**DEVELOP A PROTOTYPE SYSTEM THAT USES BLOCKCHAIN TECHNOLOGY TO SECURE DATA TRANSMISSION IN MOBILE NETWORKS**

**ABSTRACT**

In this project, we have developed a **secure messaging application** that enables users to **register, log in, and communicate with other users** in a decentralized and tamper-proof manner. The system ensures **data integrity and security** by incorporating **blockchain technology** into message transmission.

Each user can **view all registeredusers** and send messages to any user within the platform. To achieve security, we implemented **cryptographic hashing (SHA-256)** using Python’s hashlib package, ensuring that messages are **immutable and securely stored**. Additionally, timestamps from Python’s datetime module are used to maintain the integrity of message sequences.

For **efficient data storage and retrieval**, we utilized a **SQLite database**, where user information and messages are managed. Each message is **hashed and linked** to a previous transaction, forming a **blockchain-like structure**that prevents unauthorized modifications. This approach enhances **privacy, security, and trust** in message exchanges.

**EXISTING SYSTEM**

In traditional messaging applications, user authentication and message storage rely on centralized servers, where all user data and communications are stored in a single database. These systems often use encryption methods like SSL/TLS for securing messages in transit, but they do not inherently prevent data tampering, unauthorized access, or server breaches. Additionally, if the server is compromised, hackers can gain access to user messages, leading to privacy concerns. Many existing systems also lack transparency and do not provide a way to verify message integrity once stored.

### ****DRAWBACKS OF THE EXISTING SYSTEM****

* **Centralized Data Storage** A single point of failure makes data vulnerable to attacks.
* **Data Tampering** Messages stored in a database can be modified by unauthorized users.
* **Security Risks** If the server is hacked, all messages can be exposed or deleted.
* L**ackofTransparency** Users cannot verify whether their messages have been altered.
* **No Decentralization** Users rely on a single authority to manage and protect data.

**PROPOSED SYSTEM**

The proposed system introduces a blockchain-based messaging application that ensures secure,tamper-proof communication between users. Unlike traditional systems, this application hashesevery message using SHA-256 and links it to the previous message, forming a blockchain structure. This ensures that once a message is stored, it cannot be altered or deleted, enhancing data integrity. The system allows users to register, log in, and send messages to any otherregistered user, with all messages stored securely in a SQLite database while maintaining blockchain-based verification. The hashlib and datetime Python packages are used to implement cryptographic security and track message timestamps.

### ****ADVANTAGES OF THE PROPOSED SYSTEM****

* **Tamper-Proof Messaging**Messages are hashed and linked in a blockchain structure.
* **Enhanced Security**SHA-256 hashing ensures messages remain **secure and immutable**.  
  **User Transparency**Users can verify message integrity using cryptographic hashes.
* **Decentralized Trust**No central authority controls or manipulates the stored messages.
* **Scalability**Can be extended with smart contracts and decentralized storage in the future.

**MODULE DESCRIPTION**

**MODULES**

* User Registration
* User Authentication
* Secure Transfer
* Message Hashing
* Message Decryption

**User Registration**

Users sign up by providing credentials, which are securely stored with **hashed passwords** in a **SQLite database**.

**User Authentication**

Users log in using **hashed passwords**or sessions manage secure access control.

**Secure Transfer**

Messages are **encrypted** before transmission and sent over a **secure connection** to prevent interception.

**Message Hashing**

Each message is **hashed using SHA-256**, ensuring **data integrity** and preventing unauthorized modifications.

**Message Decryption**

If encryption is applied, the recipient **decrypts the message** using a **private key**, ensuring secure communication.

