Quantum Grid Project Documentation

Team Members

- **JoyJudy Wangui** – Database Design and Connectivity
- **Angela Achieng** – System Functionality
- **Brian Mathara** – Use Case Diagram and System Architecture.
- **Benir Odeny** – Database and Programming Logic
- **Gabriel Osugo** – User Payment and Functionality
- **Suraj Kumar** – System Design, Programming and Documentation.

Project Overview

Quantum Grid is a tech-driven electricity service platform designed to modernize Kenya's token-based electricity billing system. It aims to solve recurring challenges such as delayed token delivery, frequent outages, high costs, and poor service access, all while promoting sustainability and efficiency through its simple user functionality.

This documentation provides a detailed explanation of how Quantum Grid functions using Java, with reference to Object-Oriented Programming (OOP) concepts and SOLID design principles.

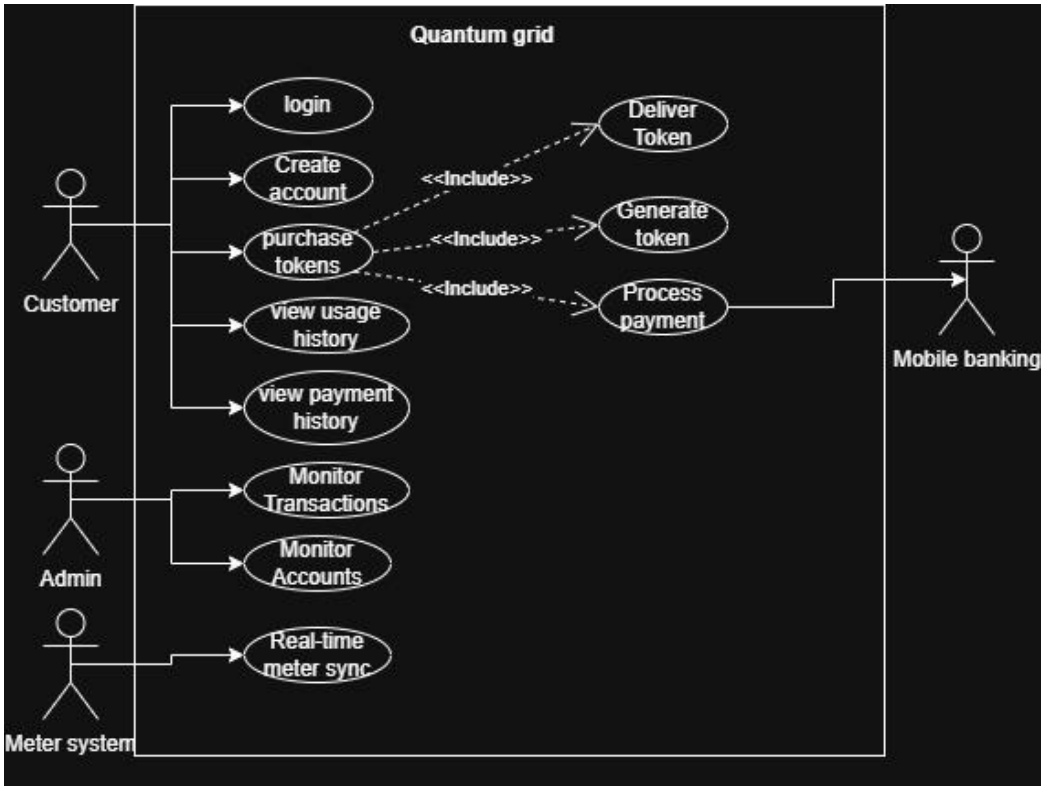
Functional Requirements

- User account creation and login
- Purchase of electricity tokens via M-Pesa, banking, or in-branch
- Real-time token generation and delivery
- Viewing usage and payment history
- Admin system monitoring
- Real-time syncing with electricity meters

Non-Functional Requirements

- Fast response for token generation
- Secure encrypted storage of user data
- High availability and uptime
- Scalable and user-friendly platform

System Functionality



1. **User enters amount and submits the token purchase request.**
2. **The payment system then routes the request.**
3. **The service layer then validates the payment.**
4. **Tokens are generated using a token generator.**
5. **Tokens are then saved to the database.**
6. **Notification is sent to the user based on success of conversion of tokens.**
7. **User is informed that tokens have been successfully delivered.**

User Account Creation and Login

Users interacting with the electricity monitoring application are obliged to have an account to monitor and make updates on payments with regards to electricity consumption. It then follows that users are required to have an account, and their credentials are stored in the database. To model the framework, users have a name, email and password.

When a new user is created or instantiated, then the user id and house number are generated based on the ID allocation which is handled by the SQL relation that auto increments IDs.

How it works:

- Our abstract framework is a user; A customer will extend from the user as a customer will have certain user functionality.
- A customer uses a certain payment service, which stores information in a certain Database. The relationship will require database connection to facilitate payment processing.
- A customer can be able to make payments based on certain conditions or check their token balance at any one time.
- Our administrator is different from the user as the administrators work is to monitor other users in the database, this may include SQL queries such as checking the number of users in the database or checking house numbers within the database.

Key concepts:

- *Inheritance*: A customer will extend from User.
- *Abstraction*: A customer does not need the complex functionality required to initiate database connection while making a payment.
- *Polymorphism*: There are different ways a customer may initiate database connections, depending on the payment service that they use.
- *Encapsulation*: The user fields are set to private, with public getters, given the need for the fields to be accessed by other methods.

