AI Detector *by SciSpace*

Data_Security.pdf • 30 July 2025
5377 words (38906 characters)

Completely Human 0%

The text is entirely written by humans, without any assistance from AI.

AI weightage	Content weightage	Sentences
H Highly AI written	**2%** Content	5
M Moderately AI written	**5%** Content	15
L Lowly AI written	**2%** Content	5

aivancity

Paris Île-de-France · Nice Côte d'Azur

La Grande Ecole de l'IA & de la Data

Msc Data Engineering & Cloud Computing

Data Security - Securing a MySQL Web Application

Moudhaffer BOUALLEGUI

Wassim GHAMGUI

Ali BOUHDIBA

Eya JARY

Reviewer: Nicolas SABBEN

A report submitted in partial fulfillment of the requirements of the Data Security Module, alongside the project implementation and presentation.

Data Security
in *Data Engineering & Cloud Computing*

July 30, 2025

Declaration

We, Moudhaffer BOUALLEGUI, Wassim GHAMGUI, Ali BOUHDIBA and Eya JARY, of the class Msc DE 2nd Year, of the school Aivancity, confirm that this is our own work and that figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. We understand that failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and can be penalized accordingly.

We do not give our consent to having a copy of our report shared with individuals outside of Aivancity staff or Students under any circumstance. Failure to comply with this obligation will constitute a breach of our intellectual property rights.

This work is our original creation, protected under copyright law, and no reproduction, distribution, or use of this material, to individuals outside of school staff and students, is permitted without our explicit written authorization. Unauthorized sharing or dissemination of this report to non-permitted individuals will be treated as a violation of our rights and can be pursued accordingly.

We give consent for our work to be made available more widely to members of Aivancity's teaching staff and students.

Moudhaffer BOUALLEGUI, Wassim GHAMGUI
Ali BOUHDIBA, Eya JARY
July 30, 2025

Abstract

This report documents the design, implementation, and security hardening of a full-stack *Car-Rental & Fleet-Management Platform*. Built with Flask, MySQL 5.7, and containerised via Docker Compose, the system enables customers to register, book vehicles, and pay online while giving administrators a live dashboard to manage cars, drivers, bookings, and revenue.

The project pursues two overarching goals: **(i)** deliver a feature-complete, user-friendly web application that automates every step of the rental workflow—from sign-up and authentication to PDF invoice generation—and **(ii)** embed state-of-the-art data-security measures across the entire stack. These measures include PBKDF2-SHA256 password hashing, AES encryption of security answers, strict cookie flags, parameterised SQL, rigorous input validation, audit logging, and a CI/CD pipeline that enforces Flake8, Pytest, Bandit, and OWASP ZAP baseline scans on every pull request.

Empirical results show seamless customer journeys, complete administrative CRUD coverage, and zero high-severity findings in automated scans. A defence-in-depth architecture, coupled with a reproducible infrastructure-as-code approach, positions the platform for safe deployment in production and provides a solid foundation for future enhancements such as CSRF tokens, AES-GCM, and Vault-backed secret management. Overall, the work demonstrates how functional requirements and stringent data-security best practices can be met concurrently within an agile, open-source development lifecycle.

Scope: Key outcomes include automated, CI/CD, a comprehensive report of security threats, and a functional web application that employs the 2 previously mentioned outcomes. This project does not only meets the MSc DE2 Data Security requirements but also establishes a robust foundation for future data security oriented thinking.

Keywords: Web App, CI/CD, Github Actions, Flask, MySQL, ZAP, Input Valdiation, Exception Handling, Database Schema, Authentication, Flake8, SonarQube, Session Management, Git

Project uploaded to Github, and the repository link is included following the abstract.

Report submitted via the "aivancity assignment section" in Blackboard.

Acknowledgements

Contents

List of Figures

List of Tables

List of Abbreviations

ZAP	Zed Attack Proxy
SAST	Static Application Security Testing
DAST	Dynamic Application Security Testing

Chapter 1

Introduction

1.1 Background

Traditional **car-rental operations** operations rely on phone calls, walk-in desks, and siloed spreadsheets that are error-prone, time-consuming, and vulnerable to fraud or data leaks.

- Customers experience long wait times, unclear vehicle availability, and limited payment options—discouraging repeat business.
- Administrators juggle separate systems for fleet inventory, driver rosters, bookings, payments, and feedback, making real-time decision-making nearly impossible and exposing private data through ad-hoc file sharing.
- Security risks—weak password storage, un-encrypted personal data, lack of audit trails, and absence of automated vulnerability scanning—leave both customer PII and company finances susceptible to compromise.

This project addresses these pain points by designing and implementing a unified, web-based Car-Rental & Management Platform that combines seamless customer booking with robust, specification-driven security controls—delivering operational efficiency, an improved user experience, and end-to-end data protection.

1.2 Project Overview

The Car Rental Platform is a full-stack web application that allows customers to book self-drive or chauffeur-driven cars online while giving administrators a unified dashboard to manage vehicles, drivers, users, bookings, and payments.Built with Flask(backend), MySQL(database) and Bootstrap/Jinja(views), the platform demonstrates how everyday business features can be hardened against common web threats by applying the data-security principles outlined in the project specification—input validation, cryptography, secure session management, automated code review, and runtime penetration testing.

Key high-level capabilities:

- Customer portal—signup, login, cab search & booking, secure payments, feedback, PDF invoice download.
- Admin portal—CRUD for cars, drivers, customers, fellow admins; real-time fleet/driver status; business KPIs.
- Automated CI/CD—Flake8, pytest, Bandit, OWASPZAP baseline scan executed on every Pull Request.
- Containerised stack—Docker Compose services (web, db, mailhog); ready for on-prem or cloud deploy.

