STM JOURNALS

Journal of
Open Source Developments

ISSN: 2395-6704
Volume 11, Issue 3, 2024
September–December
DOI (Journal): 10.37591/JoOSD

https://journals.stmjournals.com/joosd

Case Study JoOSD

Design and Development of a Case Study on a Simple Banking System (SBS) by Applying the Concepts of Django Models, Views, Templates, Admin Interfaces, and Forms Using Full-Stack Development Framework

Niranjan R. Chougala[1,*], M.V. Chaitanya Kumar[2], Shivamurthaiah M.[3], Deepak S. Sakkari[4], Shridhar V.[5], Surendra Babu M.S.[6]

Abstract

The purpose of this case study is to develop and demonstrate various facilities and features available in full-stack development using Django by considering a simple example of a banking system. The concepts of Django Models, Views, Templates, Admin Interfaces, and Forms are used to ensure this case study utilizes all features of the framework to make it the complete miniature product. In addition, a well-documented report, and various test-cases related to the product are incorporated to complete the software development product cycle. For a better understanding of the product and its use simple business logic has been developed and documented so that the user can easily improve and scale up accordingly. Various interactions and interfaces among the code have been shown to make naive users or developers understand better as well as encourage them to contribute additional business logic easily. The results of executing the code have been shown with a brief explanation. This work targets and encourages naïve developers to understand the full-stack development framework better by considering a simple case study.

Keywords: Full-stack development, Django Models, views, templates, admin interfaces

***Author for Correspondence**
Niranjan R. Chougala
E-mail: niranjan.chougala@rrit.ac.in

[1,5,6]Professor, Department of Computer Science and Engineering, R.R. Institute of Technology, Bengaluru, Karnataka, India
[2]Professor, School of Engineering and Technology, Christ University, Bengaluru, Karnataka, India
[3]Professor, Department of Computer Science and Engineering, Bengaluru, Karnataka, India
[4]Professor, Department of Computer Science and Engineering, Sri Krishna Institute of Technology (SKIT), Bengaluru, Karnataka, India

Received Date: September 13, 2024
Accepted Date: September 23, 2024
Published Date: November 04, 2024

Citation: Niranjan R. Chougala, M.V. Chaitanya Kumar, Shivamurthaiah M., Deepak S. Sakkari, Shridhar V., Surendra Babu M.S. Design and Development of a Case Study on a Simple Banking System (SBS) by Applying the Concepts of Django Models, Views, Templates, Admin Interfaces, and Forms Using Full-Stack Development Framework. Journal of Open Source Developments. 2024; 11(3): 26–36p.

INTRODUCTION

Preamble on Full-Stack Web Development

Website development plays a crucial role in showcasing services and products, helping potential customers understand their relevance, and highlighting the unique features that differentiate them from competitors [1]. Beyond commercial purposes, websites are essential for delivering quick and dynamic information in various fields. For computer science and related engineering graduates, mastering the creation of well-designed, informative, responsive, and dynamic websites is becoming increasingly important. Therefore, acquiring skills in modern technologies and frameworks to build elegant websites is essential. Full-stack developers are in high demand because they possess the expertise to handle all aspects of web application development, including front-end, back-end, business logic, and integration with third-

party services such as payment gateways [2]. Among the latest advancements, the Model-View-Template (MVT) based development framework with Django stands out as a cutting-edge approach to full-stack web development. Python, known for its ease of use, enhances the Django framework, enabling developers to efficiently build responsive and secure web applications [3].

PURPOSE/OBJECTIVE

- Explore the uses of learning full-stack web development.
- Make use of rapid application development in the design of responsive web pages.
- Illustrate the Models, Views, and Templates with their connectivity in Django for full-stack web development.
- Demonstrate the use of admin interface automation in Django.
- Design and implement Django apps containing dynamic pages with databases [4].

PROBLEM STATEMENT

- SBS is a simple banking system.
- To keep things simple, we have only one bank account.
- Admin users can add/delete/modify the account balance.
- Users can credit a positive integer amount.
- The user can debit a positive integer provided that the amount is not greater than the balance [5].
- If the amount to be credited/debited is 0 or less, or if the amount to be debited is greater than the balance, a validation error occurs, and the transaction fails.
- The transactions can be seen in the order of date/time from the beginning of the admin interface [6].

