**User Incomes and Test Cases Script:-**import datetime

from django.test import TestCase, Client

from django.urls import reverse

from django.contrib.auth.models import User

from django.utils import timezone

import json

from .models import UserIncome, Source

class UserIncomeTests(TestCase):

"""Simplified test cases for UserIncome app"""

def setUp(self):

"""Set up test data for all test methods"""

# Create a test user

self.test\_user = User.objects.create\_user(

username='testuser',

email='test@example.com',

password='testpassword123'

)

self.client = Client()

self.client.login(username='testuser', password='testpassword123')

# Create a source

self.source = Source.objects.create(

name='Salary',

owner=self.test\_user

)

# Create an income record

self.today = timezone.now().date()

self.income = UserIncome.objects.create(

amount=5000.00,

date=self.today,

description='Monthly salary',

owner=self.test\_user,

source='Salary'

)

# URLs

self.index\_url = reverse('income')

self.add\_income\_url = reverse('add-income')

self.edit\_income\_url = reverse('income-edit', args=[self.income.id])

self.delete\_income\_url = reverse('income-delete', args=[self.income.id])

self.income\_summary\_url = reverse('income-summary')

def test\_models(self):

"""Test 1: Test Source and UserIncome models"""

# Test Source model

source = Source.objects.get(name='Salary')

self.assertEqual(source.name, 'Salary')

self.assertEqual(source.owner, self.test\_user)

self.assertEqual(str(source), 'Salary')

# Test UserIncome model

income = UserIncome.objects.get(description='Monthly salary')

self.assertEqual(income.amount, 5000.00)

self.assertEqual(income.description, 'Monthly salary')

self.assertEqual(income.owner, self.test\_user)

self.assertEqual(income.source, 'Salary')

self.assertEqual(str(income), 'Salary')

def test\_income\_list\_view(self):

"""Test 2: Test the index view that displays all income records"""

response = self.client.get(self.index\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'income/index.html')

self.assertEqual(len(response.context['income']), 1)

# Test sorting functionality

response = self.client.get(f"{self.index\_url}?sort=amount\_desc")

self.assertEqual(response.status\_code, 200)

self.assertEqual(response.context['sort\_order'], 'amount\_desc')

def test\_add\_income(self):

"""Test 3: Test income creation functionality"""

# First check that the add income page loads

response = self.client.get(self.add\_income\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'income/add\_income.html')

# Test successful income addition

income\_data = {

'amount': '2500.00',

'income\_date': (self.today - datetime.timedelta(days=2)).strftime('%Y-%m-%d'),

'description': 'Bonus payment',

'source': 'Salary'

}

response = self.client.post(self.add\_income\_url, income\_data)

self.assertRedirects(response, self.index\_url)

# Verify income was created

self.assertTrue(UserIncome.objects.filter(description='Bonus payment').exists())

# Test validation error - future date

future\_date = self.today + datetime.timedelta(days=5)

income\_data = {

'amount': '2500.00',

'income\_date': future\_date.strftime('%Y-%m-%d'),

'description': 'Future payment',

'source': 'Salary'

}

response = self.client.post(self.add\_income\_url, income\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Date cannot be in the future')

def test\_edit\_delete\_income(self):

"""Test 4: Test income editing and deletion"""

# Test editing

income\_data = {

'amount': '5500.00',

'income\_date': self.today.strftime('%Y-%m-%d'),

'description': 'Updated salary',

'source': 'Salary'

}

response = self.client.post(self.edit\_income\_url, income\_data)

self.assertRedirects(response, self.index\_url)

# Verify income was updated

updated\_income = UserIncome.objects.get(pk=self.income.id)

self.assertEqual(float(updated\_income.amount), 5500.00)

self.assertEqual(updated\_income.description, 'Updated salary')

# Test deletion

response = self.client.get(self.delete\_income\_url)

self.assertRedirects(response, self.index\_url)

# Verify income was deleted

self.assertFalse(UserIncome.objects.filter(pk=self.income.id).exists())

def test\_income\_summary\_and\_authentication(self):

"""Test 5: Test income summary and authentication requirements"""

# Test income summary

response = self.client.get(self.income\_summary\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'income/dashboard.html')

self.assertIn('daily\_income', response.context)

self.assertIn('weekly\_income', response.context)

self.assertIn('monthly\_income', response.context)

# Test authentication requirements

# First logout

self.client.logout()

# Try accessing protected pages

protected\_urls = [self.index\_url, self.add\_income\_url,

self.edit\_income\_url, self.income\_summary\_url]

for url in protected\_urls:

response = self.client.get(url)

self.assertRedirects(

response,

f'/authentication/login?next={url}',

fetch\_redirect\_response=False

)