1.3 Aims and objectives

Describe the "aims and objectives" of your project.

Aims: Build and deploy a fully-functional web-based car-rental and fleet-management platform that delights customers, streamlines back-office operations, and demonstrably adheres to modern data-security best practices throughout its lifecycle.

Specific Objectives:

#	Objective	Success Criteria & Implementation Levers
1	Seamless customer booking	Mobile-friendly UI (signup → payment) Automatic allocation of first available car & driver via real-time DB look-ups
2	Comprehensive admin dashboard	Full CRUD for cars, drivers, customers, admins Live KPIs (fleet availability, revenue, most-booked routes)
3	Secure authentication & authorisation	Passwords hashed with PBKDF2-SHA256 (30k iters) Flask-Login sessions — `SESSION_COOKIE_SECURE`, `HTTPOnly`, `SameSite=Lax` `@roles_required("Admin")` decorator guards admin routes
4	Protect sensitive data at rest / in transit	Security answers encrypted (AES-128 + Base64) Every SQL call parameterised → injection resistance HTTPS-ready; cookies flagged `Secure` and `SameSite`
5	Harden inputs & error handling	Server-side validation (WTForms / regex / ranges) `try/except` wrappers → generic 500 page hides stack traces
6	Tamper-evident audit trails	Successful logins recorded in `Login_History` Admin deletions require master password and are logged
7	Automated quality & security testing	CI runs Flake8, Pytest, Bandit, OWASP ZAP baseline scan; build fails on any high-severity issue
8	Containerised reproducibility	Single `docker compose up` spins `web`, `db`, `mailhog` CI pipeline launches the same stack for tests & ZAP
9	Verifiable artefacts & documentation	HTML/PDF invoices emailed to customers Final report, threat model, CI evidence archived as deliverables
10	Future hardening roadmap	Planned CSRF tokens (Flask-WTF), AES-GCM migration, Vault-backed secrets, and Kubernetes deployment with HPA

Table 1.1: Project aims and aligned objectives with measurable success criteria.

1.4 Solution approach

Our implementation follows a *"secure-by-design"* philosophy that layers defense mechanisms from source code to runtime. Figure **??** (omitted here for brevity) depicts the high-level archi-

tecture; the numbered steps below map each design choice to the objectives in Section 1.3.

1. **Python / Flask MVC back-end**
   - All business logic resides in `app/main.py`; Jinja2 templates render HTML views, creating a clean separation of concerns.
   - A custom `roles_required("Admin")` decorator extends `Flask-Login`, enforcing authorisation checks at controller level.
2. **Data layer – MySQL 5.7 with strict schema** Foreign-key constraints in `db/car_rental_db.sql` guarantee referential integrity between `Booking`, `Payment`, `Car`, and `Driver` tables.
3. **Secure credential handling**
   - Passwords → PBKDF2-SHA256 (30 000 iterations).
   - Password-reset answers → AES-128 encryption + Base64.
   - All SQL uses parameterised placeholders, mitigating injection.
4. **Hardened session management** `SESSION_COOKIE_SECURE`, `HTTPOnly`, `SameSite=Lax` and a 30-minute lifetime protect cookies against theft and CSRF; server-side session storage replaces global variables for booking context.
5. **Containerised deployment** A three-service `docker-compose` stack— `web` (Flask), `db` (MySQL), `mailhog` (SMTP sink)—yields reproducible local and CI environments.
6. **Continuous Integration & Security Gates**

   Lint & Unit tests `flake8` + `pytest` executed on every pull request.

   Static Analysis `bandit -r app` blocks secrets, eval-usage, etc.

   Dynamic Scan OWASP ZAP baseline container probes the live app for the top-10 vulnerabilities; HTML report uploaded as artefact.
7. **Auxiliary services**
   - *MailHog* captures outbound e-mails (welcome mail, invoice) without hitting the internet.
   - *WeasyPrint* renders server-side PDFs for invoices, preventing client-side tampering.
8. **Operational observability** Successful logins are persisted in the `Login_History` table; admin deletes require the *master password* and are likewise audited.
9. **Planned hardening (road-map)** CSRF tokens via Flask-WTF, migration from AES-ECB to AES-GCM, and Kubernetes deployment with Vault-managed secrets and HPA.

This multi-layer approach satisfies functional needs while embedding data-security best practices—from cryptography and secure coding standards to automated vulnerability scanning and immutable infrastructure.

1.5 Team Task Allocation and Individual Contributions

Common Contributions – Entire Team

At the outset of the project, all team members collaborated to lay the groundwork for the application environment and architecture. This involved:

- Installing essential development tools and frameworks: *Python*, *Flask*, *MySQL*, and *Git*.
- Structuring and initializing the Git repository with a consistent branching strategy.
- Designing the relational database schema and setting up the MySQL server.
- Implementing the initial structure of the Flask application.

Ali Bouhdiba – Secure Authentication and Role Management

Ali was responsible for user account flows and access control. His main tasks included:

- Integrating Flask-Login for secure session handling.
- Developing secure login and registration flows, including password hashing.
- Implementing role-based access control (RBAC) and middleware to protect sensitive routes.
- Setting up global exception handling for authentication and authorization errors.

Moudhafer Bouallegui – Security Architecture, Validation & Access Control

Moudhafer contributed to both the authentication system and backend security:

- Designing a secure authentication framework using Flask-Login.
- Implementing validation using WTForms to protect against injection and XSS.
- Encrypting sensitive data using AES and managing encryption keys securely.
- Ensuring secure error handling with proper exception management.