ARCHITECTURE

- Django uses the MVT pattern.
- The overall project is called the bank.
- It has an application called account.
- M = model is given by the models.py in the bus application.
- V = view is given by views.py in the bus application.
- T = template is given by two HTML files under account\templates\account.

MODELS

- We have an account model, which has the date/time of the last transaction and the (current) balance.
- We have a model called Transactions which keeps track of all transactions ever made.
- We have a model called CreditRequest that captures the amount that needs to be credited.
- We have a model called DebitRequest that captures the amount that needs to be debited.
- A default Sqlite3 database holds records [7].

VIEWS

- There is a view to show the balance.
- There is a view that has a form for credit.
- There is a view that has a form for debit.

FORMS

- CreditForm is based on CreditRequest, which validates that the amount should be positive [8].
- There is a form called DebitForm based on DebitRequest that validates that the amount should be positive and not greater than the balance.

TEMPLATES

There are three templates, corresponding to the three views:

1. show_balance.html (shows the balance)
2. credit.html (a form with the amount to be credited and a submit button)
3. debit.html (a form with the amount to be debited and a submit button)

URL CONF

1. Corresponding to the three views, there are three URLs in the account application.
2. These URLS get included in the bank project.

ADMIN INTERFACE

1. Admin is enabled for Account and Transaction models.
2. We created a superuser.

BANKING PROCEDURE

1. Admin user creates an Account record with the initial balance.
2. When the show balance URL is invoked, the balance is shown [9].
3. Credit and debit forms work as expected, update the balance, and create a transaction record if the validations succeed.
4. In case of validation errors, an error message is shown.

FULL-STACK DEVELOPER

Figure 1 depicts the full-stack developer.

DJANGO ARCHITECTURE

Django is built on MVT (Model-View-Template) architecture, which consists of three key components, as shown in Figure 2.

- *Model*: The Django model represents a table in the database. We can use any relational database supported by Django. We used the default SQLite database. A model is provided by a class with its fields and key relationships.
- *View*: This view corresponds to a user interface. Typically, a view is obtained by rendering a template [10].
- *Template*: A template contains static parts of the HTM output, along with variables (placeholders), tags (logic), filters (special processing functions), and comments.

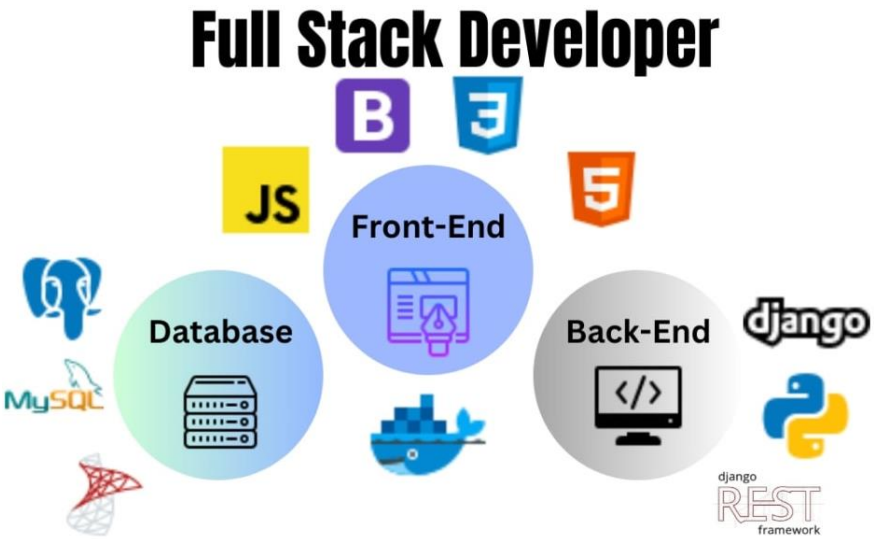

Figure 1. Full-stack developer.

