Maintenance & Support Plan

Purpose

This document describes how to maintain the Car Rental System over time, how to manage versions, and how to keep backward compatibility as the project evolves. The main goal is to keep the system stable, easy to fix, and safe to extend.

1) Maintenance (day-to-day support)

1.1 Bug handling workflow (support process)

When an issue is reported, I will follow a simple support workflow:

1) Reproduce the problem
- Record the exact steps (menu options + inputs) and the error message.

2) Identify the layer (UI / Service / Repository / DB)
- UI issues: wrong prompt / validation loop
- Service issues: incorrect rules (rental days, overlap, approval)
- Repository/DB issues: SQL, commit, DB path, missing tables

3) Fix + regression check
- Re-run the demo flow (see Health Check below) to make sure core features still work.

This makes debugging easier because responsibilities are separated by design.

1.2 Database care (SQLite)

The system uses SQLite and stores data in `data/app.db`.

Backup strategy (manual but reliable):
Before making major changes, copy the database file:

```
cp data/app.db data/app_backup_YYYYMMDD.db
```

For marking/demos, it is often better to start clean by deleting the DB so it gets recreated automatically:

```
rm -f data/app.db
```

1.3 Logging / audit trail

Admin actions (e.g., approve/reject booking) can be recorded in an audit log table (e.g., `audit_logs`).

If audit logs grow large, keep a simple retention policy:
- Keep the most recent 12 months (or a fixed maximum number of rows) and archive older entries if needed.

1.4 Data cleanup (optional policy)

To keep the database small over time:
- Cancelled bookings older than a set period (e.g., 180 days) can be archived.
- For this MVP, archiving is optional and can be added later if needed.

1.5 Health check (quick regression check)

After changes or bug fixes, run a minimal health check to ensure the core flow still works:

- Admin login - Add car - List cars
- Customer register/login - Create booking
- Admin list pending - Approve booking
- Customer view bookings - confirm booking status is approved

This matches the "Demo Script" in README and helps confirm the system still works end-to-end.

2) Versioning strategy

2.1 Semantic Versioning (SemVer)

I will use `MAJOR.MINOR.PATCH` (e.g., 1.0.0):

- PATCH: bug fixes, no breaking changes (e.g., fix overlap bug, improve input validation)
- MINOR: new backward-compatible features (e.g., add a new optional field, new CLI menu option)
- MAJOR: breaking changes (e.g., renamed CLI commands, major schema changes)

2.2 Release tagging and changelog

- Tag releases in git: `v1.0.0`, `v1.1.0`, etc.
- Maintain a short changelog (even a simple markdown list) describing what changed and why.

2.3 Example roadmap

- 1.0.0: core CLI booking flow (customer booking + admin approval)
- 1.1.0: small improvements (more add-ons, better pricing breakdown, minor reports)
- 2.0.0: larger changes (payment integration, multi-branch inventory, breaking schema changes)

3) Backward compatibility

3.1 Database schema changes (SQLite)

Preferred strategy is additive changes:

- Add new columns instead of removing old ones.
- Provide safe defaults for new columns so old rows remain valid.

Example:
If we add `discount_total`, existing records should default to `0.00`. Service code should treat missing optional fields gracefully.

3.2 Migration approach

For this MVP, schema migrations can be handled manually (run SQL once when required).

For future improvements, maintain versioned migration files (for example):

- `migrations/001_add_discount.sql`
- `migrations/002_add_coupon_code.sql`

Migrations should be applied in order and tested on a backup DB first.

3.3 CLI compatibility

To avoid breaking users:

- Avoid renaming CLI commands unless necessary.
- If a command must change, keep the old command as an alias for at least one release, and print a deprecation message.

Summary

This plan keeps the system stable and maintainable by using a clear support workflow, safe database practices, semantic versioning, and backward-compatible schema changes. It supports steady improvements without breaking existing data or core user workflows.