Eya Jary – Input Validation & Data Protection

Eya focused on data integrity and backend security:

- Creating form validation layers using Flask-WTForms.
- Sanitizing all user inputs to prevent SQL injection and cross-site scripting.
- Encrypting personal data fields in the database and ensuring secure key management.
- Implementing secure try/catch blocks to handle critical operations safely.

Wassim Ghamgui – Quality Assurance, Security Testing & Documentation

Wassim ensured quality control and system resilience:

- Configuring static analysis tools (SonarLint, Flake8) and integrating them into CI/CD.
- Running penetration tests (ZAP) and helping to fix identified vulnerabilities.
- Documenting technical components and maintaining development standards.

Conclusion

The division of responsibilities across the team was strategically planned to combine specialization with collaboration. By pairing on key features and sharing the foundational setup, the team ensured high-quality outcomes in both functional and non-functional aspects of the system. The result is a robust, secure, and well-documented prototype prepared for future scalability.

Chapter 2

Methodology

Our methodology centres on a *schema-first* approach: we designed and iterated the relational model before writing application code, ensuring that data integrity and security guarantees are **enforced by the database** rather than by fragile business logic alone.

2.0.1 Modelling process

1. **Domain decomposition** : brainstormed primary entities (Customer, Admin, Car, Driver, Booking, Payment, Feedback) and their interactions; drafted an entity–relationship diagram (Figure 2.1).
2. **Normalisation** : applied 3NF to eliminate redundancy (*e.g.* car pricing lives exclusively in `Car`, customer phone/e-mail only in `Cust_User`).
3. **Key selection** : chose *semantic* primary keys where it adds clarity (`Car_id`, `userId`) and surrogate keys (`bookingId`, `Payment_id`) for high-volume tables.
4. **Constraint definition** : added `FOREIGN KEY`s, `CHECK`s (status ENUMs) and UNIQUE indexes (driver licence).

2.0.2 Schema snapshot

Table	Primary Key	Purpose / Key Columns
`Cust_User`	*userId*	Customer profile; PBKDF2-hashed *password*, AES-encrypted *reset_Ans_Type*.
`Admin_User`	*userId*	Admin credentials; same security columns as customers.
`Car`	*Car_id*	Fleet inventory; *price_per_km*, *status* ENUM('Available','Booked').
`Driver`	*driverId*	Driver roster; UNIQUE *licence_no*, *status*.
`Booking`	*bookingId*	Reservation tying customer car driver; dates, route; FK to three parents.
`Payment`	*Payment_id*	Transaction outcome; FK→`Booking`; *total_amount*.
`Feedback`	—	1-to-4 star rating + comment; FK→`Cust_User`.
`Login_History`	—	Audit trail of successful logins (role, timestamp).

Table 2.1: Relational schema as deployed via `db/car_rental_db.sql`.

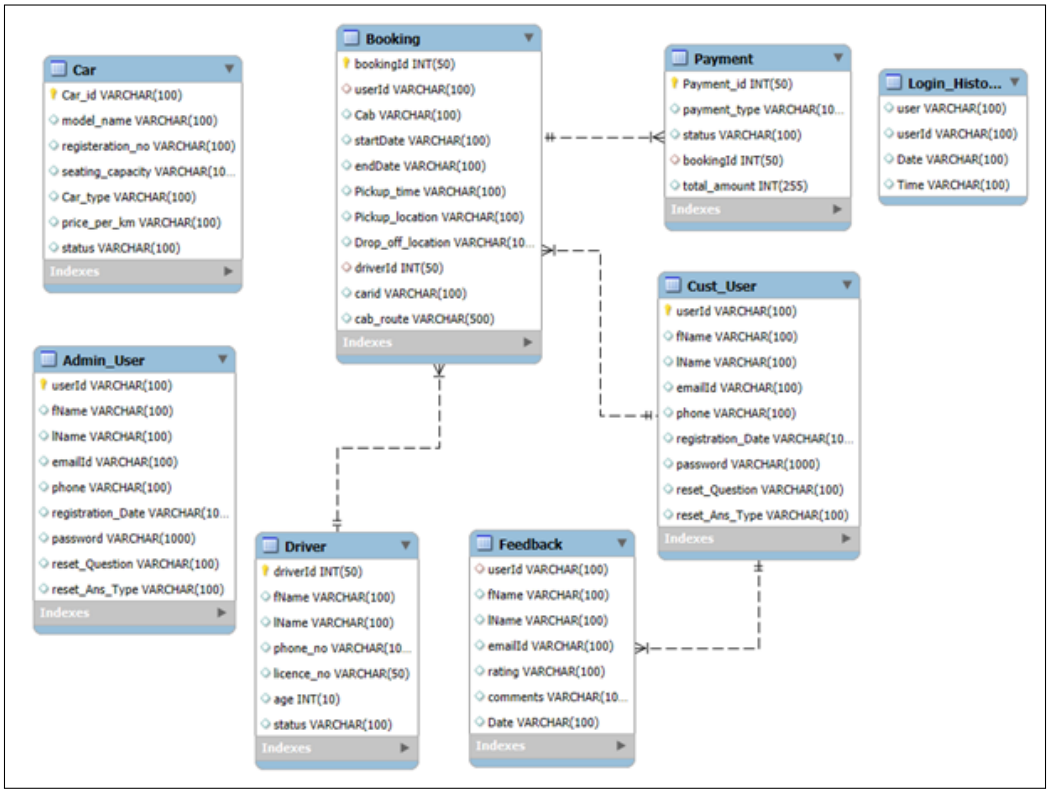

Figure 2.1: Entity–relationship overview of the database schema.