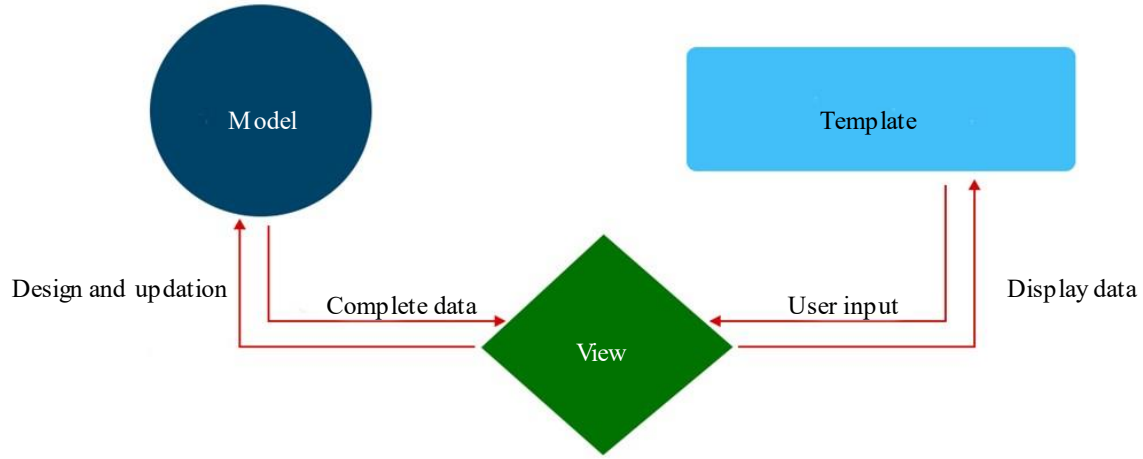

Figure 2. Django MVT architecture.

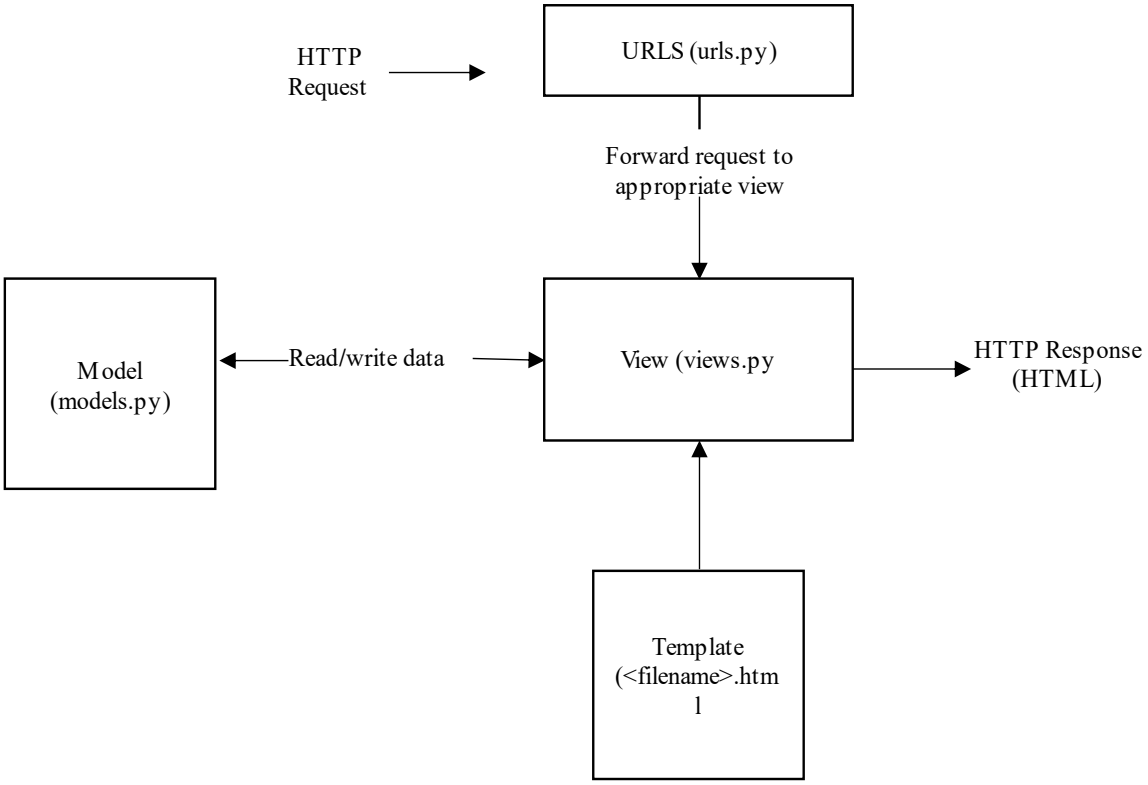

Figure 3. MVT interaction.

