Chapter 3: Internship Activities

3.1. Roles and Responsibilities

While working as a Backend Python Developer intern for Khalti, my main focus was in the development and refinement of a various feature, which involved designing and implementing backend solutions using Python and Django, ensuring these systems were robust, scalable, and seamlessly integrated with the existing digital wallet platform.I had the following tasks:

i) **Code Development and Testing**:
Worked on new backend features and enhanced existing functionalities using Python and Django, and I conducted thorough testing to ensure robustness and reliability.

ii) **System Maintenance and Optimization**:
Worked out routine maintenance and optimization tasks to improve the efficiency and scalability of the backend systems.

iii) **Database Management**:
I managed database schemas and integrated data storage solutions, ensuring data consistency and security.

iv) **Implementation of Security Protocols**:
Implemented and reviewed security measures to safeguard the digital wallet against potential threats and vulnerabilities.

v) **Collaboration with Other Teams**:
Maintained close collaboration with the front-end development team and other technical departments to ensure seamless integration and functionality across platforms.

vi) **Documentation of Development Processes**:
Created and maintained comprehensive documentation of all development processes, changes, and upgrades, providing a clear reference for future development efforts and troubleshooting.

3.2. Weekly log

Following table shows the weekly activities the intern performed throughout their internship period.

Table 3.1 : Weekly Log

Week	Activities
Week 1	• Introduced to the team and company culture. • Setup of personal development environment and necessary software. • Overview of the digital wallet codebase with a senior developer. • Attended workshops on Python and Django, the core technologies used.
Week 2	• Studied existing email functionalities within the digital wallet. • Reviewed documentation of related backend processes. • Participated in daily stand-ups to discuss ongoing projects. • Learned about security protocols relevant to user data handling.

Week 3	• Began coding the email update feature in the user profile section. • Implemented backend logic for sending verification emails. • Set up database migrations to handle new email data securely. • Developed error handling routines for failed email updates.
Week 4	• Enhanced error handling routines for failed email updates. • Improved user interface for email update feature. • Coordinated with frontend developers to align on feature integration. • Drafted initial user guides for the new feature.
Week 5	• Integrated the new email functionality with the frontend interface. • Conducted preliminary testing with dummy data to ensure stability. • Identified bugs and issues in the initial deployment. • Refined the API responses for better user feedback on errors.
Week 6	• Submitted code for peer review and incorporated feedback. • Optimized database queries for faster email updates. • Enhanced security measures based on latest best practices. • Wrote comprehensive unit tests for the new features.
Week 7	• Deployed the feature in a controlled test environment. • Monitored user interactions and collected feedback from test users. • Updated project documentation to include new features and changes. • Prepared a rollback plan for potential deployment issues.
Week 8	• Made final adjustments based on user feedback and testing results. • Conducted a final review with the project team and stakeholders. • Officially launched the email change feature to production. • Completed a detailed handover to the maintenance team, ensuring they are equipped to manage and troubleshoot the feature.
Week 9	• Analyzed post-launch metrics to measure feature adoption and performance. • Addressed any critical issues reported by users. • Continued optimizing the feature for performance and security. • Began documenting lessons learned from the project.
Week 10	• Participated in a retrospective meeting to discuss project successes and areas for improvement. • Provided training sessions for the support team on the new feature. • Assisted in resolving any outstanding user issues. • Worked on minor feature enhancements based on user feedback.

Week 11	• Collaborated with marketing to create promotional materials for the new feature. • Monitored long-term performance and user satisfaction. • Identified potential areas for future improvements. • Began planning for the next feature development cycle.
Week 12	• Completed a comprehensive project report summarizing all activities and outcomes. • Held a final project wrap-up meeting with all stakeholders. • Provided final updates to the project documentation. • Celebrated the successful completion of the project with the team.

3.3. Description of the Project Involved During Internship

One of the highlights which really helped me in understanding whole software development process was working on two minor projects during my internship both using django and django rest framework.

Use Case ID	User Case Name	Actor	Description	Preconditions	Main Flow	Alternative Flow	Postconditions
UC01	View Profile	Consumer	Users can view additional options in their profile.	User is logged into the Khalti App and is in the Profile section.	i) User accesses profile.2. User views 3 dots for more options.	None	User sees 3 dots.
UC02	Add Email	Consumer	Allows the user to add an email address to their profile.	User is in the Profile section of the Khalti App.	i) User clicks on 3 dots.2. User selects “Add Email”.3. User enters email and submits for authentication.	i) Invalid email format.2. Email linked to another account.	Email is added pending verification.