**Authentication test cases Script:-**

from django.test import TestCase, Client

from django.urls import reverse

from django.contrib.auth.models import User

import json

from django.utils.encoding import force\_bytes

from django.utils.http import urlsafe\_base64\_encode

from authentication.utils import account\_activation\_token

class AuthenticationTests(TestCase):

"""Test cases for authentication functionality"""

def setUp(self):

"""Set up test data for all test methods"""

self.client = Client()

self.register\_url = reverse('register')

self.login\_url = reverse('login')

self.validate\_email\_url = reverse('validate\_email')

# Create a test user (already active)

self.active\_user = User.objects.create\_user(

username='activeuser',

email='active@example.com',

password='password123'

)

self.active\_user.is\_active = True

self.active\_user.save()

def test\_user\_registration(self):

"""

Test Case 1: User Registration Process

This test verifies the user registration process, including:

- GET request to registration page works

- POST request creates an inactive user

- Password validation works

"""

# Test GET request to registration page

response = self.client.get(self.register\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'authentication/register.html')

# Test registration with short password (should fail)

registration\_data = {

'username': 'testuser',

'email': 'test@example.com',

'password': '123' # Too short

}

response = self.client.post(self.register\_url, registration\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Password too short')

# Check that user wasn't created

self.assertFalse(User.objects.filter(username='testuser').exists())

# Test successful registration

registration\_data['password'] = 'password123' # Valid password

response = self.client.post(self.register\_url, registration\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Account created successfully')

# Check that the user was created but is inactive

user = User.objects.get(username='testuser')

self.assertFalse(user.is\_active)

# Verify the user exists with correct email

self.assertEqual(user.email, 'test@example.com')

def test\_user\_login(self):

"""

Test Case 2: User Login Process

This test verifies the user login process, including:

- GET request to login page works

- Login for active users works

- Login for inactive users fails

- Invalid credentials are rejected

"""

# Test GET request to login page

response = self.client.get(self.login\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'authentication/login.html')

# Create inactive test user

inactive\_user = User.objects.create\_user(

username='inactiveuser',

email='inactive@example.com',

password='password123'

)

inactive\_user.is\_active = False

inactive\_user.save()

# Test login with active user (should succeed)

login\_data = {

'username': 'activeuser',

'password': 'password123'

}

response = self.client.post(self.login\_url, login\_data)

self.assertRedirects(response, reverse('expenses'))

# Logout after successful login

self.client.get(reverse('logout'))

# Test login with inactive user (should fail)

login\_data = {

'username': 'inactiveuser',

'password': 'password123'

}

response = self.client.post(self.login\_url, login\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Account is not active')

# Test login with invalid credentials (should fail)

login\_data = {

'username': 'activeuser',

'password': 'wrongpassword'

}

response = self.client.post(self.login\_url, login\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Invalid credentials')

def test\_email\_validation(self):

"""

Test Case 3: Email Validation API

This test verifies the email validation API, including:

- Valid email format validation

- Email uniqueness validation

"""

# Test invalid email format

data = {'email': 'invalid-email'}

response = self.client.post(

self.validate\_email\_url,

json.dumps(data),

content\_type='application/json'

)

self.assertEqual(response.status\_code, 400)

response\_data = json.loads(response.content)

self.assertIn('emailerror', response\_data)

self.assertEqual(response\_data['emailerror'], 'Email is invalid')

# Test email already in use

data = {'email': 'active@example.com'} # Email used by active\_user

response = self.client.post(

self.validate\_email\_url,

json.dumps(data),

content\_type='application/json'

)

self.assertEqual(response.status\_code, 409)

response\_data = json.loads(response.content)

self.assertIn('emailerror', response\_data)

self.assertEqual(response\_data['emailerror'], 'sorry email in use, choose another one')

# Test valid, unique email

data = {'email': 'new@example.com'}

response = self.client.post(

self.validate\_email\_url,

json.dumps(data),

content\_type='application/json'

)

self.assertEqual(response.status\_code, 200)

response\_data = json.loads(response.content)

self.assertIn('email\_valid', response\_data)

self.assertTrue(response\_data['email\_valid'])

**Expenses module test cases Script:-**

from django.test import TestCase, Client

from django.urls import reverse

from django.contrib.auth.models import User

from django.utils import timezone

import json

from datetime import datetime, timedelta, date

from decimal import Decimal

from .models import Expense, Category, ExpenseLimit, Bill

class ExpensesAppTests(TestCase):

"""Test suite for the Expenses application"""

def setUp(self):

"""Set up test data for all test methods"""

# Create a test user

self.test\_user = User.objects.create\_user(

username='testuser',

email='test@example.com',

password='testpass123'

)

self.client = Client()

self.client.login(username='testuser', password='testpass123')

# Create test categories

self.category1 = Category.objects.create(name='Food')

self.category2 = Category.objects.create(name='Transportation')

