**Electricity Bill Management**

**Introduction**

In today’s digitally driven era, the management and payment of utility bills have become integral aspects of modern living. The conventional methods of bill payment often involve long queues, paperwork, and potential delays. To address these inefficiencies and provide users with a seamless experience, the Electricity Bill Booking Project has been developed.

This project leverages advanced web technologies such as JSP (JavaServer Pages), Servlets, MySQL, JDBC (Java Database Connectivity), HTML, and CSS to create an efficient and user-friendly platform for managing electricity bill payments. By integrating these technologies, the project aims to streamline the process of electricity bill booking, payment, and management.

The primary objectives of the Electricity Bill Booking Project are as follows

**Efficiency:** The project aims to automate the process of electricity bill booking and payment, reducing manual intervention and minimizing errors.

**Accessibility:** By providing a web-based platform, the project ensures that users can access the system from anywhere with an internet connection, facilitating convenient bill payment.

**User-Friendly Interface:** A clean and intuitive user interface has been designed using HTML and CSS to enhance the user experience and make bill payment effortless.

Security: Robust security measures have been implemented to safeguard user data and transactions, ensuring confidentiality and integrity.

**Database Management:** MySQL database is utilized to store and manage user information, bill details, and transaction records efficiently.

**Scalability:** The project architecture is designed to accommodate future enhancements and scalability requirements, ensuring adaptability to evolving needs.

This report provides a comprehensive overview of the Electricity Bill Booking Project, detailing its objectives, features, architecture, implementation methodology, and future scope. Through diligent analysis and implementation of the aforementioned technologies, this project endeavors to revolutionize the process of electricity bill management, offering users a seamless and efficient solution for their utility bill payment needs.

**Abstraction**

The "Electricity Bill Booking Project" is a web-based application designed to streamline the process of electricity bill booking, payment, and management. In today's digital age, managing utility bills efficiently is paramount, and this project addresses this need by leveraging advanced web technologies such as JSP (JavaServer Pages), Servlets, MySQL, JDBC (Java Database Connectivity), HTML, and CSS.

The primary objective of the project is to provide users with a seamless and user-friendly platform for booking and paying electricity bills online. The application offers features such as user registration, bill generation, payment processing, and transaction history tracking. It employs robust security measures to safeguard user data and transactions, ensuring confidentiality and integrity throughout the process.

The project abstracts complexities through encapsulation, modularity, data abstraction, interface abstraction, and procedural abstraction. By encapsulating functionalities into discrete units, defining clear interfaces between components, and abstracting data and procedural details, the project achieves a high level of maintainability, scalability, and reusability.

The implementation utilizes MySQL database for efficient storage and management of user information, bill details, and transaction records. JDBC is employed for database connectivity, enabling seamless interaction between the application and the database. HTML and CSS are utilized for presentation abstraction, ensuring a visually appealing and intuitive user interface.

Overall, the "Electricity Bill Booking Project" represents a significant step towards modernizing utility bill management systems. By providing a convenient, secure, and efficient platform for electricity bill booking and payment, the project aims to enhance user experience and streamline administrative processes in the realm of utility services.

**Functionality**

There are two main users or actors in this application. Let’s see on by one

* Admin
* Users

Admin is the main user here who will manage all the activity such as

* Admin can ADD/VIEW/UPDATE/DELETE users.
* Admin can VIEW its profile.
* Admin can Calculate Bill for the registered users.
* Admin can SEARCH Users.
* Admin can VIEW the Bill Report.
* Admin can VIEW the Payment History.
* Admin can UPDATE the cost of charge/unit.

Users is the another user who will Pay Bill using this application.

* User can VIEW/UPDATE its profile.
* User can VIEW the Bill.
* User can PAY the Bill.
* User can VIEW the Payment History.

**Hardware & Software Requirement:**

**Hardware Interfaces**

* Minimum Hardware requirement
* Processor: P4 3.0 GHz
* RAM:1 GB or Higher
* Monitor
* Mouse
* Hard disk: 80 GB

All these types of software automatic configure inside operating system after installation it having Java, MYSQL, Apache and operating system base configuration file, it doesn’t need to configure manually.

**Technologies Used**

The Online Supermarket Billing System utilizes a combination of technologies to create a robust and efficient platform for managing online supermarket operations. Here's a description of the key technologies used in the project:

**Java:** Java serves as the primary programming language for both backend and frontend development. Its platform independence, robustness, and extensive ecosystem make it an ideal choice for building scalable web applications.

**JSP (JavaServer Pages):** JSP is a technology used for creating dynamic web pages in Java. It allows embedding Java code directly into HTML pages, enabling the creation of dynamic content and server-side logic for generating web responses.

**Servlet:** A servlet is a Java class that extends the functionality of a web server, enabling the server to dynamically process and respond to client requests over the HTTP protocol. Servlets run on the server-side and handle tasks such as generating dynamic web content, interacting with databases, managing user sessions, and handling form submissions. They provide a powerful and flexible framework for building interactive web applications, offering platform independence, high performance, scalability, and seamless integration with other Java technologies.

**HTML:** HTML (Hypertext Markup Language) is the standard markup language for creating web pages and web applications. It provides the structure and layout for web content, defining elements such as headings, paragraphs, links, and forms.

**CSS (Cascading Style Sheets):** CSS is a stylesheet language used for styling HTML documents. It allows developers to control the appearance and layout of web pages, including aspects such as colors, fonts, margins, and positioning.

**Bootstrap:** Bootstrap is a popular front-end framework for building responsive and mobile-first web applications. It provides a set of pre-designed CSS and JavaScript components that streamline the process of creating modern and visually appealing user interfaces.

**MySQL:** MySQL is a widely-used open-source relational database management system (RDBMS). It offers robust features for managing structured data, including tables, indexes, stored procedures, and transactions, making it suitable for storing and retrieving data in web applications.

**MySQL Workbench:** MySQL Workbench is a visual database design and modeling tool that allows developers to design, visualize, and administer MySQL databases. It provides a user-friendly interface for creating and managing database schemas, tables, and relationships.

**Eclipse:** Eclipse is an integrated development environment (IDE) widely used for Java development. It offers a rich set of features, including code editing, debugging, and version control integration, making it a preferred choice for developing Java applications.

By leveraging these technologies, the Online Supermarket Billing System achieves a balance of efficiency, scalability, and user-friendliness, enabling seamless management of online supermarket operations while providing a satisfying shopping experience for customers.

**System Architecture**

System architecture, particularly within the context of web applications, often incorporates the Model-View-Controller (MVC) architectural pattern. MVC provides a structured approach to organizing code and separating concerns within an application, facilitating modularity, scalability, and maintainability.

Here's a breakdown of the MVC architecture:

**Model (M):**

The Model represents the application's data and business logic. It encapsulates data access, manipulation, and validation operations.

In a typical MVC setup, models are responsible for querying and updating the database, processing business rules, and enforcing data integrity.

Models can be thought of as the "brains" of the application, managing the application's state and responding to requests from the controller or view.

**View (V):**

The View represents the presentation layer of the application. It is responsible for rendering user interfaces and presenting data to the user.

Views typically consist of HTML templates, CSS stylesheets, and client-side scripts (e.g., JavaScript) for dynamic interactions.

Views are passive components that receive data from the controller or model and present it to the user in a visually appealing and interactive format.

**Controller (C):**

The Controller acts as an intermediary between the Model and the View, handling user input, processing requests, and coordinating interactions between the Model and the View.

Controllers receive input from the user via the View, invoke appropriate actions on the Model based on the user's interactions, and update the View with the results.

Controllers encapsulate application logic related to request handling, routing, and business process orchestration, keeping the Model and View decoupled and independent.

Key characteristics of the MVC architecture:

**Separation of Concerns:** MVC separates the application's concerns into distinct components (Model, View, Controller), promoting code organization, reusability, and testability.

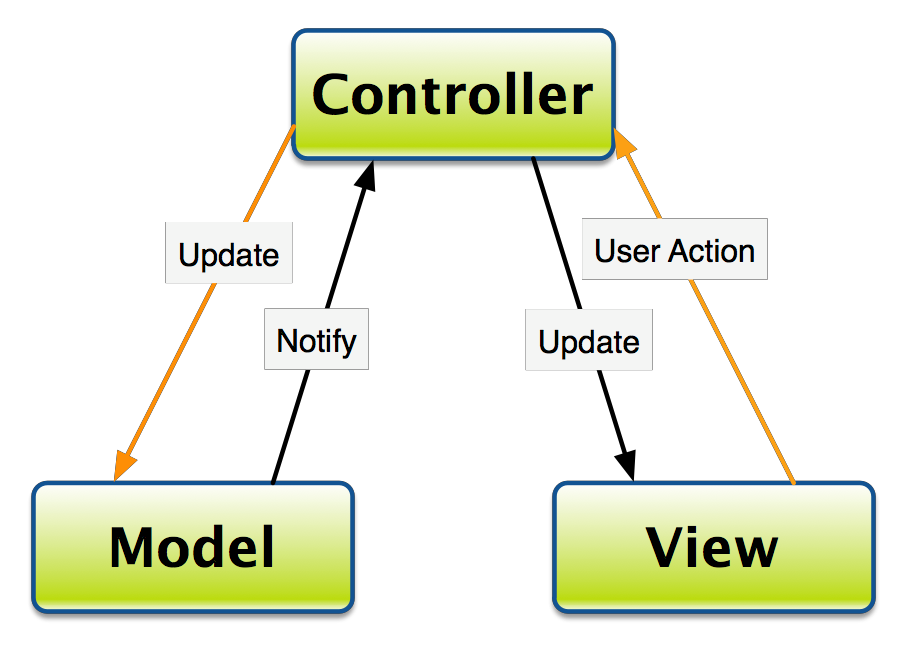
**Modularity:** MVC facilitates modularity by dividing the application into interchangeable components that can be developed, tested, and maintained independently.