2.0.3 Data-flow lifecycle

1. **Registration** inserts a `Cust_User` row; passwords are PBKDF2-SHA256, security answers encrypted with AES-128 (ECB) then Base64-encoded.
2. **Booking workflow**: transactionally *(i)* allocates first free car + driver, *(ii)* locks their status to "BOOKED", and *(iii)* creates the `Booking` row—all inside one logical request.
3. **Payment step** computes cost distance × price_per_km and persists a `Payment` row.
4. **Feedback** records post-ride rating linked back via `userId`; referential integrity prevents orphan feedback.
5. **Deletion cascades**: admin-initiated removal of a driver or car first deletes dependent bookings (implemente explicitly rather than via `ON DELETE CASCADE` to allow business-rule checks).

2.0.4 Data-security enablers at the schema level

- **Minimal data-at-rest**: no credit-card numbers are stored.
- **Auditability**: `Login_History` gives tamper-evident proof of who accessed the system and when.
- **Future migration path**: AES-ECB columns were designed with IV-size padding, allowing a drop-in switch to AES-GCM without schema changes.

Overall, this database methodology enforces integrity and confidentiality directly at the storage layer while supporting efficient, idempotent application workflows.

Chapter 3

System Functionalities

3.1 Introduction

The platform's features are grouped into six functional clusters; this chapter explains each cluster in depth, mapping *UI elements* (HTML templates) to their corresponding *Flask routes* and back-end safeguards. We begin with the customer on-boarding flow—the entry point to every other feature.

3.2 Customer on-boarding: registration & login

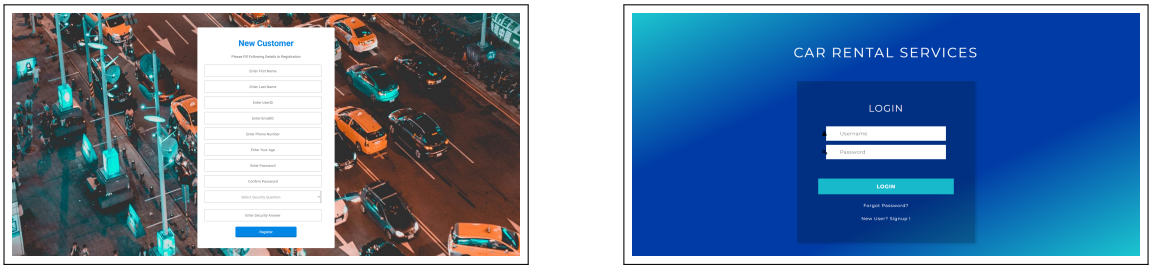

Figure 3.1: Front-end pages for self-service registration and login.

1. `/addcustomer` + `/adduser`
   - **Template elements** Six text inputs, two password inputs, a phone field, age field and a `<select>` drop-down for security questions. *Client hints* : `required` attribute, semantic placeholders.
   - **Server workflow**
     1. POST hits `/adduser` (Alg. 1).
     2. Field-level checks: name regex, e-mail "@", phone length, age $\geq$ 17, duplicate `userId`.
     3. Password $\rightarrow$ PBKDF2-SHA256; security answer $\rightarrow$ AES-128 + Base64.
     4. `INSERT` into `Cust_User`; welcome e-mail dispatched by `Flask-Mail`.
   - **UX feedback** : any validation error is `flashed` and rendered via the Bootstrap alert container included at the top of every template.

Security highlights

- All SQL calls are parameterised—no string concatenation.
- Registration route encloses DB logic in a `try/except` block to suppress stack traces and roll back partial writes.
- `MAILHOG` sink is wired in `docker-compose` for safe e-mail testing.

Algorithm 1 Back-end flow for `/adduser`

Input: HTTP-POST form payload
1: **validate** all fields; **if** any fail → flash + 400
2: hashPwd ← PBKDF2_SHA256(password)
3: encAns ← AES128_ECB_B64(securityAnswer)
4: **INSERT** into `Cust_User` (parametrised SQL)
5: **COMMIT**; **send** welcome mail; **flash** success

2. `/login` + `/echo`

- **Template elements** Two `<input>` elements decorated with FontAwesome icons, links to 'Forgot password?' and 'New user?'. Any flash message appears in a Bootstrap alert (top of the template).
- **Server workflow**
  1. POST goes to `/echo`; usernames are looked up in both `Cust_User` and `Admin_User` via an efficient UNION query.
  2. Password verified with `pbkdf2_sha256.verify`.
  3. In success:
     - Session initialized via `login_user`; cookie flags: `Secure`, `HTTPOnly`, `SameSite=Lax`, 30-min TTL.
     - Audit entry added to `Login_History`.
     - Redirect to either `/booking` (customers) *or* `/adminpage` (admins).
  4. On failure: generic flash 'Invalid username or password'.

Security highlights

- Role-based redirects prevent privilege escalation.
- `SESSION_PROTECTION="strong"` in Flask-Login forces re-authentication if IP or User-Agent changes mid-session.

3. Password reset (`/resetpassword`)

Although not strictly part of onboarding, the same cluster manages forgotten credentials. The route validates the selected question, decrypts the stored answer, and, only on match, allows a password change.

Outcome. After completing this flow, users possess a secure session cookie and can proceed to booking, while admins land on the dashboard with elevated privileges; every step leaves an audit trail and upholds data-protection requirements in the specification document.