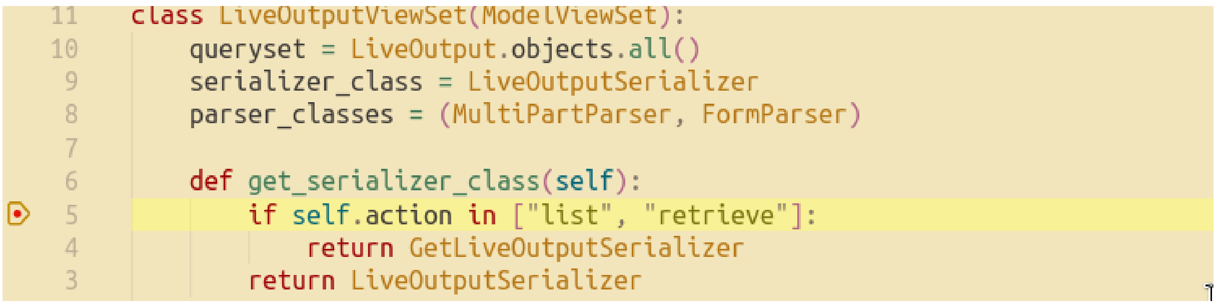
Use Case ID	User Case Name	Actor	Description	Preconditions	Main Flow	Alternative Flow	Postconditions
UC03	Authenticate Email Addition	Consumer	Handles authentication required for adding an email.	User has entered a valid email address.	i) User is prompted for authentication.2. User completes authentication.	i) Biometric fails, try password.2. Incorrect password input.	Email is authenticated, verification link sent.
UC04	Update Email	Consumer	Allows the user to change their email address.	User has previously added an email.	i) User accesses "Update Email" from profile.2. User enters new email and authenticates.	Same as Add Email use case.	Email update process initiated.
UC05	Handle Authentication Failures	Consumer	Manages repeated authentication failures.	User has failed authentication attempts.	User attempts to authenticate and fails.	User fails multiple times leading to lock-out.	User is temporarily locked out after specified failures.
UC06	Handle Expired Verification Link	Consumer	Manages scenarios where a user clicks an expired verification link.	User has received a verification link.	User clicks expired verification link.	None	User is directed to a webpage to request a new link.
UC07	Resend Verification Email	Consumer	Allows users to resend the verification email if not received.	User has not received the initial verification email.	User selects the option to resend verification email.	None	Verification email is resent.
UC08	Handle Network Issues During	Consumer	Manages scenarios with network connectivity	User is adding an email.	User faces network issues during	None	Error message is displayed, advises

Use Case ID	User Case Name	Actor	Description	Preconditions	Main Flow	Alternative Flow	Postconditions
	Email Submission		issues during email submission.		the submission.		checking network.

3.4. Tools used

i) Python Debugger

The utilization of the VS Code debugger significantly enhanced my development workflow by allowing me to efficiently identify and resolve issues within the codebase. This tool provided an interactive environment where I could set breakpoints, inspect variables, and step through the code line by line. Consequently, it helped me understand the flow of the application and pinpoint the exact locations of bugs. This experience not only increased my debugging efficiency but also deepened my comprehension of the underlying code structure, leading to more robust and error-free implementations.

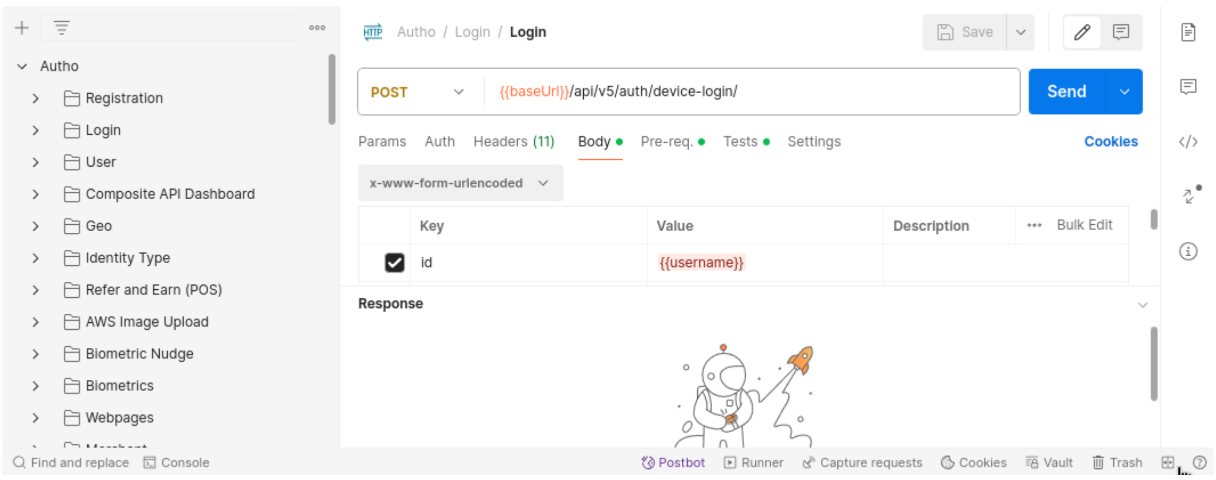


```python
class LiveOutputViewSet(ModelViewSet):
    queryset = LiveOutput.objects.all()
    serializer_class = LiveOutputSerializer
    parser_classes = (MultiPartParser, FormParser)

    def get_serializer_class(self):
        if self.action in ["list", "retrieve"]:
            return GetLiveOutputSerializer
        return LiveOutputSerializer
```

Figure 3.6 : Breakpoint in VS Code Debugger

ii) Postman

Using Postman was instrumental in testing and verifying the APIs I developed. This tool enabled me to create, send, and manage HTTP requests, facilitating thorough testing of endpoints and ensuring they behaved as expected. By simulating various scenarios and examining the responses, I was able to identify and rectify issues early in the development process. This practice improved the reliability and performance of the APIs, ensuring seamless integration with other system components and delivering a better user experience.



