ONLINE LAUNDRY SERVICE PLATFORM

Group Members:

- **BABAR Muhammad**
- **EGHAGHE Edoseghe George**
- **FIFEN MENEGUEM Idiatou**
- **YOUMBI NJANDA Steven Alan**
- **Anas TAGUI**
- **NGUELIEBOU NGUEUKAM Flavien Briator**

Contents

1. Project Overview

This is a full-stack web application designed to connect users with laundry service providers. The system allows users to book services, while providers can manage their offerings and orders through a secure dashboard.

2. Technology Stack

Frontend: HTML5, CSS3, JavaScript (Vanilla)

Backend: Node.js with Express.js framework

Database: MongoDB Atlas (Cloud Database)

Authentication: JSON Web Tokens (JWT)

File Handling: Multer (for image uploads)

3. Security Features

Password Encryption: Using bcryptjs to hash passwords before storing them.

Secure Authentication: JWT-based login system for session management.

Role-Based Access Control (RBAC): Different permissions for 'User', 'Service Provider', and 'Admin'.

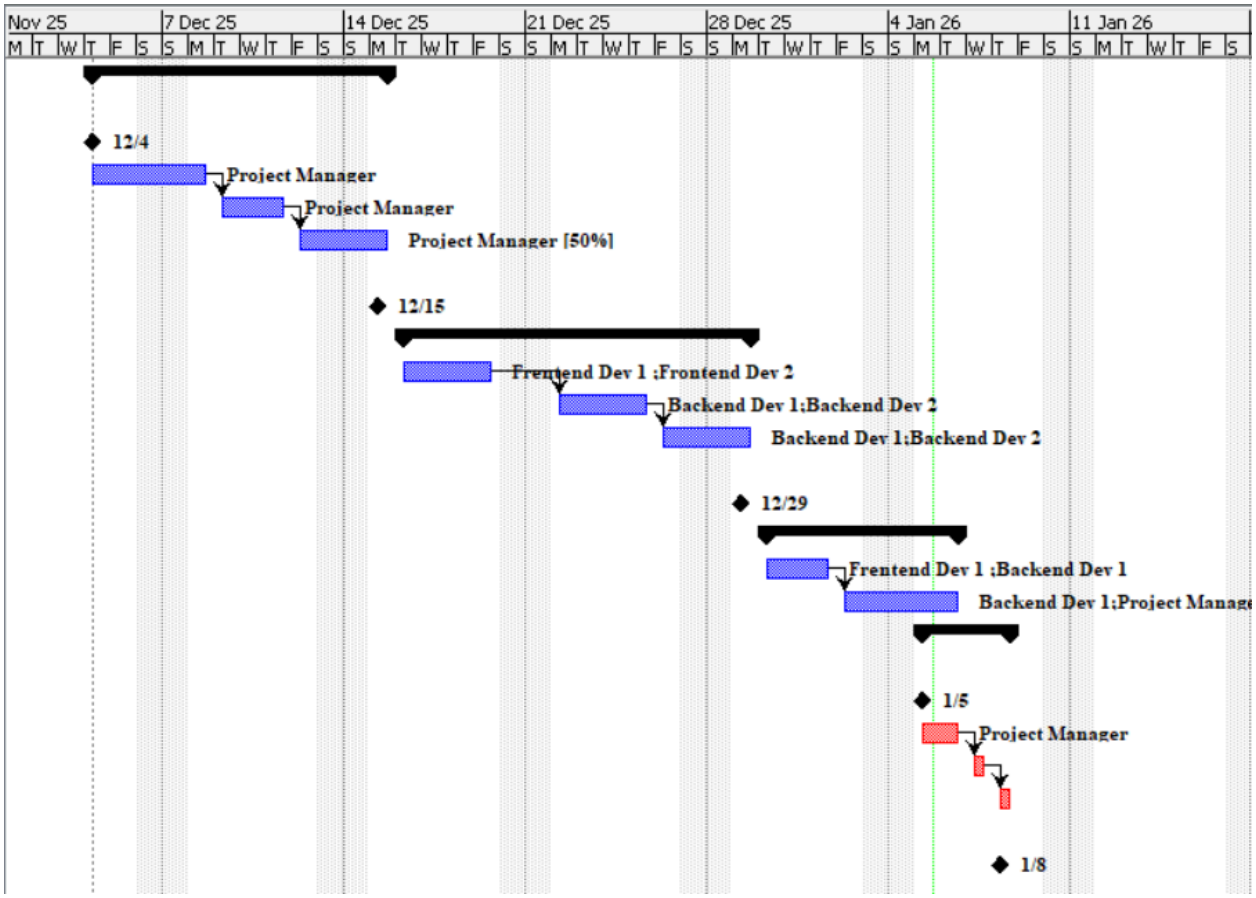
Environment Protection: Sensitive keys (API URLs, Secrets) are stored in .env files.

