PROJECT REPORT

“ONLY FRIENDS : A Secure rent WEBAPP”

Prepared By:

Rafael Josh Isaac
001202300024
IT Class 2

DISTRIBUTED AND PARALLEL SYSTEM CLASS
President University
2025

INTRODUCTION

This project was initially developed as a part of a midterm assignment to deepen my understanding of Docker and how it is used in real-world application deployment. The main objective at first was to learn how to containerize applications using Docker Compose and manage multiple services such as frontend, backend, database, and proxy servers. However, as the project evolved, I decided to expand the scope by developing a **secure web-based marketplace application** to apply and explore full-stack development principles along with web application security concepts.

The marketplace app I developed allows users to register and log in, browse profiles of other users, express interest through a swipe feature (similar to Tinder), and engage in chat conversations. This simulation mimics the functionality of a modern matchmaking or social platform, where users can interact and possibly find a partner.

Given the sensitive nature of user data and the interaction features, I saw this as an opportunity to not only build the app but also explore how to **secure it by following the OWASP Top 10 standards**, which outline the most critical security risks for web applications.

Project Objectives:

- To learn and practice **Docker and Docker Compose** for managing multi-service environments.
- To build a fully functional **web-based marketplace platform** with features such as registration, login, browsing, matching, and chat.
- To understand and implement **web application security best practices**, particularly based on the **OWASP Top 10**list.
- To use **Flask** (Python) for the frontend and **Go** for the backend to strengthen multi-language full-stack development skills.
- To design and use a **PostgreSQL** database to handle user and order data securely.
- To configure **Nginx** as a reverse proxy that supports **HTTPS with SSL certificates** for secure client-server communication.
- To explore **Redis** for simple caching functionality and understand how it can help optimize performance and security.
- To gain experience in building a **modular, scalable application** using container-based infrastructure.

This project aims to simulate a real-world secure web development environment, combining software development, security awareness, and DevOps practices.

APPLICATION DESCRIPTION

A. Purpose of the Application

The application developed in this project is a simulation of a **"Rent a Friend" platform**, which allows users to find and connect with companions—either for friendship or casual companionship—to engage in shared activities or hobbies. While the concept may sound unusual at first, this type of platform has become increasingly popular in various parts of the world, especially for individuals seeking social interaction, emotional support, or simply a friendly presence without long-term commitment.

This web application enables users to **rent a friend or even a girlfriend** for a period of time, with the goal of providing companionship for activities such as attending events, studying together, playing games, or exploring mutual hobbies. In addition to the rental feature, the app also incorporates a **matching system** similar to popular dating apps (e.g., Tinder), allowing users to potentially connect with others based on mutual interests and preferences. A built-in **chatbox** feature is provided to help users communicate after registering and browsing available profiles.

This project, while fictional and experimental, is designed to mimic the functional and structural needs of a modern, user-driven social or matchmaking web application

B. Application Flow

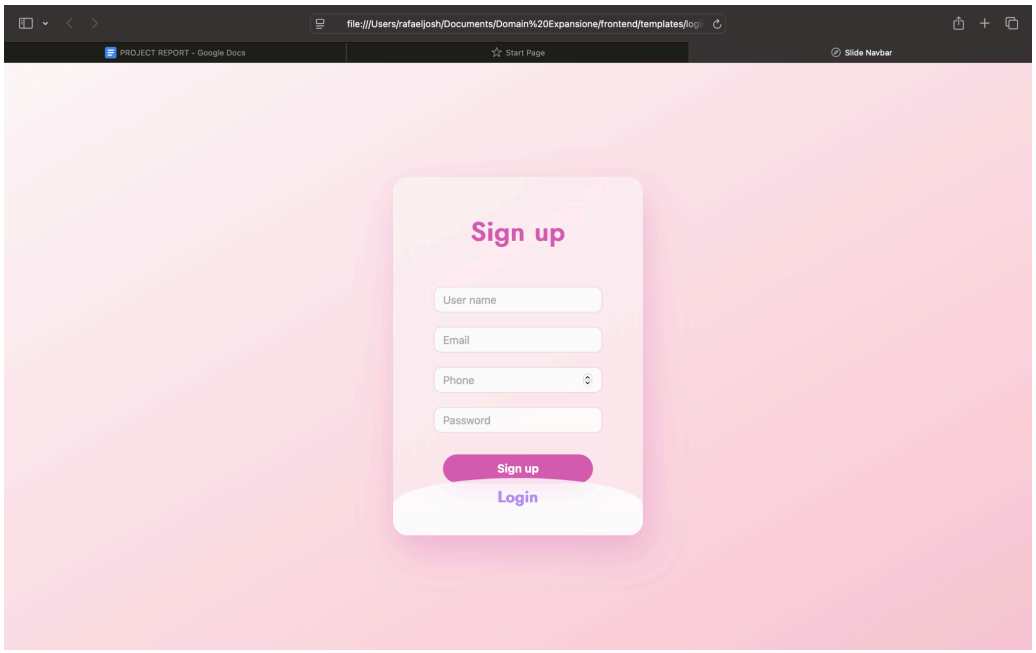
The general flow of the application is structured as follows:

