

Courier Management System — Project Report

Author: [Your Name]

Course: [Course Name]

Supervisor: [Supervisor Name]

Date: 2025-11-04

Abstract

This project implements a Courier Management System — a web application that allows users to create courier shipments, make payments, track deliveries, and receive notifications. Admins can manage shipments and assign delivery agents, while agents can view assigned shipments and mark them delivered. The system uses Flask (Python) with SQLAlchemy for the backend and MySQL for persistent storage.

Objectives

- Build a functional web application for creating and tracking courier shipments.
- Provide role-based access (User, Admin, Agent).
- Implement payment flow with server-side and client-side validations.
- Provide notifications (email and SMS) with a DB-backed fallback.
- Ensure reliable tracking history and minimal duplication through DB procedures/triggers.

Key Features

- User registration and login.
- Create courier shipments (sender & receiver details, addresses, weight, type).
- Payment page with validation (credit/debit/UPI) and server-side checks.
- Admin dashboard: list couriers, assign agents, view payments.
- Agent dashboard: view assigned couriers, mark delivered.
- Courier tracking history stored in `Courier_tracking`.
- Notifications: email and SMS (configurable in DB); in-app persistence if external services unavailable.
- Database stored procedures, functions, views and triggers to centralize event handling.

Technologies Used

- Python 3.11
- Flask (web framework)
- Jinja2 (templating)
- SQLAlchemy (ORM)
- MySQL (mysql-connector)

- Bootstrap 5 (UI)
- Optional: Twilio (SMS), SMTP for email

System Architecture (high-level)

- Client (browser): interacts with Flask server via HTML forms and JS validation.
- Server (Flask): handles routes, validation, session-based auth, calls stored procedures for key operations, sends notifications.
- Database (MySQL): stores domain data (Courier, Payments, Courier_tracking, Delivery_agent, Users) and contains stored procedures, functions, views, and triggers to ensure consistent tracking and enable reporting.

Database Design (summary)

Main tables:

- ``Courier`` — stores shipments (cid, sender/receiver details, addresses, billno, agentid, priceid, date)
- ``Courier_tracking`` — timeline of status updates for a courier (trackid, cid, status, current_location, updated_at)
- ``Payments`` — payment information (pid, cid, uid, amount, payment_mode, payment_status, transaction_date)
- ``Delivery_agent`` — delivery agents (agentid, name, email, phone)
- ``Notification_config`` — SMTP/Twilio settings (admin-editable)

Stored routines created:

- ``sp_mark_payment_completed(p_cid)`` — mark payments as Completed and insert ``Payment Received`` tracking if needed.
- ``sp_assign_agent(p_cid, p_agentid)`` — set ``agentid`` for courier and insert an assignment tracking entry if changed.

Stored functions:

- ``fn_payment_status(p_cid)`` — returns payment status.
- ``fn_last_tracking_status(p_cid)`` — returns latest tracking status.

Views:

- ``vw_courier_summary`` — summary per courier (billno, amount, payment_status, last_status, agentid).
- ``vw_agent_assignments`` — per-agent assigned-count summary.

Triggers:

- ``trg_payments_after_insert`` — AFTER INSERT on ``Payments``, inserts ``Payment Received`` tracking when appropriate (deduplicates using last status and 120s window).
- ``trg_courier_after_update_agent`` — AFTER UPDATE on ``Courier``, inserts assignment/unassignment tracking entries similar to above.

Notes: Triggers and procedures include deduplication logic to avoid duplicate ``Courier_tracking`` rows when both app and DB insert similar events.

Implementation Highlights

- `app.py` contains Flask routes for all pages (index, login/register, dashboard, create_courier, payment, agent/admin dashboards, assign, update status, notify helpers).
- A context processor `inject_now()` provides `now()` to templates for footer timestamps.
- Payment validation:
 - Client-side: JS ensures digits-only card entry (16 digits), expiry format, CVV checks, UPI validation.
 - Server-side: validates card digits (16), expiry against IST `ist_now()`, CVV length. Calls `sp_mark_payment_completed` to finalize payment.
- Notifications: `notify_parties()` constructs email/SMS messages using DB-backed configuration; `send_email()` and `send_sms()` handle external sends where configured. The DB stores notification settings via `admin_notification_config` route.
- Stored routines centralize key operations. The app was updated to call procedures for payment completion and agent assignment.

How to Run Locally (development)

1. Ensure MySQL is running and a database `courierdb` exists. Adjust `app.config['SQLALCHEMY_DATABASE_URI']` in `app.py` if needed.

2. Install Python dependencies (example):

```
pip install -r requirements.txt
```

(If no requirements file, install Flask, SQLAlchemy, mysql-connector-python, werkzeug)

3. Set environment variables for optional SMTP/Twilio if needed:

```
$env:SMTP_SERVER='smtp.example.com'; $env:SMTP_PORT='587'; $env:SMTP_USERNAME='user'; $env:SMTP_PASSWORD='password'
```

4. Run the app:

```
python app.py
```

5. Open <http://127.0.0.1:5000> in a browser.

Testing & Verification

- Manual tests performed:
 - Create courier → verify `Payments` row created (Pending) and `Courier_tracking` initial Pending entry.
 - Payment page validation (client + server) → call `sp_mark_payment_completed` and verify `Courier_tracking` has 'Payment Received'.
 - Assign agent via admin dashboard → `sp_assign_agent` called, courier `agentid` updated and tracking entry added.
 - Notifications: `admin_notify_test` and `notify_test/` endpoints to exercise email/SMS sending; if external not configured, messages are logged and persisted (in-app fallback).

Security Considerations

- Passwords hashed using Werkzeug. Do not store raw passwords.
- Card details are NOT stored (CVV is not persisted). For production use, integrate with a PCI-compliant payment gateway (Stripe/PayU) and store only tokens.
- Notification credentials in `Notification_config` are stored in DB; consider encrypting these fields or using a secrets manager.

Limitations

- Payments are simulated (no real gateway). Integration required for production.
- Timezone handling: app uses IST, DB uses NOW(); consider aligning using CONVERT_TZ or setting DB timezone.
- Potential duplication: app and DB triggers/procedures may both insert tracking rows; dedupe logic mitigates but centralizing one approach is recommended.

Future Work

- Integrate a payments gateway and tokenization.
- Add automated tests (unit + integration) and CI pipeline.
- Add a user notifications history page and admin audit page.
- Improve secrets handling (encryption or secrets manager).
- Add analytics (delivery times, agent performance).

Appendix: Useful SQL commands

- List triggers:

```
SELECT TRIGGER_NAME FROM INFORMATION_SCHEMA.TRIGGERS WHERE TRIGGER_SCHEMA = DATABASE();
```

- Show CREATE for an object:

```
SHOW CREATE PROCEDURE sp_mark_payment_completed\G
SHOW CREATE TRIGGER trg_payments_after_insert\G
SHOW CREATE VIEW vw_courier_summary\G
```

End of report.