Autho
- Registration
- Login
- User
- Composite API Dashboard
- Geo
- Identity Type
- Refer and Earn (POS)
- AWS Image Upload
- Biometric Nudge
- Biometrics
- Webpages

Autho / Login / Login

POST {{baseUrl}}/api/v5/auth/device-login/

Params | Auth | Headers (11) | Body | Pre-req. | Tests | Settings | Cookies

x-www-form-urlencoded

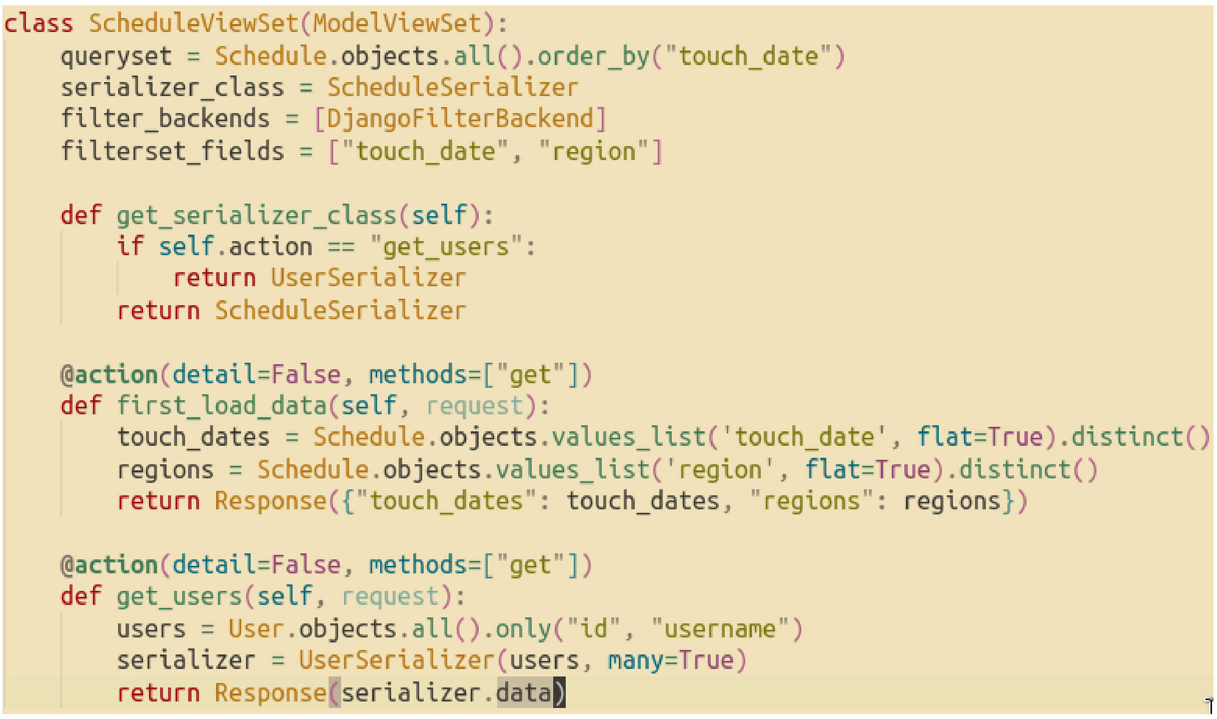
	Key	Value	Description
✓	id	{{username}}	

Response

Figure 3.7 : API collections in Postman

iii) Using Git (Merge, Fetch, Cherry-pick)
Mastering Git commands such as merge, fetch, and cherry-pick significantly streamlined my version control processes. Git merge allowed me to combine multiple branches into a cohesive codebase, ensuring all features and fixes were integrated smoothly. Git fetch helped keep my local repository up to date with the remote repository, preventing discrepancies and potential conflicts. Git cherry-pick was particularly useful for selectively applying specific commits from one branch to another, facilitating precise and controlled code updates. These skills enhanced my ability to manage code changes effectively, maintain a clean project history, and collaborate seamlessly with team members.