Protected Routes: Only authorized users can access specific actions like adding or deleting services.

4. Project Management Structure

The project followed a structured project management approach to ensure clear task allocation, timeline control, and coordination between development and security activities.

The structure defines leadership roles, development responsibilities, and security oversight to ensure smooth collaboration throughout the project lifecycle.



5. Setup & Installation Instructions

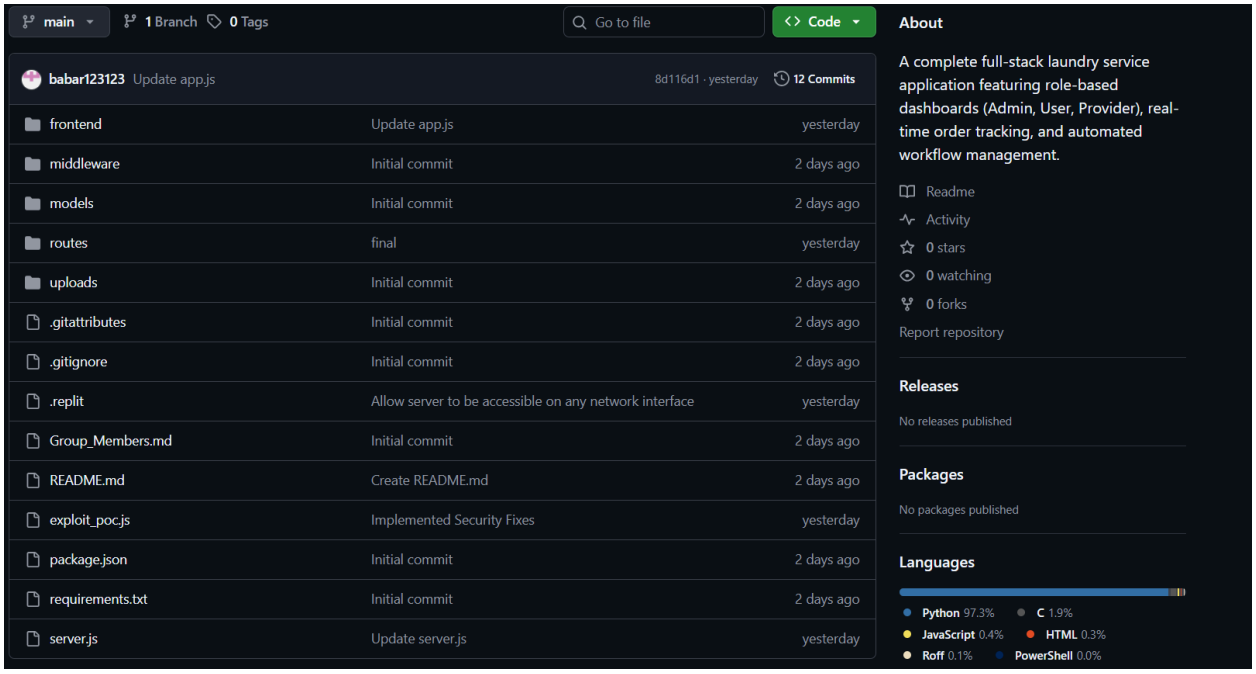
1. **Clone/Copy Project:** Transfer the project folder to the local machine.
2. **Install Dependencies:** Run npm install to download all required libraries.
3. **Environment Setup:** Configure the .env file with MongoDB URI and JWT Secret.
4. **Create Folders:** Ensure an uploads/ folder exists for image storage.
5. **Run Server:** Execute npm start or nodemon server.js to begin.

The project source code is hosted on GitHub to support version control, collaboration, and transparency.

