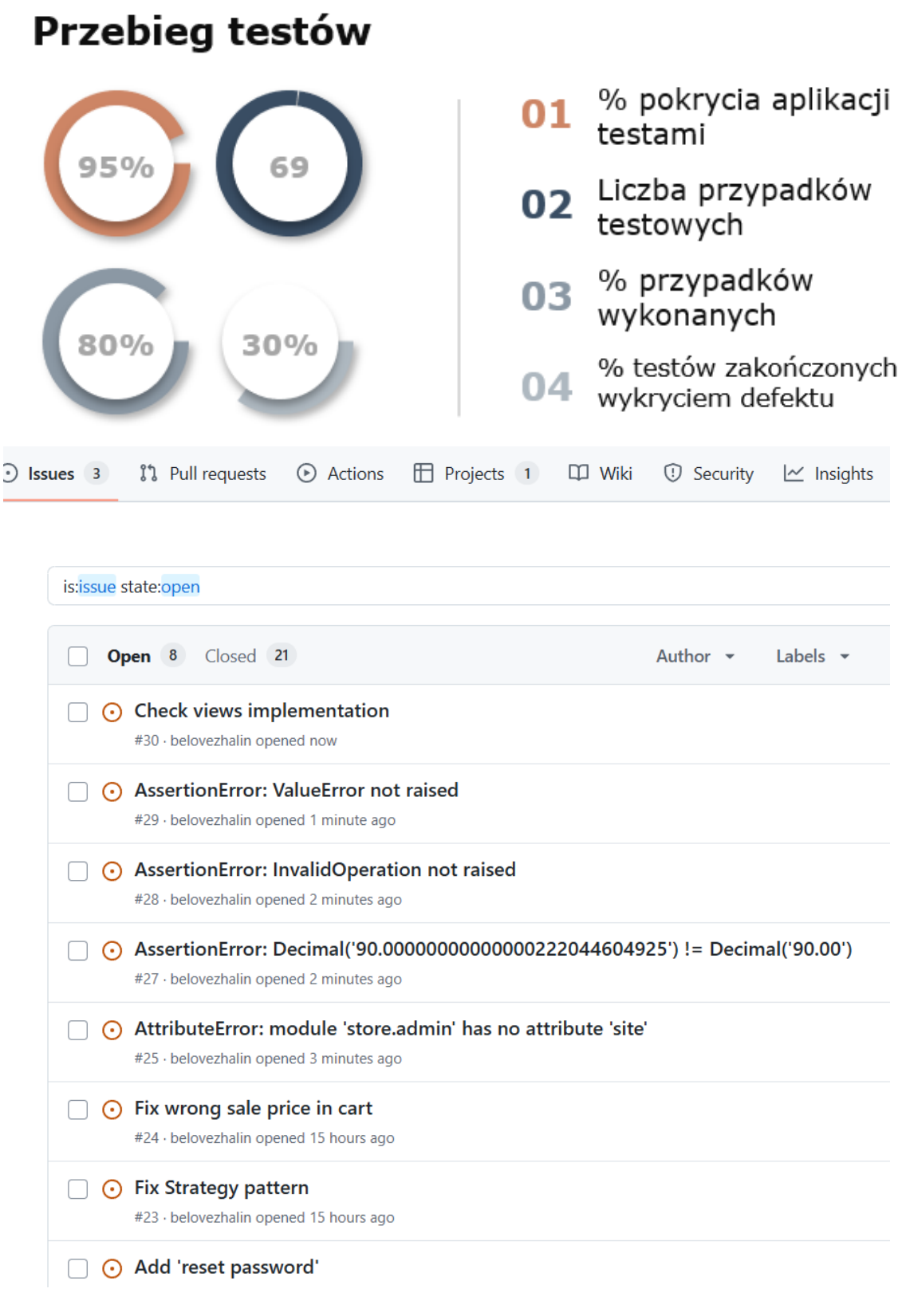
Testowanie oprogramowania Giftify

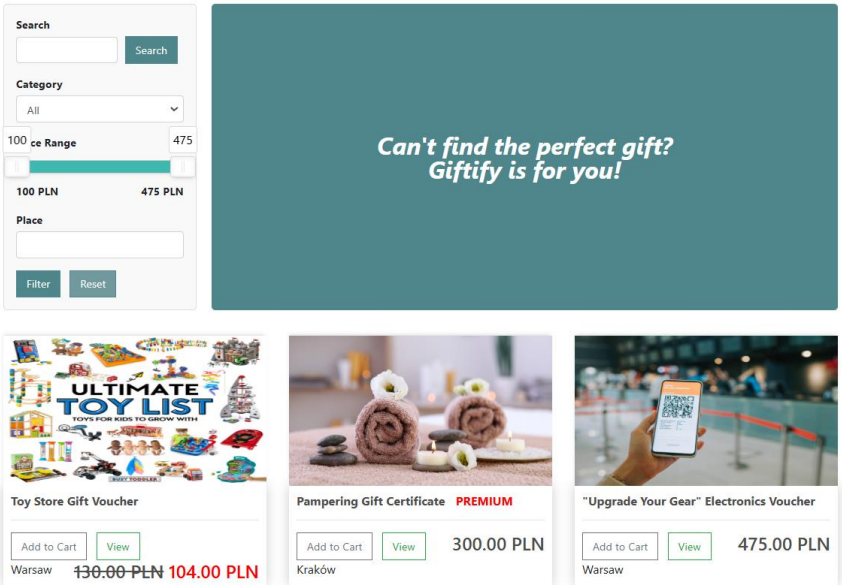
Implementacja projektu, pisanie przypadków testowych oraz wykonanie testów: Anhelina Belavezha

https://github.com/belovezhalin/giftify



Opis oprogramowania

Giftify to platforma sprzedażowa oferująca certyfikaty prezentowe w różnych kategoriach, takich jak SPA, moda, kulinaria, aktywności i rozwój osobisty. Aplikacja umożliwia użytkownikom wygodne przeglądanie ofert, filtrowanie certyfikatów według preferencji oraz zarządzanie koszykiem zakupów.

