

Supermarket Automation

Software (SAS)

Testing Report

- 1. Agniva Saha - 22CS10003**
- 2. Sayandeep Bhowmick - 22CS10069**
- 3. Srinjoy Das - 22CS30054**

March-April 2024

**Prepared for Software Engineering Laboratory,
CS29006**

**Under the guidance of Prof. Sourangshu Bhattacharya
and Prof. Debashis Samanta**

Test Report for Form Validations for Dashboard

Test Environment

Framework: Django (version unspecified)

Language: Python

Test Suite: Django's `TestCase` class

Test Cases and Results

1. ProductForm with Valid Data

Objective: To ensure the `ProductForm` correctly validates when provided with a complete and valid dataset.

Data Provided: Name, Category, Weight, Quantity, Buying Price, Selling Price

Expected Result: Form is valid.

Outcome: Passed 

2. ProductForm with No Data

Objective: To verify that the `ProductForm` correctly identifies and reports errors when no data is provided.

Data Provided: None

Expected Result: Form is invalid, with errors reported for 6 fields.

Outcome: Passed 

3. InformationForm with Valid Data

Objective: To test if the `InformationForm` is valid with proper content data.

Data Provided: Content

Expected Result: Form is valid.

Outcome: Passed 

4. OrderUpdateForm with Valid Data

Objective: To check the validity of the `OrderUpdateForm` when updating an order status.

Data Provided: Status (COMPLETED)

Expected Result: Form is valid.

Outcome: Passed 

5. ProductEditFormStaff with Valid Data

Objective: To ensure that the `ProductEditFormStaff` can be validated with correct quantity data.

Data Provided: Quantity

Expected Result: Form is valid.

Outcome: Passed 

6. ProductEditFormAdmin with Valid Data

Objective: To assess the `ProductEditFormAdmin`'s ability to validate when provided with a new selling price.

Data Provided: Selling Price

Expected Result: Form is valid.

Outcome: Passed 

Test Report for Database validations for Dashboard

Test Environment

Test Suite: Django's `TestCase` class


Tests Conducted and Results

1. Product Model String Representation

Objective: To verify that the `Product` model's string representation correctly returns the product's name.

Method: A `Product` instance was created with the name 'Test Product', category 'Stationary', and quantity 10.

Expected Result: The string representation of the product instance should be 'Test Product'.

Outcome: Passed 

2. Order Model String Representation

Objective: To ensure that the `Order` model's string representation accurately returns a string combining the product's name and the username of the staff who placed the order.

Method: An `Order` instance was created with a linked `Product` named 'Test Product' and a `User` named 'testuser'. The order quantity was set to 5.

Expected Result: The string representation of the order should be 'Test Product ordered by testuser'.

Outcome: Passed 

3. Information Model String Representation

Objective: To test if the `Information` model's string representation correctly returns the content of the information instance.

Method: An `Information` instance was created with the content 'Test Information'.

Expected Result: The string representation of the information should be 'Test Information'.

Outcome: Passed 

Test Report for View function Validations for Dashboard

Test Environment

Testing Tools: Django's `TestCase`, `Client`, and `RequestFactory`; Python's `unittest.mock`

Tests Conducted and Results

1. Index View

Objective: Verify that the index view returns a 200 status code and uses the correct template.

Method: A GET request was made to the 'dashboard-index' URL.

Expected Result: A 200 status code and the 'dashboard/index.html' template is used.

Outcome: Passed 

2. Add to Cart View

Objective: Test that products can be successfully added to the cart, resulting in a redirect (302 status code).

Method: A POST request to 'add_to_cart' URL with product details.

Expected Result: A 302 status code, indicating a successful redirect after adding to cart.

Outcome: Passed 

3. Cart View

Objective: Ensure the cart view is accessible and renders the correct template.

Method: A GET request to the 'cart' URL.

Expected Result: A 200 status code and the 'dashboard/cart.html' template is used.


Outcome: Passed 

4. Generate Barcode Function

- **Objective:** Validate that the `generate_barcode` function creates a barcode image successfully.

- **Method:** Invoking `generate_barcode` with a sample barcode data string.

- **Expected Result:** A truthy value (barcode image exists).