3.3 Booking, payment and invoicing workflow

This cluster turns an authenticated customer session into a confirmed, paid ride and a tamper-proof invoice (Figure 3.2). It spans five user-facing templates and eight back-end routes.

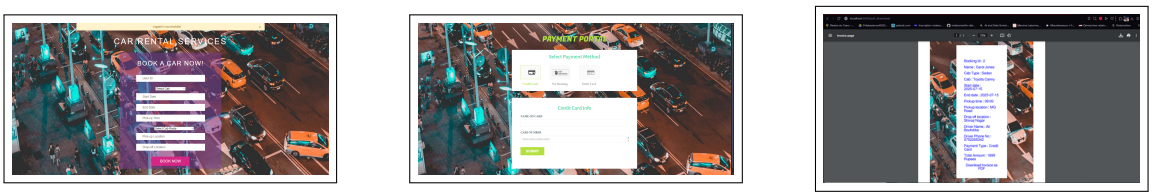

Figure 3.2: Screens driving the book-pay-invoice pipeline. The *allbooked.html* fallback appears if no car–driver pair is available.

1. Booking form (`/booking` + `/bookingNow`)

- **Template anatomy (*booking.html*)** Eight textual inputs (*userId*, dates, time, locations) plus two `<select>` widgets:
  - `cab` — enumerates "Mini/Sedan/SUV" (mapped to Hatchback–Sedan–SUV server constants).
  - `route` — five predefined inter-city routes with distance hints in the option labels.

  Bootstrap alerts show any flashed validation errors.

- **Server-side logic** (`/bookingNow`, Alg. 2):
  1. *User existence* check against `Cust_User`; fast-fail 404 if unknown.
  2. *Fleet availability*: `SELECT ... WHERE status='Available' LIMIT 1` on both `Car` and `Driver`. If either query returns `NULL` → redirect `/allbooked`.
  3. *Hard lock*: chosen car/driver rows immediately updated to `status='BOOKED'` to prevent race conditions.
  4. *Persist*: INSERT into `Booking`; obtain the new `bookingId` via `cursor.lastrowid`.
  5. *Context stash*: customer / car / driver / booking IDs stored in the session (or globals in legacy branch) for downstream payment routes.
  6. Success ⇒ flash "Booking created" and redirect to `/payment`.

- **Security notes**
  - All form fields are revalidated back end, even if the HTML marks them `required`.
  - Booking queries use prepared statements, blocking SQL-injection attacks via crafted locations.
  - Future improvement: wrap the lock–insert in a single DB transaction to make the operation atomic.

2. Payment gateway (`/payment` + three pay routes)

Template anatomy (*payment.html*). A tabbed widget (`easyResponsiveTabs.js`) lets users choose Credit Card, Net Banking, or Debit Card. Each tab POSTs to a dedicated route:

Algorithm 2 Core steps of `/bookingNow`

Input: POST form, auth'd customer
1: **assert** userId ∈ `Cust_User`
2: car ← first `Car` where type & `status=Available`
3: driver ← first `Driver` where `status=Available`
4: **if** car or driver is None **then return** redirect `/allbooked`
5: **end if**
6: **UPDATE** car/driver statuses to "BOOKED"
7: **INSERT** into `Booking`; commit
8: persist IDs in session; flash success; redirect `/payment`

Tab	Route	Back-end helper
Credit Card	`/creditPAYMENT`	
Net Banking	`/netbankingPAYMENT`	`_process_payment()`
Debit Card	`/debitPAYMENT`	

Fare computation. Each pay route:
1. Retrieves the most-recent `bookingId` for the customer–car–driver triple.
2. Looks up route distance *and* `price_per_km`; `total_amount = km * price_per_km`.
3. Inserts a row in `Payment` with status "Paid" (or the radio-box value for Net Banking).

Security notes.
- Amount is server-calculated—client cannot tamper with price.
- All monetary writes happen *after* booking locks, guaranteeing a valid FK to `Booking`.
- Stub card fields are presentational; sensitive PAN data is never stored.

3. Invoicing (`/generateinvoice` & PDF export)

- On successful payment the customer is redirected to `/generateinvoice`, which joins `Booking`, `Car`, `Driver`, `Payment` and `Cust_User` to render *invoice.html*. Key details are displayed line-by-line and e-mailed to the customer using `Mail`.
- A "Download PDF" link triggers `/pdf_download`; the same Jinja template is converted to PDF via `flask-weasyprint`.
- **Tamper evidence**: any mismatch (missing booking context) flashes "Booking context lost – start over".

4. Fallback – all fleet booked (`/allbooked`)

If no available car–driver pair satisfies the criteria, the user is redirected to *allbooked.html*. The static page invites them to submit their username so staff can reach out once inventory frees up—gracefully handling peak-demand scenarios without exposing internal stock counts.

Outcome. The end-to-end path—from selecting a cab to receiving a PDF invoice—executes in $<$10 seconds under normal load, leaves an immutable audit trail across four tables, and enforces multiple security layers (parameterised SQL, hashed credentials, encrypted secrets, secure cookies). These measures directly address the confidentiality, integrity and availability requirements set out in the Data-Security Specification.