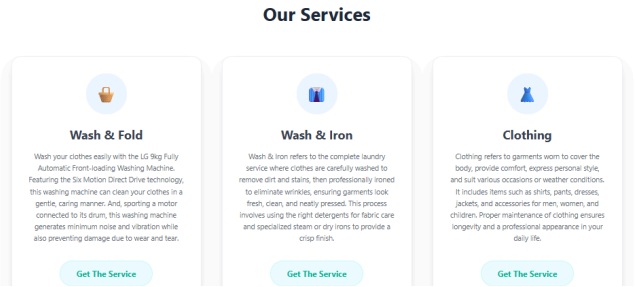
The repository contains:

- Frontend and backend source code
- Middleware, models, and routes
- Security-related implementations
- README file with setup and usage instructions

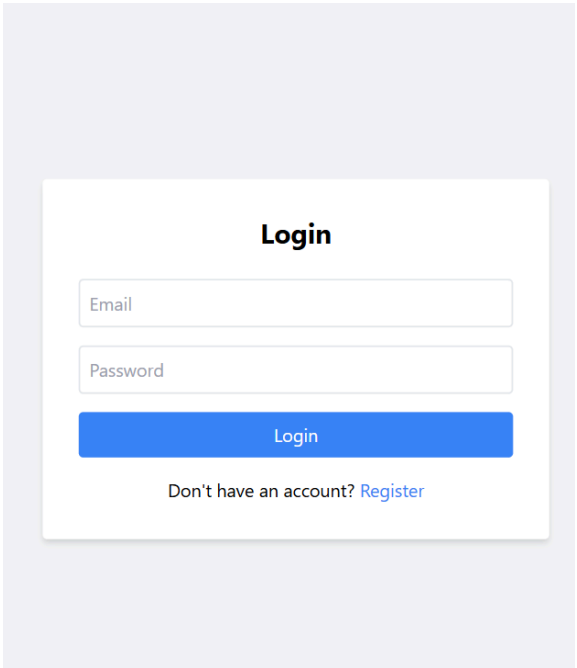
GitHub commits were used to track progress and manage updates efficiently16: https://github.com/babar123123/Smart-Laundry-Platform.git