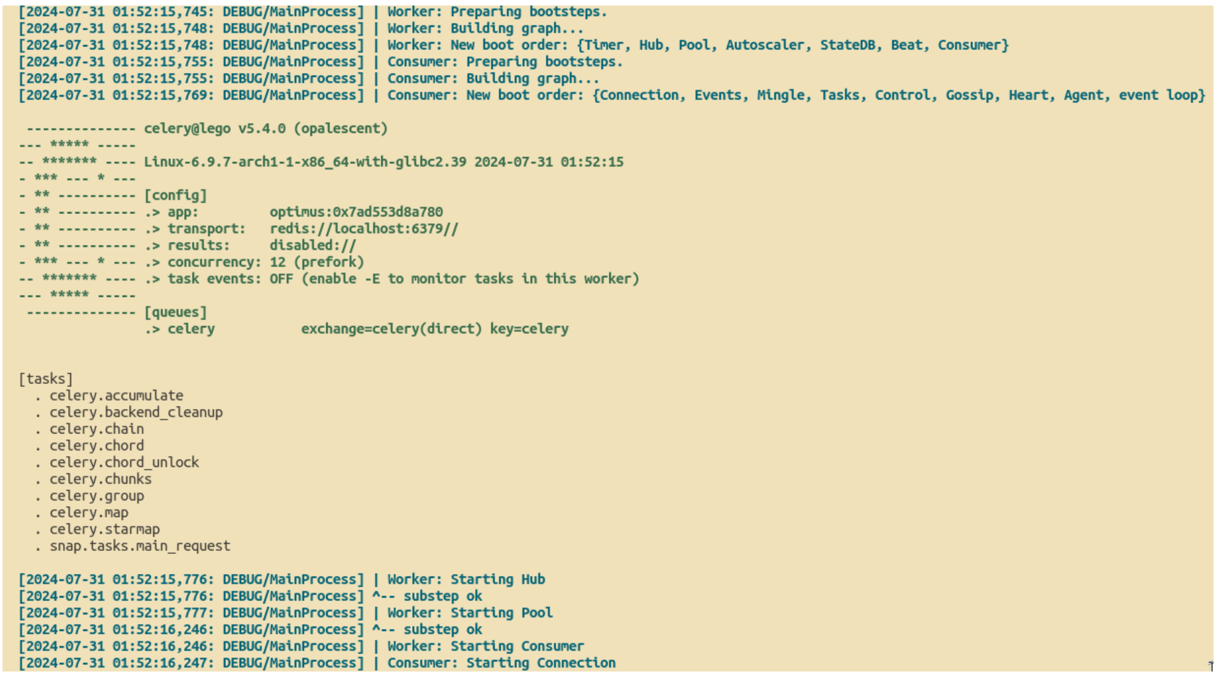
iv) Using DRF (Django Rest Framework)
Developing an in-depth understanding of how the Django Rest Framework (DRF) works, including viewsets and routers, was pivotal in building scalable and maintainable APIs. By comprehending the intricacies of viewsets, I could encapsulate common patterns for interacting with the data in a reusable manner. Understanding routers enabled me to automate URL routing, reducing boilerplate code and potential errors. This knowledge empowered me to construct more efficient and organized APIs, facilitating easier maintenance and extensibility of the codebase.



```python
class ScheduleViewSet(ModelViewSet):
    queryset = Schedule.objects.all().order_by("touch_date")
    serializer_class = ScheduleSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ["touch_date", "region"]

    def get_serializer_class(self):
        if self.action == "get_users":
            return UserSerializer
        return ScheduleSerializer

    @action(detail=False, methods=["get"])
    def first_load_data(self, request):
        touch_dates = Schedule.objects.values_list('touch_date', flat=True).distinct()
        regions = Schedule.objects.values_list('region', flat=True).distinct()
        return Response({"touch_dates": touch_dates, "regions": regions})

    @action(detail=False, methods=["get"])
    def get_users(self, request):
        users = User.objects.all().only("id", "username")
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)
```

Figure 3.8 : ModelViewSet of Django Rest Framework

v) Using Celery Tasks
Learning about Celery tasks and their implementation in my projects played a crucial role in optimizing performance and handling asynchronous operations. By offloading time-consuming tasks, such as sending emails or processing large datasets, to Celery, I was able to ensure that the main application remained responsive and efficient. This capability not only enhanced the user experience by reducing latency but also improved the scalability of the application, allowing it to handle a higher volume of concurrent operations seamlessly.



```
[2024-07-31 01:52:15,745: DEBUG/MainProcess] | Worker: Preparing bootsteps.
[2024-07-31 01:52:15,748: DEBUG/MainProcess] | Worker: Building graph...
[2024-07-31 01:52:15,748: DEBUG/MainProcess] | Worker: New boot order: {Timer, Hub, Pool, Autoscaler, StateDB, Beat, Consumer}
[2024-07-31 01:52:15,755: DEBUG/MainProcess] | Consumer: Preparing bootsteps.
[2024-07-31 01:52:15,755: DEBUG/MainProcess] | Consumer: Building graph...
[2024-07-31 01:52:15,769: DEBUG/MainProcess] | Consumer: New boot order: {Connection, Events, Mingle, Tasks, Control, Gossip, Heart, Agent, event loop}

 -------------- celery@lego v5.4.0 (opalescent)
--- ***** -----
-- ******* ---- Linux-6.9.7-arch1-1-x86_64-with-glibc2.39 2024-07-31 01:52:15
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:         optimus:0x7ad553d8a780
- ** ---------- .> transport:   redis://localhost:6379//
- ** ---------- .> results:     disabled://
- *** --- * --- .> concurrency: 12 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery


[tasks]
  . celery.accumulate
  . celery.backend_cleanup
  . celery.chain
  . celery.chord
  . celery.chord_unlock
  . celery.chunks
  . celery.group
  . celery.map
  . celery.starmap
  . snap.tasks.main_request

[2024-07-31 01:52:15,776: DEBUG/MainProcess] | Worker: Starting Hub
[2024-07-31 01:52:15,776: DEBUG/MainProcess] ^-- substep ok
[2024-07-31 01:52:15,777: DEBUG/MainProcess] | Worker: Starting Pool
[2024-07-31 01:52:16,246: DEBUG/MainProcess] ^-- substep ok
[2024-07-31 01:52:16,246: DEBUG/MainProcess] | Worker: Starting Consumer
[2024-07-31 01:52:16,247: DEBUG/MainProcess] | Consumer: Starting Connection
```

Figure 3.9 : Celery tasks for asynchronous operations

3.5. Tasks / Activities Performed

In this project, my main goal was to streamline the process of editing/verifying email address focusing on ease of use and security in mind.

i) The project is done using Django, Django Rest Framework, Django Templates, and Celery.
ii) The project involved creating a new API endpoint for users to change their email address.
iii) The project also involved creating a Celery task to send an email verification link to the new email address.
iv) The system was designed to ensure that the email change process was secure and user-friendly.

3.5.1 System Analysis

System analysis involves the detailed study of the current system model, leading to specification of a new system. A model provided the blueprint of the system. The system will be user friendly and will fulfill the user requirements as well.

3.5.1.1 Functional Requirement

The functional requirements for a system describe what the system should do. Those requirements depend on the kind of software being developed and the expected software users. These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in specific situations.