- **Outcome:** Passed 

5. To Counter View

Objective: Test moving orders from 'IN_PROGRESS' to 'WAITING' status and render the correct template.

Method: A GET request to the 'to-counter' URL after creating sample orders.

Expected Result: Orders are moved to 'WAITING' status, and the 'dashboard/counter.html' template is used with a 200 status code.

Outcome: Passed 

6. Counter View (Partial Test Description)

Objective: To verify that the counter view correctly lists orders with 'WAITING' status and handles them appropriately.

Method: Creation of sample orders with various statuses, followed by a GET request to a related URL (assumed to be 'counter' or similar).

Expected Result: Only orders with 'WAITING' status are processed or listed, adhering to the application's business logic.

Outcome: Test description incomplete; additional details required for full evaluation.

7. Automated Weighing Machine Test

Objective: Ensure that the automated weighing machine correctly calculates the total price and weight of orders.

Method: Mocking `barcode_reader` to simulate scanning a product barcode, then calling `automated_weighing_machine` with an order list.

Expected Result: Total price and weight are accurately calculated for given orders.

Outcome: Passed 


Asserted Values: Total price = 500, Total weight = 10

8. Checkout View Test

Objective: Verify that the checkout view correctly displays total price and weight, using the appropriate template.

Method: Mocking `barcode_reader` for product identification and accessing the checkout view.

Expected Result: Response status 200, usage of 'dashboard/checkout.html' template, and correct context variables (`total_price`, `total_weight`).


Outcome: Passed 

9. Billing View Test

Objective: Assess the billing view's ability to generate a bill, update order and product statuses, and display relevant information.

Method: Mocking `automated_weighing_machine` to provide total price and weight, then accessing the billing view.

Expected Result: Order status updated to 'COMPLETED', product quantity updated, generation of bill number and date, and correct total price and weight displayed.


- **Outcome:** Passed 

10. Remove from Cart View Test


- **Objective:** Ensure that products can be successfully removed from the cart, with appropriate updates to order status and product quantity.

- **Method:** Creating a sample product and order, then removing the product from the cart via the view.


- **Expected Result:** Order is deleted, product quantity is correctly updated, and the user is redirected appropriately.

- **Outcome:** Passed 


11. Clear Cart View Test

- **Objective:** Ensure the clear cart view correctly removes all orders and updates product quantities.
- **Outcome:** Passed 
- **Assertion Details:** After execution, all `Order` instances were deleted, and the related `Product` quantity was updated as expected.


12. Staff View Test

- **Objective:** Verify the staff view displays correct information including counts of workers, orders, products, and informational content.
- **Outcome:** Passed 
- **Assertion Details:** The response context contained all expected keys (`workers`, `workers_count`, `orders_count`, `products_count`, `information_content`).


13. Staff Detail View Test

- **Objective:** Assess the detail view for staff members for correct context data.
- **Outcome:** Passed 
- **Assertion Details:** Similar to the staff view, this test confirmed the presence of expected context data related to the application's operational metrics.


14. Product View Test

- **Objective:** Ensure the product view lists items correctly and provides necessary context for operations.
- **Outcome:** Passed 
- **Assertion Details:** The product view's context included `items`, `form`, and statistical counts (`workers_count`, `orders_count`, `products_count`), validating UI readiness for interactions.


15. Product Create Post Test

- **Objective:** Test the product creation functionality through a POST request with form data.
- **Outcome:** Passed 
- **Assertion Details:** After submission, the product was successfully created, confirming the form's functionality and backend processing.


16. Product Delete View Test

- **Objective:** Verify that the product delete view correctly displays the item to be deleted.
- **Outcome:** Passed 
- **Assertion Details:** The context contained the `item` to be deleted, ensuring users have clear information before taking action.


17. Sales Statistics Test

- **Objective:** Validate the sales statistics view for correctness and template usage.
- **Outcome:** Passed 
- **Assertion Details:** The view rendered the `sales_statistics.html` template successfully, indicating proper setup for displaying sales data.


18. Edit Information View Test (GET Request)

- **Objective:** Confirm that the edit information view renders correctly for GET requests.
- **Outcome:** Passed 
- **Assertion Details:** The correct template ('edit_information.html') was used, ensuring the view is accessible and editable for users.


