukcesie i przekierowuje na stronę główną.
- Jeśli żądanie nie jest typu POST, użytkownik również jest przekierowywany na stronę główną.

```python
def create_order(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        product_id = request.POST.get('product_id')
        quantity = int(request.POST.get('quantity', 0))
        payment_method = request.POST.get('payment_method')

        customer, created = Customer.objects.get_or_create(name=name, email=email)

        product = get_object_or_404(Product, id=product_id)

        total_amount = product.price * quantity

        order = Order.objects.create(
            customer=customer,
            product=product,
            quantity=quantity,
            payment_method=payment_method,
            amount=total_amount
        )
        messages.success(request, "Zamówienie zostało dodane pomyślnie.")

        print(f"Zamówienie nr {order.id} zostało utworzone")
        return redirect('index')

    return redirect('index')
```

Widok create_order jest dobrym przykładem złożonego widoku w Django, który łączy w sobie odbieranie danych z formularza, interakcję z bazą danych i przekierowywanie użytkownika. Jest to typowy scenariusz dla aplikacji e-commerce, gdzie proces tworzenia zamówienia musi być dobrze przemyślany i zaimplementowany w sposób zapewniający poprawność danych i dobre doświadczenie użytkownika.

Widoki analityczne, które potem są wykorzystane w ramach wykresów i tabel:

customer_sales_stats:

- Dekorowana przez @login_required, co oznacza, że dostępna jest tylko dla zalogowanych użytkowników.
- Wylicza łączną kwotę sprzedaży i łączną liczbę zakupionych produktów na klienta.
- Używa ORM Django do agregacji danych pochodzących z zamówień (Order) według nazwy klienta.
- Dane są przekazywane do szablonu sales_stats.html i renderowane jako strona statystyk sprzedaży dla klientów.

```python
@login_required
def customer_sales_stats(request):
    total_sales_per_customer = Order.objects.values('customer__name').annotate(
total_spent=Sum('amount'))

    total_products_per_customer = Order.objects.values('customer__name').annotate(
total_products=Count('id'))

    return render(request, 'sales_stats.html', {'sales': total_sales_per_customer,
'products': total_products_per_customer})

def product_list(request):
    products = Product.objects.all().values('id', 'name')
    return JsonResponse(list(products), safe=False)

def product_sales_stats(request):
    product_id = request.GET.get('product_id')
    if not product_id:
        return JsonResponse({'error': 'Brak podanego ID produktu'}, status=400)

    sales_data = (
        Order.objects.filter(product_id=product_id)
        .values('order_date')
        .annotate(total_quantity=Sum('quantity'))
        .order_by('order_date')
    )

    return JsonResponse(list(sales_data), safe=False)
```

product_list:

- Zwraca listę produktów w formacie JSON.
- Pobiera wszystkie produkty z bazy danych i przekształca je na format, który może być łatwo wykorzystany przez frontend aplikacji.

```python
def product_list(request):
    products = Product.objects.all().values('id', 'name')
    return JsonResponse(list(products), safe=False)
```

product_sales_stats:

- Zwraca dane o sprzedaży dla konkretnego produktu, identyfikowanego przez product_id przekazane w żądaniu GET.
- W przypadku braku product_id, zwraca błąd.
- Dane dotyczące sprzedaży są agregowane według daty zamówienia i ilości sprzedanych produktów.

```python
def product_sales_stats(request):
    product_id = request.GET.get('product_id')
    if not product_id:
        return JsonResponse({'error': 'Brak podanego ID produktu'}, status=400)

    sales_data = (
        Order.objects.filter(product_id=product_id)
        .values('order_date')
        .annotate(total_quantity=Sum('quantity'))
        .order_by('order_date')
    )

    return JsonResponse(list(sales_data), safe=False)
```

sales_in_date_range:

- Również dostępna tylko dla zalogowanych użytkowników.
- Zwraca dane o sprzedaży w określonym zakresie dat, przekazanych jako parametry GET (startDate i endDate).
- W przypadku braku dat, zwraca błąd.
- Dane są agregowane i zwracane w formacie JSON, zawierają informacje o klientach, produktach i ilości sprzedanych produktów.

```python
@login_required
def sales_in_date_range(request):
    start_date = request.GET.get('startDate')
    end_date = request.GET.get('endDate')

    if not start_date or not end_date:
        return JsonResponse({'error': 'Nie podano daty początkowej lub końcowej'},
status=400)

    start_date_aware = make_aware(datetime.strptime(start_date, '%Y-%m-%d'))
    end_date_aware = make_aware(datetime.strptime(end_date, '%Y-%m-%d'))

    sales_data = Order.objects.filter(order_date__range=[start_date_aware,
end_date_aware]).values(
        'customer__id',
        'customer__name',
        'product__name'
    ).annotate(total_quantity=Sum('quantity')).order_by('customer')

    return JsonResponse(list(sales_data), safe=False)
```

customer_expenses_stats:

- Dostępna tylko dla zalogowanych użytkowników.
- Zwraca statystyki wydatków klienta na podstawie przekazanego customer_id.
- Możliwe jest filtrowanie danych po zakresie dat.
- W przypadku braku customer_id, zwraca błąd.

```python
@login_required
def customer_expenses_stats(request):
    customer_id = request.GET.get('customer_id')
    start_date = request.GET.get('start_date')
    end_date = request.GET.get('end_date')

    if customer_id:
        query = Order.objects.filter(customer_id=customer_id)
        if start_date:
            query = query.filter(order_date__gte=start_date)
        if end_date:
            query = query.filter(order_date__lte=end_date)

        data = list(query.values('order_date')
                    .annotate(total_amount=Sum('amount'))
                    .order_by('order_date'))
        return JsonResponse(data, safe=False)

    return JsonResponse({'error': 'No customer ID provided'}, status=400)
```

payment_method_stats:

- Dostępna tylko dla zalogowanych użytkowników.
- Zwraca dane o sprzedaży z podziałem na metody płatności w określonym zakresie dat.
- Wymaga podania payment_method_id, start_date i end_date w żądaniu GET.
- W przypadku braku wymaganych parametrów, zwraca błąd.
- Dane są agregowane według daty i metody płatności, a następnie zwracane w formacie JSON.

```python
@login_required
def payment_method_stats(request):
    payment_method_id = request.GET.get('payment_method_id')
    start_date_str = request.GET.get('start_date')
    end_date_str = request.GET.get('end_date')

    if not all([payment_method_id, start_date_str, end_date_str]):
        return JsonResponse({'error': 'Brak wymaganych parametrów'}, status=400)

    start_date_naive = datetime.strptime(start_date_str, '%Y-%m-%d')
    end_date_naive = datetime.strptime(end_date_str, '%Y-%m-%d')

    end_date_naive = end_date_naive.replace(hour=23, minute=59, second=59)

    start_date_aware = timezone.make_aware(start_date_naive, timezone.
get_default_timezone())
    end_date_aware = timezone.make_aware(end_date_naive, timezone.
get_default_timezone())

    data = (
        Order.objects.filter(payment_method=payment_method_id, order_date__range=[
start_date_aware, end_date_aware])
        .annotate(date=TruncDay('order_date'))
        .values('date')
        .annotate(total_amount=Sum('amount'))
        .order_by('date')
    )

    response_data = list(data)
    return JsonResponse(response_data, safe=False)
```

Każdy z tych widoków obsługuje konkretną funkcję analityczną lub informacyjną w aplikacji, służąc do wyświetlania i przetwarzania danych z bazy danych w sposób, który jest przyjazny dla użytkownika dashboardu. Widoki te są istotnym elementem aplikacji, pozwalając na wgląd w różne aspekty działalności - od statystyk sprzedaży produktów, przez analizę wydatków klientów, po szczegółowe dane dotyczące poszczególnych metod płatności.

Pozostałe Widoki:

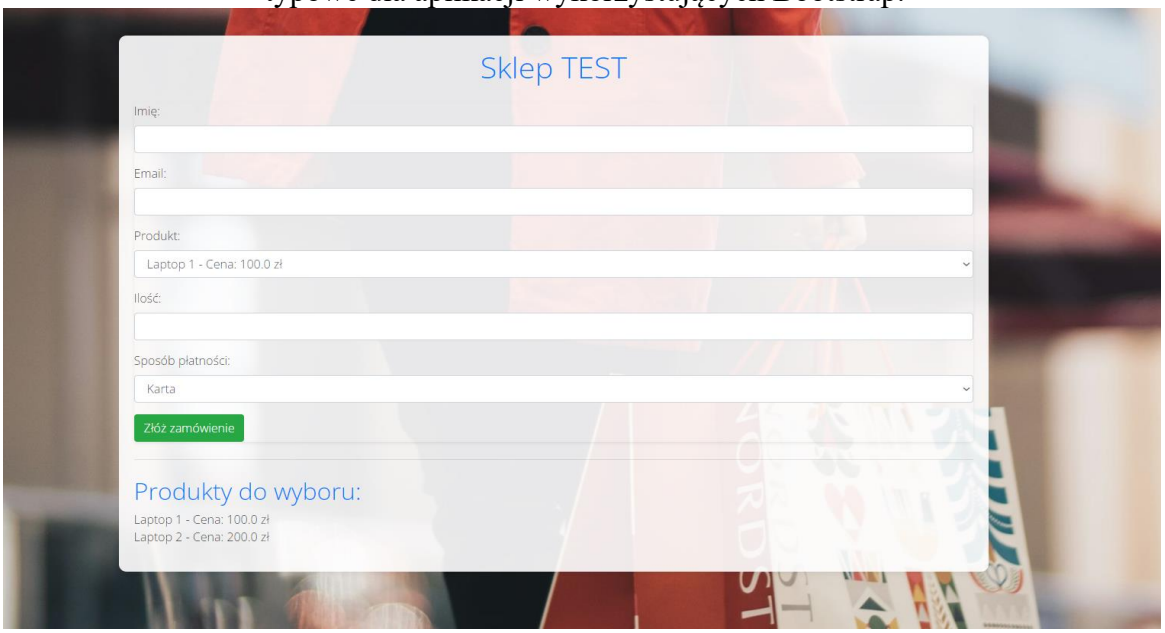
Widoki your_customers_view, your_payment_methods_view, product_sales_stats_view, sales_data_table_view, customer_expenses, payment_method_stats_page to głównie widoki renderujące szablony html lub zwracające dane w formacie JSON do tych szablonów. Są one zabezpieczone dekoratorem login_required, co oznacza, że dostęp do nich mają tylko zalogowani użytkownicy ze statusem admina.

```python
@login_required
def product_sales_stats_view(request):
    return render(request, 'product_sales_stats.html')

@login_required
def sales_data_table_view(request):
    return render(request, 'sales_data_table.html')

@login_required
def customer_expenses(request):
    return render(request, 'customer_expenses_stats.html')

@login_required
def payment_method_stats_page(request):
    return render(request, 'payment_method_expenses_stats.html')
```