Token Purchase via Payment Service of Choice

As discussed earlier, a customer, which is also a user, uses a payment service, which in turn uses a specific database to store payment data. It then follows that

based on the payment service that a user will use to make payments; it then stores payment data to that specific database.

We used Kenyan conversion rate of tokens as per June 13th, 2025. One token corresponds to around 20.57 Ksh. We implemented this in our system to prevent users for making payments of less than the amount, as it will not purchase 1 token (1Kwh).

How it works:

- A user can only make a payment so long as their input amount is not less than 20.57 Ksh. If they make a payment greater than the amount, then the amount is converted into tokens and the token system updates to reflect the new amount, based on the user input.
- Since making a payment requires database connection, the payment can only reflect if the payment service initiates a successful connection to its corresponding database. Otherwise, we display to the user that there is an error in connection, and they need to redo their payment.
- On successful payment, a token is generated, stored, and returned to the user.

Core concepts:

- *Polymorphism*: Many databases use many different payment services; therefore the means of connection may slightly vary due to these differences.
- *Encapsulation*: A user only makes payments and checks tokens, although database connection is required for the two processes, these details are captured within the two functions.

View Usage and Payment History

The customer is only allowed to view their individual usage, while the administrator has authority to view token data of all users within the database. This is where the separation of roles comes in.

For a customer, when viewing the database, they are restricted to only view their credentials, using their customer identification. Once again, database connection is instantiated during checking of payments. This time, the database connection is towards the repository of the specific payment service that a customer used to make payments.

From that, the customer can view their token usage, token balance, as well as the amount they paid.

How it works:

- Customer can check their token balance using the appropriate function.
- The function then initiates database connection according to the payment service that the customer used. To authenticate database connection, the user id and password is used.
- If the database connection is successful, token usage data and statistics is displayed, for the specific customer.

Administrator Monitoring

The purpose of the administrator is to monitor user payments, check the number of users in the database, and display information based on customer updates, as well as overseeing of the payment services that interact with the system.

The administrator can therefore check payment data of all users, payment history for all users, given the need, as well as information on payment services that users may have utilized during the payment process.

How it works:

- Admin initiates an SQL query that returns all users in the database.
- Admin can check payments made by specific users in the database, as well as the payment services that the users may have used. Again, database connection methods may vary depending on the type of payment service that a user wishes to choose.

Meter data syncing

When a payment is made, the amount should be converted into tokens and the tokens updated into the database immediately. This allows the administrator to receive accurate credentials when monitoring the system. Aside from that, users should be able to view their tokens accurately to make accurate decisions based on electricity consumption. This helps in sustainability.

How it works:

- A customer initiates database connection when checking tokens.
- Data is validated and displayed to the user (in most cases customer), giving a general idea on the electricity consumption.

Code implementation

Classes

The Quantum Grid system is built with modularity and maintainability in mind. There are four main classes. DatabaseConnection, PaymentService, User, Customer and Admin.

1. DatabaseConnection – It is primarily used for connecting to the database via the interface Connectable, which has single responsibility of initiating database connection by finding the port number, database name and password.
2. Payment Service - Also implements Connectable, therefore initiates database connection via this interface. A payment service typically has a name. An example can be MPESA, PayPal, or any payment service.
3. User - interacts with the database through making payments, checking tokens or monitoring other users in the database, as is the case of the Administrator. A user has attributes of name, email and password. When a user is saved to the database, additional attributes such as ID and house number are assigned are assigned.
4. Customer – A customer typically has the primary function of checking tokens and making payments, both of which database connection is involved. A customer uses a specific payment service and belongs to a certain house; therefore, these credentials is added to the database when saving the customer.
5. Admin – The admin class is used to check the number of users in a database, and monitor user activity in the whole database.

Interfaces

Aside from this, there are three interfaces: Connectable, VerifyPayments and TokenConversion.

1. Connectable – It is used to initiate database connection and is implemented by any class with which its usage is needed for establishing connections. The method in this interface is getConnection()- It throws an SQL exception.
2. VerifyPayments- It is used to ensure that the user has met the condition to buy tokens and returns true or false based on the conditions, truth value. The interface is used in the customer's payment making function and a Lambda expression is utilized, and the result is stored in a Boolean variable. This is because the function is designed to return a Boolean value as per the design in the interface.
3. TokenConversion – It is used to convert Kenyan shillings into tokens, and the function in the interface is designed to return a number that may have decimals (Double data type). We also utilize a lambda expression that returns

the number of tokens based on the user input and stores it in a variable in the payment making function.

General Functionality

The main method is located within the DatabaseConnection class, where we instantiate new customers or administrators. From this, instantiated customers can make payments and updates based on the conditions described above. We have also used a database simulator that generates random names, emails based on the random names and random house numbers which are unique, while utilising the Random Library.

Conclusion

Quantum Grid is a practical and academic system that blends real-world energy challenges with solid software engineering solutions. It demonstrates how modern technologies, when structured with good principles, can improve everyday life.

The system allows for:

- Secure and fast token delivery,
- Clear history tracking,
- Admin-level transparency,
- Extensibility depending on various types of users who interact with the system.

This document serves as both a technical and academic reference for anyone reviewing the project.