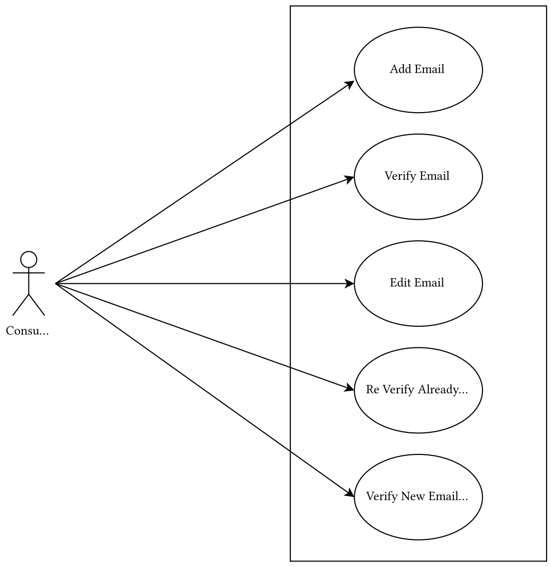


Figure 3.1 : Use case diagram

The Figure 3.1 represents a consumer and system interaction that shows the basic working features of the application itself, where the user can change their email address and verify it.

3.5.1.2 Non-Functional Requirement

Non-functional requirements are requirements that are not directly concerned with the specified function delivered by the system.

- **Maintainability**:
  Adhere to coding standards that promote readability and maintainability.
- **Compliance**:
  Ensure the system complies with relevant data protection regulations (like GDPR, if applicable) regarding user data.

3.5.1.3 Feasibility Study

Before starting the project, a feasibility study is carried out to measure the system's viability. A feasibility study is necessary to determine if creating a new or improved system is friendly with the cost, benefits, operation, technology, and time. The following are the feasibility concerns in this project:

i) **Technical Feasibility**:

The technical feasibility of enhancing the email edit/update feature in a Django-based system considers both the current technological framework and specific security requirements essential in the financial sector. Given that Django is robust in managing secure user interactions and database modifications (necessary for updating email addresses), the existing infrastructure is likely sufficient. However, due to the sensitive nature of financial data, the security implementations need to be rigorously evaluated. This includes ensuring encryption for data transmissions and safeguarding against vulnerabilities specific to financial applications, such as phishing and session hijacking.

i) **Operational Feasibility**:
From an operational feasibility standpoint, integrating this feature into Khalti's digital wallet platform is seamless and minimally disruptive to existing operations. Since digital wallets require high reliability and user trust, any changes that involve user account details, like email addresses, must be handled with extreme caution to avoid eroding trust or introducing errors. The introduction of the feature should be accompanied by comprehensive user documentation and possibly a brief tutorial within the app to facilitate adoption. Additionally, since this feature is fundamental to account security, ensure that your customer support team is well-prepared to address any issues that arise quickly and effectively. This might involve specialized training and updating internal operational protocols.

ii) **Economic Feasibility**:
The economic feasibility for Khalti involves a detailed analysis of costs versus anticipated benefits. The direct costs include development, testing, additional security measures, and training. Operationally, the feature should lead to a reduction in support costs, as users can self-manage their email details, potentially lowering the volume of support requests related to account access issues. From a benefits perspective, enhancing user autonomy and security can improve customer satisfaction and retention—a crucial metric in the competitive fintech space. Calculating the return on investment should factor in these indirect benefits, such as enhanced user trust and reduced risk of security breaches, which can have significant financial implications. As a fintech entity, projecting the feature's impact on enhancing regulatory compliance and reducing fraud incidents could further justify the investment.

3.5.2 System Design

System design defines the components, modules, interfaces, and data for a system to satisfy specified requirements. It can also be defined as creating or altering systems and the processes, practices, models, and methodologies used to develop them. The main objective of the detailed system design is to prepare a system blueprint that meets the goals of the conceptual system design requirements. The system designs for building this project include database schema, input-output design, class diagram, sequence diagram, and activity diagram.

3.5.2.1 Architectural Design

The architectural design shows the architecture of the overall system. The system is based on MVC architecture.