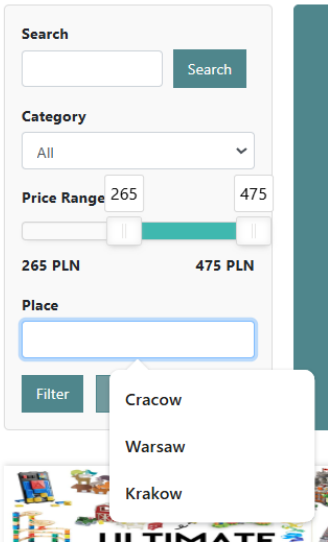


Wymagania funkcjonalne

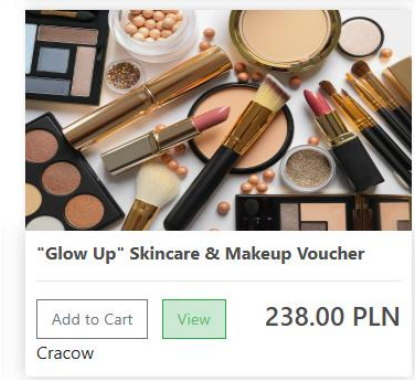
1. ***Przeglądanie katalogu certyfikatów***
   - Użytkownicy mogą przeglądać dostępne certyfikaty w różnych kategoriach oraz filtrować je według podstawowych kryteriów, takich jak cena, kategoria, lokalizacja.
2. ***Wyświetlanie szczegółów certyfikatu***
   - Użytkownik ma możliwość obejrzenia szczegółowych informacji o wybranym certyfikacie, w tym opisu, warunków realizacji oraz sugestii podobnych certyfikatów.
3. ***Dodawanie do koszyka***
   - Użytkownik może dodawać certyfikaty do koszyka. Aplikacja będzie przechowywać stan koszyka na czas trwania sesji użytkownika.
4. ***Zarządzanie koszykiem***
   - Użytkownik może dodawać i usuwać certyfikaty z koszyka oraz przeglądać łączny koszt zakupów. Finalizacja zakupów nie obejmuje integracji płatności, jednak aplikacja powinna umożliwiać przechowywanie informacji o wybranych certyfikatach.
5. ***Powiadomienia o nowościach i dostępności***
   - Użytkownicy mogą otrzymywać powiadomienia o nowych dostępnych certyfikatach w kategoriach, które ich interesują.
6. ***Zarządzanie ofertą przez administratora***
   - Administrator ma możliwość dodawania, usuwania oraz edycji certyfikatów dostępnych w katalogu. Zmiany te są od razu widoczne dla użytkowników przeglądających ofertę.

Interface testing

1. Dynamiczna zmiana najmniejszej ceny podczas filtrowania może wprowadzić w błąd użytkownika.



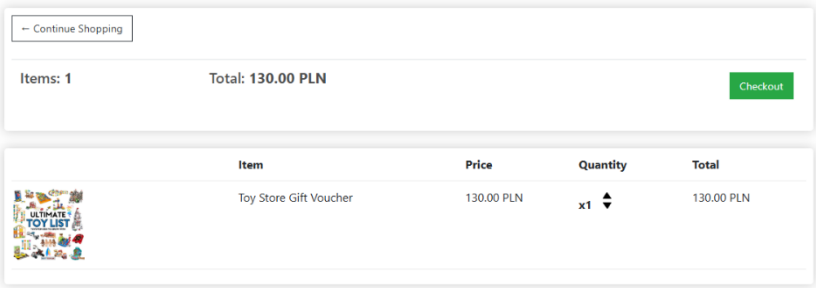
2. Brak listy rozwijanej: różne napisanie nazwy miasta (Cracow / Krakow w języku angielskim, lub pomijanie znaków polskich: Poznan) spowoduje brak wyszukiwań, o ile stosuje się ścisłe poszukiwanie.



3. Brak działań przy naciśnięciu View na każdym z certyfikatów wprowadza w błąd, brak dodatkowej informacji o certyfikacie (tylko nazwa, cena i miasto).

4. Nieistniejący adres mailowy do kontaktu.

5. Produkty ze zniżką już nie są objęte promocją w koszyku i dalszych krokach zakupu.



Unit tests

test_admin.py

Te testy weryfikują konfigurację interfejsu administracyjnego Django dla modeli Product i Category.

1. **test_list_display**: Upewnia się, że atrybut list_display w ProductAdmin jest ustawiony na wyświetlanie pól ('name', 'price', 'category', 'place', 'discount', 'special_mark').
2. **test_list_filter**: Sprawdza, czy atrybut list_filter w ProductAdmin zawiera pole category.
3. **test_search_fields**: Weryfikuje, czy atrybut search_fields w ProductAdmin zawiera ('name', 'category__name').
4. **test_fields**: Potwierdza, że atrybut fields w ProductAdmin zawiera ('name', 'price', 'category', 'image', 'place', 'discount', 'special_mark').
5. **test_category_registered**: Sprawdza, czy model Category jest zarejestrowany w interfejsie administracyjnym Django.
6. **test_product_registered**: Sprawdza, czy model Product jest zarejestrowany w interfejsie administracyjnym Django.

store > tests > test_admin.py > ProductAdminTest

```python
class ProductAdminTest(TestCase):
    def setUp(self):
        self.site = AdminSite()
        self.product_admin = ProductAdmin(Product, self.site)
        self.category = Category.objects.create(name="Spa")
        self.product = Product.objects.create(
            name="Spa Package",
            price=100,
            category=self.category,
            place="Warsaw",
            discount=10,
            special_mark="Special"
        )

    def test_list_display(self):
        self.assertEqual(
            self.product_admin.list_display,
            ('name', 'price', 'category', 'place', 'discount', 'special_mark')
        )

    def test_list_filter(self):
        self.assertEqual(self.product_admin.list_filter, ('category',))

    def test_search_fields(self):
        self.assertEqual(self.product_admin.search_fields, ('name', 'category__name'))

    def test_fields(self):
        self.assertEqual(
            self.product_admin.fields,
            ('name', 'price', 'category', 'image', 'place', 'discount', 'special_mark')
        )

    def test_category_registered(self):
        self.assertTrue(admin.site.is_registered(Category))

    def test_product_registered(self):
        self.assertTrue(admin.site.is_registered(Product))
```

test_decorators.py

Te testy sprawdzają poprawność działania dekoratorów SaleDecorator i SpecialOccasionDecorator.

1. **test_sale_decorator**: Tworzy instancję z produktem i zniżką 10%. Sprawdza, czy cena po zniżce jest poprawnie obliczona jako 90.00 (100 - 10%).
2. **test_special_occasion_decorator**: Tworzy instancję z produktem i okazją "Christmas". Sprawdza, czy mark (oznaczenie specjalnej okazji) jest poprawnie ustawione jako "Special Occasion: Christmas".

Te testy sprawdzają, jak dekoratory radzą sobie z nieprawidłowymi wartościami i przypadkami brzegowymi:

1. **test_sale_decorator_invalid_discount**: Ustawia zniżkę na -10 (nieprawidłowa wartość). Sprawdza, czy wywołanie sale_price rzuca wyjątek InvalidOperation.
2. **test_sale_decorator_high_discount**: Ustawia zniżkę na 150 (zniżka większa niż 100%). Sprawdza, czy rzuca wyjątek InvalidOperation (100 - 150%).
3. **test_special_occasion_decorator_empty_occasion**: Ustawia okazję na pusty ciąg znaków. Sprawdza, czy mark jest poprawnie ustawione jako "Special Occasion: ".