### ****HARDWARE REQUIREMENTS****

* **System :** Pentium i3 Processor.
* **Hard Disk :** 500 GB.
* **Monitor :** 15’’ LED
* **Input Devices :** Keyboard, Mouse
* **Ram :** 6 GB

### ****SOFTWARE REQUIREMENTS****

* **Operating system :** Windows 10 Pro.
* **Coding Language :** Python.
* **Web Framework :** Django

**4. SOFTWARE DESCRIPTION**

**SOFTWRAE DESCRIPTION**

**DJANGO**

Django is a powerful and flexible web framework for Python that simplifies the process of building robust web applications. Developed by a team of experienced developers, Django emphasizes rapid development, clean design, and a focus on best practices. Here's a more detailed look at its features and benefits:

**KEY FEATURES**

1. **MTV ARCHITECTURE**:
   * Django follows the Model-Template-View (MTV) pattern, which separates application logic into three interconnected components.
   * **Model**: Defines the data structure and business logic, handling database interactions.
   * **Template**: Manages the presentation layer, rendering HTML pages using a templating language.
   * **View**: Acts as the intermediary between models and templates, processing user requests and returning responses.
2. **BUILT-IN ADMIN INTERFACE**:
   * Django automatically generates a fully functional admin interface for managing application data.
   * This feature allows developers to quickly set up CRUD (Create, Read, Update, Delete) operations without extra coding.
3. **OBJECT-RELATIONAL MAPPING (ORM)**:
   * Django’s ORM simplifies database interactions by allowing developers to define database models as Python classes.
   * It abstracts the database layer, enabling developers to work with data using Python code rather than SQL, which enhances productivity and reduces errors.
4. **SECURITY FEATURES**:
   * Django incorporates several built-in security measures to protect against common vulnerabilities:
     + **SQL Injection**: The ORM automatically escapes queries to prevent injection attacks.
     + **Cross-Site Scripting (XSS)**: Django automatically escapes HTML to prevent XSS attacks.
     + **Cross-Site Request Forgery (CSRF)**: The framework includes protection against CSRF attacks by validating requests.
     + **Clickjacking**: Django can help protect against clickjacking by implementing security headers.
5. **SCALABILITY**:
   * Django is designed to handle high traffic loads and can scale effectively as applications grow.
   * It is used by many high-traffic sites, demonstrating its capability to manage significant data volumes and user requests.
6. **COMPREHENSIVE DOCUMENTATION**:
   * Django’s documentation is thorough and well-organized, making it easy for developers to learn and troubleshoot.
   * The community is active, providing numerous tutorials, guides, and third-party packages to extend functionality.
7. **REUSABILITY AND PLUGGABILITY**:
   * Django’s modular approach allows developers to create reusable applications or components (called "apps").
   * Each app can be easily integrated into different projects, promoting DRY (Don't Repeat Yourself) principles.
8. **INTERNATIONALIZATION (I18N)**:
   * Django supports multiple languages and time zones, making it suitable for applications targeting global audiences.
   * Developers can easily translate strings and format dates/times according to user preferences.
9. **DEVELOPMENT TOOLS**:
   * The framework includes a built-in development server, allowing for quick testing and debugging.
   * It also features tools for managing static files (CSS, JavaScript, images) and database migrations, streamlining the development process.
10. **COMMUNITY AND ECOSYSTEM**:
    * Django has a strong community that contributes to its growth, with a wide range of third-party packages available through the Python Package Index (PyPI).
    * This ecosystem allows developers to extend Django's functionality with plugins for various needs, from authentication to payment processing.

**SQLITE**

SQLite is a lightweight, serverless, and self-contained relational database management system (RDBMS). It is designed for simplicity, efficiency, and portability, making it an ideal choice for many applications, particularly for small to medium-sized projects. Here are some key features and characteristics of SQLite:

**KEY FEATURES**

1. **FILE-BASED**:
   * SQLite stores the entire database as a single file on disk, making it easy to manage and distribute. This file can be easily copied, moved, or backed up.
2. **SERVERLESS ARCHITECTURE**:
   * Unlike traditional databases, SQLite does not require a separate server process. Applications can directly access the SQLite database file, which simplifies setup and deployment.
3. **LIGHTWEIGHT**:
   * SQLite has a minimal footprint, with a small binary size and low memory requirements. This makes it suitable for applications where resources are limited, such as mobile apps or embedded systems.
4. **CROSS-PLATFORM COMPATIBILITY**:
   * SQLite works across various platforms, including Windows, macOS, Linux, and mobile operating systems (iOS and Android). This portability allows developers to use the same database across different environments.
5. **TRANSACTIONAL SUPPORT**:
   * SQLite supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data integrity even in the event of failures or crashes.
6. **SQL SUPPORT**:
   * SQLite supports most of the SQL standards, making it easy for developers familiar with SQL to use it effectively. It includes features like joins, indexes, triggers, and views.
7. **ZERO CONFIGURATION**:
   * SQLite requires no configuration or administration, making it an excellent choice for developers looking for a hassle-free database solution. There are no user accounts, permissions, or server setup required.
8. **CONCURRENCY**:
   * While SQLite is not designed for high-concurrency environments like traditional client-server databases, it supports multiple readers and limited writers. It uses locking mechanisms to manage access to the database file.
9. **EXTENSIBLE**:
   * SQLite can be extended with user-defined functions and custom data types, allowing developers to tailor it to their specific needs