MVT INTERACTION

The given Figure 3 is the representation of MVT Interactions.

THE CODE

Models.py

A Django model is represented by a class extending the Django Models, and the model and data are stored in a relational database supported by Django, such as MySQL, SQL*Server, Oracle, or PostgreSQL. The default SQLite database was used. Once the models are defined, we run py manage.py makemigrations and py manage.py migrates to create the tables in the database. Here, we have models for Account and Transaction that need to be persisted, as well as for Debit and Credit on which the forms are based.

models.py

```
from django.db import models
# Create your models here.
class Account(models.Model):
    last_transaction_date_time = models.DateTimeField()
    balance = models.IntegerField()
    def __str__(self):
        return str(self.balance) + ‘ ’ + str(self.last_transaction_date_time)

class Transaction(models.Model):
    date_time = models.DateTimeField()
    amount = models.IntegerField()
    debit = models.BooleanField()
    def __str__(self):
        debit_str = ‘CREDIT’
        if self.debit:
            debit_str = ‘DEBIT’
        return str(self.amount) + ‘ ’ + str(self.date_time) + ‘ ’ + debit_str

class CreditRequest(models.Model):
    amount = models.IntegerField()

class DebitRequest(models.Model):
    amount = models.IntegerField()
```

Views.py

A view is a function that takes on a web request, usually an HttpRequest, and returns a WebResponse, usually an HttpResponse. It can also render an HTML template rather than returning a constructed HTMLstring. In this case study, we have three views: showing the balance, a form for credit, and a debit form.

views.py

```
from django.shortcuts import render
from django.http import HttpResponse

import datetime
from.models import Account, Transaction, CreditRequest, DebitRequest
from.forms import CreditForm, DebitForm

# Create your views here.

def show_balance(request):
    if Account.objects.all().count() == 0:
        Account.objects.create(last_transaction_date_time = datetime.datetime.now(), balance = 0)
    account = Account.objects.get(id=1)
    balance = account.balance
    return render(request, ‘account/show_balance.html’, {‘balance’: balance})

def credit(request):
    if request.POST:
        form = CreditForm(request.POST)
        if form.is_valid():
```



```
        amount = form.cleaned_data['amount']
        date_time = datetime.datetime.now()
        Transaction.objects.create(date_time=date_time,amount=amount, debit=False)
        account = Account.objects.get(id=1)
        account.balance += amount
        account.save()
        return HttpResponse('<h1 style="background-color:green;">Credited Successfully!</h>')
    else:
        form = CreditForm()
    return render(request, 'account/credit.html', {'form': form})

def debit(request):
    if request.POST:
        form = DebitForm(request.POST)
        if form.is_valid():
            amount = form.cleaned_data['amount']
            date_time = datetime.datetime.now()
            Transaction.objects.create(date_time=date_time,amount=amount, debit=True)
            account = Account.objects.get(id=1)
            account.balance -= amount
            account.save()
            return HttpResponse('<h1 style="background-color:cyan;">Debited Successfully!</h>')
    else:
        form = DebitForm()
    return render(request, 'account/debit.html', {'form': form})
```

Admin.py

Models are registered through the admin interface, allowing for various operations to be performed. The admin interface can be used to insert, delete, view, and update database records for the Account and Transaction models registered here. We created a superuser account using py manage.py create super user. The superuser creates the account and sets its initial balance.

admin.py

```
from django.contrib import admin

from.models import Account, Transaction

# Register your models here.

admin.site.register(Account)

admin.site.register(Transaction)
```

Forms.py

Forms can be built very simply in Django based on the models they represent. For example, here, we have a form called a debit form based on the debit model and a Credit Form based on the credit model. Forms can have validations that check the appropriateness or correctness of the entered data. Here, we check whether the amount of credit or debit is always positive. We also verify that the amount to be debited is not greater than the current balance. Validation was achieved by raising forms. Validation Error when conditions fail. This is performed using the clean method of the associated form.