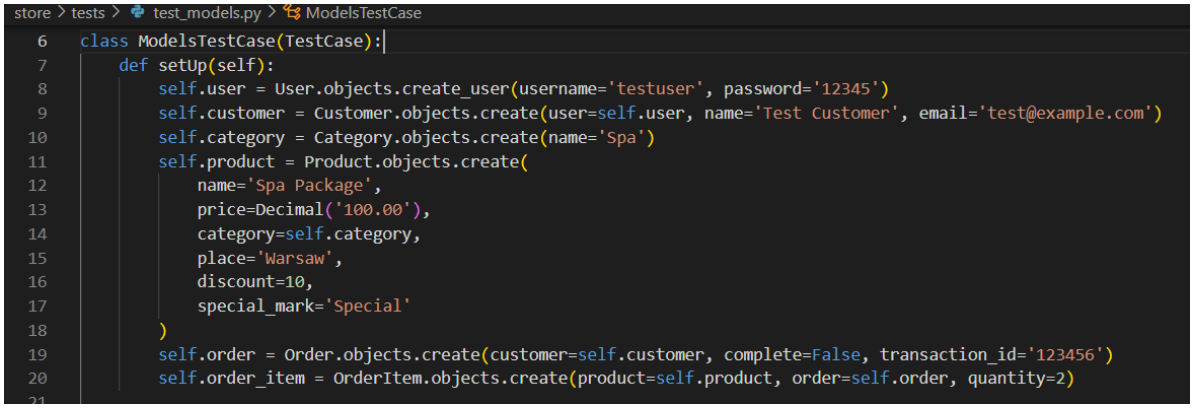
store > tests > test_decorators.py > DecoratorTestCase

```python
class DecoratorTestCase(TestCase):
    def setUp(self):
        self.product = Product(name="Test Product", price=Decimal('100.00'))

    def test_sale_decorator(self):
        discount = 10
        sale_decorator = SaleDecorator(self.product, discount)
        expected_price = Decimal('90.00')
        self.assertEqual(sale_decorator.sale_price, expected_price)

    def test_special_occasion_decorator(self):
        occasion = "Christmas"
        special_decorator = SpecialOccasionDecorator(self.product, occasion)
        expected_mark = "Special Occasion: Christmas"
        self.assertEqual(special_decorator.mark, expected_mark)

    def test_sale_decorator_invalid_discount(self):
        discount = -10
        with self.assertRaises(InvalidOperation):
            SaleDecorator(self.product, discount).sale_price

    def test_sale_decorator_high_discount(self):
        discount = 150
        with self.assertRaises(InvalidOperation):
            SaleDecorator(self.product, discount).sale_price

    def test_special_occasion_decorator_empty_occasion(self):
        occasion = ""
        special_decorator = SpecialOccasionDecorator(self.product, occasion)
        expected_mark = "Special Occasion: "
        self.assertEqual(special_decorator.mark, expected_mark)
```

test_models.py

Te testy sprawdzają poprawność działania modeli Customer, Category, Product, Order i OrderItem w aplikacji.

1. **test_customer_str**: Sprawdza, czy metoda `__str__` dla modelu Customer zwraca poprawną nazwę klienta.
2. **test_category_str**: Sprawdza, czy metoda `__str__` dla modelu Category zwraca poprawną nazwę kategorii.
3. **test_product_str**: Sprawdza, czy metoda `__str__` dla modelu Product zwraca poprawną nazwę produktu.
4. **test_product_sale_price**: Sprawdza, czy metoda sale_price dla modelu Product poprawnie oblicza cenę po zniżce.
5. **test_product_no_discount**: Sprawdza, czy metoda sale_price zwraca `None` dla produktu bez zniżki.
6. **test_product_special_occasion_mark**: Sprawdza, czy metoda special_occasion_mark dla modelu Product poprawnie zwraca oznaczenie specjalnej okazji.
7. **test_product_no_special_mark**: Sprawdza, czy metoda special_occasion_mark zwraca `None` dla produktu bez specjalnego oznaczenia.
8. **test_order_str**: Sprawdza, czy metoda `__str__` dla modelu Order zwraca poprawne ID zamówienia.
9. **test_order_get_cart_total**: Sprawdza, czy metoda get_cart_total poprawnie oblicza całkowitą wartość koszyka.
10. **test_order_get_cart_items**: Sprawdza, czy metoda get_cart_items poprawnie oblicza całkowitą liczbę przedmiotów w koszyku.
11. **test_order_no_items**: Sprawdza, czy zamówienie bez przedmiotów ma wartość koszyka równą 0.
12. **test_order_item_get_total**: Sprawdza, czy metoda get_total dla modelu OrderItem poprawnie oblicza całkowitą wartość przedmiotu.
13. **test_order_item_zero_quantity**: Sprawdza, czy metoda get_total zwraca 0 dla przedmiotu z ilością równą 0.
14. **test_order_item_negative_quantity**: Sprawdza, czy próba utworzenia OrderItem z ujemną ilością rzuca wyjątek ValueError.
15. **test_order_negative_cart_total**: Sprawdza, czy próba utworzenia OrderItem z produktem o ujemnej cenie rzuca wyjątek ValueError.



store > tests > test_models.py > ModelsTestCase

```python
class ModelsTestCase(TestCase):

    def test_customer_str(self):
        self.assertEqual(str(self.customer), 'Test Customer')

    def test_category_str(self):
        self.assertEqual(str(self.category), 'Spa')

    def test_product_str(self):
        self.assertEqual(str(self.product), 'Spa Package')

    def test_product_sale_price(self):
        self.assertEqual(self.product.sale_price, Decimal('90.00'))

    def test_product_no_discount(self):
        product_no_discount = Product.objects.create(
            name='No Discount Product',
            price=Decimal('50.00'),
            category=self.category
        )
        self.assertIsNone(product_no_discount.sale_price)

    def test_product_special_occasion_mark(self):
        self.assertEqual(self.product.special_occasion_mark, 'Special Occasion: Special')

    def test_product_no_special_mark(self):
        product_no_special_mark = Product.objects.create(
            name='No Special Mark Product',
            price=Decimal('50.00'),
            category=self.category
        )
        self.assertIsNone(product_no_special_mark.special_occasion_mark)

    def test_order_str(self):
        self.assertEqual(str(self.order), str(self.order.id))

    def test_order_get_cart_total(self):
        self.assertEqual(self.order.get_cart_total, Decimal('200.00'))
```