SYSTEM DESIGN AND DEVELOPMENT

INPUT DESIGN

Input design is the process of converting user-originated inputs to a computer-based format. Input design is one of the most expensive phases of the operation of computerized system and is often the major problem of a system.

In the project, the input design is made in various web forms with various methods. For example, in the user creation form, the empty username and password is not allowed. The username if exists in the database, the input is considered to be invalid and is not accepted. Likewise, during the login process, the username is a must and must be available in the user list in the database. Then only login is allowed.

Input forms are

**Signup**

**Login**

**OUTPUT DESIGN**

Output design generally refers to the results and information that are generated by the system for many end-users; output is the main reason for developing the system and the basis on which they evaluate the usefulness of the application.

In the project, the user details, search, are the web forms in which the output is available.

Outputs are

**Add cart**

View Order details

DATABASE DESIGN

The database design is a must for any application developed especially more for the data store projects. Since the chatting method involves storing the message in the table and produced to the sender and receiver, proper handling of the table is a must. In the project, login table is designed to be unique in accepting the username and the length of the username and password should be greater than zero. The different users view the data in different format according to the privileges given.

**TYPES OF TESTING**

**UNIT TESTING**

This is the first level of testing. The different modules are tested against the specifications produced during the integration. This is done to test the internal logic of each module. Those resulting from the interaction between modules are initially avoided. The input received and output generated is also tested to see whether it falls in the expected range of values. Unit testing is performed from the bottom up, starting with the smallest and lowest modules and proceeding one at a time.

The units in a system are the modules and routines that are assembled and integrated to perform a specific function. The programs are tested for correctness of logic applied and detection of errors in coding. Each of the modules was tested and errors are rectified. They were then found to function properly.

**INTEGRATION TESTING**

In integration testing, the tested modules are combined into sub-systems, which are then tested. The goal of integration testing to check whether the modules can be integrated properly emphasizing on the interfaces between modules. The different modules were linked together and integration testing done on them.

**VALIDATION TESTING**

The objective of the validation test is to tell the user about the validity and reliability of the system. It verifies whether the system operates as specified and the integrity of important data is maintained. User motivation is very important for the successful performance of the system.

All the modules were tested individually using both test data and live data. After each module was ascertained that it was working correctly and it had been "integrated" with the system. Again the system was tested as a whole. We hold the system tested with different types of users. The System Design, Data Flow Diagrams, procedures etc. were well documented so that the system can be easily maintained and upgraded by any computer professional at a later

**SYSTEM TESTING**

The integration of each module in the system is checked during this level of testing. The objective of system testing is to check if the software meets its requirements. System testing is done to uncover errors that were not found in earlier tests. This includes forced system failures and validation of total system as the user in the operational environment implements it. Under this testing, low volumes of transactions are generally based on live data. This volume is increased until the maximum level for each transactions type is reached. The total system is also tested for recovery after various major failures to ensure that no data are lost during the breakdown

**SYSTEM DEVELOPEMENT**

A Systems Development Life Cycle (SDLC) adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation, and are explained in the section below. A number of system development life cycle (SDLC) models have been created: waterfall, fountain, spiral, build and fix, rapid prototyping, incremental, and synchronize and stabilize. The oldest of these, and the best known, is the waterfall model: a sequence of stages in which the output of each stage becomes the input for the next.