Wygląd aplikacji

Struktura i Design

- Szablon HTML: Używa standardowego szablonu HTML5 (<!DOCTYPE html>).
- Język: Określony jako angielski (<html lang="en">).
- Meta Tagi: Zawiera meta tagi dla kodowania znaków, kompatybilności z IE i widoku na urządzeniach mobilnych.
- Tytuł Strony: Tytuł strony ustawiony na "Sklep".
- Linki do zewnętrznych zasobów:
    - Bootstrap: Do stylizacji strony wykorzystuje Bootstrap 4.5.2.
    - Google Fonts: Używa fontu 'Open Sans' z Google Fonts.
- Styl CSS wewnętrzny: Zdefiniowano niestandardowe style CSS, aby dostosować wygląd, w tym tło, kolory i stylizację przycisków.
- Główna Zawartość
    - Kontener: Zawartość strony zawarta jest w divie o klasie container, co jest typowe dla stylów Bootstrapa.
    - Nagłówek: Tytuł strony "Sklep TEST" wyróżniony w nagłówku.
    - Formularz Zamówienia:
        - Formularz służy do składania zamówień. Przesyła dane do widoku create_order.
        - Zawiera pola do wprowadzenia danych przez użytkownika, takie jak imię, email, wybór produktu, ilość i sposób płatności.
        - Lista produktów oraz metody płatności są generowane dynamicznie z dostępnych danych.
        - Przycisk "Złóż zamówienie" służy do wysłania formularza.
    - Funkcjonalności
        - Komunikaty: Obsługa komunikatów Django - wyświetla komunikaty zwrotne po wykonaniu akcji (np. po złożeniu zamówienia).
        - Lista Produktów: Pod formularzem znajduje się lista produktów z ich cenami, wygenerowana dynamicznie za pomocą pętli w szablonie Django.
        - Skrypty JS i Bootstrap
        - Skrypty JavaScript: Dołączone skrypty jQuery i Bootstrap, co jest typowe dla aplikacji wykorzystujących Bootstrap.



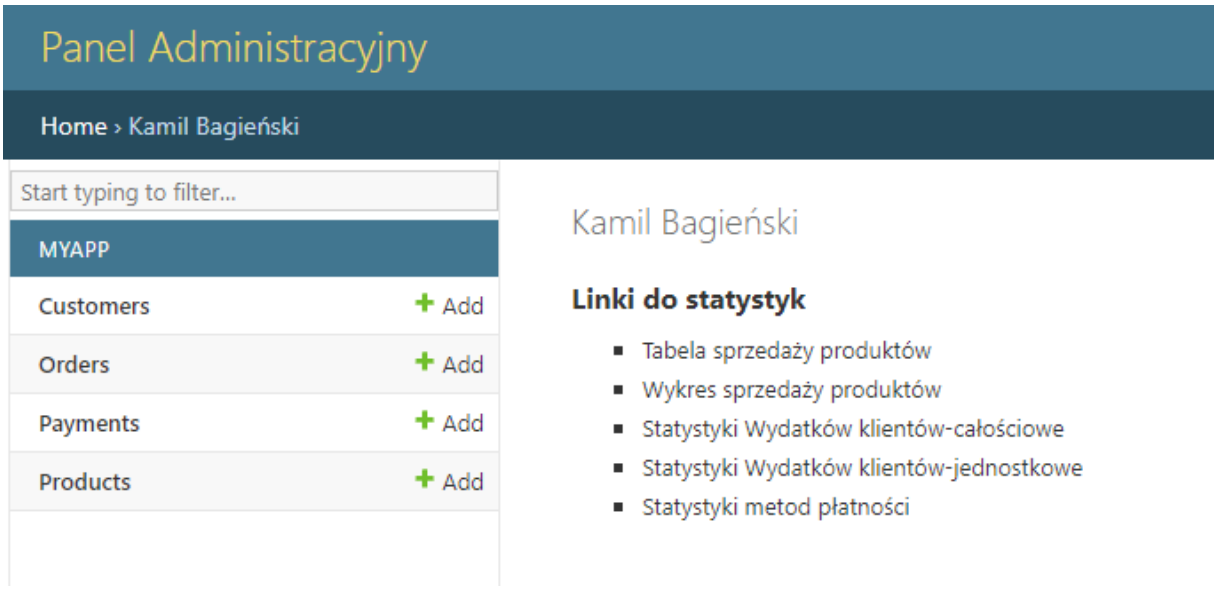
Podsumowanie

Strona główna aplikacji Django jest prosto zaprojektowana, ale funkcjonalna. Umożliwia użytkownikom składanie zamówień, wybierając produkty i określając szczegóły takie jak ilość i sposób płatności. Design jest czysty i uporządkowany, z dobrze zorganizowanym układem i łatwym w użyciu interfejsem. Wykorzystanie Bootstrapa zapewnia responsywność i estetyczny wygląd, a dynamiczne generowanie treści (produkty, metody płatności) sprawia, że strona jest interaktywna i aktualna w zależności od dostępnych danych w aplikacji.

Panel administracyjny

Panel administracyjny to typowy panel, który zapewnie framework Django. Za pomocą dodania modelów w admin.py jest możliwość dodania wszystkiego do bazy przez administratora(albo superusera w przypadku widoku developerskiego).

```python
from django.contrib import admin
from django.urls import path
from .models import Product, Order, Customer, Payment
from .views import customer_sales_stats
from django.contrib.admin import AdminSite



class MyAdminSite(AdminSite):
    site_header = 'Panel Administracyjny'
    index_template = 'admin/custom_admin_template.html'
    index_title = "Kamil Bagieński"

my_admin_site = MyAdminSite(name='myadmin')

my_admin_site.register(Customer)
my_admin_site.register(Product)
my_admin_site.register(Order)
my_admin_site.register(Payment)
```