19. Edit Information View POST Test

- **Objective:** Verify the ability to update informational content through a POST request.
- **Outcome:** Passed 
- **Details:** After submitting updated content via a POST request, the response correctly redirected to the dashboard index, and the content was updated in the database.


20. Edit Information View Invalid Form Test

- **Objective:** Ensure the application properly handles invalid form submissions when editing information.
- **Outcome:** Passed 
- **Details:** Submitting an empty string for content resulted in a re-rendering of the form with appropriate error messages, indicating robust form validation.


21. Edit Information View Nonexistent Instance Test

- **Objective:** Assess the system's handling of attempts to edit non-existent information instances.
- **Outcome:** Passed 
- **Details:** The test confirmed that accessing the edit view for a nonexistent information instance still returns a 200 status code, allowing for graceful handling of such scenarios.


22. Product Update Admin GET Test

- **Objective:** Test the accessibility of the product update page by an admin through a GET request.
- **Outcome:** Passed 
- **Details:** The admin successfully accessed the product update page, indicating proper permissions and routing.


23. Product Update Admin POST Test

- **Objective:** Examine the functionality of updating product details as an admin via POST request.
- **Outcome:** Needs Review 
- **Details:** Despite attempting to update the selling price of a product, the price remained unchanged after submission, suggesting potential issues in form handling or permission levels.


24. Product Update Staff GET Test

- **Objective:** Verify that staff members can access the product update page.
- **Outcome:** Passed 
- **Details:** Similar to the admin test, staff members could access the update page, underscoring consistent access control.

25. Product Update Staff POST Test

- **Objective:** Test the ability of staff members to update product quantities.
- **Outcome:** Passed 
- **Details:** The product's quantity was successfully updated through a POST request, highlighting effective permission settings and form processing for staff roles.

26. Order Update Test with Mocking

- **Objective:** Validate the order update functionality, with a focus on mocking external dependencies.
- **Outcome:** Needs Review 
- **Details:** The order's quantity did not change as expected after a POST request, suggesting either an issue with the test setup (mocking) or the application logic.

Code Run Output:

python manage.py test dashboard

Found 50 test(s).

Creating a test database for alias 'default'...

System check identified no issues (0 silenced).

```
.....ResolverMatch(func=dashboard.views.add_to_cart, args=(), kwargs={},
url_name='add_to_cart', app_names=[], namespaces=[], route='add_to_cart/')
.ResolverMatch(func=dashboard.views.billing, args=(), kwargs={}, url_name='billing', app_names=[],
namespaces=[], route='billing/')
.ResolverMatch(func=dashboard.views.cart, args=(), kwargs={}, url_name='cart', app_names=[],
namespaces=[], route='cart/')
.ResolverMatch(func=dashboard.views.checkout, args=(), kwargs={}, url_name='checkout',
app_names=[], namespaces=[], route='checkout/')
.ResolverMatch(func=dashboard.views.clear_cart, args=(), kwargs={}, url_name='clear_cart',
app_names=[], namespaces=[], route='clear/')
.ResolverMatch(func=dashboard.views.counter, args=(), kwargs={}, url_name='counter', app_names=[],
namespaces=[], route='counter/')
.ResolverMatch(func=dashboard.views.index, args=(), kwargs={}, url_name='dashboard-index',
app_names=[], namespaces=[], route='dashboard/')
.ResolverMatch(func=dashboard.views.order_update, args=(), kwargs={'pk': 1},
url_name='dashboard-order-update', app_names=[], namespaces=[], route='order/update/<int:pk>',
captured_kwargs={'pk': 1})
.ResolverMatch(func=dashboard.views.order, args=(), kwargs={}, url_name='dashboard-order',
app_names=[], namespaces=[], route='order/')
.ResolverMatch(func=dashboard.views.product_delete, args=(), kwargs={'pk': 1},
url_name='dashboard-product-delete', app_names=[], namespaces=[], route='product/delete/<int:pk>',
captured_kwargs={'pk': 1})
.ResolverMatch(func=dashboard.views.product_update, args=(), kwargs={'pk': 1},
url_name='dashboard-product-update', app_names=[], namespaces=[], route='product/update/<int:pk>',
captured_kwargs={'pk': 1})
.ResolverMatch(func=dashboard.views.product, args=(), kwargs={}, url_name='dashboard-product',
app_names=[], namespaces=[], route='product/')
.ResolverMatch(func=dashboard.views.edit_information, args=(), kwargs={},
url_name='edit-information', app_names=[], namespaces=[], route='edit-information/')
.ResolverMatch(func=dashboard.views.remove_from_cart, args=(), kwargs={'product_id': 1},
url_name='remove_from_cart', app_names=[], namespaces=[], route='remove/<int:product_id>',
captured_kwargs={'product_id': 1})
.ResolverMatch(func=dashboard.views.sales_statistics, args=(), kwargs={}, url_name='sales_statistics',
app_names=[], namespaces=[], route='sales_statistics/')
```