# Create test expenses

self.today = timezone.now().date()

self.yesterday = self.today - timedelta(days=1)

self.expense1 = Expense.objects.create(

amount=100.00,

date=self.today,

description='Lunch at restaurant',

owner=self.test\_user,

category='Food'

)

self.expense2 = Expense.objects.create(

amount=30.00,

date=self.yesterday,

description='Bus ticket',

owner=self.test\_user,

category='Transportation'

)

# Create test expense limit

self.expense\_limit = ExpenseLimit.objects.create(

owner=self.test\_user,

daily\_expense\_limit=200

)

# Create test bill

self.bill = Bill.objects.create(

owner=self.test\_user,

name='Electricity',

amount=Decimal('75.50'),

due\_date=self.today + timedelta(days=5),

frequency='Monthly',

description='Monthly electricity bill'

)

# URLs for testing

self.expenses\_url = reverse('expenses')

self.add\_expense\_url = reverse('add-expenses')

self.edit\_expense\_url = reverse('expense-edit', args=[self.expense1.id])

self.delete\_expense\_url = reverse('expense-delete', args=[self.expense1.id])

self.stats\_url = reverse('stats')

self.add\_bill\_url = reverse('add-bill')

self.edit\_bill\_url = reverse('bill-edit', args=[self.bill.id])

self.delete\_bill\_url = reverse('bill-delete', args=[self.bill.id])

def test\_expense\_model\_and\_listing(self):

"""

Test Case 1: Test Expense model and expense listing

This test verifies:

- Expense model creation and string representation

- Expense listing view displays correct data

- Sorting functionality in listing view

"""

# Test Expense model

expense = Expense.objects.get(description='Lunch at restaurant')

self.assertEqual(expense.amount, 100.00)

self.assertEqual(expense.category, 'Food')

self.assertEqual(expense.owner, self.test\_user)

self.assertEqual(str(expense), 'Food')

# Test expense listing view

response = self.client.get(self.expenses\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'expenses/index.html')

# Check that expenses are in the context

self.assertIn('expenses', response.context)

self.assertEqual(len(response.context['expenses']), 2)

# Test sorting functionality

response = self.client.get(f"{self.expenses\_url}?sort=amount\_desc")

self.assertEqual(response.status\_code, 200)

self.assertEqual(response.context['sort\_order'], 'amount\_desc')

# Check that expenses are sorted correctly (highest amount first)

expenses\_list = list(response.context['expenses'])

self.assertEqual(expenses\_list[0].amount, 100.00)

self.assertEqual(expenses\_list[1].amount, 30.00)

def test\_bill\_model\_and\_functionality(self):

"""

Test Case 2: Test Bill model and its functionality

This test verifies:

- Bill model creation and string representation

- Next due date calculation based on frequency

- Bill editing functionality

"""

# Test Bill model

bill = Bill.objects.get(name='Electricity')

self.assertEqual(bill.amount, Decimal('75.50'))

self.assertEqual(bill.frequency, 'Monthly')

self.assertEqual(bill.owner, self.test\_user)

# Check that string representation works correctly

self.assertEqual(str(bill), 'Electricity - 75.50')

# Verify next due date was calculated correctly for monthly frequency

self.assertIsNotNone(bill.next\_due\_date)

# Test bill editing

edit\_data = {

'bill\_name': 'Updated Electricity',

'bill\_amount': '80.00',

'due\_date': (self.today + timedelta(days=10)).strftime('%Y-%m-%d'),

'frequency': 'Yearly',

'description': 'Updated electricity bill'

}

response = self.client.post(self.edit\_bill\_url, edit\_data)

self.assertRedirects(response, self.expenses\_url)

# Verify bill was updated

updated\_bill = Bill.objects.get(id=self.bill.id)

self.assertEqual(updated\_bill.name, 'Updated Electricity')

self.assertEqual(updated\_bill.amount, Decimal('80.00'))

self.assertEqual(updated\_bill.frequency, 'Yearly')

# Test bill deletion

response = self.client.get(self.delete\_bill\_url)

self.assertRedirects(response, self.expenses\_url)

# Verify bill was deleted

self.assertEqual(Bill.objects.count(), 0)

def test\_expense\_management(self):

"""

Test Case 3: Test expense management (add, edit, delete)

This test verifies:

- Adding a new expense

- Editing an existing expense

- Deleting an expense

- Validation for required fields and future dates

"""

# Test expense creation

add\_data = {

'amount': '50.00',

'expense\_date': self.yesterday.strftime('%Y-%m-%d'),

'description': 'Groceries shopping',

'category': 'Food'

}

response = self.client.post(self.add\_expense\_url, add\_data)

self.assertRedirects(response, self.expenses\_url)

# Verify expense was created

self.assertEqual(Expense.objects.count(), 3)

new\_expense = Expense.objects.get(description='Groceries shopping')

self.assertEqual(new\_expense.amount, 50.00)

