**Advanced Software Engineering Banking System Documentation**

Agile methodology, particularly when implemented with Scrum, can be highly beneficial for developing banking applications due to its iterative and flexible nature. Hence this methodology was taken for developing our banking software. Here's how Agile with Scrum was applied:

**Project Planning**

|  |
| --- |
|  |

**Fig 1: Gantt chart showing project timeline for banking application.**

A Gantt chart is a tool used for project planning and management that visually represents the scheduled task with their timeline. In Fig 1, we can see the Gantt chart for this banking project which gave us valuable insight. The above chart depicts when different stages of project such as Requirement Gathering, Development, testing, bugfixes would start and end. It gives a clear idea of how the project delivery is planned right from requirement gathering.

**Prototype design**

**System design**  
System design is the implementation of the plan for the software system. It includes a designing of the interface, architecture, components and modules in order to meet specific requirements and objectives. Engineers shall look at scalability, databases, communication protocols, security, testing, caching, fault tolerance and monitoring. The purpose is to develop a trusted, easy-to-manage, well maintained end-to-end system .

**Behavioral Diagram**

A behavioral diagram in software engineering is a type of diagram that focuses on illustrating the dynamic behavior of a system, including how different components interact with each other over time.

1. **Use Case Diagram**: Use case diagrams describe the functional requirements of a system from the perspective of its users (actors) and the interactions between users and the system. They depict the various use cases (tasks or functionalities) supported by the system and how users interact with it to accomplish those tasks.

|  |
| --- |
| A diagram of a customer transaction  Description automatically generated  **Fig 2: Use Case Diagram for transaction.** |

Fig 2 shows the use case diagram for the transaction. Here, we can see the various functionalities available for a customer which is related to transactions. Customer has options to deposit amount, withdraw amount, transfer amount and download the receipt of any transaction.

1. **Sequence Diagram**: A sequence diagram is a graphical representation that explains how objects and classes collaborate during a specific scenario and exchange the messages in a particular sequence/order.It gives overview about the collaboration, scenario flow and how classes and objects interact.

|  |
| --- |
| A diagram of a server  Description automatically generated |

**Fig 3: Sequence Diagram for Customer Logout Process**

Fig 3 shows the sequence diagram for logout process. In Fig 3, we can see sequence of actions for user to logout. It depicts that once the customer click on logout, that request will be sent to the browser and server. Server then sends close session request, and authentication service clears the authentication and sends a logout response to server. Then server logs out the customer and redirect the user to login screen.

**Structural Diagram**

A structural diagram is a graphical representation that illustrates the structure or organization of a system, often focusing on its components, relationships, and interactions. It provides a visual overview of the system's architecture, highlighting key elements such as classes, objects, modules, components, and their interconnections. Structural diagrams are commonly used in software engineering to depict the static aspects of a system, including its composition, dependencies, and hierarchy. Examples of structural diagrams include class diagrams, object diagrams, component diagrams, and package diagrams, each serving a specific purpose in capturing different facets of the system's structure. These diagrams aid in understanding system complexity, facilitating communication among stakeholders, and guiding the design and implementation process.

**Class Diagram**

A class diagram is a type of structural diagram in software engineering that depicts the structure and relationships of classes within a system. It illustrates the attributes, methods, and associations of classes, as well as their inheritance and dependencies. Class diagrams are widely used during the design phase to visualize the organization of classes and their interactions, serving as a blueprint for the implementation of software systems.

A diagram of a bank account

Description automatically generated

**Fig 4: Class diagram for customer, online transaction and account**

Fig 4 provides a overview of the class and this diagram explains different parameters and methods used in this project

**GUI design**

Visual Designs were developed on Figma to have better visuality on how the developed screens would look like. Some of the screens are as follows:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Developed Screens**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Codes**

**Code for OTP**

from django.views import View

from django.shortcuts import render, redirect

from django.views.decorators.cache import never\_cache

from django.utils.decorators import method\_decorator

from ..utils.cryptography import Cryptography

from ..utils.otp import OTPHandler

from ..repositories.customer import CustomerRepository

@method\_decorator(never\_cache, name='dispatch')

class LoginView(View):

    def get(self, request):

        return render(request, 'login.html')

    def post(self, request):

        error\_message = None

        try:

            CustomerRepository.deleteUserSession(request)

            emailID = request.POST['email']

            password = request.POST['password']

            customerDetails = CustomerRepository.findByEmail(emailID)

            if customerDetails == None:

                error\_message = 'Invalid email ID. Please enter valid Email ID'

                return render(request, 'login.html', {"error\_message": error\_message})

            # decrypt password

            decryptedPass = Cryptography.decryption(customerDetails.password)

            if password != decryptedPass:

                error\_message = 'Invalid password. Please enter correct password.'

                return render(request, 'login.html', {"error\_message": error\_message})

            request.session['email'] = emailID

            request.session['customerId'] = customerDetails.id

            request.session['otpAction'] = 'login'

            otpObj = OTPHandler(request, emailID)

            otpObj.sentOTP()

            return redirect('otp')

        except Exception as e:

            print(e)

            return render(request, 'login.html', {"error\_message": 'Something went wrong!!'})

**Code for Transaction**

from django.views import View

from django.shortcuts import render

from django.views.decorators.cache import never\_cache

from django.utils.decorators import method\_decorator

from ..repositories.transaction import TransactionRepository, Transaction

from django.core.serializers import serialize

from datetime import datetime, time

@method\_decorator(never\_cache, name='dispatch')

class TransactionView(View):

    def get(self, request):

        date = ''

        amount = ''

        transType = 'None'

        customerId = request.session['customerId']

        tableData = []

        try:

            tableData, \_, \_, \_ = TransactionRepository.getTransactionList(

                customerId)

            totalUnFilteredDataCount = tableData.count()

        except Exception as e:

            print(e)

        return render(request, 'transaction.html', {"tableData": tableData, "date": date, "amount": amount, "transType": transType, "totalUnFilteredDataCount": totalUnFilteredDataCount, "transactionData": serialize('json', tableData)})

    def post(self, request):

        date = ''

        amount = ''

        transType = 'None'

        tableData = []

        totalUnFilteredDataCount = 0

        try:

            customerId = request.session['customerId']

            if request.POST['actionBtn'] == "Filter":

                date = request.POST['date']

                amount = request.POST['amount']

                transType = request.POST['transType']

                if (date or amount or transType):

                    tableData = TransactionView.filterTransaction(

                        date, amount, transType, customerId)

                    totalUnFilteredDataCount = tableData.count()

            if request.POST['actionBtn'] == "Clear Filter":

                tableData, \_, \_, \_ = TransactionRepository.getTransactionList(

                    customerId)

                totalUnFilteredDataCount = tableData.count()

        except Exception as e:

            print(e)

        return render(request, 'transaction.html', {"tableData": tableData, "date": date, "amount": amount, "transType": transType, "totalUnFilteredDataCount": totalUnFilteredDataCount, "transactionData": serialize('json', tableData)})

    @staticmethod

    def filterTransaction(date="", amount="", transType="", customer\_id=""):

        activeFiler = {

            "customer\_id": customer\_id

        }

        if date:

            date = datetime.strptime(date, "%Y-%m-%d")

            start\_date = datetime.combine(date, time.min)

            end\_date = datetime.combine(date, time.max)

            activeFiler['date\_\_range'] = (start\_date, end\_date)

        if transType and transType != 'None':

            activeFiler['transType'] = transType

        if amount:

            activeFiler['amount'] = amount

        tableDataFilter = Transaction.objects.filter(

            \*\*activeFiler).order\_by('-pk')

        return tableDataFilter