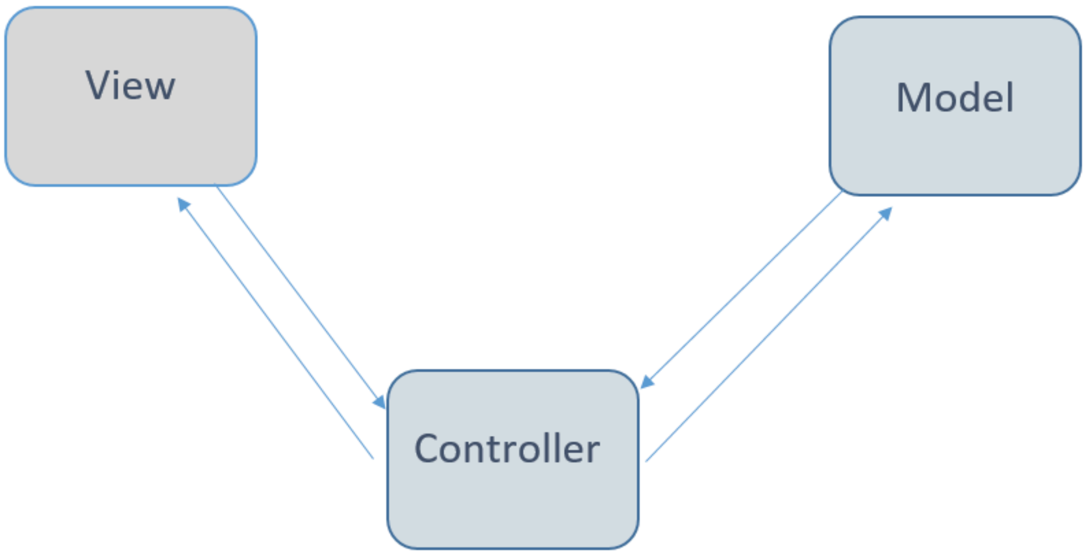
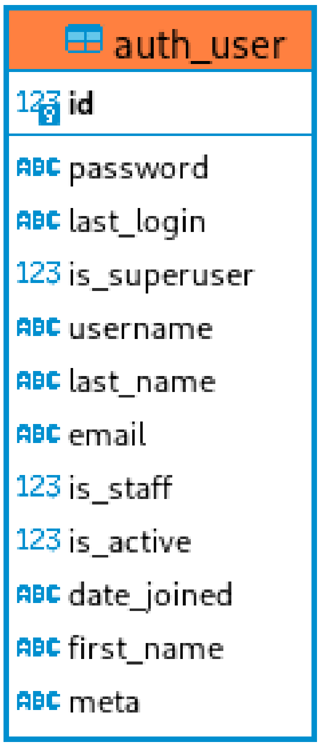
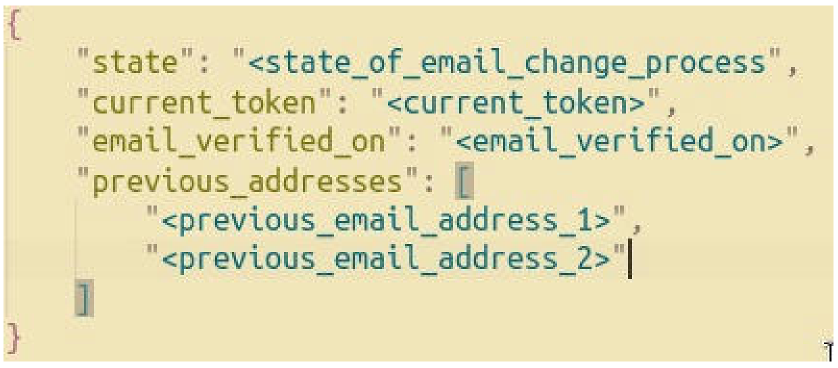


Figure 3.2 : Architectural Design of the System

3.5.2.2 Database Design

The following database schema is the general structure used in the application.


auth_user
id
password
last_login
is_superuser
username
last_name
email
is_staff
is_active
date_joined
first_name
meta



```
{
    "state": "<state_of_email_change_process",
    "current_token": "<current_token>",
    "email_verified_on": "<email_verified_on>",
    "previous_addresses": [
        "<previous_email_address_1>",
        "<previous_email_address_2>"
    ]
}
```

Figure 3.3 : Comparison of User Models

3.5.2.3 Class Diagram

The class diagram describes the relationships and source code dependencies among other classes.

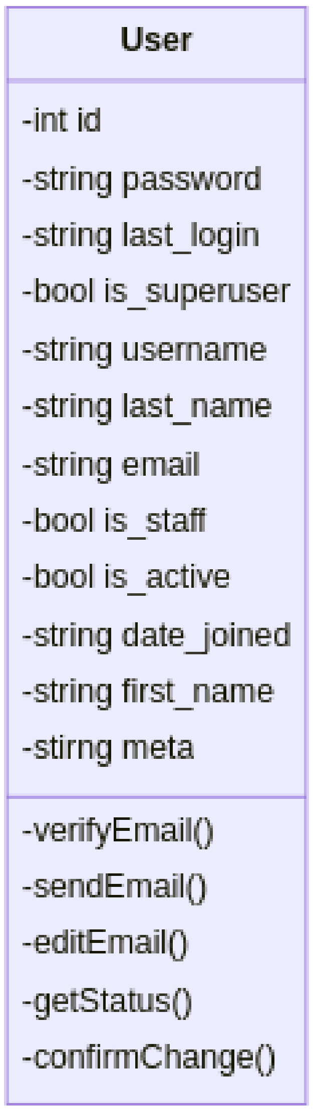


Figure 3.4 : Class Diagram of User Model

The Figure 3.3 illustrates the class diagram of the system. There is only one class, User, which is responsible for handling user data and operations on itself.

3.5.2.4 Sequence Diagram

A sequence diagram is an interaction diagram because it describes how and in what order a group of objects works together. Sequence diagrams are sometimes known as event diagrams. The sequence diagram of the system is shown below.