The waterfall model is a popular version of the systems development life cycle model for software engineering. Often considered the classic approach to the systems development life cycle, the waterfall model describes a development method that is linear and sequential. Waterfall development has distinct goals for each phase of development. Imagine a waterfall on the cliff of a steep mountain. Once the water has flowed over the edge of the cliff and has begun its journey down the side of the mountain, it cannot turn back. It is the same with waterfall development. Once a phase of development is completed, the development proceeds to the next phase and there is no turning back.

The advantage of waterfall development is that it allows for departmentalization and managerial control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process like a car in a carwash, and theoretically, be delivered on time. Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order, without any overlapping.

**FILE DESIGN**

A file system provides the infrastructure necessary to support various project tasks. At its core, a file system is a method for organizing, storing, retrieving, and managing information on a permanent storage medium such as a disk. File systems handle permanent storage and are a fundamental component of all operating systems. Different approaches exist for managing permanent storage. On one end of the spectrum are simple file systems that impose enough restrictions to inconvenience users and make the system difficult to use. When choosing an appropriate file system for a particular operating system, one must balance the needs of the problem with the constraints of the project.

The basic abstractions of files and directories form the foundation of a file system. Many operations can be performed on files and directories. All file systems must provide some basic level of support, but beyond these basic primitives, there are additional features, extensions, and more sophisticated operations.

The structure of a file can vary significantly. A file system may impose no structure on a file, treating it as an unstructured "stream of bytes" with no inherent organization. In this case, the file system only records the size of the file and allows programs to read the bytes in any order or manner desired. Conversely, if a file system enforces a formal structure, it typically does so using records. With record-based structures, a programmer specifies the size and format of each record, and all input/output operations must align with record boundaries and be multiples of the record length.

**IMPLEMENTATION**

Implementation is the most crucial stage in achieving a successful system and giving the user’s confidence that the new system is effective and workable. Implementation of this project refers to the installation of the package in its real environment to the full satisfaction of the users and operations of the system.

Testing is done individually at the time of development using the data and verification is done the way specified in the program specification. In short, implementation constitutes all activities that are required to put an already tested and completed package into operation. The success of any information system lies in its successful implementation.

System Implementation is the stage in the project where the theoretical design is turned into a working system. The most critical stage is achieving a successful system and in giving confidence on the new system for the user that it will work efficiently and effectively. The existing system was long time process.

The project execution was checked with live environment and the user requirements are satisfied. Proper implementation is essential to provide a reliable system to meet the organization requirements.

**CONCLUSION**

The development of this Blockchain-Based Secure Chat Application successfully addresses the challenges of centralized messaging platforms by integrating blockchain technology with cryptographic encryption. By implementing SHA-256 hashing and a blockchain-like structure, the system ensures that messages remain immutable, tamper-proof, and securely stored. The use of timestamps further strengthens the integrity of message sequences, preventing unauthorized modifications. Additionally, the decentralized nature of the system enhances privacy, security, and trust in user communication. With a secure and transparent messaging framework, this approach provides a reliable solution for confidential and secure digital communication.

**FUTURE SCOPE**

The Blockchain-Based Secure Chat Application has significant potential for future enhancements to improve security, scalability, and usability. Integrating decentralized storage solutions like IPFS can eliminate reliance on traditional databases, further securing message storage. Implementing smart contract-based authentication can enhance user verification and prevent unauthorized access. The system can also be extended to support multi-platform accessibility, including mobile (Android & iOS) and web applications. Additionally, incorporating end-to-end encrypted file sharing and AI-driven threat detection can strengthen security measures against cyber threats. The integration of cryptocurrency-based transactions could enable secure, anonymous payments for premium features. To improve scalability and performance, blockchain scaling techniques such as sharding and Layer 2 solutions can be adopted. Furthermore, federated identity management using decentralized identity (DID) frameworks can allow seamless authentication across multiple platforms. By incorporating these advancements, the application can evolve into a highly secure, decentralized, and efficient communication platform, ensuring privacy and trust in digital messaging.

**BIBLIOGRAPHY**

Textual Reference

* **William S. Vincent.**Django: **Django for Beginners**. O'Reilly Media, 2020.
* **Andrew Pinkham.** Django Unleashed. Packt Publishing, 2020.