forms.py

```
from Django import forms
from.models import Account, CreditRequest, DebitRequest

class CreditForm(forms.ModelForm):
    class Meta:
        model = CreditRequest
        fields = '__all__'

    def clean(self):
        amount = self.cleaned_data['amount']
        balance = Account.objects.get(id=1).balance
        if amount <= 0:
            raise forms.ValidationError('Amount should be positive.')
        return self.cleaned_data

class DebitForm(forms.ModelForm):
    class Meta:
        model = DebitRequest
        fields = '__all__'

    def clean(self):
        amount = self.cleaned_data['amount']
        balance = Account.objects.get(id=1).balance
        if amount <= 0:
            raise forms.ValidationError('Amount should be positive.')
        if amount > balance:
            raise forms.ValidationError('Insufficient Balance')
        return self.cleaned_data
```

Urls.py

URLs are provided by path instances in urls.py in the account application. The URL corresponds to each view. That is, Show _balance, credit, and debit are URLs. They can be referred to as optional. These URLs were included in the URLs of the main bank project.

urls.py

```
from django.urls import path
from.views import show_balance, credit, debit


urlpatterns = [
    path('show_balance', show_balance),

    path('credit', credit, name='credit'),

    path('debit', debit, name='debit'),
]
```

Templates: Static view

Template: show_balance.html

```
<h1>The balance in the bank is Rs. {{balance}}.</h1>
```


Template for credit form

Template: credit.html

```
<h1>Credit</h1>
<form method='post' action='credit'>
    {% csrf_token %}
    {{form.as_p}}
    <input type='submit'/>
</form>
```

Template for debit form

Template: debit.html

```
<h1>Debit</h1>
<form method='post' action='debit'>
    {% csrf_token %}
    {{form.as_p}}
    <input type='submit'/>
</form>
```

RUNNING AND RESULTS

How To Run

On the command line, in the outer case study, the directory gives the command py manage.py run server, as shown in Figure 4.

The server starts. Point browser at http://127.0.0.1:8000/account/show_balance.

You will see static information like this:

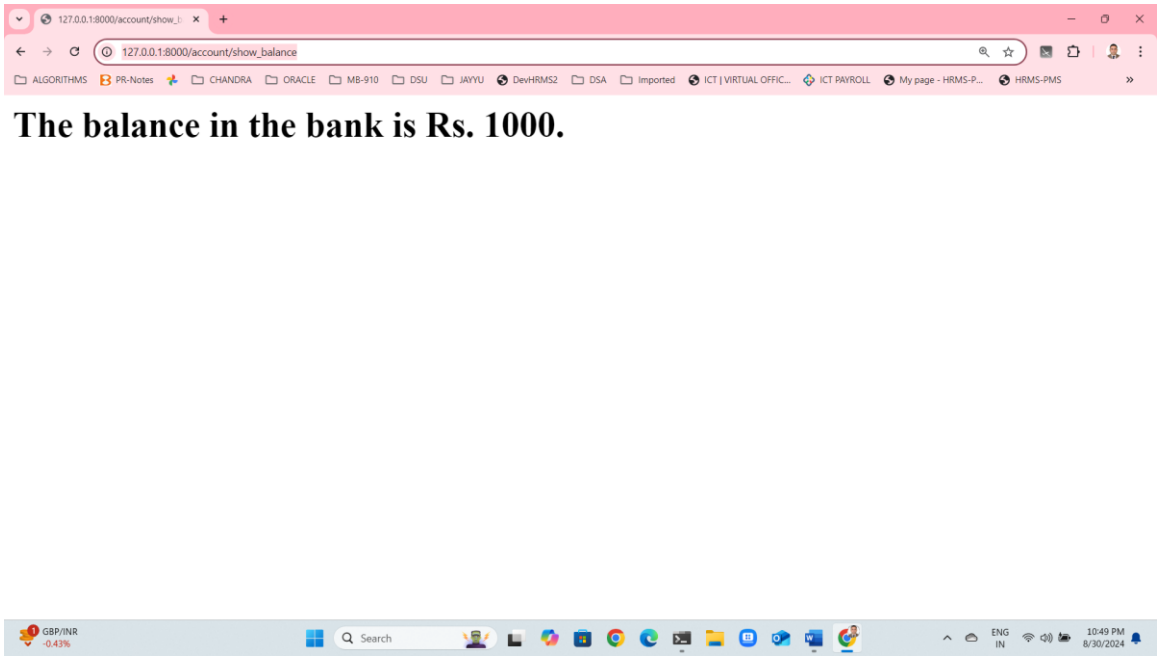

Figure 4. Show the balance page.