**Scalability:** MVC supports scalability by allowing developers to add or modify components (e.g., controllers, views, models) without affecting other parts of the application, enabling the system to evolve over time.

**Maintainability:** MVC promotes maintainability by isolating changes within specific components, making it easier to identify, understand, and update code without impacting the overall system.

**Flexibility:** MVC provides flexibility in choosing technologies and frameworks for implementing each component, allowing developers to leverage the best tools for their specific requirements.

In summary, the MVC architecture provides a structured and flexible approach to designing web applications, emphasizing the separation of concerns and facilitating code organization, modularity, and maintainability. By dividing the application into three distinct layers (Model, View, Controller), MVC enables developers to build robust and scalable software solutions that meet the needs of users and businesses effectively.



**Development Process**

The development process of the Online Supermarket Billing System was a systematic and collaborative effort aimed at transforming design specifications into a functional software solution. It involved several key stages, each contributing to the creation of a robust and user-friendly application.

Starting with requirements gathering and analysis, the team meticulously documented the needs and expectations of stakeholders to ensure that the system would meet user requirements. Database design played a crucial role in structuring data storage and management efficiently, laying the groundwork for the rest of the development process.

During the development phase, the team implemented the business logic, user interface components, and data access layer using Java, JSP and Servlet , HTML, CSS, Bootstrap, and other technologies. Coding standards and best practices were followed to ensure readability, maintainability, and scalability of the codebase.

Testing was an integral part of the development process, with both unit testing and integration testing being conducted to validate the functionality, performance, and reliability of the system. Manual testing played a vital role in identifying and addressing issues, ensuring a high-quality user experience.

Throughout the development lifecycle, collaboration, communication, and feedback were emphasized, enabling the team to adapt to changing requirements and deliver value incrementally. Continuous integration and deployment practices facilitated the automated build, test, and deployment process, streamlining the development workflow.

In conclusion, the development process of the Online Supermarket Billing System exemplified the principles of modern software engineering, leveraging technology, collaboration, and best practices to deliver a robust and user-friendly solution. By embracing iterative development, testing, and continuous improvement, the team successfully translated design concepts into a functional and scalable application that meets the needs of stakeholders and end-users.

**Requirements gathering and analysis:**

Requirements gathering and analysis is a crucial phase in the software development lifecycle, as it sets the foundation for building a successful system that meets the needs of stakeholders and end-users. This phase involves gathering, documenting, analyzing, and prioritizing the requirements of the project. Here's a detailed overview of each step:

* Conduct interviews, workshops, or surveys with stakeholders to gather requirements.
* Document functional requirements, such as user roles, features, and workflows.
* Define non-functional requirements, including performance, security, and scalability criteria.
* Prioritize requirements and create a requirements specification document.
* Review and validate requirements with stakeholders to ensure alignment with business objectives.

**Identify Stakeholders:** Begin by identifying all stakeholders involved in the project. Stakeholders may include end-users, customers, project sponsors, business owners, subject matter experts, and other relevant parties. Each stakeholder may have unique perspectives, requirements, and expectations for the system.

**Conduct Stakeholder Interviews and Workshops:** Engage with stakeholders through interviews, workshops, surveys, or focus groups to gather insights into their needs, goals, and pain points. Use open-ended questions to encourage stakeholders to articulate their requirements and preferences. Record all feedback and insights gathered during these sessions.

**Elicit Requirements:** Use various techniques to elicit requirements from stakeholders, such as:

* **Requirements workshops:** Bring together stakeholders to collaborate on defining requirements.
* **Brainstorming sessions:** Encourage stakeholders to generate ideas and requirements collectively.
* **Prototyping:** Create mockups or prototypes to visualize and validate requirements.
* Use case analysis: Identify and document specific use cases to understand how users will interact with the system.
* **User stories:** Capture requirements from the perspective of end-users in the form of user stories.
* **Surveys and questionnaires:** Collect feedback from a larger audience to validate and prioritize requirements.

**Document Requirements:** Document all gathered requirements in a structured format, such as a Requirements Specification Document (RSD) or a Product Backlog. Include both functional requirements (what the system should do) and non-functional requirements (qualities or constraints of the system, such as performance, security, usability, etc.). Use clear, concise language and provide examples or use cases to illustrate each requirement.

**Analyze Requirements:** Analyze the gathered requirements to ensure they are complete, consistent, unambiguous, and feasible. Look for dependencies and conflicts between requirements and resolve them collaboratively with stakeholders. Prioritize requirements based on their importance to the project goals, business value, and constraints.

**Validate Requirements:** Validate the requirements with stakeholders to ensure they accurately reflect their needs and expectations. Use techniques such as reviews, walkthroughs, and prototypes to gather feedback and make necessary revisions. Continuous validation and refinement of requirements help minimize the risk of misunderstandings and scope creep.

**Manage Requirements Changes:** Establish a process for managing changes to requirements throughout the project lifecycle. Use a version control system to track changes and updates to the requirements documentation. Communicate changes effectively to all stakeholders and assess their impact on project scope, schedule, and resources.

**Obtain Stakeholder Sign-off:** Seek formal approval or sign-off from key stakeholders once the requirements are finalized and agreed upon. This indicates their commitment to the documented requirements and provides a baseline for subsequent phases of the project.

By following a systematic approach to requirements gathering and analysis, software development teams can ensure that the resulting system aligns with stakeholders' needs and expectations, leading to a successful outcome for the project. Effective communication, collaboration, and documentation are essential throughout this phase to capture and validate requirements accurately.

**Database design**

Database design is a critical aspect of software development, as it lays the foundation for storing, managing, and retrieving data efficiently and accurately within an application. Effective database design ensures data integrity, performance, scalability, and flexibility. Here's a detailed overview of the database design process:

**Requirements Analysis:**

Begin by understanding the requirements of the application and the data it needs to store and manipulate. Identify the entities (objects or concepts) within the domain of the application and the relationships between them.

Conduct interviews and discussions with stakeholders to gather insights into the data requirements, business rules, and constraints.

Analyze existing documents, such as business requirements documents, use cases, and user stories, to extract relevant data requirements.

**Conceptual Database Design:**

Create a conceptual data model that represents the high-level structure of the database, independent of any specific database management system (DBMS).

Use conceptual modeling techniques, such as Entity-Relationship Diagrams (ERDs) or Unified Modeling Language (UML) diagrams, to visualize the entities, attributes, and relationships in the domain.

Define entity types, their attributes, and the relationships between entities. This stage focuses on understanding the semantics of the data without considering implementation details.

**Logical Database Design:**

Translate the conceptual data model into a logical data model that can be implemented in a specific DBMS.

Choose an appropriate data model, such as relational, document-oriented, or graph-based, based on the requirements of the application and the characteristics of the data.

Design the tables (for relational databases) or collections (for NoSQL databases) to represent the entities and relationships identified in the conceptual model.

Define the primary keys, foreign keys, indexes, constraints, and data types for each table or collection.

Normalize the data model to eliminate redundancy and ensure data integrity. Use normalization techniques, such as First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF), to reduce data redundancy and dependency.

**Physical Database Design:**

Translate the logical data model into a physical schema that can be implemented in the chosen DBMS.

Decide on storage structures, such as tables, indexes, partitions, and tablespaces, to optimize data storage and access.

Define data partitioning and clustering strategies to improve performance and scalability.

Consider factors such as data volume, access patterns, concurrency, and security requirements when designing the physical schema.

Optimize the database design for performance by denormalizing tables, creating appropriate indexes, and partitioning data as needed.

**Data Integrity and Constraints:**

Enforce data integrity by defining constraints and rules that govern the validity of data in the database.

Define primary key constraints to ensure uniqueness of records, foreign key constraints to enforce referential integrity between tables, and other constraints such as NOT NULL, UNIQUE, CHECK, and DEFAULT constraints.

Implement business rules and validation logic within the database using triggers, stored procedures, or constraints to maintain data consistency and integrity.

**Data Security and Access Control:**

Implement security measures to protect sensitive data and ensure that only authorized users have access to the database.

Define user roles and privileges to restrict access to specific tables, views, or operations based on the principle of least privilege.

Implement encryption, authentication, and auditing mechanisms to safeguard data privacy and prevent unauthorized access or tampering.

**Database Maintenance and Optimization:**

Plan for ongoing database maintenance tasks, such as backup and recovery, database tuning, and performance monitoring.