Online Reference:

* www.wikipedia.com
* [www.w3schools.com](http://www.w3schools.com)
* [docs.djangoproject.com](https://docs.djangoproject.com/en/stable/)

**TABLES**

**USER TABLE**

**Primary Key – id**

|  |  |  |
| --- | --- | --- |
| **FIELD** | **DATATYPE** | **DESCRIPTION** |
| Id | Int | Id of the user |
| Username | Varchar | Name of the user |
| Email | Varchar | Email of the user |
| Password | Varchar | Password of the user |

**MESSAGES TABLE**

**Primary Key – id**

|  |  |  |
| --- | --- | --- |
| **FIELD** | **DATATYPE** | **DESCRIPTION** |
| Id | Int | Id of the message |
| Receiver\_id | Int | Id of the receiver |
| Sender\_id | Int | Id of the sender |
| Message | Varchar | Encrypted message |
| Timestamp | Date | Timestamp of the message |
| Blockchain\_id | Varchar | Hash block of the message |

**DATAFLOW DIAGRAM**

**Level 0**

DB

User 1

User 2

**Level 1**

User

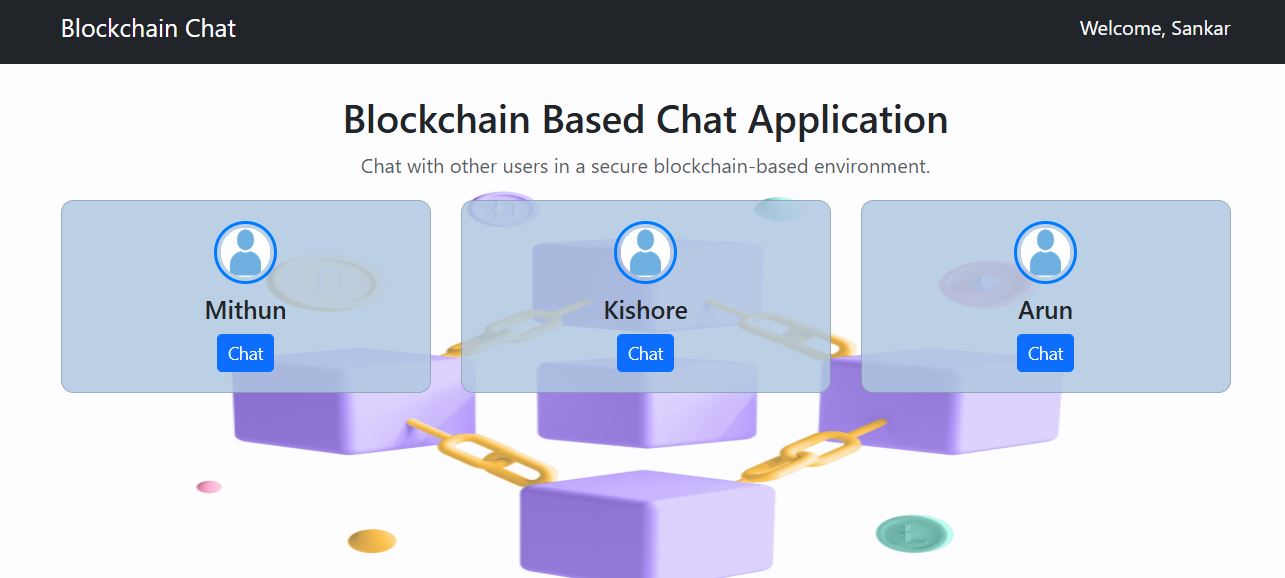
messages

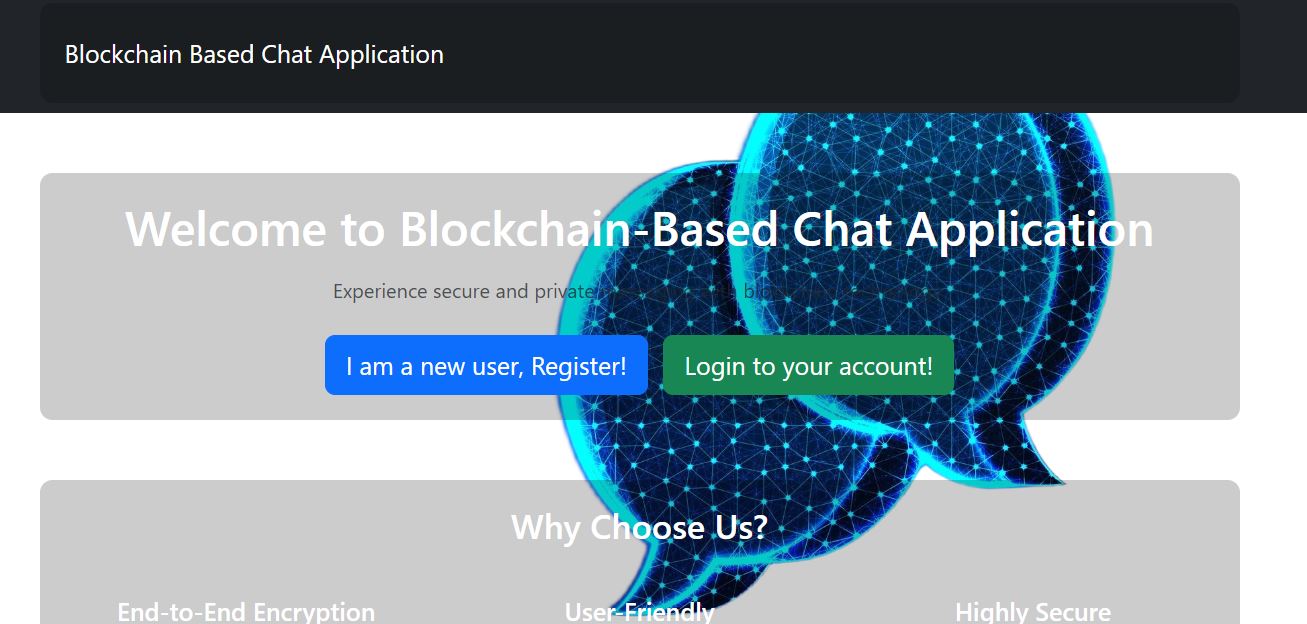
Messages

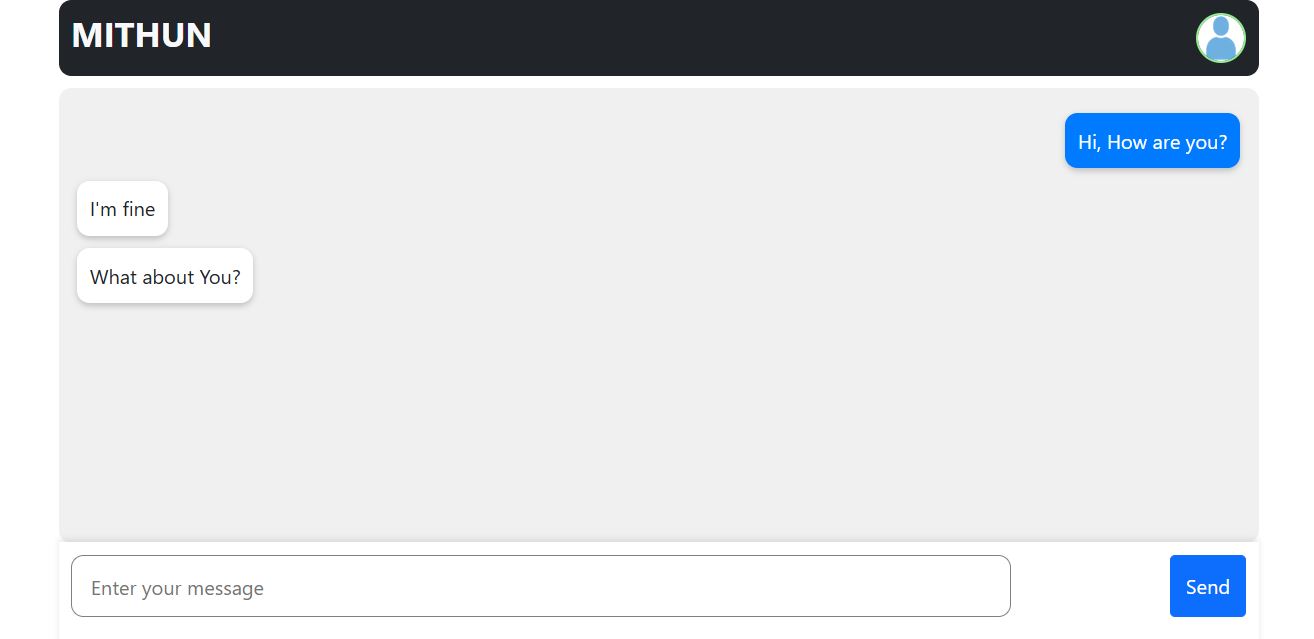
Encrypt

Decrypt

**SCREENSHOTS**

****

****

****

**SAMPLE CODE**

from django.shortcuts import render, redirect

from .models import\*

from django.db.models import Q

import hashlib

import json

from datetime import datetime

classBlockchain:

    def\_\_init\_\_(self):

        self.chain = []

        self.create\_block(sender="Genesis", receiver="Genesis", message="Start of blockchain", previous\_hash="0")

    defcreate\_block(self, sender, receiver, message, previous\_hash):

        block = {

            'index': len(self.chain) +1,

            'timestamp': str(datetime.now()),

            'sender': sender,