Ran 50 tests in 15.694s

OK

Destroying test database for alias 'default'...


Form Validation Testing Report for User Management

Test Methodology


The testing strategy involved submitting these forms with both valid and invalid data to observe their behavior in different scenarios. Specifically, we assessed the forms' ability to validate data completeness, correctness, and ensure that user feedback is provided through errors when necessary.

Test Outcomes


1. CreateUserForm with Valid Data

- **Objective:** Verify that the CreateUserForm accepts valid user data.
- **Result:** Passed 
- **Details:** The form correctly validated the provided user data, including matching passwords, thus proving its functionality in accepting valid user registration data.


2. CreateUserForm with No Data

- **Objective:** Assess the CreateUserForm's response to submissions without any data.
- **Result:** Passed 
- **Details:** As expected, the form was invalid when no data was provided, and it correctly identified errors in four fields, aligning with the form's requirements for username, email, and password fields.


3. UserUpdateForm with Valid Data

- **Objective:** Ensure that the UserUpdateForm correctly handles valid updates to a user's username and email.
- **Result:** Passed 
- **Details:** The form validation passed when valid data for username and email updates were provided, indicating its effectiveness in processing user updates.


4. UserUpdateForm with No Data

- **Objective:** Determine the UserUpdateForm's behavior when no data is submitted.
- **Result:** Passed 
- **Details:** The form was appropriately invalidated when submitted without data, which is consistent with expectations, although specific error details were not mentioned.

5. ProfileUpdateForm with Valid Data

- **Objective:** Test the ProfileUpdateForm's capability to validate correct profile-related information such as address and phone number.
- **Result:** Passed 
- **Details:** With valid data for address and phone, the form validation succeeded, indicating that it properly handles valid profile updates.

6. ProfileUpdateForm with No Data

- **Objective:** Evaluate how the ProfileUpdateForm responds to being submitted without any data.
- **Result:** Passed 
- **Details:** The form was invalidated as expected when no data was provided, which is in line with the requirement for at least some profile information to be submitted.

Model Testing Report of User

Introduction

The integrity and functionality of models within a Django application are paramount for the consistent handling and storage of data. As part of our ongoing quality assurance processes, we conducted a test on the `Profile` model to ensure its reliability, specifically focusing on its string representation method (`__str__`). This report outlines the methodology, results, and conclusions of the test conducted on the `Profile` model associated with the Django `User` model.


Test Methodology

The testing framework employed was Django's built-in `TestCase` class, which provides a wide range of methods and tools for testing the components of Django applications in isolation. For this particular test, our objective was to validate the `__str__` method of the `Profile` model, ensuring it returns the correct string representation that includes the related user's username followed by "-Profile".

The test involved the following steps:

- 1. Setup Test Data:** A test user was created using Django's `create_user` method. This process automatically triggers the creation of a corresponding `Profile` instance due to the application's design.
- 2. Test Execution:** The `__str__` method of the created `Profile` instance was invoked.
- 3. Assertion:** The output of the `__str__` method was compared against the expected string value, which is the username of the associated user appended with "-Profile".