Monitor database performance metrics, such as query execution times, resource utilization, and throughput, to identify bottlenecks and optimize performance.

Implement indexing strategies, query optimization techniques, and database caching to improve query performance and response times.

Regularly review and optimize the database schema, data storage structures, and indexing strategies to ensure optimal performance and scalability as the application evolves.

**Documentation and Communication:**

Document the database design, including the conceptual, logical, and physical models, as well as any design decisions, assumptions, and trade-offs made during the process.

Communicate the database design to stakeholders, developers, and other members of the project team to ensure a common understanding of the data model and its implications.

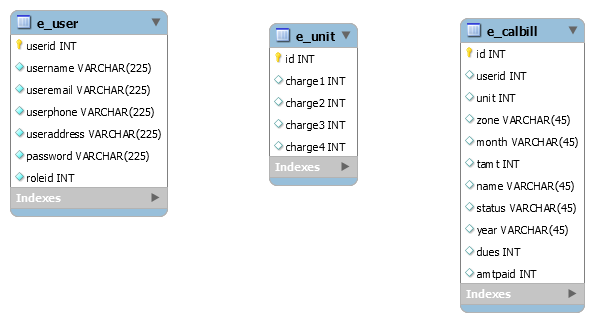
Provide documentation and training materials to support database administrators, developers, and users in understanding and working with the database effectively.

By following a structured approach to database design, software development teams can create well-designed databases that meet the requirements of the application, optimize performance, ensure data integrity, and support future growth and evolution. Effective database design is essential for building robust and scalable software systems that efficiently manage and leverage data to drive business value.

* Analyze the application's data requirements based on the gathered requirements.
* Identify entities, attributes, and relationships to model the data effectively.
* Create an Entity-Relationship Diagram (ERD) using tools like Lucidchart or draw.io.
* Normalize the database schema to eliminate redundancy and improve data integrity.
* Define database constraints, such as primary keys, foreign keys, and unique constraints.
* Consider indexing strategies to optimize query performance for frequently accessed data.

**Entity Relationship Diagram**

An Entity-Relationship Diagram (ERD) is a graphical representation used in database design to visualize the relationships between entities in a domain and the attributes associated with those entities. ERDs are an essential tool for understanding and communicating the structure of a database schema.

****

Here's a breakdown of key components and concepts of an ERD:

**Entities:**

* An entity represents a real-world object, concept, or thing with distinct properties.
* In an ERD, entities are typically represented as rectangles or boxes.
* Each entity has a name that describes the category of objects it represents (e.g., "Customer," "Product," "Order").

**Attributes:**

* Attributes are the properties or characteristics of entities that describe them.
* Each attribute is associated with an entity and represents a specific piece of information about the entity.
* Attributes are represented as ovals or ellipses connected to their respective entities by lines.

**Relationships:**

* Relationships define associations and connections between entities.
* A relationship describes how entities are related to each other and can have various cardinality constraints, such as one-to-one, one-to-many, or many-to-many.
* Relationships are represented as lines connecting related entities, with optional symbols indicating cardinality and participation constraints.

**Cardinality:**

* Cardinality specifies the number of instances of one entity that are associated with the number of instances of another entity through a relationship.
* Cardinality constraints are often indicated using symbols such as "1" (one), "M" (many), "0" (zero), "N" (any number), or specific ranges (e.g., "1..\*" for one or more).
* Cardinality constraints help define the nature of the relationship between entities and influence database schema design and query formulation.

**Keys:**

* Keys are attributes or combinations of attributes that uniquely identify instances of an entity within a database.
* Primary keys are special attributes designated to uniquely identify each instance of an entity, and they play a crucial role in maintaining data integrity and enforcing constraints.
* Foreign keys are attributes that establish relationships between entities by referencing the primary key of another entity.

**Associative Entities:**

* Associative entities, also known as junction entities or relationship entities, represent entities that connect other entities in a many-to-many relationship.
* Associative entities typically contain attributes that describe the relationship between the connected entities.
* They are represented similarly to regular entities in an ERD but are connected by relationships indicating their association with other entities.

Overall, Entity-Relationship Diagrams provide a visual representation of the structure and relationships within a database schema, helping stakeholders, designers, and developers to understand, communicate, and design database systems effectively. ERDs serve as a blueprint for database implementation and play a crucial role in database design and modeling processes.

**System Design**

System design, also known as architectural design, is the process of defining the architecture, components, modules, interfaces, and data for a software system to meet specified requirements. It involves translating the requirements gathered during the analysis phase into a blueprint that guides the implementation of the system. Here's a detailed overview of the system design process:

* Architect the system using the Model-View-Controller (MVC) pattern to separate concerns and improve maintainability.
* Create a high-level architecture diagram illustrating the components and their interactions.
* Develop detailed component diagrams or class diagrams to specify the structure and behavior of each component.
* Define the data flow between components using Data Flow Diagrams (DFDs) or sequence diagrams.
* Choose appropriate design patterns and architectural styles to address specific requirements, such as dependency injection for managing dependencies and inversion of control for decoupling components.

**Understand Requirements:**

* Begin by thoroughly understanding the functional and non-functional requirements of the system.
* Analyze use cases, user stories, and other requirements documentation to identify the system's features, constraints, and goals.
* Clarify any ambiguities and resolve conflicting requirements through discussions with stakeholders.

**Define System Architecture:**

* Choose an appropriate architectural style or pattern based on the requirements, such as layered architecture, client-server architecture, microservices architecture, or event-driven architecture.
* Define the high-level structure of the system, including the major components, layers, subsystems, and their interactions.
* Determine how components will communicate with each other, including protocols, data formats, and APIs.

**Design Data Model:**

* Design the data model based on the requirements gathered during the analysis phase.
* Identify entities, attributes, relationships, and constraints using techniques such as Entity-Relationship Diagrams (ERDs) or UML class diagrams.
* Normalize the data model to eliminate redundancy and ensure data integrity, applying normalization techniques such as First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

**Define Component Interfaces:**

* Specify the interfaces between different components, modules, or subsystems of the system.
* Define input and output parameters, data formats, method signatures, and communication protocols for each interface.
* Ensure that interfaces are well-defined, clear, and consistent to facilitate integration and communication between components.

**Design Module Structure:**

* Decompose the system into smaller modules or units of functionality based on the system architecture and requirements.
* Identify cohesive and loosely coupled modules that encapsulate related functionality and minimize dependencies between modules.
* Define the responsibilities and interfaces of each module, specifying how they interact with other modules and the external environment.

**Specify System Behavior:**

* Define the behavior of the system in response to different inputs, events, and scenarios.
* Specify the business logic, algorithms, workflows, state transitions, and error handling mechanisms of the system.
* Use techniques such as flowcharts, state diagrams, activity diagrams, or sequence diagrams to illustrate system behavior.

**Consider Non-Functional Requirements:**

* Address non-functional requirements such as performance, scalability, reliability, security, and usability during system design.
* Design architectural strategies and mechanisms to meet these requirements, such as caching, load balancing, encryption, authentication, and user interface design principles.

**Evaluate Design Decisions:**

* Review and validate the system design against the requirements, constraints, and quality attributes.
* Conduct design reviews, walkthroughs, or architectural reviews with stakeholders and subject matter experts to solicit feedback and identify potential issues.
* Refine and iterate on the design based on feedback and lessons learned from the evaluation process.

**Document Design Artifacts:**

* Document the system design artifacts, including architectural diagrams, data models, interface specifications, module specifications, and behavioral descriptions.
* Create design documents, architectural blueprints, and technical specifications to communicate the system design to developers, testers, and other stakeholders.
* Ensure that design documentation is comprehensive, well-organized, and accessible to support implementation, testing, and maintenance activities.

**Iterate and Refine:**

* System design is an iterative process, and it may require multiple iterations to refine and optimize the design based on feedback, changing requirements, and emerging constraints.
* Continuously evaluate and refine the design as the project progresses, incorporating new insights, addressing issues, and adapting to evolving needs.

By following a systematic approach to system design, software development teams can create well-structured, modular, and scalable systems that meet the functional and non-functional requirements of the project. Effective system design lays the foundation for successful implementation, testing, and deployment of software systems, enabling the delivery of high-quality and reliable solutions to users and stakeholders.

**Data FlowDiagram**

A Data Flow Diagram (DFD) is a graphical representation that depicts the flow of data within a system, showing how data moves from one process to another and how it is stored, processed, and transformed along the way. DFDs are commonly used in software engineering and systems analysis to visualize the structure and behavior of information systems. Here's a detailed explanation of the key components and concepts of a DFD:

**Processes:**

* Processes represent activities or functions that manipulate data within the system.
* Each process performs a specific task or operation on the input data to produce output data.
* Processes are depicted as circles or rectangles in a DFD diagram, labeled with a descriptive name or identifier.
* Examples of processes include data transformation, computation, validation, storage, retrieval, and transmission.

**Data Flows:**

* Data flows represent the movement of data between processes, data stores, and external entities within the system.
* Data flows are depicted as arrows in a DFD diagram, indicating the direction of data flow.
* Each data flow is labeled with a meaningful name that describes the type or content of the data being transmitted.
* Data flows may represent inputs, outputs, or intermediate data exchanged between processes and external entities.