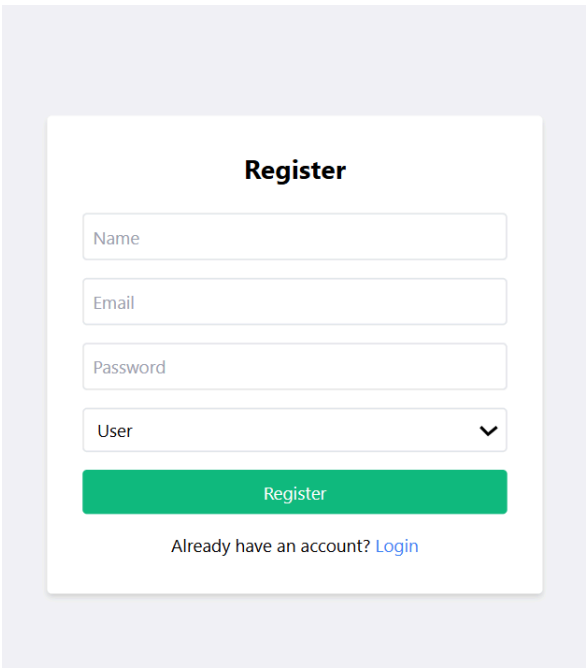


6. Frontend:



7. Login / Register





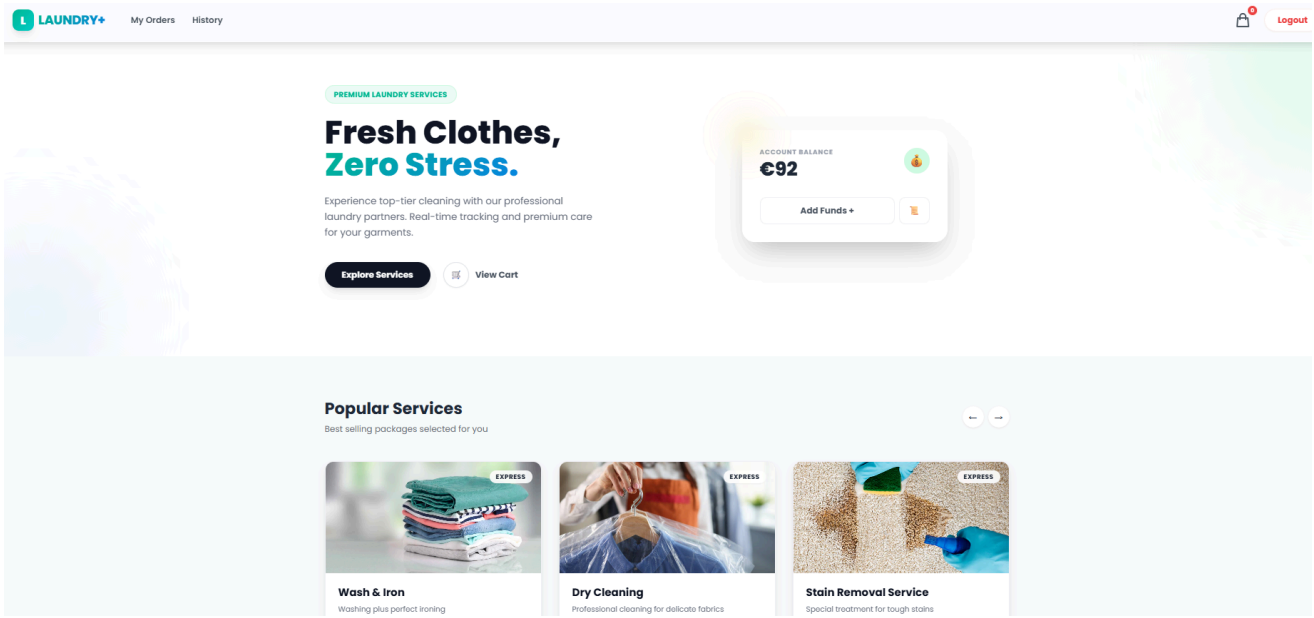
8. Team Roles and Responsibilities

Each team member had a clearly defined role combining development, management, and security responsibilities.

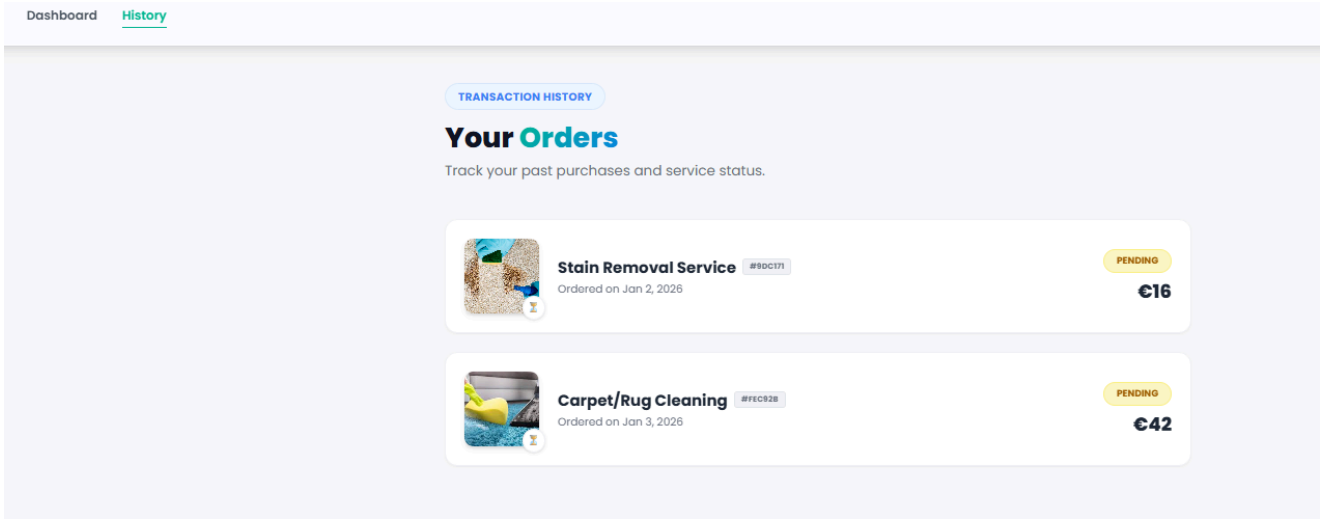
- **EGHAGHE Edoseghe George** – Project Manager & Security
- **BABAR Muhammad** – Development Lead & Security
- **FIFEN MENEGUEM Idiatou** – Project Management Structure Designer & Security
- **YOUMBI NJANDA Steven Alan** – Backend Developer & Security
- **Anas TAGUI**– Frontend Developer & Security
- **FNGUELIEBOU NGUEUKAM Flavien Briator** – Frontend Developer & Security

This role distribution ensured accountability, efficient development, and continuous security integration.