# Test expense editing

edit\_data = {

'amount': '60.00',

'expense\_date': self.yesterday.strftime('%Y-%m-%d'),

'description': 'Updated lunch cost',

'category': 'Food'

}

response = self.client.post(self.edit\_expense\_url, edit\_data)

self.assertRedirects(response, self.expenses\_url)

# Verify expense was updated

updated\_expense = Expense.objects.get(id=self.expense1.id)

self.assertEqual(updated\_expense.amount, 60.00)

self.assertEqual(updated\_expense.description, 'Updated lunch cost')

# Test validation - future date (should fail)

future\_date = self.today + timedelta(days=5)

invalid\_data = {

'amount': '70.00',

'expense\_date': future\_date.strftime('%Y-%m-%d'),

'description': 'Future expense',

'category': 'Food'

}

response = self.client.post(self.add\_expense\_url, invalid\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'Date cannot be in the future')

# Test validation - missing description (should fail)

invalid\_data = {

'amount': '70.00',

'expense\_date': self.yesterday.strftime('%Y-%m-%d'),

'description': '',

'category': 'Food'

}

response = self.client.post(self.add\_expense\_url, invalid\_data)

self.assertEqual(response.status\_code, 200)

self.assertContains(response, 'description is required')

# Test expense deletion

response = self.client.get(self.delete\_expense\_url)

self.assertRedirects(response, self.expenses\_url)

# Verify expense was deleted

self.assertEqual(Expense.objects.count(), 2)

with self.assertRaises(Expense.DoesNotExist):

Expense.objects.get(id=self.expense1.id)

def test\_expense\_limit\_functionality(self):

"""

Test Case 4: Test expense limit functionality

This test verifies:

- Setting and updating expense limits

- Warning when adding expense that exceeds daily limit

"""

# Test setting expense limit

limit\_data = {

'daily\_expense\_limit': '300'

}

response = self.client.post(reverse('set-daily-expense-limit'), limit\_data)

self.assertEqual(response.status\_code, 302) # Should redirect

# Verify limit was updated

updated\_limit = ExpenseLimit.objects.get(owner=self.test\_user)

self.assertEqual(updated\_limit.daily\_expense\_limit, 300)

# Test adding expense below the limit

expense\_data = {

'amount': '50.00',

'expense\_date': self.today.strftime('%Y-%m-%d'),

'description': 'Small expense',

'category': 'Food'

}

response = self.client.post(self.add\_expense\_url, expense\_data)

self.assertRedirects(response, self.expenses\_url)

# Test adding expense that exceeds the limit

# First, update limit to a small value

ExpenseLimit.objects.filter(owner=self.test\_user).update(daily\_expense\_limit=10)

# Then try to add an expense that exceeds this limit

expense\_data = {

'amount': '100.00',

'expense\_date': self.today.strftime('%Y-%m-%d'),

'description': 'Big expense',

'category': 'Food'

}

response = self.client.post(self.add\_expense\_url, expense\_data)

# Should still create the expense but show a warning

self.assertRedirects(response, self.expenses\_url)

# Verify the expense was added despite exceeding limit

self.assertTrue(Expense.objects.filter(description='Big expense').exists())

def test\_expense\_statistics\_and\_search(self):

"""

Test Case 5: Test expense statistics and search functionality

This test verifies:

- Statistics view loads correctly

- Category summary data is correctly calculated

- Expense search functionality works

"""

# Test statistics view

response = self.client.get(self.stats\_url)

self.assertEqual(response.status\_code, 200)

self.assertTemplateUsed(response, 'expenses/stats.html')

# Test expense category summary data

response = self.client.get(reverse('expense\_category\_summary'))

self.assertEqual(response.status\_code, 200)

# Parse the JSON response

data = json.loads(response.content)

self.assertIn('expense\_category\_data', data)

# Verify category totals

category\_data = data['expense\_category\_data']

self.assertEqual(category\_data['Food'], 100.0)

self.assertEqual(category\_data['Transportation'], 30.0)

# Test expense search

search\_data = json.dumps({'searchText': 'lunch'})

response = self.client.post(

reverse('search\_expenses'),

data=search\_data,

content\_type='application/json'

)

self.assertEqual(response.status\_code, 200)

search\_results = json.loads(response.content)

# Verify search results

self.assertEqual(len(search\_results), 1)

self.assertEqual(search\_results[0]['description'], 'Lunch at restaurant')

# Test search by category

search\_data = json.dumps({'searchText': 'transportation'})

response = self.client.post(

reverse('search\_expenses'),

data=search\_data,

content\_type='application/json'

)

self.assertEqual(response.status\_code, 200)

search\_results = json.loads(response.content)

# Verify search results

self.assertEqual(len(search\_results), 1)

self.assertEqual(search\_results[0]['category'], 'Transportation')