Go to the admin page http://127.0.0.1:8000/account/credit.Try, crediting 100 rupees, as shown in Figure 5.

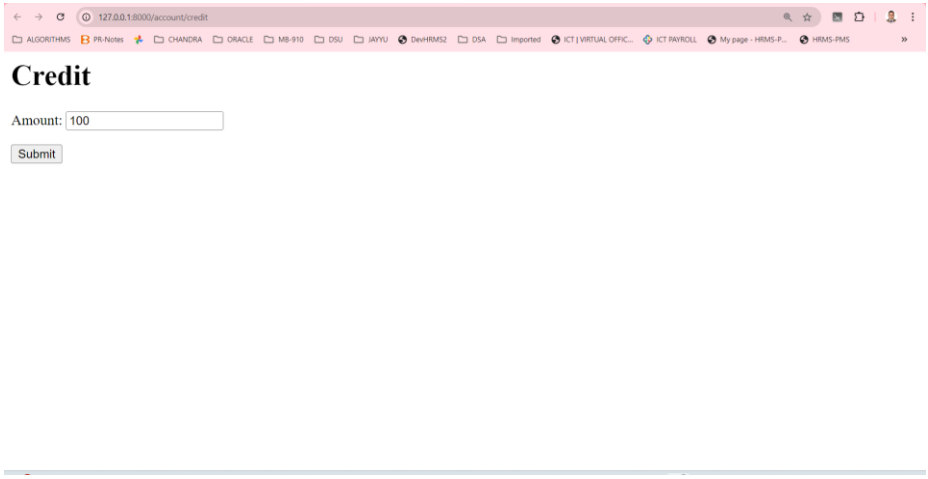

Figure 5. Credit form.

After clicking submit, the following shows:

Figure 6. Credit success message.

The balance was rechecked after crediting. It will increase by 100 rupees, as shown in Figure 6. Similarly, we tried http://127.0.0.1:8000/account/debit, as shown in Figure 7.

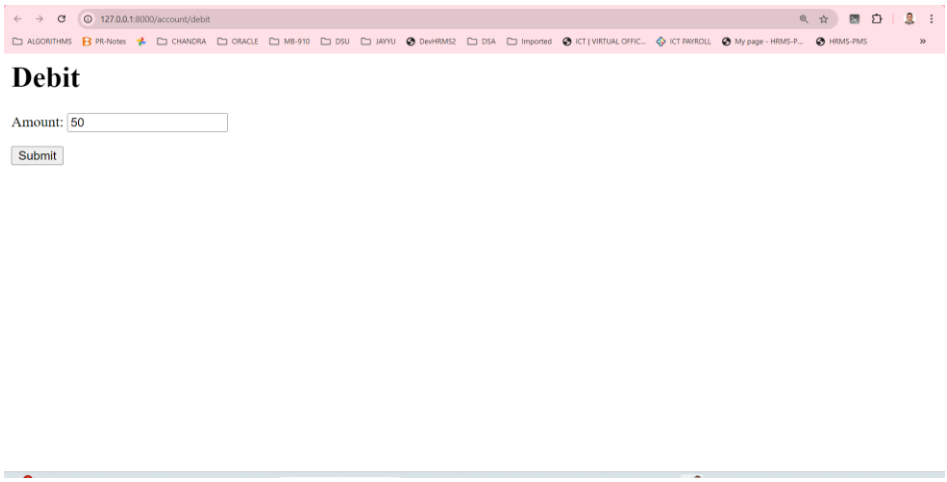

Figure 7. Debit form.

After clicking submit, the following shows as shown in Figure 8.

Figure 8. Debit success message.