store > tests > test_models.py > ModelsTestCase

```python
class ModelsTestCase(TestCase):

    def test_order_get_cart_items(self):
        self.assertEqual(self.order.get_cart_items, 2)

    def test_order_no_items(self):
        empty_order = Order.objects.create(customer=self.customer, complete=False, transaction_id='654321')
        self.assertEqual(empty_order.get_cart_total, Decimal('0.00'))
        self.assertEqual(empty_order.get_cart_items, 0)

    def test_order_item_get_total(self):
        self.assertEqual(self.order_item.get_total, Decimal('200.00'))

    def test_order_item_zero_quantity(self):
        zero_quantity_item = OrderItem.objects.create(product=self.product, order=self.order, quantity=0)
        self.assertEqual(zero_quantity_item.get_total, Decimal('0.00'))

    def test_order_item_negative_quantity(self):
        with self.assertRaises(ValueError):
            OrderItem.objects.create(product=self.product, order=self.order, quantity=-2)

    def test_order_negative_cart_total(self):
        negative_price_product = Product.objects.create(
            name='Negative Price Product',
            price=Decimal('-50.00'),
            category=self.category
        )
        OrderItem.objects.create(product=negative_price_product, order=self.order, quantity=1)
        self.assertEqual(self.order.get_cart_total, Decimal('50.00'))
```

test_observers.py

Te testy pomagają wykryć potencjalne błędy w implementacji wzorca obserwatora, takie jak nieprawidłowe dodawanie lub usuwanie obserwatorów oraz poprawne logowanie i wysyłanie powiadomień.

1. **test_attach_observer**: Sprawdza, czy obserwator (UserObserver) został poprawnie dołączony do obiektu Subject. Używa assertIn, aby upewnić się, że obserwator znajduje się na liście _observers obiektu Subject.
2. **test_detach_observer**: Sprawdza, czy obserwator został poprawnie odłączony od obiektu Subject. Używa assertNotIn, aby upewnić się, że obserwator nie znajduje się na liście _observers obiektu Subject po jego odłączeniu.
3. **test_notify_observers**: Sprawdza, czy metoda notify w obiekcie Subject poprawnie powiadamia wszystkich dołączonych obserwatorów. Używa mocka (patch) do sprawdzenia, czy metoda `send_mail` została wywołana z odpowiednimi argumentami, co symuluje wysłanie e-maila do klienta.
4. **test_observer_update**: Sprawdza, czy metoda Update w UserObserver poprawnie wysyła e-mail z powiadomieniem. Używa mocka (patch) do sprawdzenia, czy metoda `send_mail` została wywołana z odpowiednimi argumentami, co symuluje wysłanie e-maila do klienta.
5. **test_subject_notify_no_observers**: Odłącza obserwatora od obiektu Subject. Sprawdza, czy wywołanie metody notify nie powoduje wysłania e-maila, ponieważ nie ma żadnych obserwatorów.
6. **test_observer_update_logging**: Używa mocka do sprawdzenia, czy metoda update w UserObserver poprawnie loguje wiadomość. Sprawdza, czy metoda logger.info została wywołana z odpowiednim komunikatem.
7. **test_subject_attach_duplicate_observer**: Sprawdza, czy próba dodania tego samego obserwatora do obiektu Subject nie powoduje dodania duplikatu. Porównuje liczbę obserwatorów przed i po próbie dodania duplikatu.
8. **test_subject_detach_nonexistent_observer**: Próbuje odłączyć obserwatora, który nie jest dołączony do obiektu Subject. Sprawdza, czy liczba obserwatorów pozostaje taka sama po próbie odłączenia nieistniejącego obserwatora.

store > tests > test_observers.py > ObserverTestCase

```python
class ObserverTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='12345')
        self.customer = Customer.objects.create(user=self.user, name='Test Customer', email='test@example.com')
        self.subject = Subject()
        self.observer = UserObserver(self.customer)
        self.subject.attach(self.observer)

    def test_attach_observer(self):
        self.assertIn(self.observer, self.subject._observers)

    def test_detach_observer(self):
        self.subject.detach(self.observer)
        self.assertNotIn(self.observer, self.subject._observers)

    def test_notify_observers(self):
        with patch('store.observers.send_mail') as mock_send_mail:
            self.subject.notify('Test message')
            mock_send_mail.assert_called_once_with(
                'New Offer Notification',
                'Test message',
                'from@example.com',
                [self.customer.email],
                fail_silently=False,
            )

    def test_observer_update(self):
        with patch('store.observers.send_mail') as mock_send_mail:
            self.observer.update('Test message')
            mock_send_mail.assert_called_once_with(
                'New Offer Notification',
                'Test message',
                'from@example.com',
                [self.customer.email],
                fail_silently=False,
            )
```

store > tests > test_observers.py > ObserverTestCase

```python
class ObserverTestCase(TestCase):

    def test_subject_notify_no_observers(self):
        self.subject.detach(self.observer)
        with patch('store.observers.send_mail') as mock_send_mail:
            self.subject.notify('Test message')
            mock_send_mail.assert_not_called()

    def test_observer_update_logging(self):
        with patch('store.observers.logger.info') as mock_logger_info:
            self.observer.update('Test message')
            mock_logger_info.assert_called_once_with('New offer on testuser: Test message')

    def test_subject_attach_duplicate_observer(self):
        initial_count = len(self.subject._observers)
        self.subject.attach(self.observer)
        self.assertEqual(len(self.subject._observers), initial_count)

    def test_subject_detach_nonexistent_observer(self):
        non_existent_observer = UserObserver(self.customer)
        initial_count = len(self.subject._observers)
        self.subject.detach(non_existent_observer)
        self.assertEqual(len(self.subject._observers), initial_count)
```

test_urls.py

Te testy sprawdzają, czy każda ścieżka URL jest poprawnie skonfigurowana i prowadzi do odpowiedniego widoku. Pomagają one upewnić się, że routing w aplikacji działa zgodnie z oczekiwaniami.

store > tests > test_urls.py > UrlsTestCase

```python
class UrlsTestCase(SimpleTestCase):
    def test_store_url(self):
        url = reverse('store')
        self.assertEqual(resolve(url).func, views.store)

    def test_cart_url(self):
        url = reverse('cart')
        self.assertEqual(resolve(url).func, views.cart)

    def test_checkout_url(self):
        url = reverse('checkout')
        self.assertEqual(resolve(url).func, views.checkout)

    def test_update_item_url(self):
        url = reverse('update_item')
        self.assertEqual(resolve(url).func, views.updateItem)

    def test_process_order_url(self):
        url = reverse('process_order')
        self.assertEqual(resolve(url).func, views.processOrder)

    def test_login_url(self):
        url = reverse('login')
        self.assertEqual(resolve(url).func.view_class, auth_views.LoginView)

    def test_logout_url(self):
        url = reverse('logout')
        self.assertEqual(resolve(url).func, views.custom_logout)

    def test_register_url(self):
        url = reverse('register')
        self.assertEqual(resolve(url).func, views.register)

    def test_subscribe_to_offers_url(self):
        url = reverse('subscribe_to_offers')
        self.assertEqual(resolve(url).func, views.subscribe_to_offers)
```

test_utils.py

Te testy sprawdzają poprawność działania funkcji cookieCart, cartData, i guestOrder, w tym przypadki brzegowe, takie jak puste koszyki, koszyki z przedmiotami, użytkowników zalogowanych i niezalogowanych oraz zamówienia gości. Pomagają one wykryć potencjalne błędy w logice aplikacji.