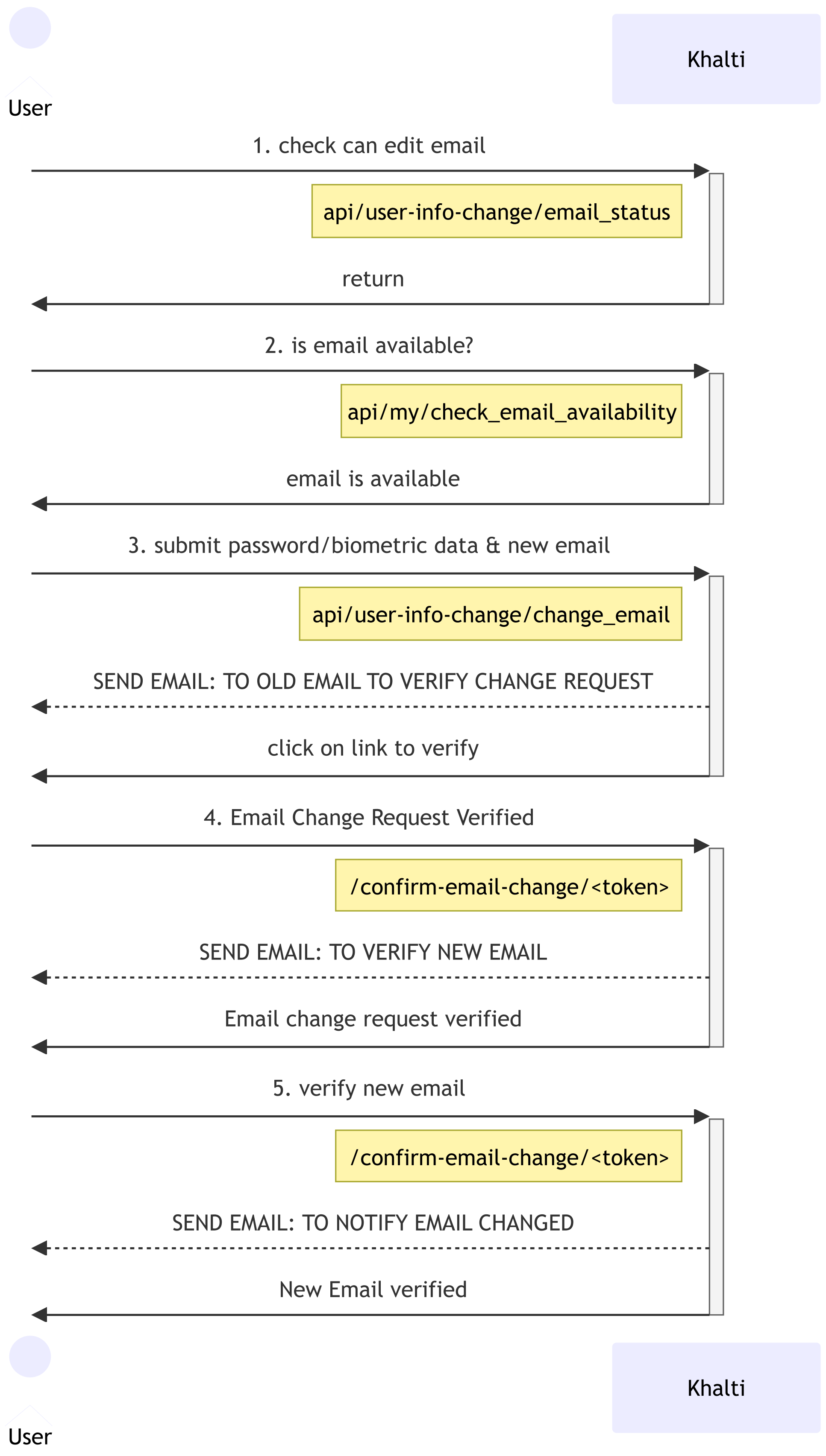

User
Khalti
1. check can edit email
api/user-info-change/email_status
return
2. is email available?
api/my/check_email_availability
email is available
3. submit password/biometric data & new email
api/user-info-change/change_email
SEND EMAIL: TO OLD EMAIL TO VERIFY CHANGE REQUEST
click on link to verify
4. Email Change Request Verified
/confirm-email-change/<token>
SEND EMAIL: TO VERIFY NEW EMAIL
Email change request verified
5. verify new email
/confirm-email-change/<token>
SEND EMAIL: TO NOTIFY EMAIL CHANGED
New Email verified
User
Khalti

Figure 3.5 : Sequence Diagram of Email change

The Figure 3.3 illustrates how the frontend system will interact with the backend api to change the email address of the user.

3.5.3 Implementation

The implementation of the email change process was designed with a focus on security, user experience, and scalability. The process involved multiple stages, including authentication, email verification, and the handling of potential errors or misuse scenarios. This section details the step-by-step implementation of the user email change process, highlighting key components and their interactions within the system.

i) **Authentication and Security**:
The first step in the email change process involves authenticating the user to ensure that the request is legitimate. Users can authenticate using either their password or biometric data. The chosen method is validated against the stored credentials. To mitigate security risks, the system tracks the number of incorrect authentication attempts. If the user exceeds a predefined threshold of failed attempts, the email change process is temporarily locked for a configured amount of time. This security measure is logged in the User model, where the state of the request and the number of failed attempts are recorded. This ensures that any potential fraudulent activity is monitored and controlled.

ii) **Initiating the Email Change Request**:
Once authenticated, the user can initiate the email change process. A new entry is created in the User model, where the state is set to REQUESTED, and the new email address is stored. This entry serves as the starting point for the subsequent verification steps. The system then sends a verification email to the new address. This email contains a unique verification link generated by concatenating the index (id), timestamp, and email type. This string is then encrypted using HMAC (Hash-based Message Authentication Code) with a salt retrieved from the application settings. The generated token is stored in the User model under the info field with the key current_token.

iii) **Email Verification Process**:
The user receives the verification email and clicks the provided link. Upon clicking the link, the system decodes the verification string to extract the user information and the token. The token is then compared against the stored value in the User model. If the tokens match, the process continues, and the current_token field is set to None to prevent reuse. At this point, the user's email in the User model is updated with the new address, and the is_email_verified field is set to true. The state of the User instance is updated to APPROVED, marking the completion of the email change process. Additionally, the system sets the email_verified_on field in the User model to record the timestamp of the verification.