**Data Stores:**

* Data stores represent repositories or storage locations where data is persistently stored within the system.
* Data stores are depicted as rectangles with two parallel lines on one side in a DFD diagram.
* Each data store is labeled with a descriptive name that identifies the type or purpose of the stored data.
* Data stores may include databases, files, tables, queues, or any other storage medium used by the system.

**External Entities:**

* External entities represent sources or destinations of data that interact with the system but are external to it.
* External entities can be users, devices, systems, or other entities that exchange data with the system.
* External entities are depicted as squares or rectangles with rounded corners in a DFD diagram.
* Each external entity is labeled with a descriptive name that identifies its role or function in the system.

**Data Flow Paths:**

* Data flow paths represent the paths along which data flows through the system from its source to its destination.
* Data flow paths are traced by following the arrows that connect processes, data stores, and external entities in the DFD diagram.
* Data flow paths illustrate the sequence of operations performed on the data as it moves through the system, showing how data is processed and transformed.

**Levels of Detail:**

* DFDs can be organized into multiple levels of detail to represent different perspectives or views of the system.
* Level 0 DFD represents the highest level of abstraction, showing the overall structure and major processes of the system.
* Lower-level DFDs provide more detailed views of specific processes or subsystems, breaking down complex processes into smaller components.

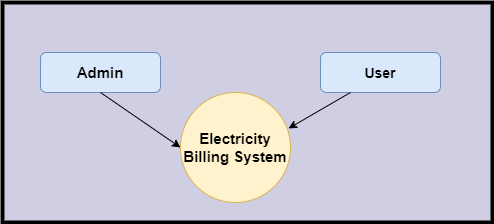
**Context Diagram:**

* A context diagram is a high-level DFD that provides an overview of the system and its interactions with external entities.
* The context diagram shows the external entities that interact with the system and the data flows between them, without going into detail about internal processes or data stores.
* It serves as a starting point for understanding the scope and boundaries of the system and its external interfaces.

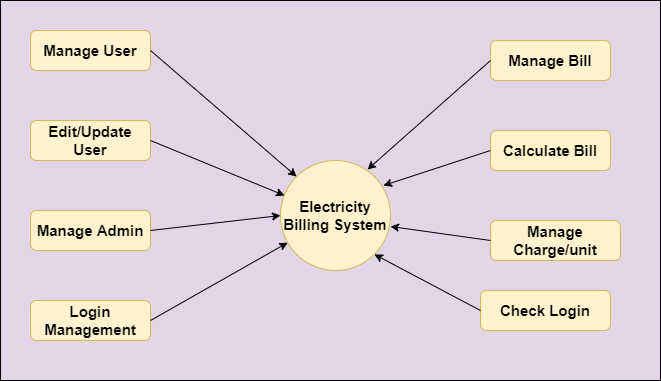
**DFD Diagrams**

**Data Flow Diagram (DFD)**

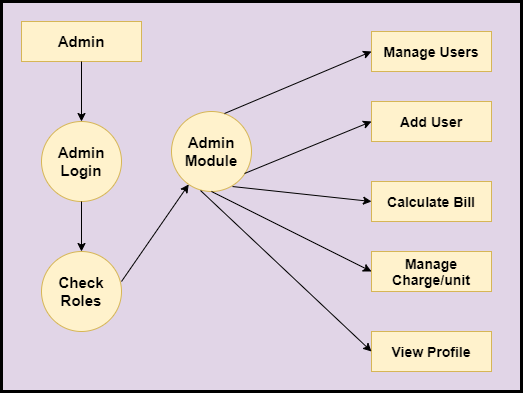
**Level 0:**

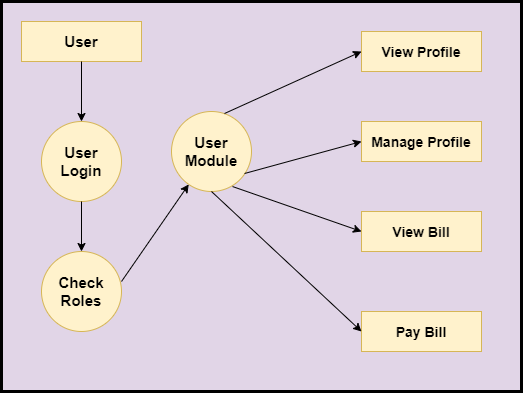
****

**Level 1**

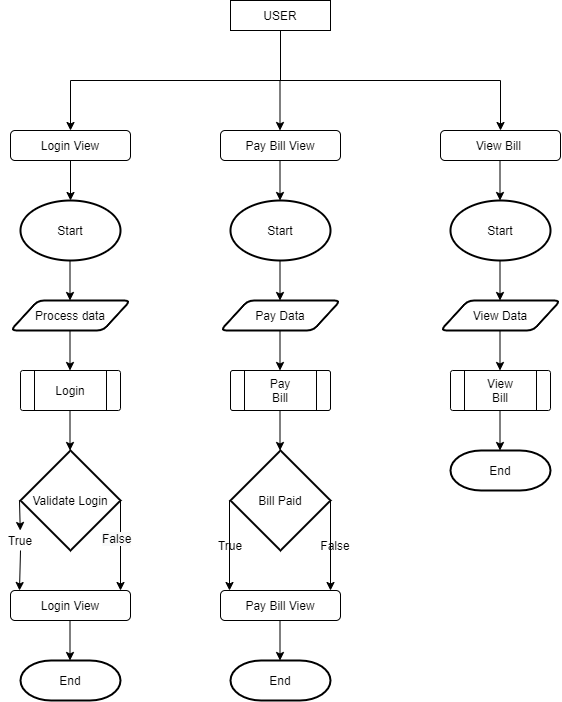
****

**Level 2: Admin**

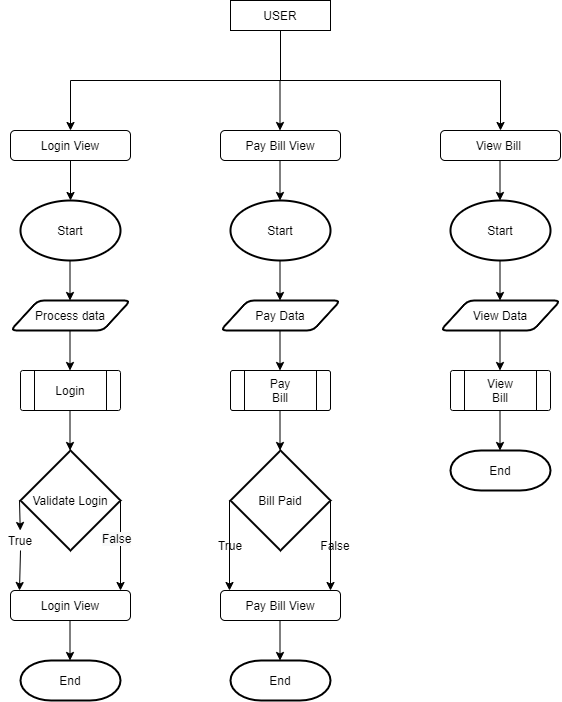
****

**Level 2: User**

**Functional DFD:**

****

**System Flow Chart:**

****

Overall, Data Flow Diagrams provide a visual representation of how data moves through a system, helping stakeholders, designers, and developers to understand, analyze, and communicate the structure and behavior of information systems effectively. DFDs are valuable tools for requirements analysis, system design, and documentation in software engineering and systems analysis.

**Development**

The development phase of a software project is where the design specifications are translated into actual code, leading to the creation of the software product. This phase involves writing, testing, and debugging code, as well as integrating various components to build a functional system. Here's a detailed overview of the development process.

* Utilize Object-Relational Mapping (ORM) frameworks like Hibernate to simplify database interactions.
* Design the View layer using HTML, CSS, and frontend frameworks such as Bootstrap or Materialize for responsive and visually appealing user interfaces.
* Develop controller classes to handle HTTP requests, route them to appropriate actions, and interact with the Model layer.
* Use version control systems like Git for collaborative development and code management.
* Follow coding standards, conventions, and best practices to ensure consistency and maintainability.
* Implement the Model layer to represent the application's data model and business logic.

**Select Development Methodology:**

* Choose an appropriate development methodology based on the project's requirements, team size, and organizational culture.
* Common methodologies include Waterfall, Agile, Scrum, Kanban, and DevOps, each offering different approaches to managing the development process.

**Setup Development Environment:**

* Set up the development environment with the necessary tools, libraries, frameworks, and resources required for coding.
* Install and configure development tools such as Integrated Development Environments (IDEs), version control systems (e.g., Git), build automation tools (e.g., Maven, Gradle), and testing frameworks.

**Write Code:**

* Developers write code according to the specifications and design documents created during the previous phases.
* Use appropriate programming languages, frameworks, and best practices to implement the desired functionality.
* Follow coding standards, naming conventions, and design patterns to ensure consistency and maintainability of the codebase.

**Implement Business Logic:**

* Implement the business logic of the application, including algorithms, calculations, workflows, and decision-making processes.
* Write code to handle user interactions, process input data, and generate output responses according to the requirements.