store > tests > test_utils.py > UtilsTestCase

```python
class UtilsTestCase(TestCase):
    def setUp(self):
        self.factory = RequestFactory()
        self.user = User.objects.create_user(username='testuser', password='12345')
        self.customer = Customer.objects.create(user=self.user, name='Test Customer', email='test@example.com')
        self.product = Product.objects.create(
            name='Test Product',
            price=Decimal('100.00'),
            category=None,
            place='Test Place',
            discount=0,
            special_mark=''
        )

    def test_cookieCart_empty_cart(self):
        request = self.factory.get('/')
        request.COOKIES['cart'] = json.dumps({})
        response = cookieCart(request)
        self.assertEqual(response['cartItems'], 0)
        self.assertEqual(response['order']['get_cart_total'], 0)
        self.assertEqual(response['order']['get_cart_items'], 0)
        self.assertEqual(response['items'], [])

    def test_cookieCart_with_items(self):
        request = self.factory.get('/')
        request.COOKIES['cart'] = json.dumps({str(self.product.id): {'quantity': 2}})
        response = cookieCart(request)
        self.assertEqual(response['cartItems'], 2)
        self.assertEqual(response['order']['get_cart_total'], Decimal('200.00'))
        self.assertEqual(response['order']['get_cart_items'], 2)
        self.assertEqual(len(response['items']), 1)
        self.assertEqual(response['items'][0]['id'], self.product.id)
```

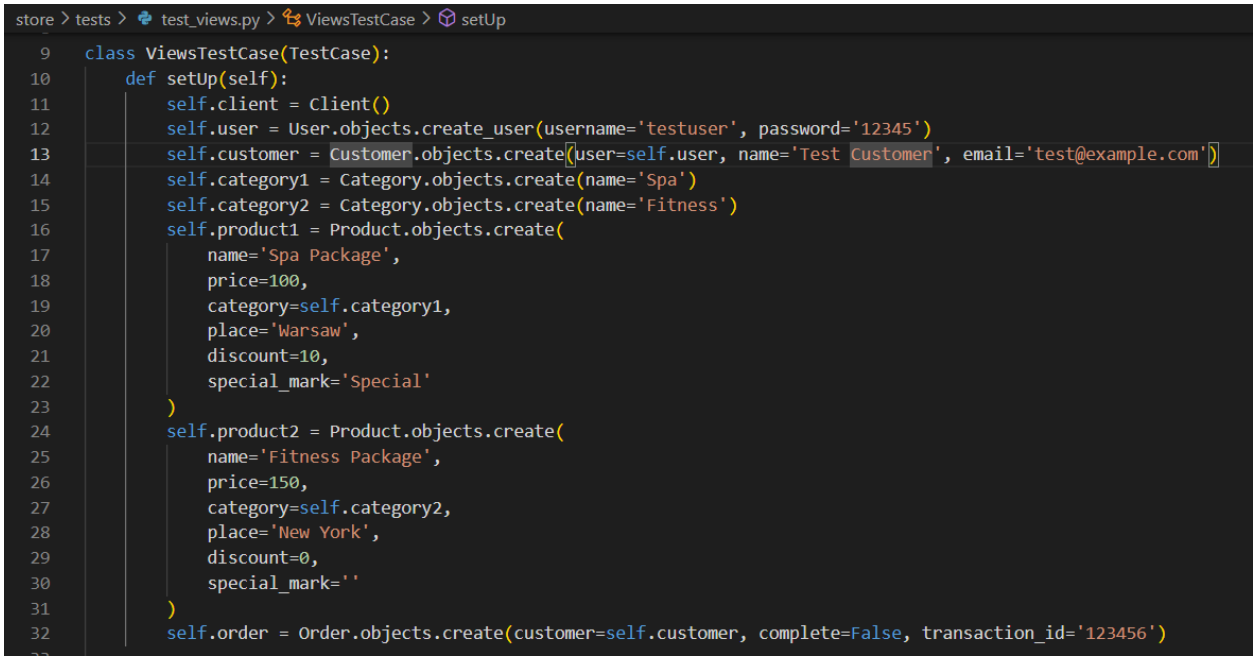
store > tests > test_utils.py > UtilsTestCase

```python
class UtilsTestCase(TestCase):
    def test_cartData_authenticated_user(self):
        request = self.factory.get('/')
        request.user = self.user
        response = cartData(request)
        self.assertEqual(response['cartItems'], 0)
        self.assertEqual(response['order'].customer, self.customer)
        self.assertEqual(response['items'], [])

    def test_cartData_unauthenticated_user(self):
        request = self.factory.get('/')
        request.COOKIES['cart'] = json.dumps({str(self.product.id): {'quantity': 2}})
        request.user = AnonymousUser()
        response = cartData(request)
        self.assertEqual(response['cartItems'], 2)
        self.assertEqual(response['order']['get_cart_total'], Decimal('200.00'))
        self.assertEqual(response['order']['get_cart_items'], 2)
        self.assertEqual(len(response['items']), 1)
        self.assertEqual(response['items'][0]['id'], self.product.id)

    def test_guestOrder(self):
        request = self.factory.get('/')
        request.COOKIES['cart'] = json.dumps({str(self.product.id): {'quantity': 2}})
        data = {'form': {'name': 'Guest', 'email': 'guest@example.com'}}
        customer, order = guestOrder(request, data)
        self.assertEqual(customer.name, 'Guest')
        self.assertEqual(customer.email, 'guest@example.com')
        self.assertEqual(order.customer, customer)
        self.assertEqual(order.complete, False)
        self.assertEqual(order.orderitem_set.count(), 1)
        self.assertEqual(order.orderitem_set.first().product, self.product)
        self.assertEqual(order.orderitem_set.first().quantity, 2)
```

test_views.py

Te testy jednostkowe sprawdzają różne aspekty działania widoków w aplikacji.