Check the balance again: http://127.0.0.1:8000/account/show_balance, as shown in Figure 9.

Figure 9. Balance again.

Validation Errors

There will be a validation error if the amount (credit or debit) is not positive or if the debit amount exceeds the balance. For example, try debiting 2000 rupees as shown in Figure 10.

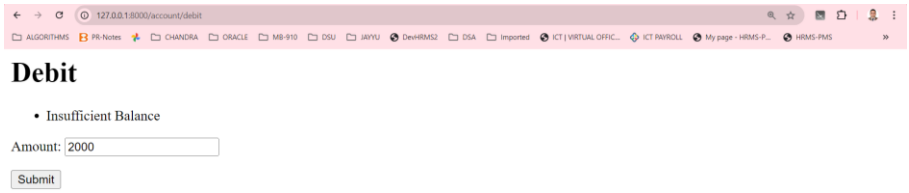

Figure 10. Insufficient balance error.

Table 1. Test case for SBS.

TC	Test case description	Expected output	Actual output	Result
TC1	Create an Account Record using the Admin interface and try to show the balance	Balance should be as set by the Admin	Balance should be as set by the Admin	Pass
TC2	Credit an amount > 0	Credited successfully page	Credited successfully page	Pass
TC3	Check balance again	Balance should increase by the amount	Balance should increase by the amount	Pass
TC4	Try crediting 0 or a negative integer amount	Validation error	Validation error	Pass
TC5	Debit an amount less than or equal to the balance	Debit success message	Debit success message	Pass
TC6	Check balance after debit	Balance will be less than the debited amount	Balance will be less than the debited amount	Pass
TC7	Try debiting 0 or a negative amount	Validation error	Validation error	Pass
TC8	Try debiting an amount greater than the balance	Validation error: insufficient balance	Validation error: insufficient balance	Pass

TESTING

Testing is an important phase of software development. Testing ensures that a product is suitable for use. As we have discussed in the running part, the following 'Test-Cases' can be written thus ensuring its reliability and functionality as shown in Table 1.

CONCLUSION

We demonstrated a Django project to maintain a simple banking system. The account application has suitable models for accounts, transactions, credit requests, and bit requests. It has a model form based on the credit request model and another based on the debit request model. The view for processing posted requests makes use of the form, and if the form is valid, accepts the request and makes the necessary changes to the account record and transaction records. The admin interface for SBS allows users to view account balances and monitor transactions.

REFERENCES

1. Holovaty A, Kaplan-Moss J. The Definitive Guide to Django: Web Development Done Right. New York City: Apress; 2009. p. 3–433.
2. W3Schools.com. (2024). Django tutorial [online]. W3Schools.com. Available from: https://www.w3schools.com/django/
3. GeeksforGeeks. (2020). Django tutorial. Learn Django framework [online]. GeeksforGeeks. Available from: https://www.geeksforgeeks.org/django-tutorial/
4. Van Rossum G. Python programming language. 2007 USENIX Annual Technical Conference, June 17, 2007–June 22, 2007, Santa Clara, CA, United States. 2007.
5. Northwood C. The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer. New York: Apress, 2018. p. 1–9.
6. Bendoraitis A, Kronika J. Django 3 Web Development Cookbook: Actionable Solutions to Common Problems in Python Web Development. Birmingham, United Kingdom: Packt Publishing Ltd.; 2020.
7. Stangl M, Pielmeier J, Berger C, Braunreuther S, Reinhart G. Development of a web-based monitoring system for a distributed and modern production. Procedia CIRP. 2016;52:222–7. DOI: 10.1016/j.procir.2016.07.073.
8. Duckett J. JavaScript and JQuery: Interactive Front-End Web Development. New Jersey, United States: Wiley Publishing; 2014.
9. Kofler M. What is MySQL? In: The Definitive Guide to MySQL 5. 3rd ed. Berkeley, CA: Apress; 2005. p. 3–16. DOI: 10.1007/978-1-4302-0071-0.
10. Cico O, Jaccheri L, Nguyen-Duc A, Zhang H. Exploring the intersection between software industry and software engineering education – a systematic mapping of Software Engineering trends. J Syst Softw. 2021;172:110736. DOI: 10.1016/j.jss.2020.110736.