**Develop User Interface (UI):**

* Design and develop the user interface (UI) components, including screens, forms, menus, buttons, and widgets.
* Use frontend technologies such as HTML, CSS, JavaScript, and frontend frameworks to create interactive and responsive user interfaces.

**Integrate Components:**

* Integrate different modules, components, and libraries to build a cohesive and functional system.
* Ensure that components communicate effectively with each other and adhere to the defined interfaces and protocols.

**Implement Data Access Layer:**

* Develop the data access layer to interact with the database or external data sources.
* Write code to perform CRUD (Create, Read, Update, Delete) operations, query data, and handle transactions.
* Use Object-Relational Mapping (ORM) frameworks (e.g., Hibernate, Entity Framework) or data access libraries to abstract database interactions and improve productivity.

**Write Unit Tests:**

* Write unit tests to validate the functionality of individual units or components of the software.
* Use testing frameworks (e.g., JUnit, NUnit) and mocking libraries to simulate dependencies and isolate units for testing.
* Write test cases to cover different scenarios, edge cases, and error conditions to ensure robustness and reliability of the code.

**Perform Integration Testing:**

* Conduct integration testing to verify that different components work together as expected.
* Test the interactions between modules, APIs, and external dependencies to identify and resolve integration issues.
* Use techniques such as black-box testing, white-box testing, and end-to-end testing to validate system behavior across various integration points.

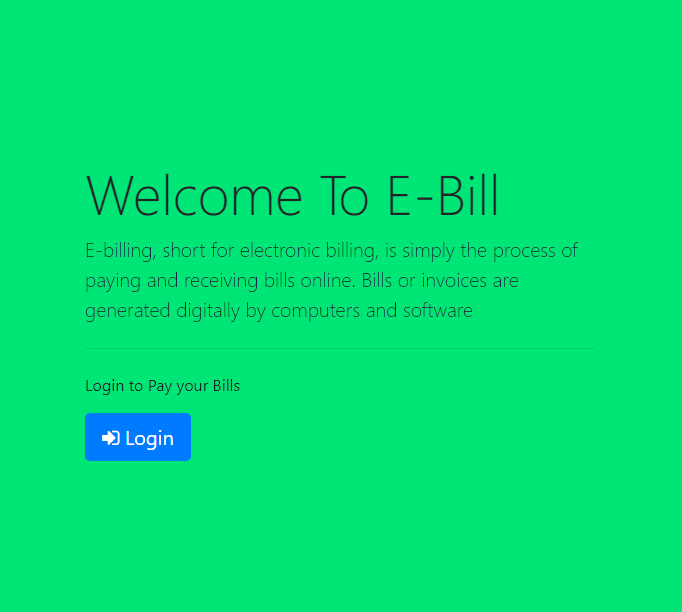
**Debug and Refactor Code:**

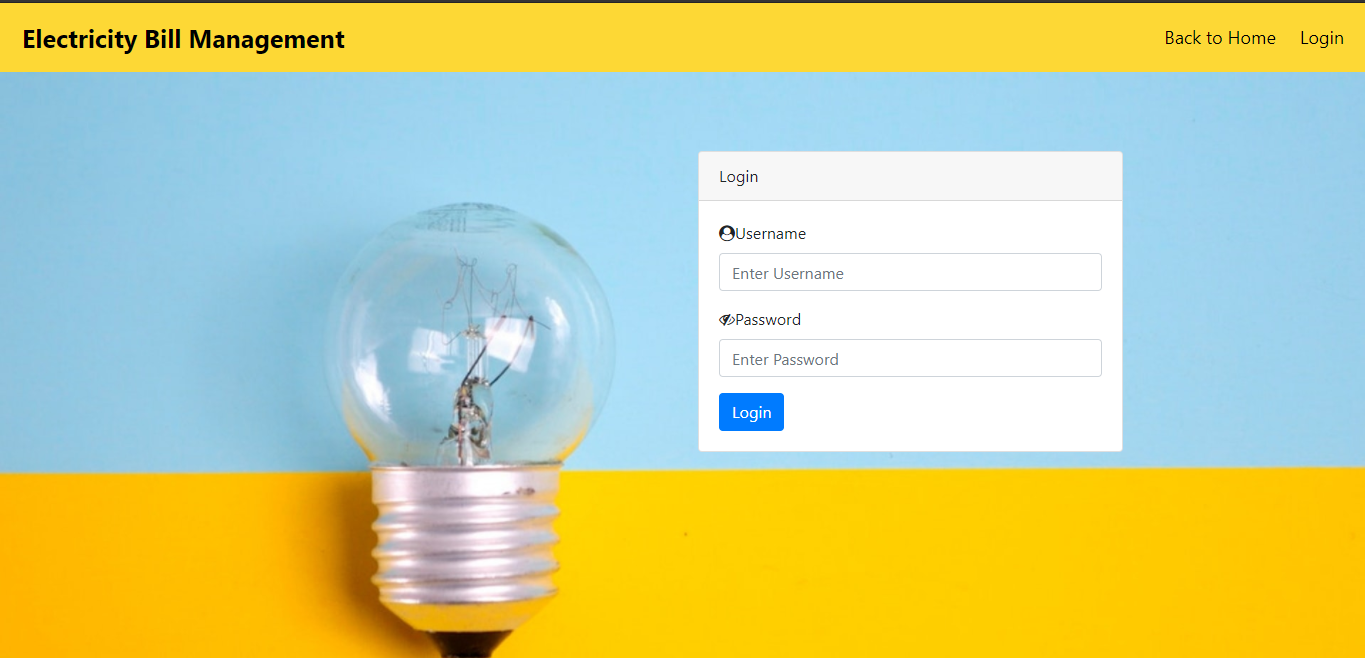
* Debug code to identify and fix defects, errors, and anomalies in the software.
* Use debugging tools, logging frameworks, and diagnostic utilities to trace and troubleshoot issues.
* Refactor code to improve readability, performance, and maintainability while preserving the existing functionality.