Dodane zostały linki do panelu administracyjnego, które przywołują szablony html odnośnie analiz statystycznych. Jest to dodatkowa funkcjonalność, bo dodawanie linków nie mieści się w działaniu standardowego panelu administracyjnego Django. Dodałem to w następujący sposób:

```
{% extends "admin/base_site.html" %}

{% block content %}
    {{ block.super }}

    <h2>Linki do statystyk</h2>
    <ul>
        <li><a href="{% url 'sales-data-table' %}">Tabela sprzedaży produktów</a></
li>
        <li><a href="{% url 'product-sales-stats-view' %}"
>Wykres sprzedaży produktów</a></li>
        <li><a href="{% url 'sales_stats' %}"
>Statystyki Wydatków klientów-całościowe</a></li>
        <li><a href="{% url 'customer_expenses' %}"
>Statystyki Wydatków klientów-jednostkowe</a></li>
        <li><a href="{% url 'payment-method-stats-page' %}"
>Statystyki metod płatności</a></li>
    </ul>
{% endblock %}
```

Szablon ten ma za zadanie załadownie na początek głównego panelu administracyjnego i dodanie bloku odnośnie linków. Dzięki czemu mamy dostęp do tabel i wykresów z poziomu nie wpisywanych linków w przeglądarkę, a standardowych hiperłączy. Aby skorzystać z tego panelu należy wejść na adres:
http://localhost:8000/admin/

Analiza danych

Jedyna rzecz, która nie jest intyuicyjna w Django to budowanie panelu administracyjnego, jest wiele dodatkowych bibliotek, ale nie są one zaktualizowane do najnowsze Django. Najczęściej zostały one w wersji 3.0, i niestety nikt już nie postarał się o ich aktualizację. Początkowo wszystkie dane były widoczne w panelu administracyjnym za pomocą widgetów. Jednakże te rozwiązanie na dłuższa metę nie było czytelne. Postawiłem więc na prostsze i czytelniejsze rozwiązanie jakim jest chart.js w szablonach html. Które świetnie nadają się do wizualizacji i analizy danych do aplikacji tego typu.
W rozdziale z modelami, opisałem w jaki sposób są one przekazywane do szablonów. Teraz przedstawię ich wygląd oraz budowę.

Statystyki Sprzedaży Produktów – Tabela

Szablon sales_data_table.html w aplikacji Django pełni rolę interaktywnej strony do wyświetlania statystyk sprzedaży produktów. Oto prosty opis jego funkcji i zawartości:

Struktura i Wygląd

Podstawowy HTML: Strona rozpoczyna się od standardowego szablonu HTML5.

Ustawienia Meta: Zawiera meta tagi dla kodowania znaków i responsywności.

Tytuł: Tytuł strony to "Statystyki Sprzedaży Produktów - Tabela".

Stylizacja: Stylizacja jest prosta, wykorzystuje Arial jako font i zawiera podstawowe style dla tabeli i przycisków.

Główne Elementy

- Przycisk Powrotu: Na początku strony znajduje się przycisk, który po kliknięciu przekierowuje użytkownika na stronę administracyjną (/admin).
- Nagłówek: Wyświetla tytuł strony.
- Formularz Wyboru Daty: Pozwala użytkownikowi wybrać zakres dat (data początkowa i końcowa) dla wyświetlanych danych.
- Tabela Sprzedaży: Pusta tabela z nagłówkami dla ID klienta, imienia klienta, nazwy produktu oraz ilości zakupionych produktów.

Skrypty i Interakcja

- Obsługa Formularza: Skrypt JavaScript obsługuje zdarzenie submit formularza. Po wysłaniu formularza skrypt wykonuje zapytanie do odpowiedniego endpointu API aplikacji, przekazując wybrane daty jako parametry.
- Wypełnianie Tabeli Danych: Po otrzymaniu odpowiedzi z API, skrypt dynamicznie wypełnia tabelę danymi o sprzedaży produktów w wybranym zakresie dat.

Statystyki Sprzedaży Produktów - Tabela

Data początkowa: 01.01.2024 Data końcowa: 29.01.2024 Pokaż dane

Dane Sprzedaży Produktów

Id Klienta	Imię Klienta	Nazwa Produktu	Ilość
13b12869-4c23-4561-b112-cac0db891588	Lilia	Laptop 1	3
2ffa037b-a4fb-46c5-9c19-7fd864afaf75	data	Laptop 2	10
3077b891-6f13-4513-8cf1-42267f940389	Godzinas	Laptop 2	44
3a20ae4d-b5ef-4909-aca7-f352a0e8fb6b	test2	Laptop 1	11
4ffb89fc-2d48-4d7a-82c5-93331c3a8f92	Kamil	Laptop 1	1
5fff2a55-ac47-4fba-93f3-b691606e6f60	Test pritna	Laptop 2	10
70b94155-fb25-4427-aa8c-b7bd00484ec9	te4	Laptop 1	9
9b1c2252-d97e-4676-95c6-abe8e4bd1116	Bonjour	Laptop 1	2
be1b1816-812b-4320-a899-b07f7964242a	test_css	Laptop 1	3
c0affb65-1d5e-4d66-ac75-3019b9e3da8e	tert	Laptop 2	3
cc842407-6304-4147-aab8-d371911a68ee	Kamil	Laptop 1	220
cc842407-6304-4147-aab8-d371911a68ee	Kamil	Laptop 2	338
deaa62a2-f019-4875-9ce6-1796701e7d83	data	Laptop 2	2
dfe4f8fc-87fa-4f57-b84a-498faaa00e0f	rest3	Laptop 1	2

Podsumowanie

Szablon sales_data_table.html jest przeznaczony do dynamicznego wyświetlania danych sprzedaży produktów w aplikacji. Użytkownik może określić zakres dat, a tabela zostanie wypełniona odpowiednimi danymi sprzedaży, pobranymi z backendu aplikacji. Jest to przykład interaktywnej strony, która zapewnia użytkownikom dostęp do szczegółowych danych w łatwy do analizowania sposób. Wykorzystanie JavaScript do dynamicznego ładowania danych sprawia, że strona jest bardziej interaktywna i przyjazna użytkownikowi.