Test Outcome

- **Test Name:** `test_str_method`
- **Objective:** To verify that the `__str__` method of the `Profile` model returns the expected string format.
- **Result:** Passed 
- **Details:** The `__str__` method of the `Profile` model correctly returned the string combining the username of the associated user and "-Profile". The test user with the username 'test' resulted in the `Profile` model's `__str__` method returning "test-Profile".

URL Resolution for User


Test Methodology

The URL resolution tests were implemented using Django's `SimpleTestCase` class, which provides utilities for testing Django-specific functionality without database interactions. For each view function, the test involved the following steps:


- 1. Generate URL:** Using Django's `reverse` function, the URL for the respective view function was generated based on the URL name specified in the URL configuration.
- 2. Resolve URL:** The generated URL was resolved using Django's `resolve` function, which matches the URL to the corresponding view function.
- 3. Assertion:** The resolved view function was compared against the expected view function to verify that the URL is correctly mapped.

Test Outcomes


1. Register URL Resolution Test

- **Objective:** Validate the resolution of the registration view URL.
- **Result:** Passed 
- **Details:** The test successfully resolved the URL for the registration view (`register`) to the expected view function.


2. Check URL Resolution Test

- **Objective:** Confirm the resolution of the check view URL.
- **Result:** Passed 
- **Details:** The URL for the check view (`check`) was correctly resolved to the expected view function.


3. Profile URL Resolution Test

- **Objective:** Verify the resolution of the profile view URL.
- **Result:** Passed 
- **Details:** The URL for the profile view (`profile`) was resolved as expected to the corresponding view function.

4. Profile Update URL Resolution Test

- **Objective:** Ensure the resolution of the profile update view URL.
- **Result:** Needs Review 
- **Details:** Although the URL for the profile update view was successfully generated, no assertion was made regarding the resolved view function. This oversight requires attention to ensure comprehensive testing.

5. Logout URL Resolution Test

- **Objective:** Validate the resolution of the logout view URL.
- **Result:** Passed 
- **Details:** The URL for the logout view (`logout_view`) was resolved correctly to the expected view function.

User Views Testing Report

Introduction


Testing views in a Django application is essential to ensure that each view behaves as expected, handling requests and rendering templates correctly. In this report, we conducted tests on user-related views, including registration, profile management, and logout functionalities. The tests were designed to cover both GET and POST requests, validating the responses and behavior of each view under different scenarios.

Test Methodology


Using Django's built-in `TestCase` class, we implemented tests for the user views. The `setUp` method was used to set up necessary data and URLs for each test. The tests were categorized based on the views being tested, covering both GET and POST requests where applicable. For each test, we made assertions regarding the HTTP status codes, template usage, and redirections, ensuring that the views respond correctly to different types of requests.

Test Outcomes


1. Register View GET Test

- **Objective:** Verify that the register view renders the registration form correctly.
- **Result:** Passed 
- **Details:** The test confirmed that the register view renders the expected template (`user/register.html`) and returns a 200 OK status code.


2. Register View POST Test

- **Objective:** Ensure that the register view processes registration form submissions correctly.
- **Result:** Passed 
- **Details:** The test validated that the register view redirects users to the login page after successful registration, with a status code of 302 (Found).


3. Logout View Test

- **Objective:** Confirm that the logout view renders the logout confirmation page.
- **Result:** Passed 
- **Details:** The test verified that the logout view renders the expected template (`user/logout.html`) and returns a 200 OK status code.


4. Profile View Test

- **Objective:** Validate that the profile view renders the user's profile page when logged in.
- **Result:** Passed 
- **Details:** The test ensured that the profile view renders the correct template ('user/profile.html') and returns a 200 OK status code when the user is logged in.

5. Profile Update View GET Test

- **Objective:** Verify that the profile update view renders the profile update form.
- **Result:** Passed 
- **Details:** The test confirmed that the profile update view renders the expected template ('user/profile_update.html') and returns a 200 OK status code when accessed via GET request.

6. Profile Update View POST Test

- **Objective:** Ensure that the profile update view processes profile update form submissions correctly.
- **Result:** Passed 
- **Details:** The test validated that the profile update view redirects users to their profile page after successfully updating their profile, with a status code of 302 (Found).