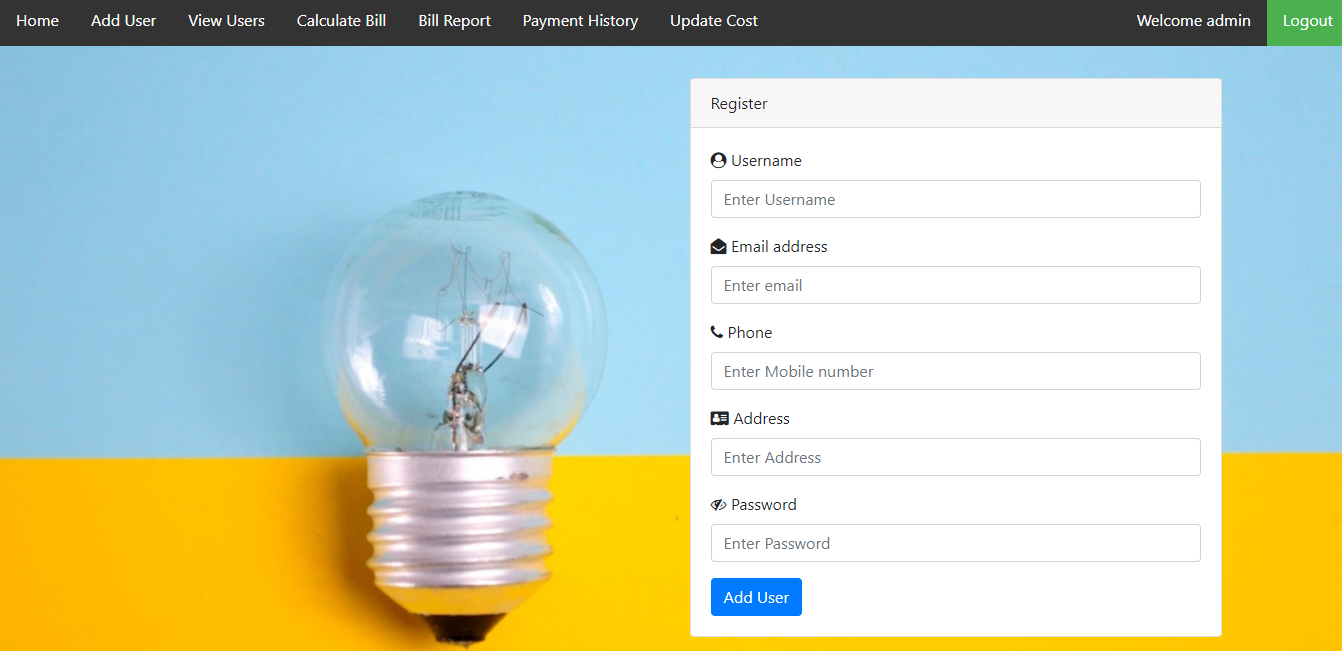
**Optimize Performance:**

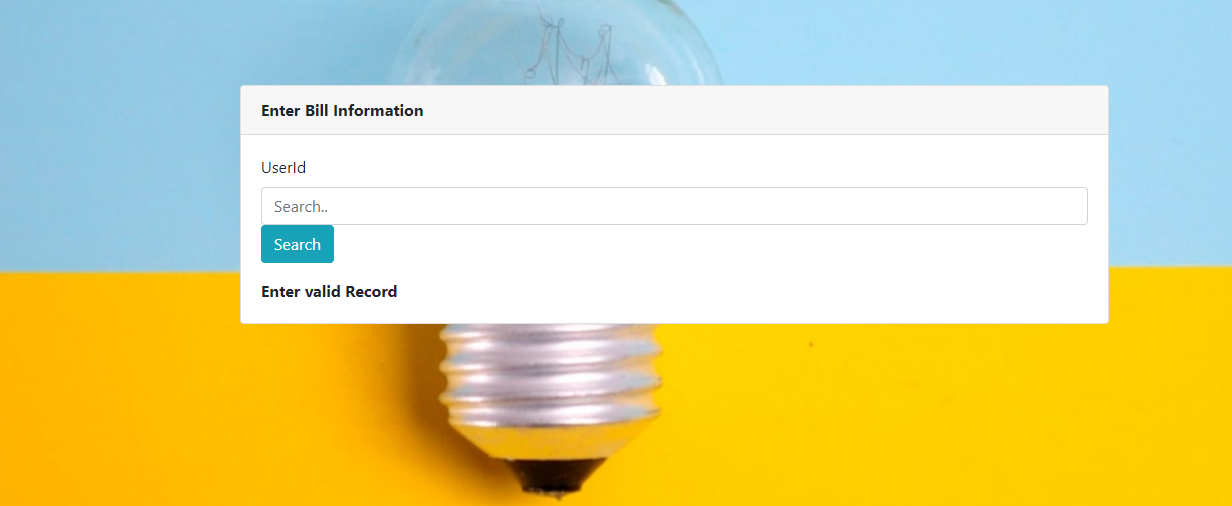
* Identify and address performance bottlenecks, inefficiencies, and resource constraints in the code.
* Use profiling tools to analyze code execution, memory usage, and I/O operations to identify areas for optimization.
* Optimize algorithms, data structures, and database queries to improve the overall performance and responsiveness of the system.

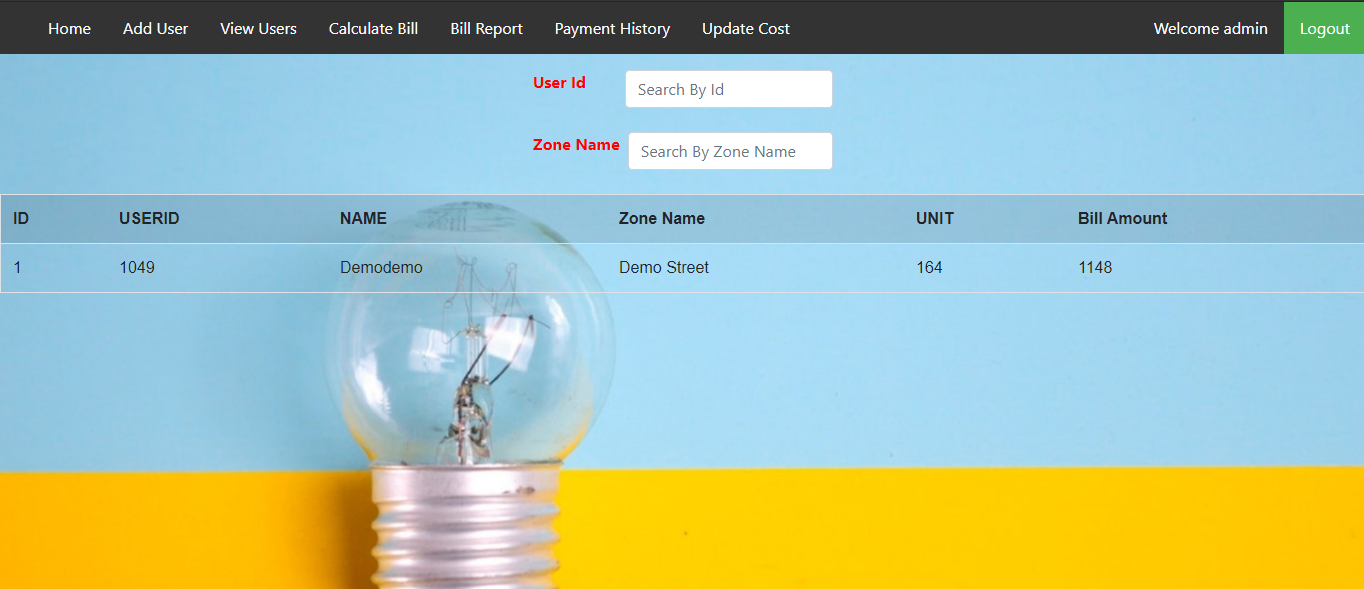
**Screen shot**

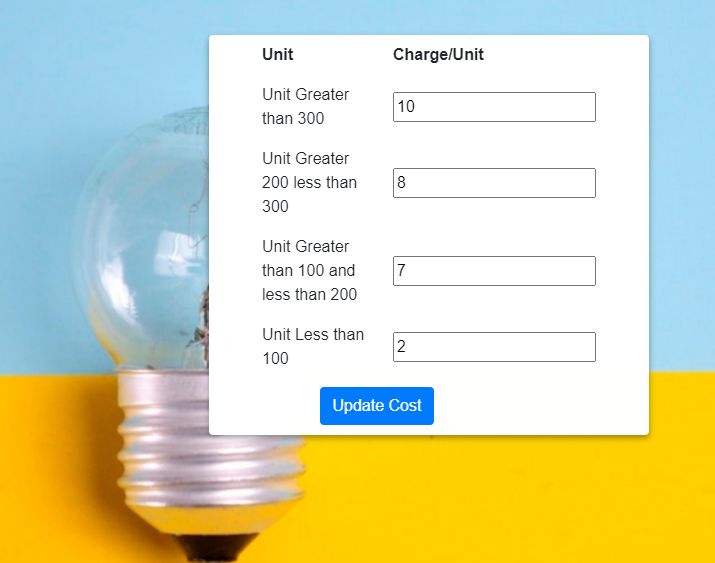


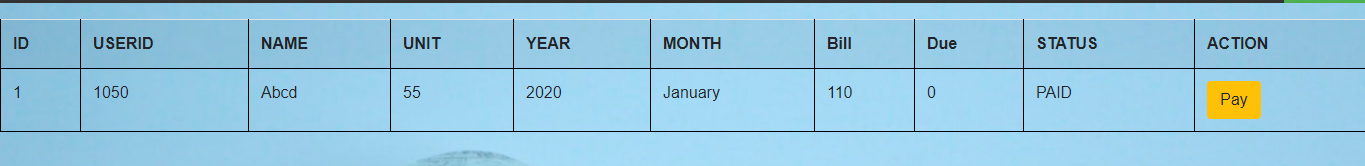












**Source code**

package com.ebilling.DBConnection;

import java.sql.\*;

public class DBConnect {

public static Connection getConn() {

Connection con = null;

String loadDriver = "com.mysql.cj.jdbc.Driver"; // driver name for mysql

String dbURL = "jdbc:mysql://localhost:3306/ebill"; // url of the

// database

String dbUSERNAME = "root"; // username to coonect db

String dbPASSWORD = "khan"; // password to connect db

try {

Class.*forName*(loadDriver); // load the driver

con = DriverManager.*getConnection*(dbURL, dbUSERNAME, dbPASSWORD);// get

// the

// connection

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return con; // return the connection obj.

}

}

Bean Classes

User Bean

package com.ebilling.bean;

public class UserBean {

private int userid;

private String username;

private String useremail;

private String userphone;

private String useraddress;

private String password;

private int roleid;

public int getUserid() {

return userid;

}

public void setUserid(int userid) {

this.userid = userid;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getUseremail() {

return useremail;

}

public void setUseremail(String useremail) {

this.useremail = useremail;

}

public String getUserphone() {

return userphone;

}

public void setUserphone(String userphone) {

this.userphone = userphone;

}

public String getUseraddress() {

return useraddress;

}

public void setUseraddress(String useraddress) {

this.useraddress = useraddress;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public int getRoleid() {

return roleid;

}

public void setRoleid(int roleid) {

this.roleid = roleid;

}

}

Cost Bean

package com.ebilling.bean;

public class CostBean {

private int id;

private int cost1;

private int cost2;

private int cost3;

private int cost4;

public int getCost4() {

return cost4;

}

public void setCost4(int cost4) {

this.cost4 = cost4;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public int getCost1() {

return cost1;

}

public void setCost1(int cost1) {

this.cost1 = cost1;

}

public int getCost2() {

return cost2;

}

public void setCost2(int cost2) {

this.cost2 = cost2;

}

public int getCost3() {

return cost3;

}

public void setCost3(int cost3) {

this.cost3 = cost3;

}

}

**CalcBill Bean**

package com.ebilling.bean;

public class CalcBill {

private int cid;

private int userid;

private int unitConsumed;

private String zoneName;

private String month;

private String name;

private String year;

private String status;

private int dues;

private int payamt;

public int getPayamt() {

return payamt;