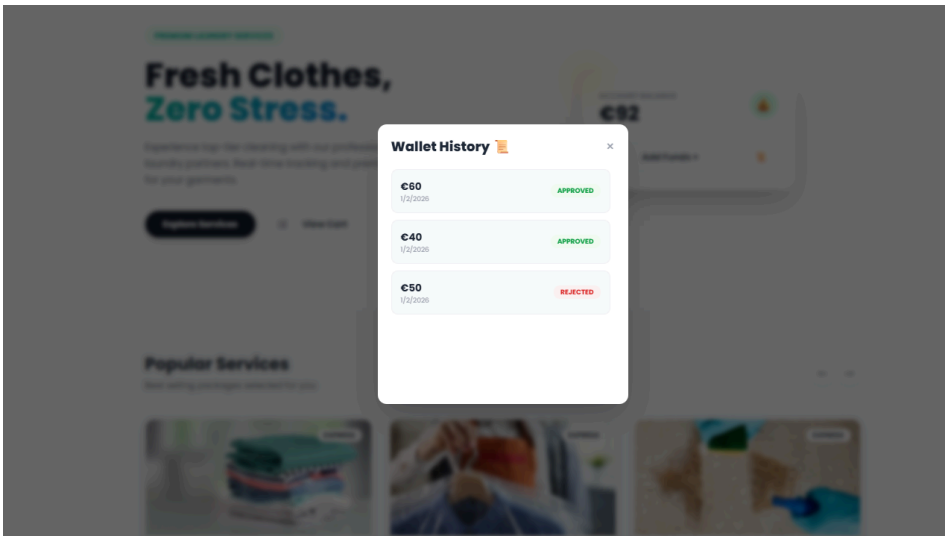
9. User Interface:



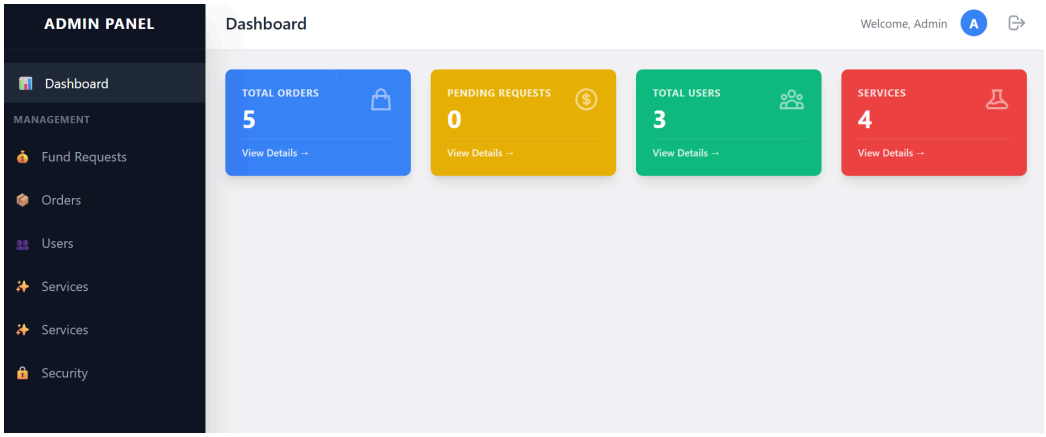
10. Order History:



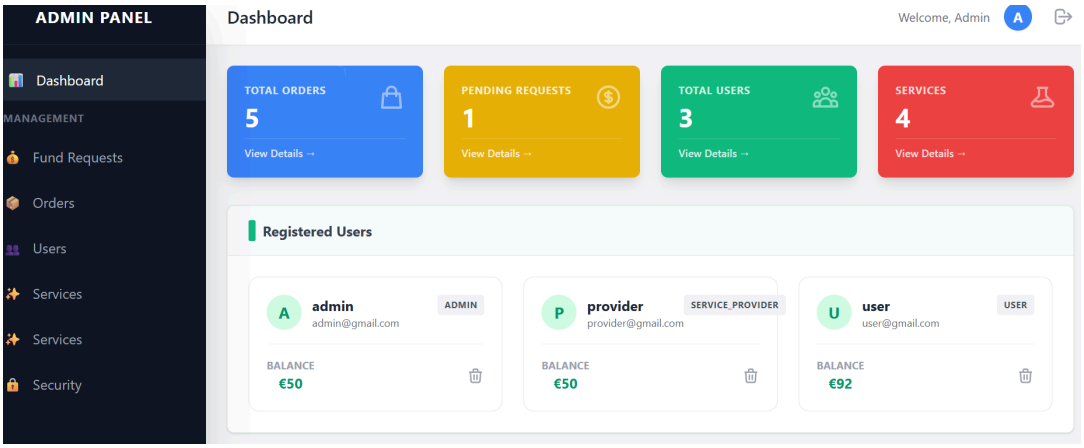
11. Balance Tracking:



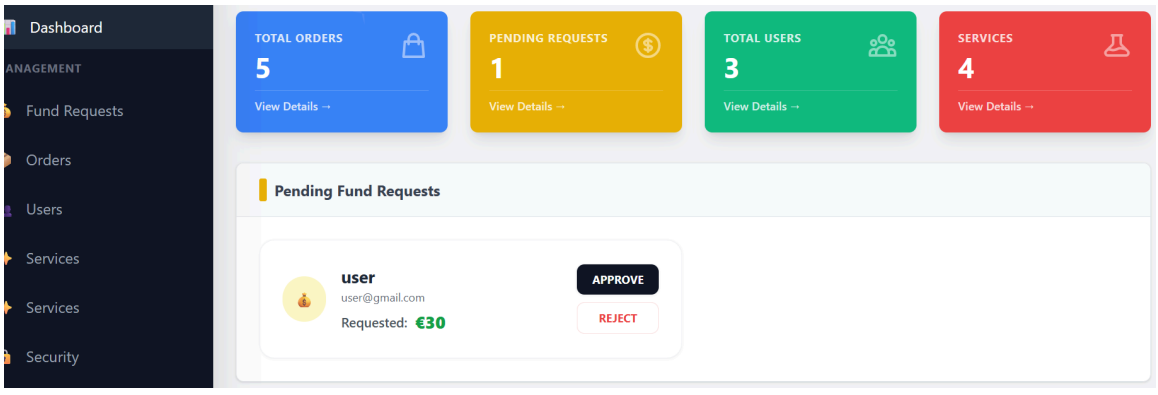
12. Admin Panel:



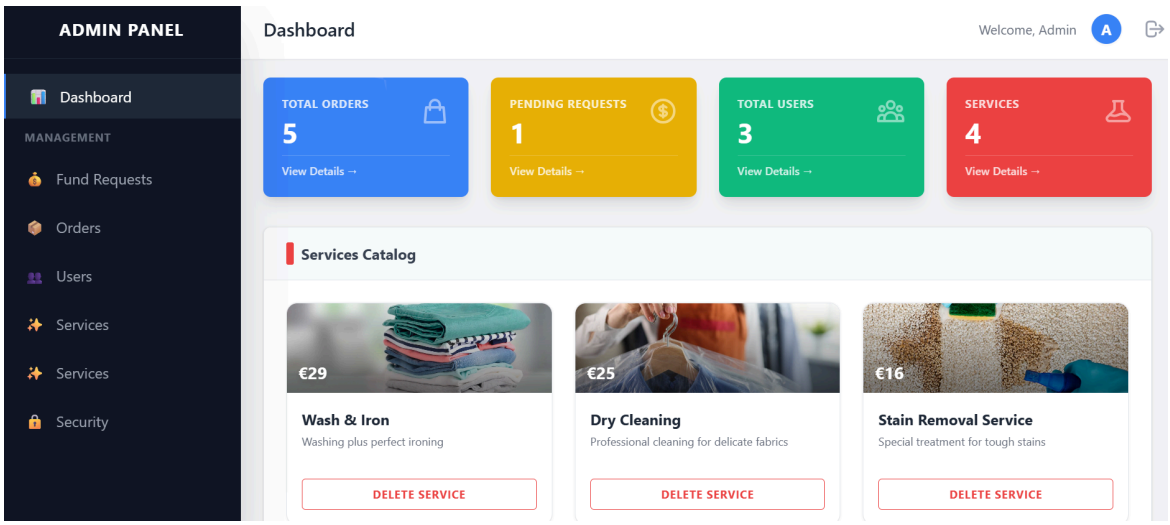
13. All User Data:



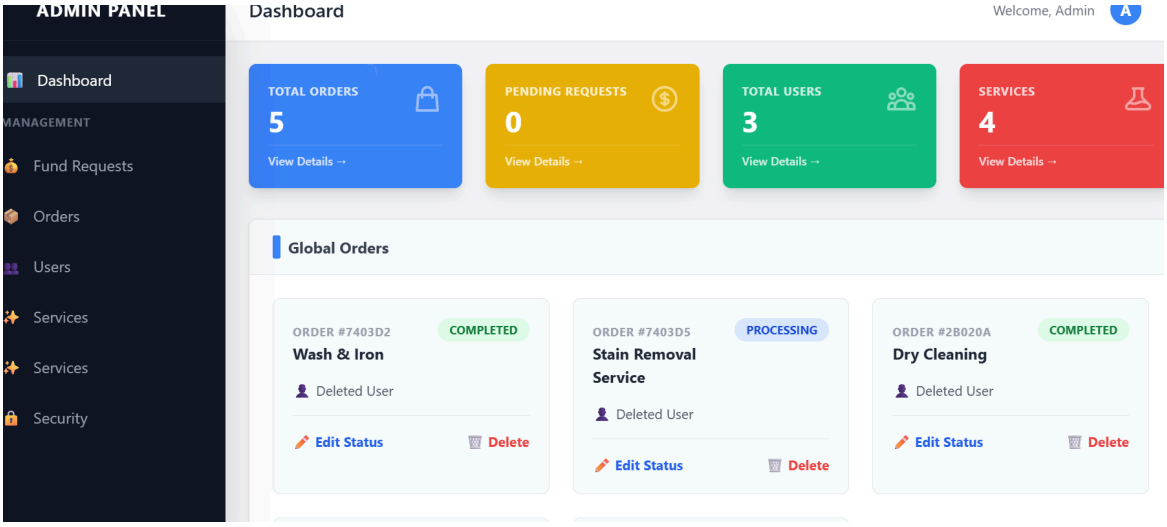
14. Pending Request:



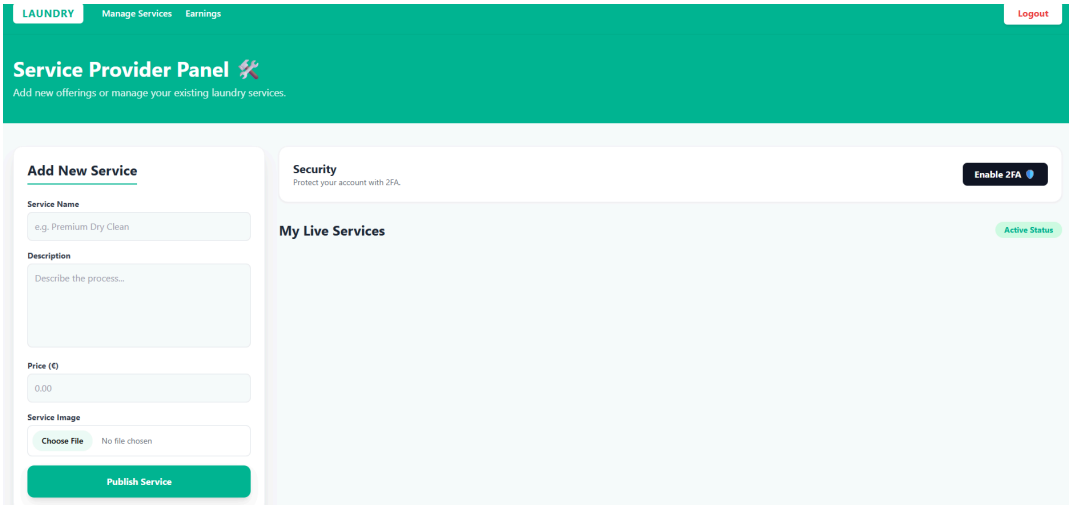
15. All Services:



16. All Orders:



17. Service Provider Interface:



18. Security Implementation Report

The Laundry Platform is designed with a Security-First approach.
Security is applied at every stage of the Software Development Life Cycle (SDLC).

The system includes:

- Multi-Factor Authentication (MFA)
- Role-Based Access Control (RBAC)
- Strong input validation

These controls help protect against the major security risks listed in the OWASP Top 10.

18.1 Authentication and Authorization

Secure Authentication

- **JSON Web Tokens (JWT)**
  Used for secure login sessions.
  Tokens are signed using a secret key stored in environment variables.

```
// Generating signed JWT token
const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1d'
});
```

- **Password Hashing (bcrypt)**
  User passwords are salted and hashed before storage.
  This protects against brute-force and rainbow-table attacks.

```
// Hashing password before saving
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
const user = new User({ name, email, password: hashedPassword, role });
await user.save();
```

- **Multi-Factor Authentication (MFA)**
  Time-based One-Time Password (TOTP) is implemented.
  Users can connect Google Authenticator for extra login security.

```
const verified = speakeasy.totp.verify({
    secret: user.mfaSecret.base32,
    encoding: 'base32',
    token: token
});

if (verified) {
    // Grant access
} else {
    res.status(400).json({ msg: "Invalid MFA Code" });
}
```