Statystyki Sprzedaży Produktów

Szablon product_sales_stats.html w aplikacji Django pełni funkcję interaktywnej strony do wyświetlania statystyk sprzedaży poszczególnych produktów. Oto jego opis w prostych słowach:

Główne Elementy Strony

- Struktura HTML: Strona jest zbudowana przy użyciu standardowego szablonu HTML5.
- Tytuł Strony: Tytuł strony to "Statystyki Sprzedaży Produktów".
- Stylowanie: Stylowanie strony jest proste, z fontem Arial i podstawowymi stylami dla elementów interfejsu, takich jak select i canvas.
- Wykres: Strona zawiera obszar na wykres (element <canvas>), który będzie używany do wizualizacji danych sprzedaży.
- Przycisk Powrotu: Przycisk na początku strony umożliwia powrót do panelu administracyjnego.

Interaktywność i Funkcje JavaScript

- Ładowanie Produktów: Po załadowaniu strony, skrypt JavaScript wykonuje zapytanie do API aplikacji, aby pobrać listę produktów. Produkty te są następnie wyświetlane w elemencie <select>.
- Tworzenie Wykresu: Skrypt używa biblioteki Chart.js do tworzenia wykresu słupkowego, który przedstawia ilość sprzedanych produktów w określonym okresie.
- Aktualizacja Wykresu: Po wybraniu produktu z listy rozwijanej, wykres jest aktualizowany, aby pokazać dane dotyczące wybranego produktu. Skrypt pobiera dane sprzedaży dla wybranego produktu z API aplikacji i aktualizuje wykres.
- Interaktywny Wybór Produktu: Użytkownik może wybrać produkt z rozwijanej listy, a wykres zostanie zaktualizowany, aby pokazać dane dla tego konkretnego produktu.

Podsumowanie

Szablon product_sales_stats.html jest interaktywną stroną umożliwiającą wizualizację danych sprzedaży produktów w aplikacji Django. Wykorzystanie JavaScript i Chart.js pozwala na dynamiczne generowanie wykresów w zależności od wyboru użytkownika, co czyni tę stronę użytecznym narzędziem do analizy trendów sprzedaży. Jest to przykład, jak można łączyć backend aplikacji Django z interaktywnymi elementami frontendowymi, aby stworzyć funkcjonalne i estetycznie przyjemne rozwiązanie dla użytkowników.

Statystyki Wydatków Klientów- Całościowe

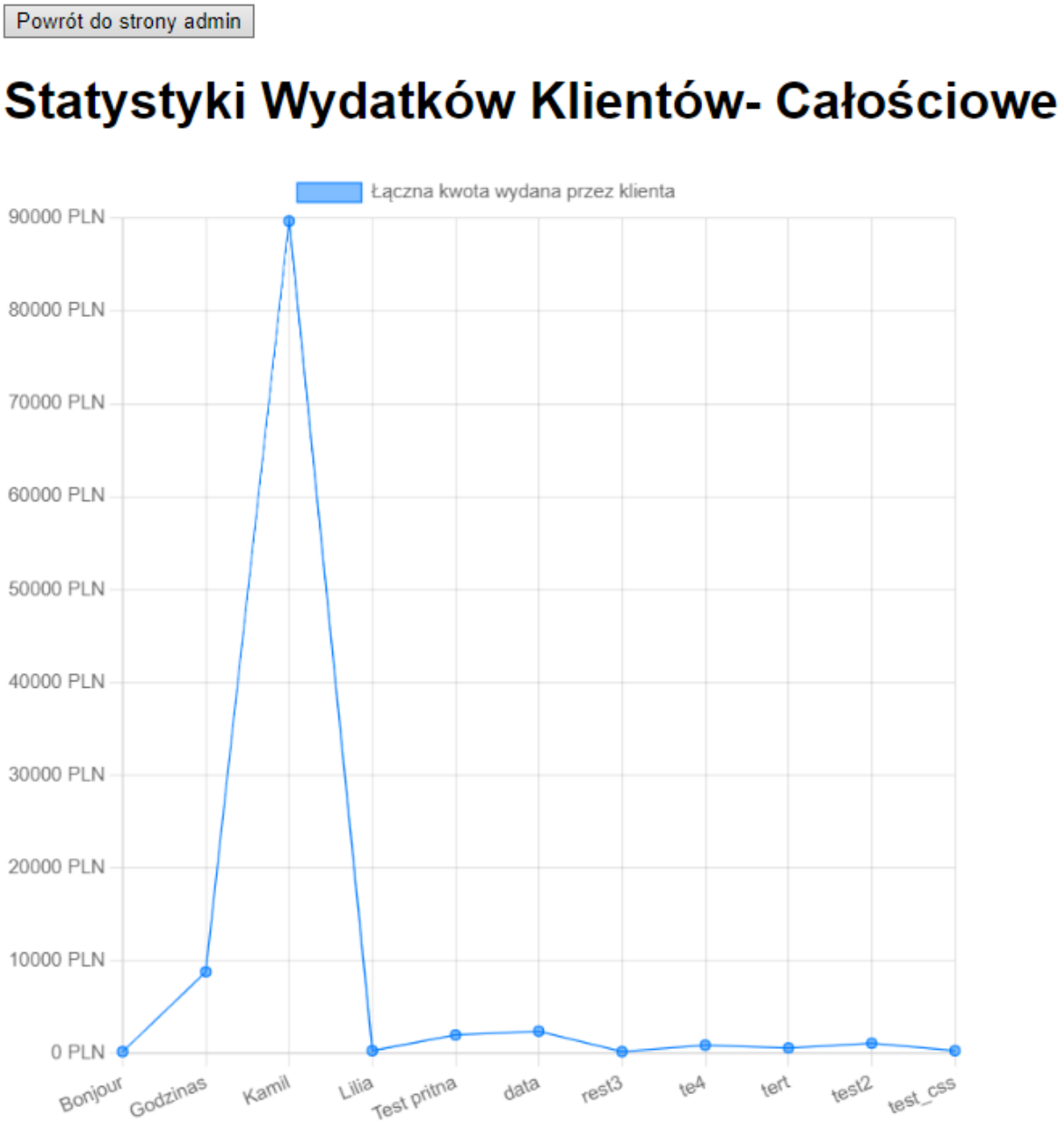
Szablon sales_stats.html w aplikacji Django służy do prezentacji statystyk wydatków klientów. Oto uproszczony opis tego, co robi ten szablon:

Główne Elementy Strony

- Struktura HTML: Używa standardowego szablonu HTML.
- Tytuł: Tytuł strony ustawiony na "Statystyki Wydatków Klientów".
- Stylowanie: Proste stylowanie CSS z fontem Arial. Szablon zawiera także stylizacje dla wykresu (element <canvas>).

Interaktywność i Funkcje JavaScript

- Przycisk Powrotu: Na górze strony znajduje się przycisk, który po kliknięciu przekierowuje użytkownika na stronę administracyjną aplikacji Django (/admin).
- Nagłówek: Duży nagłówek "Statystyki Wydatków Klientów - Całościowe" informuje użytkownika o tematyce strony.
- Obszar Wykresu: Używa elementu <canvas> do wyświetlania wykresu.
- Interaktywność i Wykres
- Generowanie Wykresu: Strona korzysta z biblioteki Chart.js do generowania wykresu liniowego, który pokazuje łączne wydatki klientów.
- Dane Wykresu: Dane do wykresu są dynamicznie wstrzykiwane z kontekstu Django (sales), gdzie każdy punkt na wykresie reprezentuje sumę wydatków klienta. Etykiety na osi X to nazwy klientów, a wartości na osi Y to łączna kwota wydatków.
- Opcje Wykresu: Wykres ma dostosowane opcje, w tym skalę osi Y zaczynającą się od zera i etykiety formatujące wartości jako wartości pieniężne (PLN).



Podsumowanie

Ta strona jest użytecznym narzędziem dla administratora aplikacji do analizy ogólnych trendów wydatków klientów. Dzięki wizualizacji danych w formie wykresu liniowego, administrator może łatwo zidentyfikować, którzy klienci wydali najwięcej, co jest przydatne do analizy zachowań zakupowych klientów i planowania strategii sprzedaży. Wykorzystanie Chart.js pozwala na elegancką i czytelną prezentację danych, co ułatwia interpretację statystyk.

Statystyki Wydatków Klientów- jednostkowe

Szablon customer_expenses_stats.html w aplikacji Django jest przeznaczony do wyświetlania statystyk wydatków poszczególnych klientów. Oto uproszczony opis jego funkcji:

Struktura i Wygląd

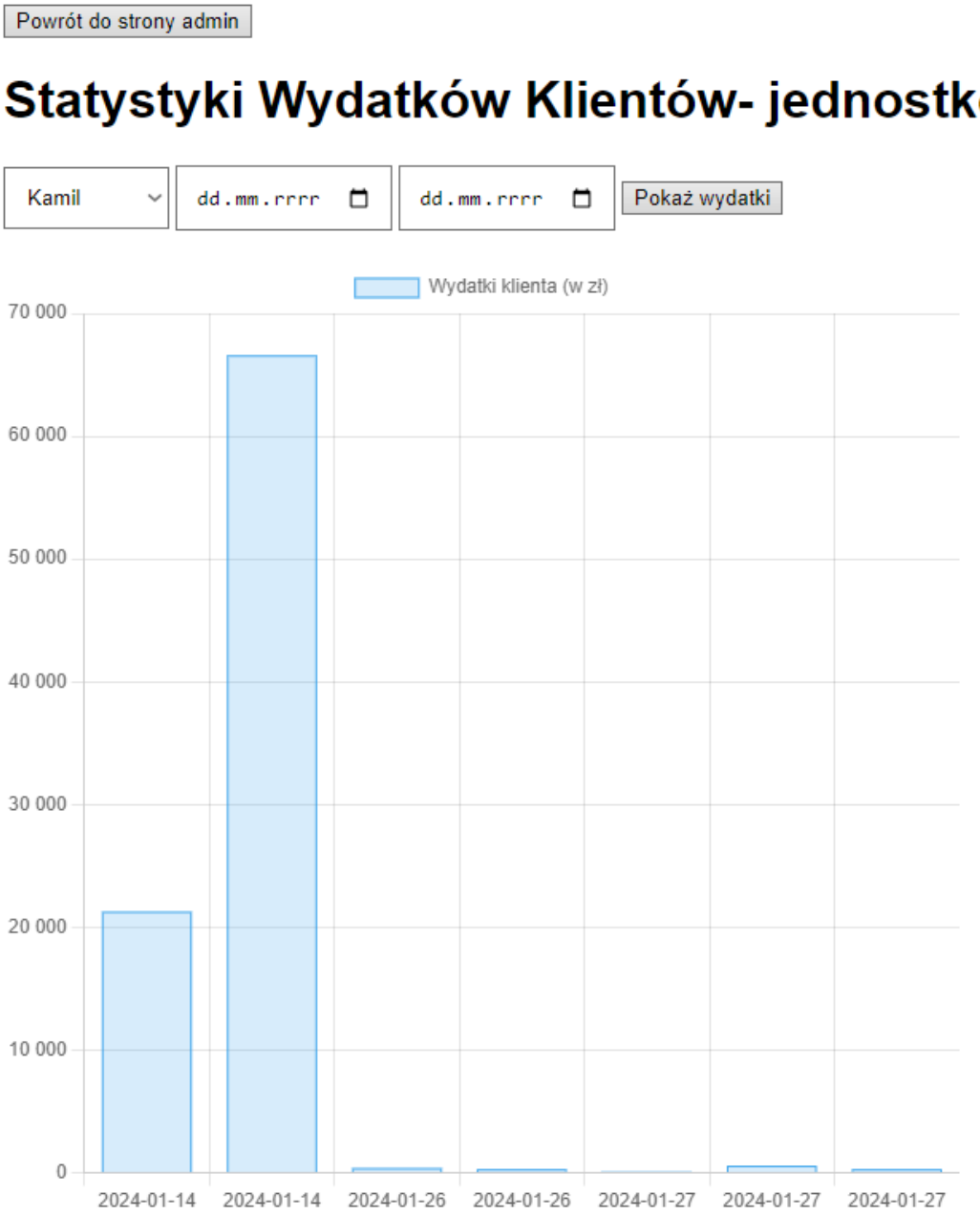
- Podstawowy HTML: Strona rozpoczyna się od standardowego szablonu HTML5.
- Tytuł: Tytuł strony to "Statystyki Wydatków Klientów".
- Stylowanie: Proste stylowanie CSS, używające fontu Arial i podstawowych stylów dla elementów wykresu i kontrolek.
- Główne Elementy
- Przycisk Powrotu: Na początku strony znajduje się przycisk, który po kliknięciu przekierowuje użytkownika na stronę administracyjną (/admin).
- Nagłówek: Wyróżniony nagłówek "Statystyki Wydatków Klientów - jednostkowe".