Code Run Output:

```
python manage.py test user
Found 18 test(s).
Creating a test database for alias 'default'...
System check identified no issues (0 silenced).
.....ResolverMatch(func=user.views.check, args=(), kwargs={}, url_name='user-check',
app_names=[], namespaces=[], route=")
.ResolverMatch(func=user.views.logout_view, args=(), kwargs={}, url_name='user-logout',
app_names=[], namespaces=[], route='logout/')
.ResolverMatch(func=user.views.profile_update, args=(), kwargs={}, url_name='user-profile-update',
app_names=[], namespaces=[], route='profile/update/')
.ResolverMatch(func=user.views.profile, args=(), kwargs={}, url_name='user-profile', app_names=[],
namespaces=[], route='profile/')
.ResolverMatch(func=user.views.register, args=(), kwargs={}, url_name='user-register', app_names=[],
namespaces=[], route='register/')
.
-----
Ran 18 tests in 2.988s

OK
Destroying test database for alias 'default'...
```

Test Report for Inventory

No test cases were required as we don't have any function in views.py and urls.py only imports the urls.py from all other apps.

Code Run Output:

```
python manage.py test inventory
```

```
Found 0 test(s).
```


```
System check identified no issues (0 silenced).
```

Ran 0 tests in 0.000s

NO TESTS RAN


Test Report for Form function Validations for Staff

1. Staff Register Form Valid Data Test


- **Objective:** Validate the StaffRegisterForm with valid data.
- **Outcome:** Passed 
- **Details:** The StaffRegisterForm validate the provided data successfully. The form's `is_valid()` method returned `True`, indicating that the form accepted the input data without any validation errors. This ensures that the form handles valid data appropriately, which is essential for registering new staff members in the system.

Test Report for Urls function Validations for Staff:

1. Activate URL Resolved Test


- **Objective:** Ensure that the URL for activating staff members is correctly resolved.
- **Outcome:** Passed 
- **Details:** The URL 'staff-activate' with an argument of '1' was successfully resolved to the `activate` function in the `staff.views` module, as expected.

2. Staff Dashboard URL Resolved Test

- **Objective:** Confirm that the URL for the staff dashboard is resolved properly.
- **Outcome:** Passed 
- **Details:** The URL 'dashboard-staff' was resolved to the `staff` function in the `staff.views` module, as intended.

3. Staff Product Page URL Resolved Test

Objective: Validate the resolution of the URL for the staff product page.

Outcome: Passed 

Details: The URL 'staff-product' was resolved to the 'product' function in the 'staff.views' module, as expected.

4. Staff Product Update URL Resolved Test


- **Objective:** Verify that the URL for updating staff products is resolved correctly.

- **Outcome:** Passed 

- **Details:** The URL 'staff-product-update' with an argument of '1' was resolved to the 'product_update' function in the 'staff.views' module, as intended.

5. Staff Product Delete URL Resolved Test

- **Objective:** Ensure the correct resolution of the URL for deleting staff products.

- **Outcome:** Passed 

- **Details:** The URL 'staff-product-delete' with an argument of '1' was resolved to the 'product_delete' function in the 'staff.views' module, as expected.

6. Staff Registration URL Resolved Test


- **Objective:** Validate the resolution of the URL for staff registration.

- **Outcome:** Passed 

- **Details:** The URL 'staff-application' was resolved to the 'staff_register' function in the 'staff.views' module, as intended.

7. Staff Order URL Resolved Test

- **Objective:** Confirm the correct resolution of the URL for staff orders.

- **Outcome:** Passed 

- **Details:** The URL 'order-staff' was resolved to the 'order' function in the 'staff.views' module, as expected.

8. Staff Order Update URL Resolved Test

- **Objective:** Ensure the proper resolution of the URL for updating staff orders.

- **Outcome:** Passed 

- **Details:** The URL 'order-staff-update' with an argument of '1' was resolved to the 'order_update' function in the 'staff.views' module, as intended.

9. Staff Product Page URL Resolved Test (Duplicate)


- **Objective:** Validate the resolution of the URL for the staff product page.

- **Outcome:** Passed 


- **Details:** The URL 'staff-product' was resolved to the 'product' function in the 'staff.views' module, as expected.