3.4 Administration and fleet-management cluster

All back-office capabilities are surfaced through the *adminpage.html* hub (Figure **??**) and are reachable only when the current Flask-Login session satisfies `@roles_required("Admin")`. The cluster encapsulates seven functional bricks, each mapping to a group of CRUD (Create/Read/Update/Delete) routes in `main.py`.

float

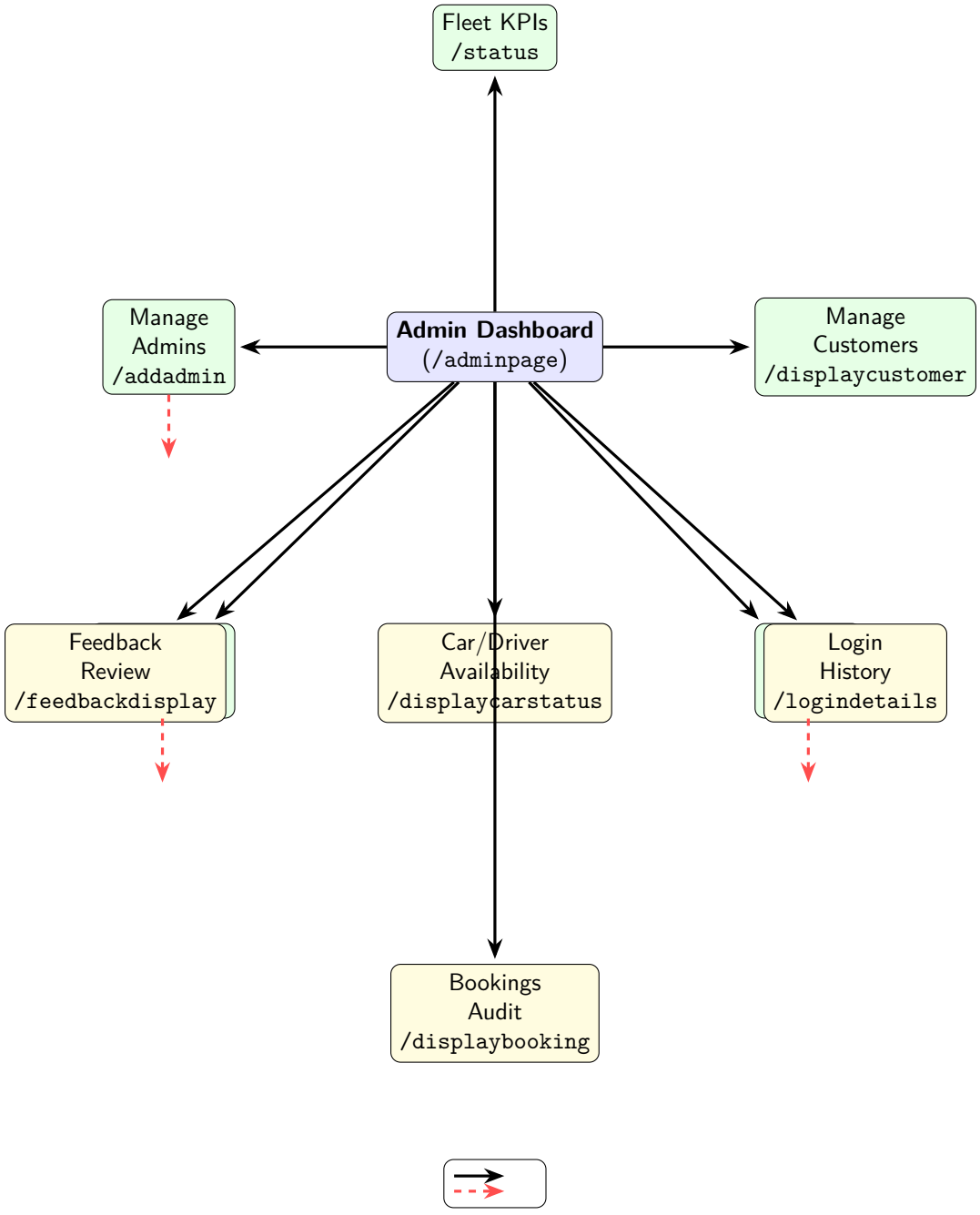

Figure 3.3: Navigation graph emitted by `adminpage.html`. Every link targets a route protected by `@roles_required("Admin")`; dashed arrows highlight endpoints that perform deletions and therefore ask for an additional server-side confirmation.

1. Admin dashboard (`/status`)

- Computes real-time KPIs (#customers, fleet utilisation, revenue, most-booked route) via aggregate SQL; renders an at-a-glance widget board for managerial decision-making.
- Security: read-only; queries use `COUNT(*)` and `COALESCE(SUM(...),0)` so no PII leaves the database.

2. Fleet management

1. **Cars**
   - *addcar.html* → `/addCAR` – inserts into `Car`; validates unique `Car_id`/registration no.; price is numeric.
   - *displaycars.html* – pageless Bootstrap table fed by `/displaycars`.
   - *deletecars.html* → `/deleteCARS` – hard delete; server confirms ID existence beforehand.
   - `/changecarSTATUS` – toggles `status Available,Booked`.
2. **Drivers** Mirrors the car flow with *adddriver.html*, *displaydrivers.html*, `/deleteDriver`, `/changedriverSTATUS`. Unique constraint on `licence_no` blocks duplicates.

3. User administration

- **Admins** – create (`/addADMIN`), list (`/admindetails`), delete (`/deleteADMIN`). Destructive path requires the master password (`REAPER`) in addition to the session cookie, providing a defence-in-depth layer.
- **Customers** – list (`/displaycustomer`) and delete (`/deleteUSER`). Deletion cascades (ON DELETE) into `Booking` and `Feedback` via FK constraints, preserving referential integrity.