Kontrolki:

- Wybór klienta z rozwijanej listy (<select>).
- Pola wyboru daty (<input type="date">) dla określenia zakresu czasowego.
- Przycisk do aktualizacji wykresu.

Wykres i Interaktywność

- Wykres Wydatków:
    - Strona używa biblioteki Chart.js do wyświetlania wykresu słupkowego, który pokazuje wydatki klienta w wybranym zakresie czasowym.
- Dynamika:
    - Po załadowaniu strony, skrypt JavaScript wykonuje zapytanie do API aplikacji, aby pobrać listę klientów, a następnie aktualizuje wykres na podstawie wybranego klienta i zakresu dat.
- Interakcja z Użytkownikiem:
    - Użytkownik może wybrać klienta i zakres dat, a wykres zostanie zaktualizowany, aby pokazać wydatki dla tego konkretnego klienta w wybranym okresie.



Podsumowanie

Szablon customer_expenses_stats.html jest interaktywną stroną do analizy wydatków klientów w aplikacji. Dzięki dynamicznemu ładowaniu danych klientów i możliwości wyboru zakresu dat, użytkownik może uzyskać szczegółowe informacje o wydatkach poszczególnych klientów. Wykorzystanie Chart.js umożliwia wizualną i intuicyjną prezentację tych danych, co jest przydatne w analizie trendów zakupowych i zachowań klientów.

Statystyki Według Metody Płatności
Szablon payment_method_expenses_stats.html w aplikacji Django służy do prezentacji statystyk związanych z wydatkami dokonanymi przy użyciu różnych metod płatności. Oto uproszczony opis tego szablonu:

Struktura i Wygląd

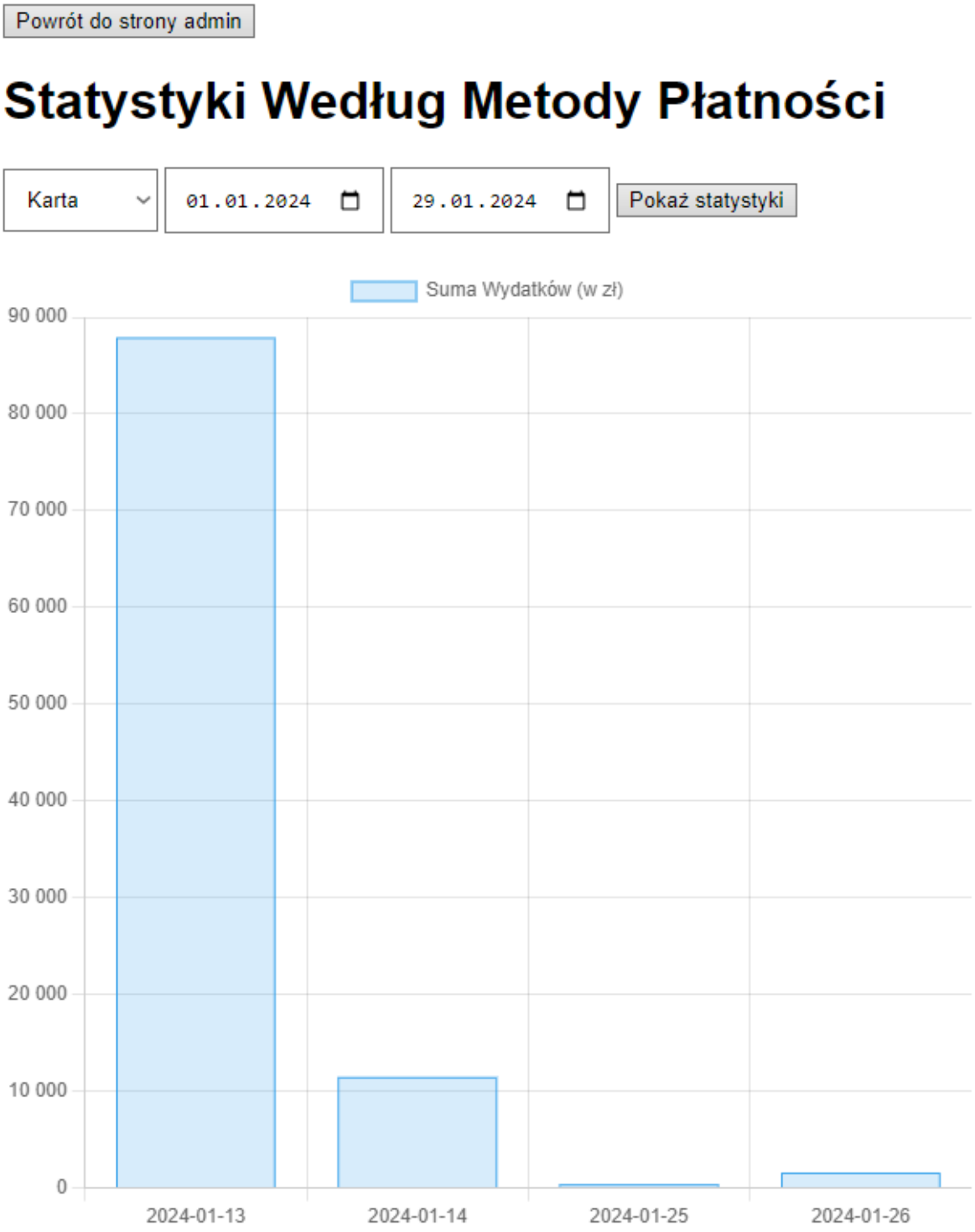
- Szablon HTML: Używa standardowej struktury HTML5.
- Tytuł Strony: "Statystyki Według Metody Płatności".
- Stylowanie: Proste stylowanie CSS z użyciem fontu Arial. Stylizuje elementy takie jak przyciski, pola wyboru daty i obszar wykresu.