}

public void setPayamt(int payamt) {

this.payamt = payamt;

}

public int getDues() {

return dues;

}

public void setDues(int dues) {

this.dues = dues;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

public String getYear() {

return year;

}

public void setYear(String year) {

this.year = year;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getCid() {

return cid;

}

public void setCid(int cid) {

this.cid = cid;

}

public int getUserid() {

return userid;

}

public void setUserid(int userid) {

this.userid = userid;

}

public int getUnitConsumed() {

return unitConsumed;

}

public void setUnitConsumed(int unitConsumed) {

this.unitConsumed = unitConsumed;

}

public String getZoneName() {

return zoneName;

}

public void setZoneName(String zoneName) {

this.zoneName = zoneName;

}

public String getMonth() {

return month;

}

public void setMonth(String month) {

this.month = month;

}

}

**Model Classes**

**LoginDAO**

package com.ebilling.Model;

import java.sql.\*;

import com.ebilling.DBConnection.DBConnect;

import com.ebilling.bean.UserBean;

public class LoginDAO {

public String authenticateUser(UserBean bean) throws SQLException {

String username=bean.getUsername();

String pass=bean.getPassword();

System.out.println(username+pass);

Connection con=null;

Statement statement=null;

ResultSet resultSet=null;

String usernamedb="";

String passdb="";

int roledb;

con=DBConnect.getConn();

statement=con.createStatement();

resultSet=statement.executeQuery("SELECT \*FROM E\_USER WHERE USERNAME='"+username+"' AND PASSWORD='"+pass+"'");

while(resultSet.next())

{

usernamedb=resultSet.getString("username");

passdb=resultSet.getString("password");

roledb=resultSet.getInt("roleid");

System.out.println(usernamedb+passdb+roledb);

if(username.equals(usernamedb) && pass.equals(passdb) && roledb==1)

return "ADMIN\_ROLE";

else if (username.equals(usernamedb) && pass.equals(passdb) && roledb==2)

return "USER\_ROLE";

}

return "INVALID";}

}

RegDAO

package com.ebilling.Model;

import java.sql.\*;

import com.ebilling.DBConnection.DBConnect;

import com.ebilling.bean.UserBean;

public class RegDAO {

/\* Add User By Admin \*/

public int addUser(UserBean bean) throws SQLException {

Connection con = null;

con = DBConnect.getConn();

String sql = "INSERT INTO E\_USER (USERNAME,USEREMAIL,USERPHONE,USERADDRESS,PASSWORD,ROLEID) VALUES"

+ "(?,?,?,?,?,?)";

PreparedStatement preparedStatement = con.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

preparedStatement.setString(1, bean.getUsername());

preparedStatement.setString(2, bean.getUseremail());

preparedStatement.setString(3, bean.getUserphone());

preparedStatement.setString(4, bean.getUseraddress());

preparedStatement.setString(5, bean.getPassword());

preparedStatement.setInt(6, bean.getRoleid());

int i = preparedStatement.executeUpdate();

if (i == 0) {

System.out.println("not inserted");

} else {

System.out.println("inserted");

}

int id = 0;

ResultSet rs = preparedStatement.getGeneratedKeys();

if (rs.next()) {

id = rs.getInt(1);

}

System.out.println("Last inserted id" + id);

return id;

}

public static UserBean getRecordById(int id) throws SQLException {

UserBean bean = new UserBean();

Connection con = DBConnect.getConn();

String sql = "SELECT \* FROM E\_USER where USERID=?";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setInt(1, id);

ResultSet rs = preparedStatement.executeQuery();

while (rs.next()) {

bean.setUserid(rs.getInt(1));

bean.setUsername(rs.getString(2));

bean.setUseremail(rs.getString(3));

bean.setUserphone(rs.getString(4));

bean.setUseraddress(rs.getString(5));

bean.setPassword(rs.getString(6));

}

return bean;

}

public int editUser(UserBean userBean) throws SQLException {

// TODO Auto-generated method stub

Connection con = DBConnect.getConn();

int status = 0;

String sql = "UPDATE E\_USER SET USEREMAIL=?,USERADDRESS=?,USERPHONE=? WHERE USERID=?";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setString(1, userBean.getUseremail());

preparedStatement.setString(2, userBean.getUseraddress());

preparedStatement.setString(3, userBean.getUserphone());

preparedStatement.setInt(4, userBean.getUserid());

status = preparedStatement.executeUpdate();

System.out.println(status);

if (status != 0) {

System.out.println(status);

return 1;

} else {

return 0;

}

}

public int editMyProfile(UserBean userBean) throws SQLException {

// TODO Auto-generated method stub

Connection con = DBConnect.getConn();

int status = 0;

String sql = "UPDATE E\_USER SET USERNAME=?,USEREMAIL=?,USERADDRESS=?,USERPHONE=?,PASSWORD=? WHERE USERID=?";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setString(1, userBean.getUsername());

preparedStatement.setString(2, userBean.getUseremail());

preparedStatement.setString(3, userBean.getUseraddress());

preparedStatement.setString(4, userBean.getUserphone());

preparedStatement.setString(5, userBean.getPassword());

preparedStatement.setInt(6, userBean.getUserid());

status = preparedStatement.executeUpdate();

System.out.println(status);

if (status != 0) {

System.out.println(status);

return 1;

} else {

return 0;

}

}

}

CalcDAO

package com.ebilling.Model;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import com.ebilling.DBConnection.DBConnect;

import com.ebilling.bean.CalcBill;

import com.ebilling.bean.CostBean;

import com.ebilling.bean.UserBean;

public class CalcDAO {

public void calculateBill(CalcBill bill) throws SQLException {

// TODO Auto-generated method stub

int cost1, cost2, cost3, cost4;

Connection connection = DBConnect.getConn();

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM E\_UNIT ORDER BY ID DESC");

ResultSet resultSet = statement.executeQuery();

if (resultSet.next())

// while(resultSet.next())

{

cost1 = resultSet.getInt(2);

cost2 = resultSet.getInt(3);

cost3 = resultSet.getInt(4);

cost4 = resultSet.getInt(5);

System.out.println(cost1 + " " + cost2 + "" + cost3 + "+" + cost4);

int due = bill.getDues();

int unit = bill.getUnitConsumed();

int tamt = +due;

if (unit > 100) {

if (unit >= 200) {

if (unit > 300) {

tamt = (unit \* cost1) + due;

} else

tamt = (unit \* cost2) + due;

} else

tamt = (unit \* cost3) + due;

} else {

tamt = (unit \* cost4) + due;

}

System.out.println(tamt);

Connection con = DBConnect.getConn();

String sql = "INSERT INTO E\_CALBILL (USERID,NAME,UNIT,ZONE,MONTH,TAMT,YEAR,STATUS,DUES) VALUES (?,?,?,?,?,?,?,?,?) ";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setInt(1, bill.getUserid());

preparedStatement.setString(2, bill.getName());

preparedStatement.setInt(3, bill.getUnitConsumed());

preparedStatement.setString(4, bill.getZoneName());

preparedStatement.setString(5, bill.getMonth());

preparedStatement.setInt(6, tamt);

preparedStatement.setString(7, bill.getYear());

preparedStatement.setString(8, bill.getStatus());

preparedStatement.setInt(9, bill.getDues());

int result = preparedStatement.executeUpdate();

System.out.println(result);

}

// return result;

}

public CalcBill payAmount(CalcBill bill2) throws SQLException {

// TODO Auto-generated method stub

Connection con = DBConnect.getConn();

String sql = "UPDATE E\_CALBILL SET AMTPAID=?,STATUS=? WHERE ID=?";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setInt(1, bill2.getPayamt());

preparedStatement.setString(2, bill2.getStatus());

preparedStatement.setInt(3, bill2.getCid());

int result = preparedStatement.executeUpdate();

if (result != 0) {

System.out.println("valid");

}

return bill2;

}

public static CalcBill getRecordById(int id) throws SQLException {

CalcBill bean = new CalcBill();

Connection con = DBConnect.getConn();

String sql = "SELECT \* FROM E\_CALBILL where ID=?";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setInt(1, id);

ResultSet rs = preparedStatement.executeQuery();

while (rs.next()) {

bean.setCid(rs.getInt(1));

bean.setUserid(rs.getInt(2));

bean.setUnitConsumed(rs.getInt(3));

}

return bean;

}

public static CostBean updateCost(CostBean bean) throws SQLException {

Connection con = DBConnect.getConn();

String sql = "INSERT INTO E\_UNIT (CHARGE1,CHARGE2,CHARGE3,CHARGE4) VALUES (?,?,?,?)";

PreparedStatement preparedStatement = con.prepareStatement(sql);

preparedStatement.setInt(1, bean.getCost1());

preparedStatement.setInt(2, bean.getCost2());

preparedStatement.setInt(3, bean.getCost3());

preparedStatement.setInt(4, bean.getCost4());

int i = preparedStatement.executeUpdate();

if (i != 0) {

System.out.println("done");

}

return bean;

// TODO Auto-generated method stub

}

}

Controller Classes

package com.ebilling.Ctrl;

import java.io.IOException;

import java.sql.SQLException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

import com.ebilling.Model.CalcDAO;

import com.ebilling.bean.CalcBill;

import com.ebilling.bean.CostBean;