            'receiver': receiver,

            'message': message,

            'previous\_hash': previous\_hash,

            'hash': self.calculate\_hash(sender, receiver, message, previous\_hash)

        }

        self.chain.append(block)

        return block

    defcalculate\_hash(self, sender, receiver, message, previous\_hash):

        block\_data =f"{sender}{receiver}{message}{previous\_hash}{str(datetime.now())}"

        return hashlib.sha256(block\_data.encode('utf-8')).hexdigest()

    defget\_last\_block(self):

        returnself.chain[-1] ifself.chain elseNone

    defverify\_chain(self):

        for i inrange(1, len(self.chain)):

            current\_block =self.chain[i]

            previous\_block =self.chain[i -1]

            if current\_block['previous\_hash'] != previous\_block['hash']:

                returnFalse

            if current\_block['hash'] !=self.calculate\_hash(current\_block['sender'], current\_block['receiver'], current\_block['message'], current\_block['previous\_hash']):

                returnFalse

        returnTrue

defhome(request):

    return render(request, 'index.html')

defregister(request):

    if request.method =="POST":

        username = request.POST['username']

        email = request.POST['email']

        password = request.POST['password']

        obj = User()

        if User.objects.filter(email=email).exists():

           msg="Email already exists"

        else:

            obj.username = username

            obj.email=email

            obj.password = password

            obj.save()

            msg="User Registered successfully"

        return render(request, 'register.html',{'message':msg})