iv) **Handling Verification Resends**:
If the user does not receive the verification email, they can request to resend it. The system retrieves the last User instance for the user with a REQUESTED state and resends the verification email. This process generates a new token and follows the same steps as the initial email verification, ensuring that the user can complete the process without starting over.

v) **Finalization and Error Handling**:
The system is designed to handle various edge cases, such as the presence of an unverified email in another user's account or multiple verification attempts. If the new email is already associated with another account but has not been verified, the system clears the email from that account and assigns it to the requesting user, marking it as verified. This prevents email address conflicts and ensures that each email is uniquely associated with a single user account. The implementation also includes a mechanism to decline email change requests if the verification process is completed by another user before the current user. This ensures that only one user can successfully verify a given email address, preventing potential security breaches.

vi) **Endpoints and API Integration**:
Several API endpoints were developed to support the email change process:

- **Check Email Availability**:
  This endpoint verifies if the new email is available for use.
- **Change Email Initiation**:
  Initiates the email change process, validating the password/biometric data and sending the initial verification email.
- **Email Validation Link**:
  Handles the verification of both old and new email addresses through a signed token.
- **Email Status**:
  Provides real-time status updates on the email change process, including any locked states or errors encountered during the process.

The implementation of these endpoints ensures a seamless and secure experience for users, allowing them to update their email addresses with confidence.

3.5.4 Testing

Testing the email change process was a critical part of the implementation, ensuring that all aspects of the system functioned as intended, particularly concerning security, reliability, and user experience. The testing process included unit tests, integration tests, and user acceptance tests (UAT), covering various scenarios, including edge cases and potential security vulnerabilities.

i) **Unit Testing**:
Unit tests were implemented to verify the functionality of individual components and methods within the email change process. These tests focused on the following key areas:

- **Authentication Validation**:
  Tests were conducted to ensure that the system correctly validates both password and biometric authentication methods. Tests included scenarios where incorrect credentials were provided, verifying that the system accurately logs the number of failed attempts and enforces the lockout policy after exceeding the allowed threshold.

- **Token Generation and Encryption**:
  The HMAC token generation process was tested to ensure that tokens were unique, correctly encrypted, and stored securely. Tests also verified the correct handling of the salt from the application settings, ensuring consistent encryption across different environments.

- **Model Integrity**:
  The UserInfoChange model was tested to ensure that data was correctly recorded, particularly the states (e.g., REQUESTED, APPROVED, DECLINED), email addresses, and tokens. Tests ensured that the transition between states was handled correctly and that the appropriate fields were updated during the process.

- **Email Sending**:
  The system's ability to send verification emails to the correct addresses (both old and new) was tested, including scenarios where the background email-sending process might fail or experience delays. Tests ensured that the system correctly handled these failures, such as by retrying email sends or logging errors for further investigation.

ii) **Integration Testing**:
Integration tests were conducted to validate the interaction between various components of the system. These tests ensured that the entire email change process worked seamlessly across different modules and external systems:

- **End-to-End Email Change Process**:
  Integration tests simulated the complete email change process, from the initial request to the final verification. These tests verified that the user could successfully change their email address, receive the correct verification emails, and have their information updated in the system. Tests also covered the case where the user requests a resend of the verification email, ensuring that the process worked consistently across multiple attempts.
- **Edge Case Handling**:
  Tests were conducted for scenarios where the email address might already be associated with another user account, particularly when that account's email is unverified. These tests ensured that the system correctly handled such cases, reassigning the email to the requesting user and clearing it from the previous account. Other edge cases included scenarios where the user might attempt to verify the same email address simultaneously in multiple sessions, testing the system's ability to handle these situations securely.
- **API Endpoint Testing**:
  Each API endpoint involved in the email change process was tested to ensure proper functionality, including the ability to handle various input conditions (e.g., invalid tokens, expired tokens, unavailable emails). Tests also verified the responses from the endpoints, ensuring that users received clear and accurate feedback regarding the status of their requests.

iii) **User Acceptance Testing (UAT)**
User Acceptance Testing involved real users interacting with the system to validate the email change process from a usability and user experience perspective. The following aspects were tested:

- **Usability**:
  Users were asked to initiate the email change process, follow the verification steps, and confirm that they could successfully update their email address. Feedback was collected to identify any usability issues or areas of confusion. The process was tested across different devices and browsers to ensure consistent user experience regardless of the platform used.
- **Error Handling and Messaging**:
  Test cases included scenarios where users entered incorrect passwords, did not receive verification emails, or attempted to use an email address already in use. The system's error messages and guidance were evaluated to ensure they were clear, helpful, and directed users on how to proceed. Users also tested the lockout feature by intentionally failing multiple authentication attempts, ensuring that the system appropriately enforced security measures and communicated the lockout status.
- **Performance**:
  The email verification process, particularly the background email-sending tasks, was tested under load to assess the system's performance. This included scenarios where multiple users initiated email changes simultaneously, ensuring that the system could handle the load without delays or failures.

iv) **Security Testing**:
Given the sensitive nature of email addresses and the potential risks of unauthorized changes, extensive security testing was conducted:

- **Token Expiration and Validation**:
  Tests were performed to ensure that the verification tokens expired after a set period and

could not be reused. The system's ability to detect and reject tampered or invalid tokens was also tested.

- **Cross-User Verification Attempts**:
  Scenarios were tested where one user might attempt to use another user's verification link. The system's ability to prevent such actions and ensure that only the intended user could verify their email address was validated.