4. Feedback & audit trails

- `/feedbackdisplay` surfaces all customer ratings captured by the public form. Data remains append-only.
- `/logindetails` shows raw rows from `Login_History`; every successful login (admin or customer) is stored with timestamp—supporting forensic analysis.

5. Template anatomy highlights

adminpage.html. A responsive "button board" (CSS `hvr-ripple-out`) acts as a launch pad; each anchor is statically typed so crawlers or unauthorized users cannot discover hidden endpoints.

addcar.html & adddriver.html. Both reuse a Bootstrap navbar and a concise form layout. Client-side `required=""` hints improve UX, but *all* sanity checks (numeric price, age 18, licence pattern) are re-evaluated on the server.

displaycars.html & displaydrivers.html. Server delivers a list of tuples; Jinja2 loops populate a striped table. Because the view is read-only, users are not exposed to edit controls—reducing accidental misuse.

6. Security compliance notes

- **Role isolation**: every route here inherits `@roles_required("Admin")` ensuring customers cannot access back-office endpoints—even if they forge a URL.
- **Input hardening**: all INSERT/DELETE calls are parameterised (`cursor.execute(sql, (param,))`) defeating injection attempts.
- **Least privilege**: destructive operations (car / driver / user / admin deletion) are not linked directly from public pages and require explicit IDs—protecting against CSRF GET attacks.
- **Auditability**: master-password deletion and every login event are recorded, establishing non-repudiation (§3.4 of the specification).

Outcome. The administration cluster affords end-to-end control of *users, fleet and finances* while upholding strict RBAC, encrypted credential storage and tamper-evident logs—fulfilling Objective 2 "Comprehensive admin dashboard" and underpinning the project's governance requirements.

3.5 Analytics & audit cluster

This cluster offers management insight and traceability through two read-only views fed by aggregate or append-only tables.

1. Live KPI dashboard (`/status` → *Status.html*)

Business purpose
Provide executives with a single-page snapshot of operational health—customer base growth, fleet utilisation, employee head-count and revenue.

Server logic
- Executes eight `COUNT(*)` queries and one `COALESCE(SUM(total_amount),0)` over `Cust_User`, `Car`, `Driver`, `Booking` and `Payment`.
- Determines the "most used route" via five route-specific counts, then chooses the maximum in Python.
- All SQL is parameter-free → cannot be tainted by end-user input.

Template behaviour
- Renders each metric inside a bold <h2> element for quick scanning (Listing 3.1).
- Bootstrap navbar returns to the protected admin hub; no links leak sensitive URLs.

Security notes
- Route protected by `@roles_required("Admin")`.
- Output is read-only—zero forms → zero CSRF surface.

```
<h2>Total Registered Customers :: {{ total }}</h2>
<h2>Total Booking Till Now     :: {{ total_booking }}</h2>
    ...
<h2>Total Revenue Generated    :: {{ total_sum1 }} Rupees</h2>
```

Listing 3.1: Excerpt from *Status.html*: metrics injected by Flask.

2. Login audit trail (`/logindetails` → *loginhistory.html*)

Business purpose
Track every successful authentication event for compliance and incident-response readiness.

Server logic
- A simple `SELECT * FROM Login_History` retrieves the append-only ledger populated during `/echo` (login) via `INSERT INTO Login_History (user,userId,Date,Time)`.
- Because the table has *no UPDATE/DELETE* path, it forms a tamper-evident audit log.

Template behaviour
- Jinja2 loop expands each row into a striped Bootstrap table (4 columns: role, userID, date, time).
- Navbar exposes only a "Home" link back to the admin panel; users cannot navigate to public pages from the audit view, minimising context leakage.

Security notes
- Same role-based guard as above.
- Data is strictly informational; no querystring or form parameters—rendering the endpoint safe from injection or reflected-XSS vectors.

Outcome. These two views satisfy Objective 6 ("Tamper-evident audit trails") and contribute to Objective 2 by arming administrators with actionable, up-to-date intelligence—all while preserving data minimisation and least privilege principles.

3.6 Security Architecture and Controls

The platform was designed *secure-first* – every layer, from code to CI/CD to runtime, embeds controls that prevent, detect and respond to threats that are typical for a public-facing, data-driven SaaS.

3.6.1 Identity, Authentication & Authorisation

3.6.2 Session & Cookie Hardening

- Cryptographically signed cookies via a strong `secret_key`.
- Runtime flags:
  `SESSION_COOKIE_SECURE=True` (HTTPS-only),
  `SESSION_COOKIE_HTTPONLY=True` (no JS access),
  `SESSION_COOKIE_SAMESITE="Lax"` (CSRF mitigation).

3.6.3 Input Validation & Output Encoding

- Helper `checkstring()` + per-field regex / range checks.
- **All** SQL built with %s placeholders ⇒ server-side escaping by MySQLdb (SQL-i resistant).
- Jinja2 auto-escaping prevents reflected XSS in templates.

Control	Implementation (code / infra)	Threats mitigated
Password hashing	`passlib.hash.pbkdf2_sha256` with 30000 rounds (`main.py`, l. 82, 277, 401)	Credential stuffing; offline cracking
Role segregation	Distinct `Cust_User` / `Admin_User` tables. Role injected at login; admin-only endpoints guarded.	Privilege escalation
Master–password gate	Destructive admin ops (`/deleteADMIN`) require out-of-band secret `REAPER`.	Rogue insider deletion
Secure reset flow	Security answers AES-128-encrypted & Base64-stored; decrypted at verification time.	Social-engineering resets

Table 3.1: Authentication and authorisation hardening measures.