    return render(request, 'register.html')

deflogin(request):

    if request.method =="POST":

        email = request.POST['email']

        password = request.POST['password']

        user = User.objects.get(email=email, password=password)

        if(user):

            request.session['username'] = user.username

            request.session['email'] = user.email

            return redirect('dashboard')

        else:

            return render(request,'login.html')

    return render(request,'login.html')

defdashboard(request):

    username = request.session.get('username', 'Guest')

    email = request.session.get('email')

    if email:

        data = User.objects.exclude(email=email)

    else:

        data = User.objects.all()

    return render(request, 'dashboard.html', {'username': username, 'email': email, 'users': data})

defcaesar\_cipher\_encrypt(message, shift):

    encrypted\_message =""

    for char in message:

        if char.isalpha():

            shift\_base =65if char.isupper() else97

            encrypted\_message +=chr((ord(char) - shift\_base + shift) %26+ shift\_base)

        else:

            encrypted\_message += char

    return encrypted\_message

defcaesar\_cipher\_decrypt(encrypted\_message, shift):

    decrypted\_message =""

    for char in encrypted\_message:

        if char.isalpha():

            shift\_base =65if char.isupper() else97

            decrypted\_message +=chr((ord(char) - shift\_base - shift) %26+ shift\_base)

        else:

            decrypted\_message += char

    return decrypted\_message

blockchain = Blockchain()

defchatarea(request, uid, email):