1. **test_register_view**: Sprawdza, czy widok rejestracji poprawnie renderuje stronę rejestracji i czy użytkownik może się zarejestrować.
2. **test_register_view_invalid**: Sprawdza, czy widok rejestracji poprawnie obsługuje nieprawidłowe dane (np. niezgodne hasła) i nie tworzy nowego użytkownika.
3. **test_custom_logout_view**: Sprawdza, czy widok wylogowania poprawnie wylogowuje użytkownika i przekierowuje go na stronę główną.
4. **test_store_view**: Sprawdza, czy widok sklepu poprawnie renderuje stronę sklepu.
5. **test_cart_view**: Sprawdza, czy widok koszyka poprawnie renderuje stronę koszyka.
6. **test_checkout_view**: Sprawdza, czy widok realizacji zamówienia poprawnie renderuje stronę realizacji zamówienia.
7. **test_updateItem_view**: Sprawdza, czy widok aktualizacji przedmiotu poprawnie aktualizuje ilość przedmiotu w zamówieniu.
8. **test_processOrder_view**: Sprawdza, czy widok przetwarzania zamówienia poprawnie przetwarza zamówienie i oznacza je jako zakończone.
9. **test_store_view_with_category_filter**: Sprawdza, czy widok sklepu poprawnie filtruje produkty według kategorii.
10. **test_store_view_with_multiple_categories**: Sprawdza, czy widok sklepu poprawnie filtruje produkty według wielu kategorii.
11. **test_store_view_with_invalid_category**: Sprawdza, czy widok sklepu poprawnie obsługuje nieprawidłową kategorię i nie wyświetla produktów.
12. **test_store_view_with_price_range_filter**: Sprawdza, czy widok sklepu poprawnie filtruje produkty według zakresu cen.
13. **test_store_view_with_place_filter:** Sprawdza, czy widok sklepu poprawnie filtruje produkty według miejsca.
14. **test_store_view_with_no_products**: Sprawdza, czy widok sklepu poprawnie obsługuje brak produktów i nie wyświetla żadnych produktów.
15. **test_subscribe_to_offers_view**: Sprawdza, czy widok subskrypcji ofert poprawnie dodaje obserwatora do produktów i przekierowuje użytkownika na stronę główną.

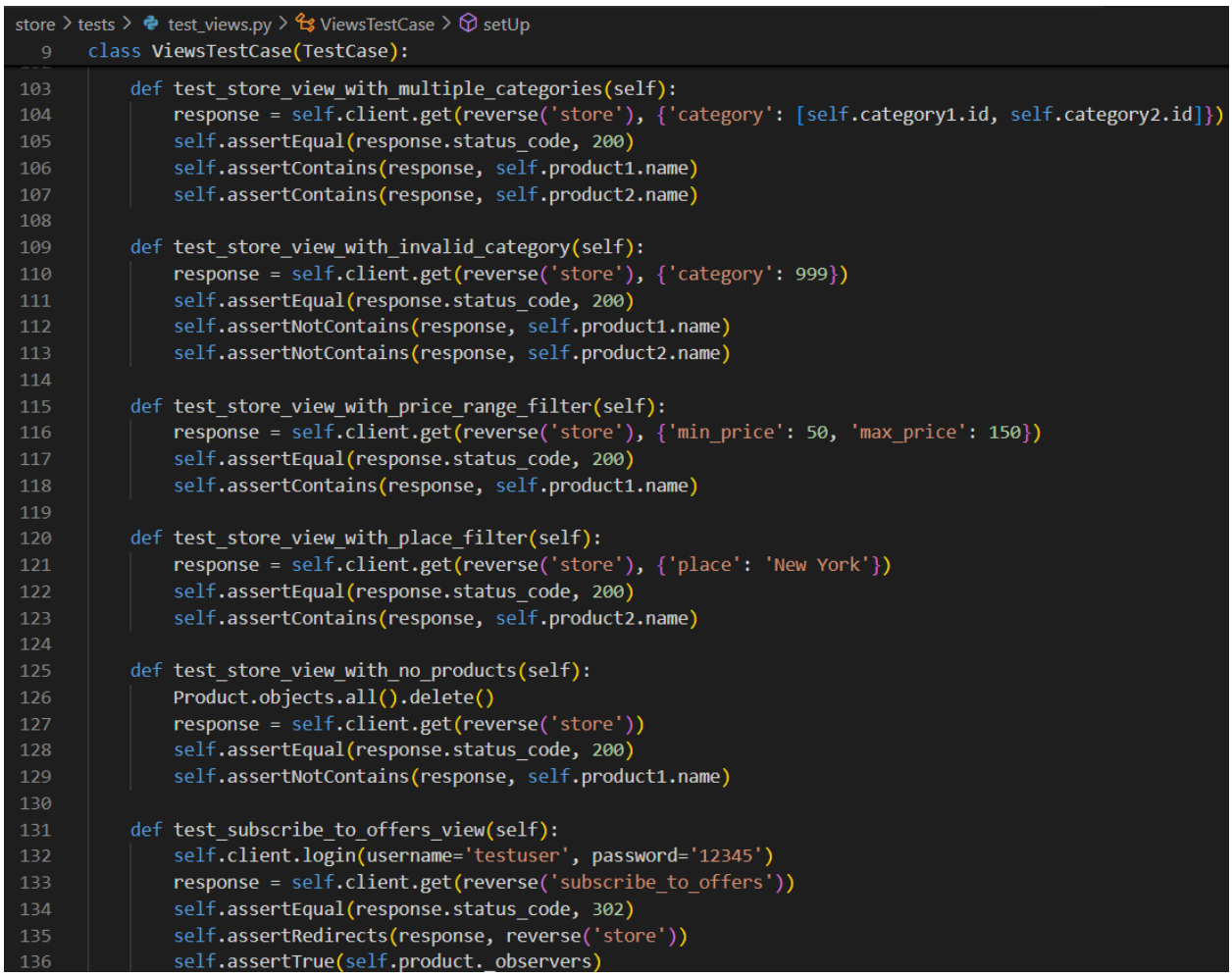


store > tests > test_views.py > ViewsTestCase > setUp

```
9   class ViewsTestCase(TestCase):
34      def test_register_view(self):
35          response = self.client.get(reverse('register'))
36          self.assertEqual(response.status_code, 200)
37          self.assertTemplateUsed(response, 'store/register.html')
38
39          response = self.client.post(reverse('register'), {
40              'username': 'newuser',
41              'password1': 'testpassword',
42              'password2': 'testpassword'
43          })
44          self.assertEqual(response.status_code, 302)
45          self.assertTrue(User.objects.filter(username='newuser').exists())
46
47      def test_register_view_invalid(self):
48          response = self.client.post(reverse('register'), {
49              'username': 'newuser',
50              'password1': 'testpassword123',
51              'password2': 'wrongpassword'
52          })
53          self.assertEqual(response.status_code, 200)
54          self.assertFalse(User.objects.filter(username='newuser').exists())
55
56      def test_custom_logout_view(self):
57          self.client.login(username='testuser', password='12345')
58          response = self.client.get(reverse('logout'))
59          self.assertEqual(response.status_code, 302)
60          self.assertRedirects(response, reverse('store'))
61
62      def test_store_view(self):
63          response = self.client.get(reverse('store'))
64          self.assertEqual(response.status_code, 200)
65          self.assertTemplateUsed(response, 'store/store.html')
```