1. **User Registration**

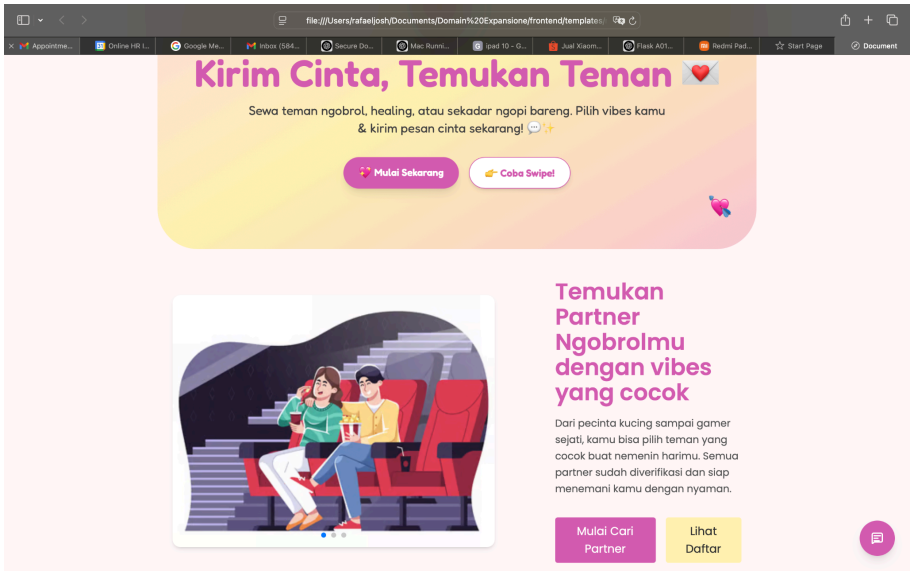
   New users begin by registering their account through a registration form. Required fields include:

   - Email address
   - Password
   - Username
   - Phone number

   The submitted data is validated and stored securely in the database with hashed passwords.



2. **User Login**
   Registered users can log in using their email and password. Upon successful login:
   - A **JWT (JSON Web Token)** is generated by the backend.
   - This token is stored securely in the user's browser as an **HTTP-only cookie** to maintain session authentication.
3. **Dashboard / Main Page**



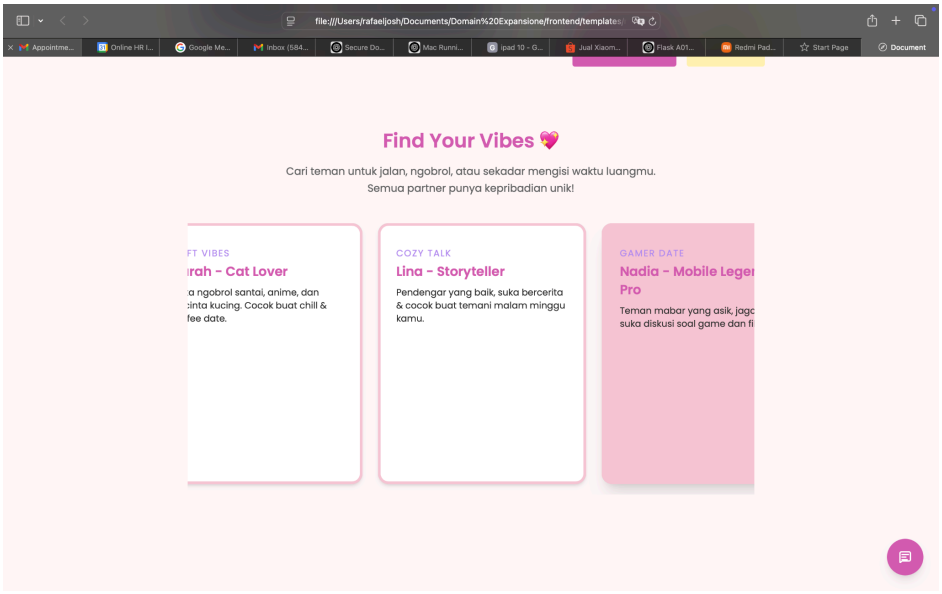
After logging in, users are redirected to the main dashboard where they can interact with the application’s core features. These include:

a. **Profile Cards Slider**

A horizontal scrollable section displaying various user profiles (e.g., friends or girlfriends available for rent). Each card contains information such as:

- Profile picture
- Name
- Hobbies and interests
- Available package (duration, price, etc.)

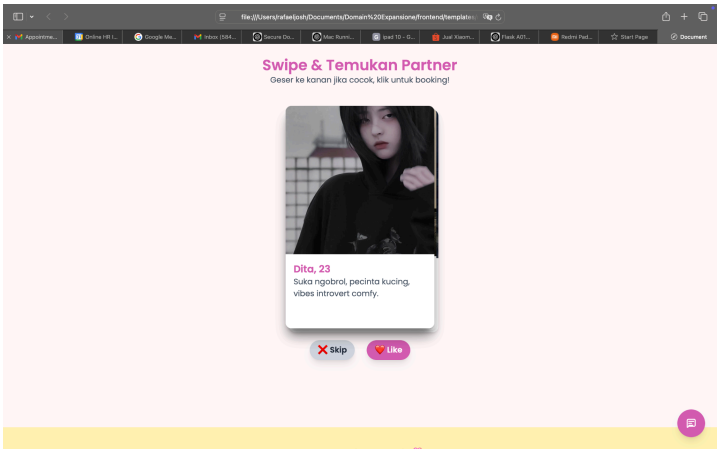
  This feature allows users to quickly explore potential companions based on shared interests.



b. **Swipe-to-Match Feature (Tinder-like)**

This section simulates a modern dating-style experience:

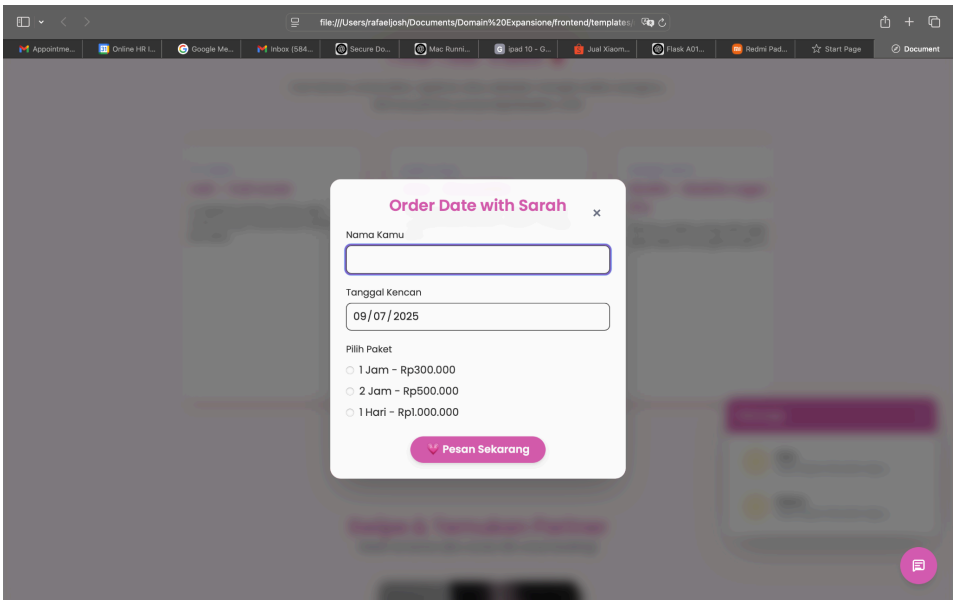
- Users are presented with one card at a time.
- Swipe right to like or swipe left to skip.



c. **Order Feature (Rent a Friend)**

If a user is interested in a particular profile, they can fill out a form to place an order (i.e., book a session with the selected friend). The order data includes:

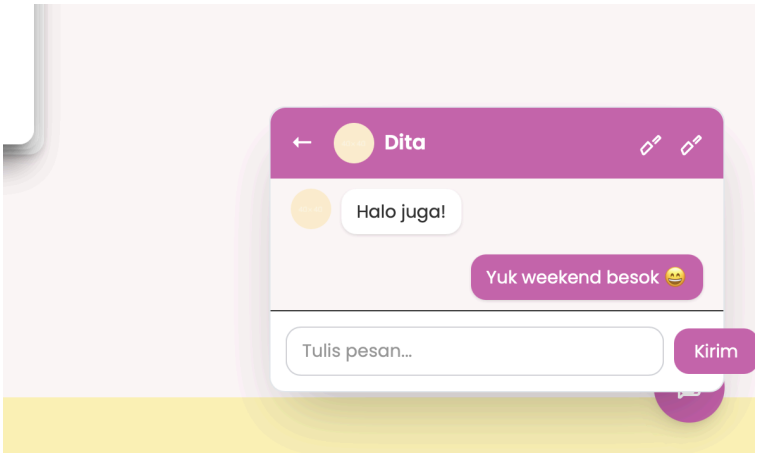
- Name of the person to be rented
- Date of booking
- Selected package
- Authenticated user info (from the token)



d. **Chatbox Feature**

At the bottom of the main page is a chat section, where users can send and receive messages. This allows:

- Post-match communication
- Casual conversations with previously rented friends
- A real-time or simulated chat experience



METHODOLOGY

This section explains the process of how the application was built, including the development flow, frameworks and tools used, as well as the implementation of security practices that align with the OWASP Top 10 standards.

A. Development Approach

The development of this web application followed a container-based architecture using Docker Compose, which allowed the frontend, backend, database, and supporting services (like NGINX and Redis) to run in isolated, yet connected environments.

The workflow was divided into two main components:

Frontend Development

Languages Used: HTML, CSS, JavaScript

Frameworks & Libraries:

- Tailwind CSS – Used for rapid and responsive UI styling.
- Flask (Python) – Used as a lightweight server framework to handle frontend routing, form handling, and integration with the backend via RESTful API calls.

Functionality:

- Handles login and registration pages.
- Displays user dashboard (main.html) after authentication.
- Dynamically shows user profiles, swipe interaction, and chat interface.
- Communicates securely with the backend using HTTPS (via NGINX).

Backend Development

Language Used: Golang (Go)

Responsibilities:

- Handles business logic like login, registration, order processing, and token generation.

- Interacts with the PostgreSQL database to perform secure data queries.
- Issues JWT tokens for session management and authentication.

Structure:

- Modular packages for handlers, auth, jwt, and redisclient.

Database

Database Engine: PostgreSQL

Structure:

- users table for user credentials and profile info.
- orders table for tracking friend rental bookings.
- Passwords are hashed using “bcrypt” before being stored.

Supporting Tools

a. NGINX:
    - Acts as a reverse proxy to handle HTTPS traffic.
    - Uses self-signed SSL certificates to secure communication over port 443.
    - Redirects HTTP (on port 81) to HTTPS automatically.
b. Redis:
    - Used for caching simple data such as usernames during login to reduce database queries.
    - Also useful for rate limiting and session control in more advanced setups.
c. Docker Compose:
    - Manages all services (frontend, backend, nginx, db, and redis) in isolated containers.
    - Enables portability and consistent development environment.

B. Security Implementation

Security is a key focus of this project. The following practices were implemented to address common vulnerabilities based on the OWASP Top 10:

1. A01:2021 – Broken Access Control

   JWT-Based Authorization:

   - After login, a JWT token is generated and stored as an HTTP-only cookie.
   - Token includes username and email to identify the user session.
   - Protected routes (e.g /main, /order) check token validity before allowing access.

```
152
153
154   JWT_ALGORITHM = "HS256"
155
156   @app.route('/main')
157   def main_page():
158       token = request.cookies.get("token")
159       if not token:
160           return redirect("/?error=Login+required")
161
162       try:
163           payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
164           username = payload.get("username", "User")
165       except Exception as e:
166           print("JWT decode error:", str(e))
167           return redirect("/?error=Invalid+token")
168
169       return render_template("main.html", user_name=username)
170
```

2. A02:2021 – Cryptographic Failures

   HTTPS with NGINX:

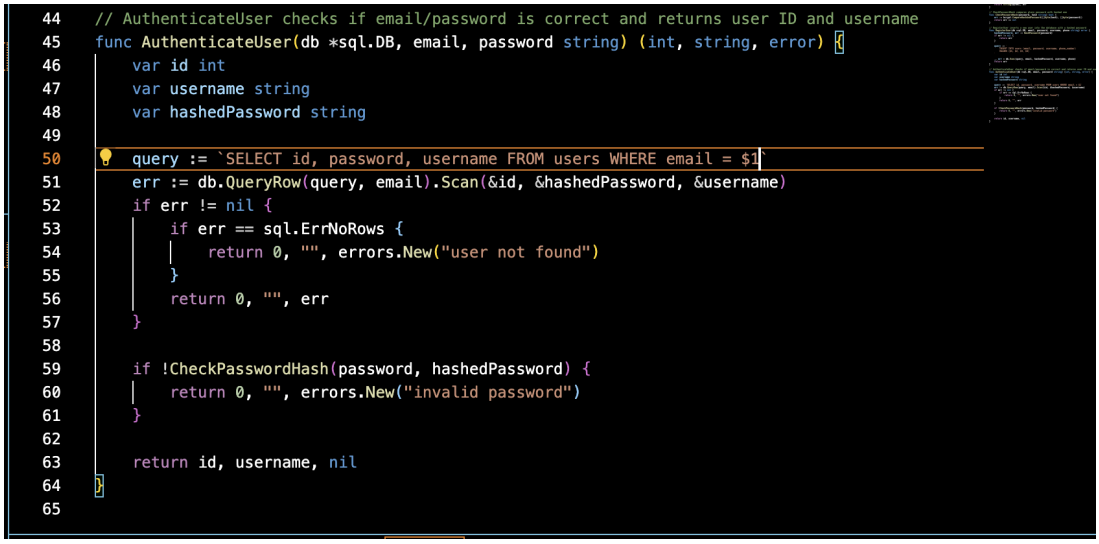
   - All frontend and backend requests are routed through HTTPS.
   - SSL certificates are used (self-signed for development).
   - HTTP requests are force-redirected to HTTPS.

nginx > nginx.conf

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    server {
        listen 81;
        server_name localhost;

        return 301 https://$host$request_uri;
    }

    server {
        listen 443 ssl;
        server_name localhost;

        ssl_certificate     /etc/nginx/certs/selfsigned.crt;
        ssl_certificate_key /etc/nginx/certs/selfsigned.key;

```

3. A03:2021 – Injection (SQL Injection)

   Prepared Statements with Parameters:
   All SQL interactions in Go use
   ? placeholders or $1, $2... to prevent SQL injection.



```
// AuthenticateUser checks if email/password is correct and returns user ID and username
func AuthenticateUser(db *sql.DB, email, password string) (int, string, error) {
    var id int
    var username string
    var hashedPassword string

    query := `SELECT id, password, username FROM users WHERE email = $1`
    err := db.QueryRow(query, email).Scan(&id, &hashedPassword, &username)
    if err != nil {
        if err == sql.ErrNoRows {
            return 0, "", errors.New("user not found")
        }
        return 0, "", err
    }

    if !CheckPasswordHash(password, hashedPassword) {
        return 0, "", errors.New("invalid password")
    }

    return id, username, nil
}
```

4. A05:2021 – Security Misconfiguration

   Docker Isolation:
   Services are isolated in Docker containers to reduce cross-service vulnerability.

Nginx Limits and Headers:
Proxy headers and limited ports are configured to prevent exposure.

5. A07:2021 – Identification and Authentication Failures

   Secure Login System:
   Passwords are hashed using bcrypt before storage.
   Authentication checks both email and hashed password.

```
14      Password string
15  }
16
17  // HashPassword hashes the plain-text password using bcrypt
18  func HashPassword(password string) (string, error) {
19      bytes, err := bcrypt.GenerateFromPassword([]byte(password), 14)
20      return string(bytes), err
21  }
22
23  // CheckPasswordHash compares plain password with hashed one
24  func CheckPasswordHash(password, hash string) bool {
25      err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
26      return err == nil
27  }
28
```

   JWT Expiry:
   Tokens are set with a 24-hour expiration limit.

6. A08:2021 – Software and Data Integrity Failures
   - Each service is built from its own Dockerfile to ensure code consistency and integrity.
   - Containers run only what is necessary for their function.
7. A09:2021 – Security Logging and Monitoring Failures
   - Application prints clear logs for important events such as login, registration, and order creation.
   - Errors are caught and redirected to the frontend with proper messages (not raw stack traces).
8. A09:2021 – Security Logging and Monitoring Failures
   - Application prints clear logs for important events such as login, registration, and order creation.
   - Errors are caught and redirected to the frontend with proper messages (not raw stack traces).

Additional Security Measures :

1. Rate Limiting (Flask):
    - Added rate limiting using Flask extensions like flask-limiter to prevent brute-force login attempts and abuse of APIs.
    - Limits are set based on IP address, reducing risk of denial-of-service attacks.