    username = request.session.get('username', 'Guest')

    receiver\_id = uid

    sender\_email = email

    receiver = User.objects.get(id=receiver\_id)

    sender = User.objects.get(email=sender\_email)

    request.session['receiver-name'] = receiver.username

    request.session['receiver-email'] = receiver.email

    messages = Messages.objects.filter(Q(sender=sender, receiver=receiver) | Q(sender=receiver, receiver=sender)).order\_by('timestamp')

    shift =3

    decrypted\_messages = []

    for msg in messages:

        decrypted\_message = caesar\_cipher\_decrypt(msg.message, shift)

        decrypted\_messages.append(decrypted\_message)

    if request.method =="POST":

        message = request.POST['msg']

        encrypted\_message = caesar\_cipher\_encrypt(message, shift)

        message\_obj = Messages.objects.create(sender=sender, receiver=receiver, message=encrypted\_message)

        message\_obj.save()

        messages = Messages.objects.filter(Q(sender=sender, receiver=receiver) | Q(sender=receiver, receiver=sender)).order\_by('timestamp')

        decrypted\_messages = []

        for msg in messages:

            decrypted\_message = caesar\_cipher\_decrypt(msg.message, shift)

            decrypted\_messages.append(decrypted\_message)

    messages\_and\_decrypted =zip(messages, decrypted\_messages)

    return render(request, "chatarea.html", {'name': receiver.username, 'messages\_and\_decrypted': messages\_and\_decrypted})

<!DOCTYPE html>

<htmllang="en">

<head>

    <metacharset="UTF-8">

    <metaname="viewport"content="width=device-width, initial-scale=1.0">

    <title>Dashboard</title>

    {%load static%}

    <linkhref="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"rel="stylesheet">

    <style>

        body {

            background-color: #f8f9fa;

        }

        .navbar {

            padding: 10px;

        }

        .user-card {

            transition: transform 0.3s ease-in-out, box-shadow 0.3s ease-in-out;

            border-radius: 10px;

            margin-bottom: 20px;

            background:rgba(175, 199, 224, 0.84);

        }

        .user-card:hover {

            transform: translateY(-5px);

            box-shadow: 0px 4px 10px rgba(0, 0, 0, 0.1);

        }

        .card-body img {

            width: 50px;

            height: 50px;

            border: 3px solid #007bff;

            padding: 3px;

        }

        #back1{

            position: absolute;

            zindex: 0;

            left:0;

            width: 100vw;

            height: 100vh;

            opacity:0.7;

            animation-name: aniback;

            animation-duration: 5s;

            animation-iteration-count: 1;

            animation-delay: 1s;

            transition: all 1ms ease

        }

        @keyframes aniback{

            0%{

                opacity:0;

            }

            100%{

                opacity:0.7;

            }

        }

        #con{

            position: relative;

            zindex:2

        }

    </style>

</head>

<body>

    <navclass="navbar navbar-expand-lg navbar-dark bg-dark">

        <divclass="container">

            <aclass="navbar-brand"href="#">Blockchain Chat</a>

            <spanclass="navbar-text text-light">Welcome, {{ username }}</span>

        </div>

    </nav>

    <imgid="back1"src="{% static 'on\_chain.jpg' %}"/>

    <divid="con"class="container mt-4">

        <h2class="text-center">Blockchain Based Chat Application</h2>

        <pclass="text-center text-muted">Chat with other users in a secure blockchain-based environment.</p>

        <divclass="row">

            {% ifusers %}

                {% foruserinusers %}

                    <divclass="col-md-4">

                        <divclass="card user-card">

                            <divclass="card-body text-center">

                                {% load static %}

                                <imgsrc="{% static 'person.png' %}"class="rounded-circle mb-2"alt="Profile">

                                <h5class="card-title">{{ user.username }}</h5>

                                <ahref="/chatArea/{{user.id}}/{{email}}"class="btn btn-primary btn-sm">Chat</a>

                            </div>

                        </div>

                    </div>

                {% endfor %}

            {% else %}

                <pclass="text-center">No users available.</p>

            {% endif %}

        </div>

    </div>

    <scriptsrc="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>

</body>

</html>