store > tests > test_views.py > ViewsTestCase > setUp

```
9   class ViewsTestCase(TestCase):
67      def test_cart_view(self):
68          response = self.client.get(reverse('cart'))
69          self.assertEqual(response.status_code, 200)
70          self.assertTemplateUsed(response, 'store/cart.html')
71
72      def test_checkout_view(self):
73          response = self.client.get(reverse('checkout'))
74          self.assertEqual(response.status_code, 200)
75          self.assertTemplateUsed(response, 'store/checkout.html')
76
77      def test_updateItem_view(self):
78          self.client.login(username='testuser', password='12345')
79          response = self.client.post(reverse('update_item'), json.dumps({
80              'productId': self.product.id,
81              'action': 'add'
82          }), content_type='application/json')
83          self.assertEqual(response.status_code, 200)
84          self.assertEqual(OrderItem.objects.get(order=self.order, product=self.product).quantity, 1)
85
86      def test_processOrder_view(self):
87          self.client.login(username='testuser', password='12345')
88          response = self.client.post(reverse('process_order'), json.dumps({
89              'form': {
90                  'total': '100.00'
91              }
92          }), content_type='application/json')
93          self.assertEqual(response.status_code, 200)
94          self.order.refresh_from_db()
95          self.assertTrue(self.order.complete)
96
97      def test_store_view_with_category_filter(self):
98          response = self.client.get(reverse('store'), {'category': self.category1.id})
99          self.assertEqual(response.status_code, 200)
100         self.assertContains(response, self.product1.name)
101         self.assertNotContains(response, self.product2.name)
```


store > tests > test_views.py > ViewsTestCase > setUp
class ViewsTestCase(TestCase):
def test_store_view_with_multiple_categories(self):
response = self.client.get(reverse('store'), {'category': [self.category1.id, self.category2.id]})
self.assertEqual(response.status_code, 200)
self.assertContains(response, self.product1.name)
self.assertContains(response, self.product2.name)
def test_store_view_with_invalid_category(self):
response = self.client.get(reverse('store'), {'category': 999})
self.assertEqual(response.status_code, 200)
self.assertNotContains(response, self.product1.name)
self.assertNotContains(response, self.product2.name)
def test_store_view_with_price_range_filter(self):
response = self.client.get(reverse('store'), {'min_price': 50, 'max_price': 150})
self.assertEqual(response.status_code, 200)
self.assertContains(response, self.product1.name)
def test_store_view_with_place_filter(self):
response = self.client.get(reverse('store'), {'place': 'New York'})
self.assertEqual(response.status_code, 200)
self.assertContains(response, self.product2.name)
def test_store_view_with_no_products(self):
Product.objects.all().delete()
response = self.client.get(reverse('store'))
self.assertEqual(response.status_code, 200)
self.assertNotContains(response, self.product1.name)
def test_subscribe_to_offers_view(self):
self.client.login(username='testuser', password='12345')
response = self.client.get(reverse('subscribe_to_offers'))
self.assertEqual(response.status_code, 302)
self.assertRedirects(response, reverse('store'))
self.assertTrue(self.product._observers)

Integration tests

test_integrations.py

Te testy integracyjne pomagają upewnić się, że różne komponenty systemu współpracują ze sobą poprawnie w rzeczywistych scenariuszach użytkowania, takich jak rejestracja, logowanie, dodawanie produktów do koszyka, realizacja zamówienia oraz subskrypcja ofert.