Test Report for Views function Validations for Staff:


1. Staff Register View Test

- **Objective:** Verify the functionality of the staff registration view.
- **Outcome:** Passed 
- **Details:** The test ensures that users can access the staff registration page ('staff-application.html'). It validates that upon submitting valid registration data, a new staff member is successfully created in the database. The HTTP response status code for the registration POST request is 302 (redirect), and the new staff member 'newstaff' exists in the database.


2. Logout View Test

- **Objective:** Confirm the functionality of the logout view.
- **Outcome:** Passed 
- **Details:** The test verifies that users can access the logout page ('logout.html') and logout successfully. The HTTP response status code for the logout GET request is 200, and the template 'user/logout.html' is used as expected.


3. Product View Test

- **Objective:** Validate the behavior of the product view.
- **Outcome:** Passed 
- **Details:** This test ensures that staff members can access the product page ('staff_page.html'). It validates that upon submitting a new product, the product is successfully created in the database. The HTTP response status code for the product POST request is 200, and the product 'Test Product' exists in the database.


4. Product Delete View Test

- **Objective:** Ensure the functionality of the product delete view.
- **Outcome:** Passed 
- **Details:** The test confirms that staff members can delete a product successfully. It verifies that upon deleting a product, the product 'Test Product' no longer exists in the database, and the HTTP response status code for the delete request is 302 (redirect).


5. Product Update View Test

- **Objective:** Verify the functionality of the product update view.
- **Outcome:** Passed 
- **Details:** This test checks that staff members can access the product update page ('staff_update.html'). It ensures that upon updating a product's name, the product's name remains unchanged. The test confirms that the product name remains 'Test Product' after the update operation.

6. Order View Test

- **Objective:** Validate the behavior of the order view.
- **Outcome:** Passed 
- **Details:** The test ensures that staff members can access the order page ('order_staff.html'). It confirms that staff members can view orders successfully. The HTTP response status code for the order GET request is 200, and the template 'staff/order_staff.html' is used as expected.

7. Order Update View Test

- **Objective:** Ensure the functionality of the order update view.
- **Outcome:** Passed 
- **Details:** This test verifies that staff members can access the order update page ('order_update.html'). It validates that upon updating an order's quantity, the order's quantity remains unchanged. The test confirms that the order quantity remains 3 after the update operation.

Code Run Output:

python manage.py test staff

Found 17 test(s).

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.....Updated Product Name: Test Product

...ResolverMatch(func=staff.views.activate, args=(), kwargs={'pk': 1}, url_name='staff-activate', app_names=[], namespaces=[], route='activate/<int:pk>', captured_kwargs={'pk': 1})

.ResolverMatch(func=staff.views.order_update, args=(), kwargs={'pk': 1}, url_name='order-staff-update', app_names=[], namespaces=[], route='order-staff/update/<int:pk>', captured_kwargs={'pk': 1})

.ResolverMatch(func=staff.views.order, args=(), kwargs={}, url_name='order-staff', app_names=[], namespaces=[], route='order-staff/')

.ResolverMatch(func=staff.views.product, args=(), kwargs={}, url_name='staff-product', app_names=[], namespaces=[], route='product-staff/')

.ResolverMatch(func=staff.views.product, args=(), kwargs={}, url_name='staff-product', app_names=[], namespaces=[], route='product-staff/')

.ResolverMatch(func=staff.views.product_delete, args=(), kwargs={'pk': 1}, url_name='staff-product-delete', app_names=[], namespaces=[], route='staff_product_delete/<int:pk>', captured_kwargs={'pk': 1})

.ResolverMatch(func=staff.views.product_update, args=(), kwargs={'pk': 1}, url_name='staff-product-update', app_names=[], namespaces=[], route='staff_product_update/<int:pk>', captured_kwargs={'pk': 1})

.ResolverMatch(func=staff.views.staff_register, args=(), kwargs={}, url_name='staff-application', app_names=[], namespaces=[], route='staff-register/')

.ResolverMatch(func=staff.views.staff, args=(), kwargs={}, url_name='dashboard-staff', app_names=[], namespaces=[], route='staff/')

.

Ran 17 tests in 4.165s

OK

Destroying test database for alias 'default'...