CarConnect - Case Study Report

**Project Title: CarConnect, a Car Rental Platform**

**Submitted by:** Siva Balan T

# INTRODUCTION

# In the modern era of digital transformation, the automobile rental industry demands efficient, reliable, and scalable systems to manage vehicles, customers, reservations, and administrative operations. CarConnect is a Python-based backend system designed to simplify the core operations of a vehicle rental platform. It integrates modular service layers, a relational database, and unit testing to ensure correctness and maintainability. Built following best practices in software engineering, this project demonstrates real-world enterprise application development using Python, SQL, and software design principles.

# IMPLEMENTATION

## The main goals of the CarConnect system are:

## To create a robust and scalable backend system for vehicle rental services.

## To implement clean and modular architecture using DAO, Entity, Service, and Exception layers.

## To support user authentication for both customers and administrators.

## To provide CRUD operations for Customers, Vehicles, and Reservations.

## To ensure data integrity via foreign key constraints and validations.

## To test and validate system functionalities using a centralized unit test suite.

**SYSTEM ARCHITECTURE & FLOW**

The CarConnect architecture follows a **layered design**, enabling clean separation of concerns and improved maintainability.

## Data Flow:

## The user (admin/customer) interacts with the system via service methods.

## Authentication checks credentials and routes requests accordingly.

## Services call DAO classes to fetch/store data.

## The DAO layer communicates with the MySQL database.

## Exceptions are raised and handled when validations or operations fail.

## KEY MODULES

## Here’s a breakdown of the critical components implemented:

## 1. Entity Classes (entity/)

## Customer, Vehicle, Reservation, Admin

## Encapsulate data with private variables, getters/setters, and authenticate() method.

## 2. DAO Layer (dao/)

## Handles database operations.

## Implements interfaces for abstraction (ICustomerService, etc.).

## Includes VehicleService, CustomerService, ReservationService, AdminService.

## 3. Authentication Module (authentication/)

## AuthenticationService.py provides methods to validate user credentials.

## 4. Utility Classes (util/)

## DBConnUtil.py – Manages MySQL database connection.

## DBPropertyUtil.py – Reads DB credentials from a .properties file.

## 5. Exception Handling (exception/)

## Custom exceptions like AuthenticationException, DatabaseConnectionException, etc., ensure graceful error management.

## 6. Testing (main/test\_services.py)

## A comprehensive test suite covering CRUD operations, validation, and data integrity.

## Uses custom output formatting for test tracking.

## DATABASE DESIGN

## The backend relies on a normalized MySQL database, ensuring referential integrity and optimized access.

## Key Tables:

## Customer

## CustomerID (PK)

## FirstName, LastName, Email, Username, Password, Phone, RegistrationDate

## Vehicle

## VehicleID (PK)

## Model, Make, Year, Color, RegistrationNumber, DailyRate, Availability

## Reservation

## ReservationID (PK)

## CustomerID (FK), VehicleID (FK)

## StartDate, EndDate, TotalCost, Status

## Admin

## AdminID (PK)

## Username, Password, Role, JoinDate

## All relationships are enforced with foreign keys, ensuring consistent, relational integrity during create/delete operations.

## TECHNICAL HIGHLIGHTS

## Python OOP: All entity classes are encapsulated using private members and accessors.

## Layered Architecture: DAO, Service, Utility, and Entity layers ensure scalability.

## Custom Exception Handling: For managing invalid credentials, DB failures, and data integrity issues.

## Database Integration: Secure and configurable MySQL access using properties file.

## Modular and Maintainable: Easy to extend functionality by adding new service or entity modules.

## Clean Testing Output: Custom messages for success and organized unit test outputs.

## UNIT TESTING

## All critical services are validated using a centralized test file test\_services.py.

## Test Cases Covered:

## Admin login verification.

## Customer registration and retrieval.

## Reservation creation and cost calculation.

## Vehicle addition and availability checks.

## Test cases use unique input combinations to avoid conflicts (like duplicate emails or registration numbers).

## CONCLUSION

## The CarConnect case study successfully demonstrates how a modular and scalable Python backend system can be built to support real-world business operations such as vehicle rental. Through clean architecture, layered components, and reliable database integration, it provides a foundation for expanding into a full-fledged enterprise application. With comprehensive unit tests, exception handling, and separation of concerns, the solution ensures reliability and future maintainability.

## This project showcases effective backend system design that is production-ready and adaptable for integration with frontend interfaces like web or mobile apps.