```
18
19  limiter = Limiter(
20      get_remote_address,
21      app=app,
22      default_limits=["200 per day", "50 per hour"]
23  )
24
25  @app.route('/')
26  def index():
27      error = request.args.get("error")
28      return render_template("login.html", error=error)
29
30  @app.route('/login', methods=['POST'])
31  @limiter.limit("5 per minute")
32  def login():
33      email = request.form['email']
34      password = request.form['pswd']
```

2. Environment Variable Security:
    - Sensitive credentials such as database passwords, JWT secret, and Redis settings are stored securely in .env files and injected via Docker Compose.
    - These variables are never hardcoded in the codebase, reducing risk of accidental exposure.

```
6
7   func main() {
8       redisclient.InitRedis()
9       // DB connection
0       dbHost := os.Getenv("POSTGRES_HOST")
1       dbPort := os.Getenv("POSTGRES_PORT")
2       dbUser := os.Getenv("POSTGRES_USER")
3       dbPassword := os.Getenv("POSTGRES_PASSWORD")
4       dbName := os.Getenv("POSTGRES_DB")
5
6       connStr := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s sslmode=disable",
7           dbHost, dbPort, dbUser, dbPassword, dbName)
8       db, err := sql.Open("postgres", connStr)
9       if err != nil {
0           log.Fatal(err)
1       }
2
```

CHALLENGES & LEARNINGS

A. Challenges

During the development of this secure marketplace web application, several challenges were encountered:

1. Integrating Multiple Technologies
   Combining Flask (Python), Go (for backend), PostgreSQL, Redis, Docker, and NGINX into a single, functioning ecosystem was complex. Ensuring they all communicated correctly, especially under Docker networking, required in-depth research and testing.
2. JWT Authentication Across Services
   Managing secure login sessions using JWT (JSON Web Token) between Flask and Go was a challenge, especially when encoding and decoding the tokens securely without leaking any sensitive data.
3. Implementing Security Best Practices
   Applying OWASP Top 10 practices in real-world code was more challenging than expected. Each type of vulnerability required a unique solution. For example, preventing SQL injection required the use of parameterized queries, while setting up HTTPS correctly involved configuring NGINX with SSL and testing certificate validity.
4. User Experience and Feature Integration
   Making sure that the swipe (Tinder-style) feature, order feature, and chatbox all work smoothly without breaking the authentication or user sessions posed front-end and back-end syncing issues.

B. Learnings

Despite the challenges, the project brought many valuable technical and conceptual learnings:

- Understanding Secure Web Architecture
  I learned how to structure a full-stack web application that is not only functional but also secure and scalable. Each component (frontend, backend, database, caching, reverse proxy) has its role, and understanding that helped me build a more stable application.
- Real-World Application of OWASP Top 10
  Applying security best practices based on OWASP helped me understand how attackers might exploit common flaws, and how to prevent them using concrete code — such as hashed passwords, CSRF-safe cookies, and input validation.

- Working with Docker & Microservices
  This project started as an exercise to understand Docker better. By containerizing each part of the app (frontend, backend, database, redis, nginx), I gained hands-on experience with Docker Compose and how containerized environments can simulate real-world deployments.
- Redis as a Lightweight Tool for Caching
  Redis turned out to be very helpful in storing temporary data like user login attempts or caching user profile data, reducing load on the database and improving performance.

Conclusion :

This project began as an exploration of Docker and evolved into a fully functional, secure marketplace web application with modern features like user registration, Tinder-style matching, chat functionality, and secure payment/order flow. Through this project, I have deepened my understanding of full-stack development, secure coding practices, and real-world deployment architectures.

Building a secure app involves not just writing code, but also thinking ahead about possible vulnerabilities, system interactions, and user behaviors. I also learned the importance of simplicity, documentation, and testing, all essential parts of delivering a reliable application.

In the future, I hope to expand this project by adding more robust features such as real-time notifications, stronger encryption, and CI/CD automation pipelines to further enhance both the user experience and security level.