Główne Elementy

- Przycisk Powrotu: Pozwala użytkownikowi na powrót do strony administracyjnej aplikacji.
- Nagłówek: Duży nagłówek informujący o treści strony - "Statystyki Według Metody Płatności".
- Kontrolki Wyboru:
    - Wybór metody płatności z rozwijanej listy.
    - Pola do wyboru daty początkowej i końcowej.
    - Przycisk do wyświetlenia statystyk.

Wykres i Interaktywność

- Wykres Statystyk:
  Strona wykorzystuje bibliotekę Chart.js do wyświetlania wykresu słupkowego, który prezentuje wydatki według wybranej metody płatności i zakresu dat.
- Dynamika Strony:
  Po załadowaniu strony, skrypt JavaScript wykonuje zapytanie do API aplikacji, aby pobrać dostępne metody płatności, a następnie aktualizuje wykres na podstawie wybranej metody płatności i dat.
- Interakcja z Użytkownikiem:
  Użytkownik może wybrać metodę płatności i zakres dat, a wykres zostanie zaktualizowany, aby pokazać wydatki dla tej konkretnej metody płatności w wybranym okresie.



Podsumowanie

Szablon payment_method_expenses_stats.html jest użytecznym narzędziem dla administratorów do analizy wydatków dokonanych za pomocą różnych metod płatności w określonym czasie. Dzięki dynamicznemu ładowaniu danych i możliwości wyboru metody płatności oraz zakresu dat, użytkownik może uzyskać szczegółowy wgląd w preferencje płatnicze klientów. Wykorzystanie Chart.js pozwala na wizualne i intuicyjne przedstawienie tych danych, co jest pomocne w analizie trendów i planowaniu strategii biznesowych.

Podsumowanie

W ramach mojego projektu "Aplikacja do Zarządzania Procesem Sprzedaży", osiągnąłem główne cele edukacyjne związane z językiem SQL i systemami zarządzania bazami danych. Udało mi się stworzyć funkcjonalną aplikację webową na frameworku Django, która efektywnie korzysta z baz danych SQL do przechowywania i analizy informacji handlowych.

Moje kluczowe osiągnięcia w tym projekcie to:

1. Efektywne Zarządzanie Elementami Procesu Sprzedaży: Skoncentrowałem się na opracowaniu skutecznego systemu do obsługi klientów, produktów, zamówień i płatności, co jest fundamentalne dla każdego systemu e-commerce.
2. Zaawansowane Zastosowanie SQL: Z sukcesem wykorzystałem różne rodzaje złączeń SQL, demonstrując moje zdolności do pracy z złożonymi strukturami danych.
3. Rozwój Interfejsu Użytkownika i Panelu Administracyjnego: Zaprojektowałem intuicyjny interfejs użytkownika i rozbudowany panel administracyjny, co znacząco ułatwia zarządzanie danymi.
4. Analiza Danych: Wprowadziłem funkcjonalności analizy danych, takie jak statystyki wydatków klientów czy preferencje płatnościowe, dodając istotną wartość biznesową do aplikacji.
5. W mojej dokumentacji szczegółowo opisałem każdy aspekt projektu, od jego uruchomienia, przez budowę i modele bazy danych, aż po omówienie widoków i funkcjonalności aplikacji.

Ten projekt nie tylko pokazuje moje umiejętności w tworzeniu aplikacji z użyciem Django i SQL, ale także moją zdolność do analizowania i prezentowania danych w przystępny sposób. Jestem przekonany, że te doświadczenia będą cennym atutem w mojej dalszej karierze zawodowej, szczególnie w obszarze technologii i informatyki.

Podsumowując, uważam, że projekt "Aplikacja do Zarządzania Procesem Sprzedaży" jest znaczącym krokiem w moim rozwoju jako programisty i analityka danych, przyczyniając się do budowania moich praktycznych umiejętności, które są niezbędne w dzisiejszym, dynamicznie rozwijającym się świecie technologii.