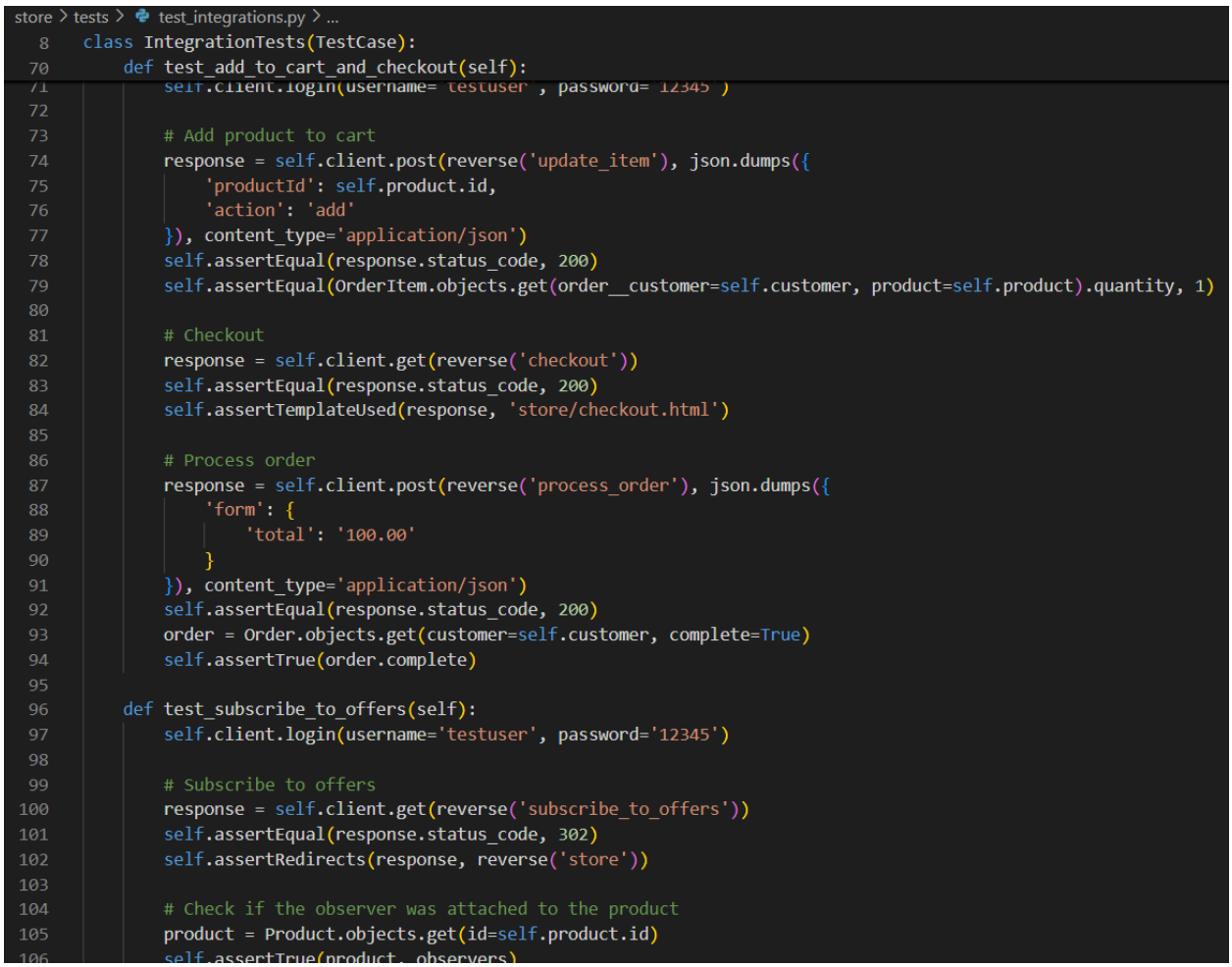
1. **test_user_registration_and_login**:
    a. Rejestracja użytkownika: Wysyła żądanie POST do widoku rejestracji z danymi nowego użytkownika. Sprawdza, czy odpowiedź ma status 302 (przekierowanie) i czy nowy użytkownik został utworzony w bazie danych.
    b. Logowanie użytkownika: Wysyła żądanie POST do widoku logowania z danymi nowego użytkownika. Sprawdza, czy odpowiedź ma status 302 (przekierowanie) i czy użytkownik jest zalogowany.
2. **test_add_and_remove_from_cart**:
    a. Dodawanie produktu do koszyka: Loguje użytkownika, wysyła żądanie POST do widoku aktualizacji przedmiotu z akcją "add". Sprawdza, czy odpowiedź ma status 200 i czy ilość przedmiotu w zamówieniu wynosi 1.
    b. Usuwanie produktu z koszyka: Wysyła żądanie POST do widoku aktualizacji przedmiotu z akcją "remove". Sprawdza, czy odpowiedź ma status 200 i czy przedmiot został usunięty z zamówienia.
3. **test_store_view_with_price_range_filter**:
    a. Wysyła żądanie GET do widoku sklepu z parametrami zakresu cen. Sprawdza, czy odpowiedź ma status 200 i czy zawiera nazwę produktu.
4. **test_store_view_with_place_filter**:
    a. Wysyła żądanie GET do widoku sklepu z parametrem miejsca. Sprawdza, czy odpowiedź ma status 200 i czy zawiera nazwę produktu.
5. **test_add_to_cart_and_checkout**:
    a. Dodawanie produktu do koszyka: Loguje użytkownika, wysyła żądanie POST do widoku aktualizacji przedmiotu z akcją "add". Sprawdza, czy odpowiedź ma status 200 i czy ilość przedmiotu w zamówieniu wynosi 1.
    b. Realizacja zamówienia: Wysyła żądanie GET do widoku realizacji zamówienia. Sprawdza, czy odpowiedź ma status 200 i czy użyto odpowiedniego szablonu.
    c. Przetwarzanie zamówienia: Wysyła żądanie POST do widoku przetwarzania zamówienia z danymi zamówienia. Sprawdza, czy odpowiedź ma status 200 i czy zamówienie zostało oznaczone jako zakończone.
6. **test_subscribe_to_offers**:
    a. Loguje użytkownika, wysyła żądanie GET do widoku subskrypcji ofert. Sprawdza, czy odpowiedź ma status 302 (przekierowanie) i czy użytkownik został przekierowany na stronę główną.
    b. Sprawdza, czy obserwator został dodany do produktu.

store > tests > test_integrations.py > ...

```python
class IntegrationTests(TestCase):
    def setUp(self):
        self.client = Client()
        self.user = User.objects.create_user(username='testuser', password='12345')
        self.customer = Customer.objects.create(user=self.user, name='Test Customer', email='test@example.com')
        self.category = Category.objects.create(name='Spa')
        self.product = Product.objects.create(
            name='Spa Package',
            price=Decimal('100.00'),
            category=self.category,
            place='Warsaw',
            discount=10,
            special_mark='Special'
        )

    def test_user_registration_and_login(self):
        # Test user registration
        response = self.client.post(reverse('register'), {
            'username': 'newuser',
            'password1': 'testpassword123',
            'password2': 'testpassword123'
        })
        self.assertEqual(response.status_code, 302)
        self.assertTrue(User.objects.filter(username='newuser').exists())

        # Test user login
        response = self.client.post(reverse('login'), {
            'username': 'newuser',
            'password': 'testpassword123'
        })
        self.assertEqual(response.status_code, 302)
        self.assertTrue(response.wsgi_request.user.is_authenticated)
```

store > tests > test_integrations.py > ...

```python
class IntegrationTests(TestCase):
    def test_add_and_remove_from_cart(self):
        self.client.login(username='testuser', password='12345')

        # Add product to cart
        response = self.client.post(reverse('update_item'), json.dumps({
            'productId': self.product.id,
            'action': 'add'
        }), content_type='application/json')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(OrderItem.objects.get(order__customer=self.customer, product=self.product).quantity, 1)

        # Remove product from cart
        response = self.client.post(reverse('update_item'), json.dumps({
            'productId': self.product.id,
            'action': 'remove'
        }), content_type='application/json')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(OrderItem.objects.filter(order__customer=self.customer, product=self.product).count(), 0)

    def test_store_view_with_price_range_filter(self):
        response = self.client.get(reverse('store'), {'min_price': 50, 'max_price': 150})
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, self.product.name)

    def test_store_view_with_place_filter(self):
        response = self.client.get(reverse('store'), {'place': 'Warsaw'})
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, self.product.name)
```


store > tests > test_integrations.py > ...
class IntegrationTests(TestCase):
def test_add_to_cart_and_checkout(self):
self.client.login(username='testuser', password='12345')
# Add product to cart
response = self.client.post(reverse('update_item'), json.dumps({
'productId': self.product.id,
'action': 'add'
}), content_type='application/json')
self.assertEqual(response.status_code, 200)
self.assertEqual(OrderItem.objects.get(order__customer=self.customer, product=self.product).quantity, 1)
# Checkout
response = self.client.get(reverse('checkout'))
self.assertEqual(response.status_code, 200)
self.assertTemplateUsed(response, 'store/checkout.html')
# Process order
response = self.client.post(reverse('process_order'), json.dumps({
'form': {
'total': '100.00'
}
}), content_type='application/json')
self.assertEqual(response.status_code, 200)
order = Order.objects.get(customer=self.customer, complete=True)
self.assertTrue(order.complete)
def test_subscribe_to_offers(self):
self.client.login(username='testuser', password='12345')
# Subscribe to offers
response = self.client.get(reverse('subscribe_to_offers'))
self.assertEqual(response.status_code, 302)
self.assertRedirects(response, reverse('store'))
# Check if the observer was attached to the product
product = Product.objects.get(id=self.product.id)
self.assertTrue(product.observers)