/\*\*

\* Servlet implementation class CalcBillCtrl

\*/

@WebServlet("/CalcBillCtrl")

public class CalcBillCtrl extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#HttpServlet()

\*/

public CalcBillCtrl() {

super();

// TODO Auto-generated constructor stub

}

/\*\*

\* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

request.getRequestDispatcher("CalculateBillView.jsp").forward(request, response);

request.getRequestDispatcher("paybill.jsp").forward(request, response);

}

/\*\*

\* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

if (request.getParameter("Action").equals("Calculate Bill")) {

CalcBill bill = new CalcBill();

bill.setUserid(Integer.parseInt(request.getParameter("userid")));

bill.setUnitConsumed(Integer.parseInt(request.getParameter("unit")));

bill.setZoneName(request.getParameter("zonename"));

bill.setMonth(request.getParameter("month"));

bill.setName(request.getParameter("name"));

bill.setYear(request.getParameter("year"));

bill.setStatus("Not Paid");

bill.setDues(Integer.parseInt(request.getParameter("dues")));

CalcDAO calcDAO = new CalcDAO();

int result;

try {

calcDAO.calculateBill(bill);

/\*

\* if(result!=0) {

\*/

HttpSession httpSession = request.getSession();

httpSession.setAttribute("bill", bill);

String msg = "Bill Calculated";

request.setAttribute("msg", msg);

request.getRequestDispatcher("CalculateBillView.jsp").forward(request, response);

// response.sendRedirect("CalculateBillView.jsp");

/\* } \*/

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

if (request.getParameter("Action").equals("Pay")) {

CalcBill bill2 = new CalcBill();

// CalcDAO calcDAO2=new CalcDAO();

}

// doGet(request, response);

if (request.getParameter("Action").equals("Pay Bill")) {

System.out.println("inside pay bill");

CalcBill bill2 = new CalcBill();

bill2.setCid(Integer.parseInt(request.getParameter("id")));

bill2.setPayamt(Integer.parseInt(request.getParameter("bamt")));

bill2.setStatus("PAID");

CalcDAO calcDAO2 = new CalcDAO();

try {

bill2 = calcDAO2.payAmount(bill2);

HttpSession httpSession = request.getSession();

httpSession.setAttribute("pay", bill2);

request.setAttribute("msg", "Bill Paid Successfully!!!");

request.getRequestDispatcher("paybill.jsp").forward(request, response);

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

if (request.getParameter("Action").equals("Update Cost")) {

CostBean bean = new CostBean();

bean.setCost1(Integer.parseInt(request.getParameter("c1")));

bean.setCost2(Integer.parseInt(request.getParameter("c2")));

bean.setCost3(Integer.parseInt(request.getParameter("c3")));

bean.setCost4(Integer.parseInt(request.getParameter("c4")));

try {

bean = CalcDAO.updateCost(bean);

request.setAttribute("msg", "Cost Updated Successfully");

request.getRequestDispatcher("updatecost.jsp").forward(request, response);

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

**Login Controller**

package com.ebilling.Ctrl;

import java.io.IOException;

import java.sql.SQLException;

import java.util.HashMap;

import java.util.Map;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

import com.ebilling.Model.LoginDAO;

import com.ebilling.bean.UserBean;

/\*\*

\* Servlet implementation class LoginCtrl

\*/

@WebServlet("/LoginCtrl")

public class LoginCtrl extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#HttpServlet()

\*/

public LoginCtrl() {

super();

// TODO Auto-generated constructor stub

}

/\*\*

\* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

request.getRequestDispatcher("login.jsp").forward(request, response);

request.getRequestDispatcher("userhome.jsp").forward(request, response);

request.getRequestDispatcher("adminhome.jsp").forward(request, response);

}

/\*\*

\* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

System.out.println("inside post");

String username = request.getParameter("username");

String pass = request.getParameter("pass");

UserBean bean = new UserBean();

bean.setUsername(username);

bean.setPassword(pass);

System.out.println("before creating object of dao");

LoginDAO dao = new LoginDAO();

try {

String result = dao.authenticateUser(bean);

System.out.println("result " + result);

if (result.equals("ADMIN\_ROLE")) {

System.out.println("inside admin");

HttpSession httpSession = request.getSession();

httpSession.setAttribute("admin", username);

request.setAttribute("username", username);

System.out.println(username);

response.sendRedirect("adminhome.jsp");

}

if (result.equals("USER\_ROLE")) {

System.out.println("inside auser");

HttpSession httpSession = request.getSession();

httpSession.setAttribute("user", username);

request.setAttribute("username", username);

System.out.println(username);

response.sendRedirect("userhome.jsp");

}

if (result.equals("INVALID")) {

System.out.println("invalid");

String msg = "Please enter valid Username or password";

request.setAttribute("msg", msg);

request.getRequestDispatcher("login.jsp").forward(request, response);

}

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

// doGet(request, response);

}

}

**RegCTRL**

package com.ebilling.Ctrl;

import java.io.IOException;

import java.sql.SQLException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

import com.ebilling.Model.RegDAO;

import com.ebilling.bean.UserBean;

/\*\*

\* Servlet implementation class RegCtrl

\*/

@WebServlet("/RegCtrl")

public class RegCtrl extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#HttpServlet()

\*/

public RegCtrl() {

super();

// TODO Auto-generated constructor stub

}

/\*\*

\* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

// response.getWriter().append("Served at:

// ").append(request.getContextPath());

request.getRequestDispatcher("adduser.jsp").forward(request, response);

request.getRequestDispatcher("viewusers.jsp").forward(request, response);

}

/\*\*

\* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// TODO Auto-generated method stub

String username = request.getParameter("username");

String emailid = request.getParameter("email");

String phone = request.getParameter("phone");

String address = request.getParameter("address");

String password = request.getParameter("pass");

UserBean bean = new UserBean();

bean.setUsername(username);

bean.setUseremail(emailid);

bean.setUserphone(phone);

bean.setUseraddress(address);

bean.setPassword(password);

bean.setRoleid(2);

RegDAO dao = new RegDAO();

try {

int id = dao.addUser(bean);

// int id=dao.getLastId();

System.out.println("called " + id);

// System.out.println(id);

System.out.println(bean);

if (id == 0) {

}

HttpSession httpSession = request.getSession();

httpSession.setAttribute("id", id);

String regmsg = "Your User ID is " + id;

// request.setAttribute("id", id);

request.setAttribute("regmsg", regmsg);

request.getRequestDispatcher("adduser.jsp").forward(request, response);

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

HttpSession httpSession = request.getSession();

httpSession.setAttribute("ERROR", "User already exist");

request.getRequestDispatcher("adduser.jsp").forward(request, response);

}

if (request.getParameter("Action").equals("Edit")) {

System.out.println("inside edit section");

UserBean userBean = new UserBean();

userBean.setUserid(Integer.parseInt(request.getParameter("id")));

userBean.setUseremail(request.getParameter("email"));

userBean.setUseraddress(request.getParameter("address"));

userBean.setUserphone(request.getParameter("phone"));

RegDAO dao2 = new RegDAO();

try {

int result = dao2.editUser(userBean);

System.out.println("result is " + result);

String msg = "User edited successfully!!";

request.setAttribute("msg", msg);

request.getRequestDispatcher("edituser.jsp").forward(request, response);

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

if (request.getParameter("Action").equals("Edit Profile")) {

System.out.println("inside edit section");

UserBean userBean = new UserBean();

userBean.setUserid(Integer.parseInt(request.getParameter("id")));

userBean.setUsername(request.getParameter("uname"));

userBean.setUseremail(request.getParameter("email"));

userBean.setUseraddress(request.getParameter("address"));

userBean.setUserphone(request.getParameter("phone"));

userBean.setPassword(request.getParameter("password"));

RegDAO dao2 = new RegDAO();

try {

int result = dao2.editMyProfile(userBean);

System.out.println("result is " + result);

String msg = "Profile edited successfully!!";

request.setAttribute("msg", msg);

request.getRequestDispatcher("editprofileView.jsp").forward(request, response);

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} }

}

}

**Implementation and Testing :**

**Black-Box Testing**:

Black Box Testing, also known as Behavioral Testing, is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.

This can be following way:

* Input interfacing
* Processing
* Output interfacing



This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

* Incorrect or missing functions
* Interface errors
* Errors in data structures or external database access
* Behavior or performance errors
* Initialization and termination errors

**White-Box Testing:**

White Box Testing ,also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential.

White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/ transparent box; inside which one clearly sees.

**Limitations and Future Application of the Project**

**Futures Enhancement:**

* In future we can expand this project on the web.
* Software can be accessed through internet also.

**Limitation :**

* In this system Email to user for bill facility is not available.
* In this system Download of bill is not available.

**Conclusion:**

This Web Application provides facility to Pay Bill Online using this web application. It saves time as it allows number of users to pay the bill online instead of standing in line for bill payment. It is automatically generated by the server.

Administrator has a privilege to create, modify and delete the Users, Add Bill payment and Update cost according to the need.