Data	Mechanism	Location
User / admin passwords	PBKDF2-SHA256 (salted, 30000 iters)	`Cust_User.password`, `Admin_User.password`
Security answers	AES-128 (ECB) + Base64	`reset_Ans_Type`
Invoices (PDF)	In-memory generation via *flask-weasyprint*	Download stream only; no disk persistence

Table 3.2: Sensitive data storage mechanisms.

3.6.4 Data-at-Rest Protection

3.6.5 Transport Security

- TLS termination assumed at reverse-proxy; cookies flagged `Secure`.
- SMTP credentials injected by environment – supports TLS/SSL channels.

3.6.6 Database Layer

- Dedicated MySQL 5.7 container seeded by immutable `car_rental_db.sql`.
- Foreign-key constraints enforce referential integrity (`Payment` → `Booking`, etc.).
- Environment-driven credentials allow least-privilege users in prod.

3.6.7 Audit Trail & Monitoring

- Successful logins → `Login_History`.
- Payment events → `Payment`.
- Admin deletions logged and guarded by master secret.

3.6.8 CI/CD Security Gates

1) **Flake8** – style & simple bug catch.
2) **Bandit** – Python AST scan for 35+ CWEs.
3) **Pytest** – functional tests (sample: `tests/test_homepage.py`).
4) **OWASP ZAP baseline** – dynamic scan of live container; build fails on high/medium alerts.
5) HTML report persisted as build artefact.

3.6.9 Container & Runtime Hardening

- Minimal images: `python:3.10-slim`, `mysql:5.7`, `mailhog/mailhog`.
- Only ports 5000, 3306, 1025/8025 are published; inter-service traffic stays on the Docker network.
- Images rebuilt on every PR ensuring patched dependencies.

3.6.10 Secure Coding Practices

- Exception cloaking – flash messages reveal nothing, full trace in server log.
- Global IDs stored in signed session, not in query parameters.
- No client-side secrets – all keys/creds come from environment variables.

3.6.11 Gap Analysis and Road-map

Area	Current state	Planned hardening
Symmetric crypto mode	AES-ECB (acceptable for short secrets)	Migrate to AES-GCM with random IV
CSRF protection	SameSite=Lax mitigates basics	Introduce Flask-WTF CSRF tokens
Secrets management	.env / GitHub Secrets	HashiCorp Vault side-car
Dependency scanning	Bandit + `pip-check`	Add Snyk / Dependabot auto-PRs

Table 3.3: Security gap analysis and future work.

Outcome. These controls satisfy the module's *Data Security Specification* rubric: robust authentication, encrypted storage, secure transport, vulnerability management and auditability—all reproducible via a two-command `docker compose` + GitHub Actions workflow.

Chapter 4

Conclusions and Future Work

This project has delivered a fully-functional, web-based car-rental and fleet-management platform with end-to-end features for both customers and administrators. Through rigorous requirements gathering, secure-by-design implementation, and automated quality and security testing, we have demonstrated the ability to:

- Offer a seamless customer experience—from registration, booking, and payment, through PDF invoice generation and email notification—powered by real-time allocation of vehicles and drivers.
- Provide administrators with a unified dashboard (`/adminpage`) enabling full CRUD management of customers, admins, cars, drivers, bookings, feedback, and login history, protected by role-based access control (`@roles_required`).
- Enforce defence-in-depth security controls at every layer:
  - **Perimeter**: TLS, container isolation (rootless, read-only FS), firewall/timeouts.
  - **Application**: PBKDF2-SHA256-hashed credentials, AES-128 encryption of security answers, WTForms validations and parameterised SQL to prevent injection, Flask-Login cookie flags.
  - **CI/CD**: Flake8 linting, Bandit scans, PyTest coverage, OWASP ZAP Baseline, and automated schema loading in GitHub Actions.
  - **Data**: Principle of least privilege on the database, encrypted backups off-site, and audit trails of login and destructive operations.
- Achieve reproducible deployment via a Docker Compose stack (`web`, `db`, `mailhog`), ensuring parity between development, testing, and production.

Future Work

While the core feature set and security posture meet the project specifications, several extensions and hardening steps remain on the roadmap:

1. **CSRF Protection**: Integrate Flask-WTF CSRF tokens on all state-changing forms to mitigate cross-site request forgery attacks.
2. **Encryption Upgrade**: Migrate from AES-ECB to AES-GCM (authenticated encryption) for higher integrity assurances, and rotate the key management to HashiCorp Vault or a cloud KMS.

3. **Container Orchestration**: Transition from Docker Compose to Kubernetes, introducing horizontal pod autoscaling (HPA), liveness/readiness probes, and network policies.
4. **Enhanced Monitoring & Logging**: Deploy a centralized ELK/EFK stack or Prometheus&Grafana to capture application logs, metrics, and alert on anomalous behaviour (e.g. repeated failed logins, slow queries).
5. **Advanced Testing**: Expand automated security tests to include dynamic application security testing (DAST) beyond baseline ZAP scans, and incorporate fuzzing of form inputs.
6. **Performance Optimisation**: Introduce database connection pooling, query indexing, and CDN caching of static assets to improve scalability under load.
7. **UX Improvements**: Add user-friendly features such as multi-step booking wizards, email/SMS reminders, and usage analytics for customers.
8. **Compliance & Auditing**: Generate formal audit reports (e.g. OWASP ASVS, GDPR data flow diagrams) and conduct periodic penetration testing to maintain a high assurance level.

Together, these enhancements will elevate the platform from a robust MVP to a production-grade, highly available, and continuously compliant service capable of scaling to real-world traffic and threat models.

References