18.2 Role-Based Access Control (RBAC)

System roles:

- User
- Service Provider
- Admin

Access is enforced using middleware:

- auth
- adminAuth

These ensure users only access features they are allowed to use.

```
const auth = async (req, res, next) => {
    try {
        const token = req.header('Authorization')?.replace('Bearer ', '');
        if(!token) return res.status(401).json({ msg: "No token" });

        const decoded = jwt.verify(token, process.env.JWT_SECRET);
        req.user = await User.findById(decoded.id).select('-password');
        next();
    } catch (err) {
        res.status(401).json({ msg: "Token is not valid" });
    }
}

const adminAuth = (req, res, next) => {
    if(req.user.role !== 'admin') return res.status(403).json({ msg: "Admin access
denied" });
    next();
```

18.3 Application Security Controls

Input Validation and Sanitization

express-validator validates user inputs
(registration and login forms)

Prevents:

- malformed data
- injection attacks

Mongoose schema enforces correct data types

```
router.post('/register', [
    check('name', 'Name is required').not().isEmpty(),
    check('email', 'Please include a valid email').isEmail(),
    check('password', 'Password must be 6 or more characters').isLength({ min: 6 })
], async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });
    // ... proceed
});
```

`

Security Headers (Helmet.js)

Helmet is used to set secure HTTP headers:

- prevents clickjacking
- reduces XSS risk

- adds protocol-level protections

Content Security Policy (CSP) allows only trusted CDNs such as Tailwind CSS.

```
const helmet = require('helmet');
// Secure headers with CSP configured for Tailwind CDN
app.use(helmet({
  contentSecurityPolicy: false,
}));
```

Rate Limiting

express-rate-limit is configured

- Maximum: 100 requests per 15 minutes per IP
- Protects against:
    - brute force login attempts
    - DoS attacks

```
const rateLimit = require('express-rate-limit');

// Rate Limiting (100 requests per 15 mins)
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100
});
app.use(limiter);
```

18.4 Data Protection and Infrastructure

Secret Management

- Sensitive data (DB URI, JWT secrets, API keys) stored in:
    - .env file
- No secrets are written in source code
- A past hard-coded secret was fixed and moved to environment variables

Secure Communication

- Application runs over HTTPS/TLS
- Encryption protects data during transfer

- Prevents Man-in-the-Middle attacks

Database Security

- MongoDB connection strings are stored securely
- Mongoose is used for queries
- Parameterized queries help prevent NoSQL injection

```
// Loading secrets from .env file
require('dotenv').config();

// Using the secret (never hardcoded)
mongoose.connect(process.env.MONGO_URI);
const decoded = jwt.verify(token, process.env.JWT_SECRET);
```

19. Segmentation Analysis

In network security, the term micro-segmentation is commonly used in Zero Trust architectures, especially in cloud and distributed environments. Since the Laundry Platform is currently deployed in a monolithic architecture on a single server, traditional network micro-segmentation does not directly apply.

However, the application implements strong logical segmentation at the application level, which effectively separates users, roles, and data. This provides many of the same security benefits.

19.1 Role-Based Segmentation (RBAC)

The platform strictly separates users based on roles:

- **Admin Segment** – can manage users and view global orders
- **Service Provider Segment** – can add and manage services
- **User Segment** – can place orders and view only their own history

Access control is enforced using middleware such as:

- auth
- adminAuth

These ensure that ordinary users cannot access admin-level routes (for example, /api/admin).

19.2 Data Segmentation (Isolation)

Data in the database is logically isolated:

- When a user calls /api/orders/my, the backend checks that:
    - The user can only access their own orders
    - A user cannot access another user's data, even if they guess an order ID

This is enforced using the authenticated user ID (req.user._id) in database queries.

19.3 Modular Code Segmentation

The codebase is separated into logical modules to reduce risk and improve maintainability:

- routes/userRoutes.js
- routes/serviceRoutes.js
- routes/orderRoutes.js

If one module encounters an error, the rest of the system remains unaffected conceptually, supporting fault isolation and maintainability.

20. Conclusion

The Laundry Platform has been developed with a strong focus on security at every level of the system. By implementing secure authentication, role-based access control, encrypted communication, secure coding practices, and strict data protection controls, the platform is well-protected against common cyber threats.

Continuous monitoring, regular security audits, and timely updates will further ensure that the platform remains secure, reliable, and trustworthy for all users and service providers.